

O'REILLY®

Early Release

RAW & UNEDITED

# Data Algorithms

RECIPES FOR SCALING UP WITH HADOOP AND SPARK

Mahmoud Parsian

# Data Algorithms:

Recipes for Scaling up with Hadoop and Spark

---

*Mahmoud Parsian*

# **Data Algorithms:**

## Recipes for Scaling up with Hadoop and Spark

Mahmoud Parsian

September 15, 2014

## **Dedication**

This book is dedicated to my dear family:  
my wife, Behnaz,  
my daughter, Maral,  
my son, Yaseen

# Contents

<b>Preface</b>	<b>xxiii</b>
<b>Preface</b>	<b>xxiv</b>
0.1 Introduction . . . . .	xxiv
0.2 Relationship of Spark and Hadoop . . . . .	xxvi
0.3 What is MapReduce? . . . . .	xxxii
0.4 Why use MapReduce? . . . . .	xxxii
0.5 What Is in This Book? . . . . .	xxxiv
0.6 What Is the Focus of This Book? . . . . .	xxxv
0.7 What are Core Concepts of MapReduce/Hadoop? . . . . .	xxxvi
0.8 Is MapReduce for Everything? . . . . .	xxxvii
0.9 What is not MapReduce . . . . .	xxxvii
0.10 Who Is This Book For? . . . . .	xxxviii
0.11 What Software Is Used in This Book? . . . . .	xxxix
0.12 Using Code Examples . . . . .	xxxix
0.13 Where NOT to use MapReduce? . . . . .	xl
0.14 Chapters in This Book? . . . . .	xli
0.15 Online Resources . . . . .	xlii
0.16 Comments and Questions for This Book? . . . . .	xliii
<b>1 Secondary Sort: Introduction</b>	<b>1</b>
1.1 What is a Secondary Sort Problem? . . . . .	1
1.2 Solutions to Secondary Sort Problem . . . . .	3
1.2.1 Sort Order of Intermediate Keys . . . . .	5
1.3 Data Flow Using Plug-in Classes . . . . .	8

1.4	Mapreduce/Hadoop Solution . . . . .	10
1.4.1	Input . . . . .	10
1.4.2	Expected Output . . . . .	10
1.4.3	map() function . . . . .	10
1.4.4	reduce() function . . . . .	11
1.4.5	Hadoop Implementation . . . . .	11
1.4.6	Sample Run of Hadoop Implementation . . . . .	13
1.4.7	Sample Run . . . . .	14
1.5	What If Sorting Ascending or Descending . . . . .	15
1.6	Spark Solution To Secondary Sorting . . . . .	16
1.6.1	Time-Series as Input . . . . .	16
1.6.2	Expected Output . . . . .	17
1.6.3	Option-1: Secondary Sorting in Memory . . . . .	17
1.6.4	Spark Sample Run . . . . .	25
1.6.5	Option-2: Secondary Sorting using Framework . . . . .	30
<b>2</b>	<b>Secondary Sorting: Detailed Example</b>	<b>31</b>
2.1	Introduction . . . . .	31
2.2	Secondary Sorting Technique . . . . .	32
2.3	Complete Example of Secondary Sorting . . . . .	37
2.3.1	Problem Statement . . . . .	37
2.3.2	Input Format . . . . .	37
2.3.3	Output Format . . . . .	38
2.3.4	Composite Key . . . . .	38
2.3.5	Sample Run . . . . .	42
2.4	Secondary Sort using New Hadoop API . . . . .	44
<b>3</b>	<b>Top 10 List</b>	<b>47</b>
3.1	Introduction . . . . .	47
3.2	Top-N Formalized . . . . .	48
3.3	MapReduce Solution . . . . .	49
3.4	Implementation in Hadoop . . . . .	54
3.4.1	Input . . . . .	55
3.4.2	Sample Run 1: find top 10 list . . . . .	55
3.4.3	Output . . . . .	57
3.4.4	Sample Run 2: find top 5 list . . . . .	57
3.5	Bottom 10 . . . . .	58
3.6	Spark Implementation: Unique Keys . . . . .	58

3.6.1	Introduction . . . . .	59
3.6.2	What is an RDD? . . . . .	60
3.6.3	Spark's Function Classes . . . . .	60
3.6.4	Spark Solution for Top-10 Pattern . . . . .	61
3.6.5	Complete Spark Solution for Top-10 Pattern . . . . .	62
3.6.6	Input . . . . .	67
3.6.7	Sample Run : find top-10 list . . . . .	67
3.7	What If for Top-N . . . . .	69
3.7.1	Shared Data Structures Definition and Usage . . . . .	70
3.8	What If for Bottom-N . . . . .	70
3.9	Spark Implementation : Non-Unique Keys . . . . .	71
3.9.1	Complete Spark Solution for Top-10 Pattern . . . . .	73
<b>4</b>	<b>Left Outer Join in MapReduce</b>	<b>84</b>
4.1	Introduction . . . . .	84
4.2	Implementation of Left Outer Join in MapReduce . . . . .	89
4.2.1	MapReduce Phase-1 . . . . .	89
4.2.2	MapReduce Phase-2: Counting Unique Locations . . . . .	94
4.2.3	Implementation Classes in Hadoop . . . . .	95
4.3	Sample Run . . . . .	95
4.3.1	Input for Phase-1 . . . . .	95
4.3.2	run Phase-1 . . . . .	96
4.3.3	View Output of Phase-1 (Input of Phase-2) . . . . .	97
4.3.4	Run Phase-2 . . . . .	97
4.3.5	View Output of Phase-2 . . . . .	98
4.4	Spark Implementation . . . . .	98
4.4.1	Spark Program . . . . .	101
4.4.2	STEP-0: Import Required Classes . . . . .	101
4.4.3	STEP-1: Read Input Parameters . . . . .	102
4.4.4	STEP-2: Create JavaSparkContext Object . . . . .	103
4.4.5	STEP-3: Create a JavaPairRDD for Users . . . . .	103
4.4.6	STEP-4: Create a JavaPairRDD for Transactions . . . . .	104
4.4.7	STEP-5: Create a union of RDD's created by STEP-3 and STEP-4 . . . . .	104
4.4.8	STEP-6: Create a JavaPairRDD(userID, List(T2)) by calling groupBy() . . . . .	105
4.4.9	STEP-7: Create a productLocationsRDD as JavaPairRDD(String, String) . . . . .	106

4.4.10	STEP-8: Find all locations for a product . . . . .	108
4.4.11	STEP-9: Finalize output by changing "value" . . . . .	108
4.4.12	STEP-10: Print the final result RDD . . . . .	109
4.4.13	Running Spark Solution . . . . .	109
4.5	Running Spark on YARN . . . . .	111
4.5.1	Script to Run Spark on YARN . . . . .	111
4.5.2	Running Script . . . . .	112
4.5.3	Checking Expected Output . . . . .	114
4.6	Left Outer Join by Spark's leftOuterJoin() . . . . .	114
4.6.1	High-Level Steps . . . . .	116
4.6.2	STEP-0: import required classes and interfaces . . . . .	117
4.6.3	STEP-1: read input parameters . . . . .	117
4.6.4	STEP-2: create Spark's context object . . . . .	118
4.6.5	STEP-3: create RDD for user's data . . . . .	118
4.6.6	STEP-4: Create usersRDD: The "right" Table . . . . .	119
4.6.7	STEP-5: create transactionRDD for transaction's data	119
4.6.8	STEP-6: Create transactionsRDD: The Left Table . . .	120
4.6.9	STEP-7: use Spark's built-in JavaPairRDD.leftOuterJoin() method . . . . .	120
4.6.10	STEP-8: create (product, location) pairs . . . . .	121
4.6.11	STEP-9: group (K=product, V=location) pairs by K .	122
4.6.12	STEP-10: create final output (K=product, V=Set(location))	122
4.6.13	Sample Run by YARN . . . . .	123
<b>5</b>	<b>Order Inversion Pattern</b>	<b>126</b>
5.1	Introduction . . . . .	126
5.2	Example of Order Inversion Pattern . . . . .	128
5.3	MapReduce for Order Inversion Pattern . . . . .	129
5.3.1	Custom Partitioner . . . . .	130
5.3.2	Relative Frequency Mapper . . . . .	131
5.3.3	Relative Frequency Reducer . . . . .	133
5.3.4	Implementation Classes in Hadoop . . . . .	134
5.4	Sample Run . . . . .	135
5.4.1	Input . . . . .	135
5.4.2	Running MapReduce Job . . . . .	135
5.4.3	Generated Output . . . . .	135

<b>6 Moving Average</b>	<b>137</b>
6.1 Introduction . . . . .	137
6.1.1 Example-1: Time Series Data . . . . .	138
6.1.2 Example-2: Time Series Data . . . . .	138
6.2 Formal Definition . . . . .	139
6.3 Moving Average by POJO . . . . .	140
6.3.1 First solution: using Queue . . . . .	140
6.3.2 Second Solution : using Array . . . . .	141
6.3.3 Testing of Moving Average . . . . .	142
6.3.4 Sample Run . . . . .	143
6.4 MapReduce Solution . . . . .	143
6.4.1 Input . . . . .	144
6.4.2 Output . . . . .	144
6.4.3 MapReduce Solution: Option-1: sort in RAM . . . . .	145
6.4.4 Hadoop Implementation: sort in RAM . . . . .	148
6.4.5 Sample Run . . . . .	148
6.4.6 MapReduce Solution: Option-2: Sort by MR Framework	151
6.5 Sample Run . . . . .	157
<b>7 Market Basket Analysis</b>	<b>160</b>
7.1 What is Market Basket Analysis? . . . . .	160
7.2 MapReduce/Hadoop Solution . . . . .	161
7.3 What are the Application areas for MBA? . . . . .	163
7.4 Market Basket Analysis using MapReduce . . . . .	163
7.4.1 Mapper Formal . . . . .	167
7.4.2 Reducer . . . . .	168
7.5 MapReduce/Hadoop Implementation Classes . . . . .	169
7.5.1 Find Sorted Combinations . . . . .	169
7.5.2 Market Basket Analysis Driver: MBADriver . . . . .	170
7.5.3 Market Basket Analysis Mapper: MBAMapper . . . . .	171
7.5.4 Sample Run . . . . .	173
7.6 Spark/Hadoop Solution . . . . .	174
7.6.1 MapReduce Algorithm . . . . .	176
7.6.2 Input . . . . .	176
7.6.3 Spark Implementation . . . . .	176
7.6.4 Creating Item Sets From Transactions . . . . .	191

<b>8</b>	<b>Common Friends</b>	<b>194</b>
8.1	Introduction . . . . .	194
8.2	Input . . . . .	195
8.3	Common Friends Algorithm . . . . .	196
8.4	MapReduce Algorithm . . . . .	197
8.4.1	MapReduce Algorithm in Action . . . . .	198
8.5	Solution 1: Hadoop Implementation using Text . . . . .	201
8.5.1	Sample Run for Solution 1 . . . . .	202
8.6	Solution 2: Hadoop Implementation using ArrayListOfLongsWritable . . . . .	205
8.6.1	Sample Run for Solution 2 . . . . .	205
8.7	Spark Solution . . . . .	207
8.7.1	STEP-0: Import Required Classes . . . . .	209
8.7.2	STEP-1: Check Input Parameters . . . . .	210
8.7.3	STEP-2: Create a JavaSparkContext Object . . . . .	210
8.7.4	STEP-3: Read Input . . . . .	210
8.7.5	STEP-4: Apply a Mapper . . . . .	211
8.7.6	STEP-5: Apply a Reducer . . . . .	212
8.7.7	STEP-6: Find Common Friends . . . . .	212
8.8	Sample Run of a Spark Program . . . . .	214
8.8.1	HDFS Input . . . . .	214
8.8.2	Script to Run Spark Program . . . . .	214
8.8.3	Log of Sample Run . . . . .	215
<b>9</b>	<b>Recommendation Engines using MapReduce</b>	<b>218</b>
9.1	Customers Who Bought This Item Also Bought . . . . .	220
9.1.1	Input . . . . .	220
9.1.2	Expected Output . . . . .	220
9.1.3	MapReduce Solution . . . . .	220
9.2	Frequently Bought Together . . . . .	226
9.2.1	Input . . . . .	227
9.2.2	MapReduce Solution . . . . .	228
9.3	Recommend People Connection . . . . .	232
9.3.1	Input . . . . .	234
9.3.2	Output . . . . .	235
9.3.3	MapReduce Solution . . . . .	236
9.4	Spark Implementation . . . . .	237
9.4.1	STEP-0: Import Required Classes . . . . .	239

9.4.2	STEP-1: Handle Input Parameters . . . . .	239
9.4.3	STEP-2: Create Spark's Context Object . . . . .	240
9.4.4	STEP-3: Read HDFS Input File . . . . .	240
9.4.5	STEP-4: Implement map() Function . . . . .	241
9.4.6	STEP-5: Implement reduce() Function . . . . .	243
9.4.7	STEP-6: Generate Final Output . . . . .	243
9.4.8	Convenient Methods . . . . .	244
9.4.9	HDFS Input . . . . .	245
9.4.10	Script to Run Spark Program . . . . .	245
9.4.11	Program Run Log . . . . .	246
<b>10</b>	<b>Content-Based Recommendation: Movies</b>	<b>252</b>
10.1	Input . . . . .	253
10.2	MapReduce PHASE-1 . . . . .	254
10.3	MapReduce PHASE-2 and PHASE-3 . . . . .	255
10.4	MapReduce-Phase-2 Mapper . . . . .	256
10.5	MapReduce-Phase-2 Reducer . . . . .	257
10.6	MapReduce-Phase-3 Mapper . . . . .	259
10.7	MapReduce-Phase-3 Reducer . . . . .	260
10.8	More Similarity Measures . . . . .	262
10.9	Movie Recommendation in Spark . . . . .	263
10.9.1	High-Level Solution in Spark . . . . .	264
10.9.2	High-Level Solution: All Steps . . . . .	264
10.9.3	STEP-0: Import Required Classes . . . . .	265
10.9.4	STEP-1: Handle Input Parameters . . . . .	265
10.9.5	STEP-2: Create a Spark's Context Object . . . . .	266
10.9.6	STEP-3: Read Input File and Create RDD . . . . .	266
10.9.7	STEP-4: Find Who Has Rated Movies . . . . .	267
10.9.8	STEP-5: Group moviesRDD by Movie . . . . .	268
10.9.9	STEP-6: Find Number of Raters per Movie . . . . .	269
10.9.10	STEP-7: Perform Self-Join . . . . .	270
10.9.11	STEP-8: Remove Duplicate (movie1, movie2) Pairs . . . . .	272
10.9.12	STEP-9: Generate All (movie1, movie2) Combinations . . . . .	273
10.9.13	STEP-10: Group Movie Pairs . . . . .	274
10.9.14	STEP-11: Calculate Correlations . . . . .	274
10.9.15	STEP-12: Print Final Results . . . . .	275
10.9.16	Helper Method: calculateCorrelations() . . . . .	275
10.9.17	Helper Method: calculatePearsonCorrelation() . . . . .	276

10.9.18 Helper Method: calculateCosineCorrelation() . . . . .	277
10.9.19 Helper Method: calculateJaccardCorrelation() . . . . .	277
10.10 Sample Run of Spark Program . . . . .	277
10.10.1 HDFS Input . . . . .	277
10.10.2 Script . . . . .	278
10.11 Log of Sample Run . . . . .	279
10.11.1 Inspecting HDFS Output . . . . .	283
<b>11 Smarter Email Marketing with Markov Model</b>	<b>285</b>
11.1 Introduction . . . . .	285
11.2 Markov Chain in a Nutshell . . . . .	286
11.3 Markov Model using MapReduce . . . . .	288
11.3.1 MapReduce to Generate Time-ordered Transactions .	290
11.3.2 MapReduce to Generate Markov State Transition .	300
11.4 Using Markov Model to Predict Next Email Marketing Date	306
<b>12 K-Means Clustering</b>	<b>308</b>
12.1 Introduction . . . . .	308
12.2 What is K-Means Clustering . . . . .	311
12.3 What are the Applications of Clustering? . . . . .	312
12.4 K-Means Clustering Method: Partitioning Approach . . . .	313
12.5 K-Means Distance Function . . . . .	314
12.6 K-Means Clustering Step-by-Step Example . . . . .	315
12.7 K-Means Clustering Formalized . . . . .	315
12.8 MapReduce Solution for K-Means Clustering . . . . .	316
12.8.1 MapReduce Solution: map() . . . . .	318
12.8.2 MapReduce Solution: combine() . . . . .	319
12.8.3 MapReduce Solution: reduce() . . . . .	320
12.9 MapReduce K-Means Clustering Step-by-Step Example . . .	321
12.10 K-Means Implementation by Spark . . . . .	321
12.10.1 Sample Run of K-Means by Spark . . . . .	324
<b>13 kNN: k-Nearest-Neighbors</b>	<b>326</b>
13.1 Introduction . . . . .	326
13.2 kNN Classification . . . . .	327
13.3 Distance Functions . . . . .	328
13.4 kNN Example . . . . .	329
13.5 An Informal kNN Algorithm . . . . .	329

13.6	Formal kNN Algorithm . . . . .	331
13.6.1	Java-like Non-MapReduce Solution for kNN . . . . .	331
13.7	kNN Implementation in Spark . . . . .	333
13.7.1	Formalizing kNN for Spark Implementation . . . . .	334
13.7.2	Input Data Set Formats . . . . .	335
13.7.3	kNN Implementation in Spark . . . . .	337
<b>14</b>	<b>Naive Bayes</b>	<b>352</b>
14.1	Introduction . . . . .	352
14.2	Training and Learning Stage . . . . .	354
14.2.1	Example: Training Data (Numeric Data) . . . . .	355
14.2.2	Example: Training Data (Symbolic Data) . . . . .	356
14.3	Conditional Probability . . . . .	357
14.4	The Naive Bayes Classifier . . . . .	358
14.4.1	The Naive Bayes Classifier Example . . . . .	359
14.5	The Naive Bayes Classifier: MapReduce Solution for Symbolic Data . . . . .	362
14.5.1	STAGE-1: Building Classifier Using Symbolic Training Data . . . . .	362
14.5.2	STAGE-2: Using Classifier To Classify New Symbolic Data . . . . .	369
14.6	The Naive Bayes Classifier: MapReduce Solution for Numeric Data . . . . .	371
14.7	Naive Bayes Classifier Implementation in Spark . . . . .	374
14.7.1	STAGE-1: Building Classifier Using Training Data . . . . .	375
14.7.2	STAGE-2: Using Classifier To Classify New Data . . . . .	385
14.8	Using Apache Mahout . . . . .	391
<b>15</b>	<b>Sentiment Analysis</b>	<b>392</b>
15.1	Introduction . . . . .	392
15.1.1	Sentiment Examples . . . . .	393
15.1.2	Sentiment Scores: Positive or Negative . . . . .	393
15.2	Steps for Sentiment Analysis . . . . .	395
15.3	A Simple MapReduce for Sentiment Analysis . . . . .	395
15.3.1	map() for Sentiment Analysis . . . . .	396
15.3.2	reduce() for Sentiment Analysis . . . . .	397

<b>16 Finding, Counting and Listing all Triangles in Large Graphs</b>	<b>398</b>
16.1 Introduction . . . . .	398
16.2 Basic Graph Concepts . . . . .	399
16.3 Importance of Counting Triangles . . . . .	402
16.4 MapReduce Solution . . . . .	402
16.5 MapReduce in Action . . . . .	403
16.6 STEP-3: Remove Duplicate Triangles . . . . .	406
16.6.1 STEP-3: Mapper . . . . .	406
16.6.2 STEP-3: Reducer . . . . .	407
16.7 Hadoop Implementation . . . . .	408
16.7.1 Sample Run . . . . .	408
16.8 Spark Implementation . . . . .	413
16.8.1 STEP-0: Import Required Classes and Interfaces . . . . .	413
16.8.2 STEP-1: Read Input Parameters . . . . .	414
16.8.3 STEP-2: Create a Spark Context Object . . . . .	414
16.8.4 STEP-3: Read Graph via HDFS Input . . . . .	415
16.8.5 STEP-4: Create All Graph Edges . . . . .	415
16.8.6 STEP-5: Create RDD To Generate Triads . . . . .	416
16.8.7 STEP-6: Create All Possible Triads . . . . .	417
16.8.8 STEP-7: Create RDD To Generate Triangles . . . . .	418
16.8.9 STEP-8: Create All Triangles . . . . .	419
16.8.10 Step-9: Create Unique Triangles . . . . .	421
16.9 Spark Sample Run . . . . .	421
16.9.1 Input . . . . .	421
16.9.2 Script . . . . .	422
16.9.3 Running Script . . . . .	422
<b>17 K-mer Counting</b>	<b>425</b>
17.1 Introduction to K-mers . . . . .	425
17.2 K-mer counting using MapReduce . . . . .	427
17.2.1 K-mer counting using MapReduce: map() . . . . .	427
17.2.2 K-mer counting using MapReduce: reduce() . . . . .	427
17.2.3 K-mer Counting with MapReduce and Hadoop . . . . .	428
17.3 Input Data for K-mer Counting . . . . .	429
17.3.1 Sample Runs of K-mer Counting . . . . .	429
17.4 K-mer Implementation in Spark . . . . .	430
17.4.1 K-mer High-Level Solution in Spark . . . . .	431
17.4.2 STEP-0: import required classes and interfaces . . . . .	432

17.4.3	createJavaSparkContext()	432
17.4.4	STEP-1: handle input parameters	433
17.4.5	STEP-2: create a Spark context object	433
17.4.6	STEP-3: broadcast global shared objects	433
17.4.7	STEP-4: read FASTQ file from HDFS and create the first RDD	434
17.4.8	STEP-5: filter redundant records	435
17.4.9	STEP-6: generate K-mers	436
17.4.10	STEP-7: combine/reduce frequent kmers	437
17.4.11	STEP-8: create a local top-N	438
17.4.12	STEP-9: Find Final top-N	439
17.4.13	STEP-10: Emit Final top-N	439
17.4.14	YARN Script for Spark	440
17.4.15	HDFS Input	440
17.4.16	Output for Final Top-N	441
<b>18</b>	<b>DNA-Sequencing</b>	<b>442</b>
18.1	Introduction	442
18.2	Input to DNA-Sequencing	446
18.3	Input data validation	446
18.4	DNA-Sequencing: Alignment	447
18.5	MapReduce Algorithms for DNA-Sequencing	448
18.6	MR Algorithms: Step-1: DNA-Sequencing: Alignment	453
18.6.1	Step-1: map(): Alignment	455
18.6.2	Step-1: reduce(): Alignment	456
18.7	Step-2: DNA-Sequencing: Recalibration	461
18.8	Step-3: DNA-Sequencing: Variant Detection	466
18.8.1	Variant Detection Mapper	467
18.8.2	Variant Detection Reducer	468
<b>19</b>	<b>Cox Regression</b>	<b>470</b>
19.1	Introduction to Survival Analysis using Cox Regression	470
19.2	Cox Model in a Nutshell	471
19.3	MapReduce Solution for Cox Regression	472
19.3.1	Cox Regression Basic Terminology	473
19.4	Cox Regression by using R Language	474
19.5	Problem Statement	475
19.6	Cox Regression POJO Solution	476

19.7	Input for MapReduce . . . . .	477
19.8	Cox Regression by MapReduce . . . . .	479
19.8.1	Cox Regression PHASE-1: map() . . . . .	479
19.8.2	Cox Regression PHASE-1: reduce() . . . . .	480
19.8.3	Cox Regression PHASE-2: map() . . . . .	481
19.8.4	Sample Output Generated by PHASE-2 reduce() . . . . .	483
19.8.5	Sample Output Generated by PHASE-2 map() . . . . .	484
19.8.6	Cox Regression by MapReduce: How Does It Work . . . . .	484
<b>20</b>	<b>Cochran-Armitage Test for Trend</b>	<b>486</b>
20.1	Introduction . . . . .	486
20.2	Cochran-Armitage Algorithm . . . . .	487
20.3	Application of Cochran-Armitage . . . . .	493
20.4	MapReduce Solution . . . . .	496
20.4.1	Input . . . . .	496
20.4.2	Expected Output . . . . .	497
20.4.3	Mapper . . . . .	498
20.4.4	Reducer . . . . .	499
20.5	MapReduce/Hadoop Implementation . . . . .	503
20.5.1	Sample Run of MapReduce/Hadoop Implementation . . . . .	504
<b>21</b>	<b>Allelic Frequency</b>	<b>507</b>
21.1	Introduction . . . . .	507
21.2	Basic Definitions . . . . .	509
21.2.1	Chromosome . . . . .	509
21.2.2	Bioset . . . . .	509
21.2.3	Allele and Allelic Frequency . . . . .	510
21.2.4	Source of Data for Allelic Frequency . . . . .	510
21.2.5	Allelic Frequency Analysis by Fisher's Exact Test . . . . .	512
21.2.6	Fisher's Exact Test . . . . .	512
21.3	Formal Problem Statement . . . . .	514
21.4	MapReduce Phase-1 . . . . .	515
21.4.1	Input . . . . .	516
21.4.2	Output/Result . . . . .	517
21.4.3	MapReduce Solution for Allelic Frequency . . . . .	519
21.4.4	Phase-1 Mapper . . . . .	519
21.4.5	Phase-1 Reducer . . . . .	520
21.4.6	Sample Run of MapReduce/Hadoop Implementation . . . . .	526

21.4.7	Sample Plot of Pvalues . . . . .	528
21.5	MapReduce Phase-2 . . . . .	530
21.5.1	Phase-2: Mapper for Bottom-100 . . . . .	532
21.5.2	Phase-2: Reducer for "Bottom 100" . . . . .	533
21.6	Is Bottom 100 List A Monoid? . . . . .	534
21.6.1	Hadoop Solution for Bottom 100 List . . . . .	535
21.6.2	Sample Run of Bottom 100 List . . . . .	536
21.7	MapReduce Phase-3 . . . . .	536
21.7.1	Phase-3: Mapper for Bottom-100 . . . . .	538
21.7.2	Phase-3: Reducer for "Bottom 100" . . . . .	539
21.7.3	Hadoop Solution for Bottom 100 List Per Chromosome	541
21.7.4	Sample Run of Bottom 100 List Per Chromosome . . . . .	541
<b>22</b>	<b>The T-Test</b>	<b>542</b>
22.1	Introduction . . . . .	542
22.2	MapReduce Problem Statement . . . . .	547
22.3	Input . . . . .	547
22.4	Expected Output . . . . .	548
22.5	MapReduce Solution . . . . .	548
22.6	Hadoop Implementation . . . . .	550
22.7	Spark Implementation . . . . .	551
22.7.1	High Level Steps . . . . .	553
22.7.2	STEP-0: import required classes and interfaces . . . . .	554
22.7.3	Create JavaSparkContext . . . . .	554
22.7.4	Create TimeTable Data Structure . . . . .	555
22.7.5	Create RDD for All Biosets . . . . .	555
22.7.6	STEP-1: handle input parameters . . . . .	556
22.7.7	Create time table data structure . . . . .	556
22.7.8	STEP-3: create a spark context object . . . . .	556
22.7.9	High Level Steps . . . . .	557
22.7.10	STEP-5: create RDD for all biosets . . . . .	557
22.7.11	STEP-6: map bioset records into JavaPairRDD(K,V) pairs . . . . .	557
22.7.12	STEP-7: group biosets by GENE-ID . . . . .	557
22.7.13	STEP-8: perform Ttest for every GENE-ID . . . . .	558
22.7.14	Ttest Algorithm . . . . .	559
22.7.15	Input for Spark Program . . . . .	560
22.7.16	Spark on YARN Script . . . . .	560

22.7.17 Sample Run of Script . . . . .	561
22.7.18 Generated Outputs . . . . .	561
<b>23 Computing Pearson Correlation</b>	<b>563</b>
23.1 Pearson Correlation Formula . . . . .	564
23.2 Pearson Correlation by Example . . . . .	569
23.3 Data Set for Pearson Correlation . . . . .	569
23.4 POJO Solution for Pearson Correlation . . . . .	569
23.5 MapReduce Solution for Pearson Correlation . . . . .	572
23.5.1 map() for Pearson Correlation . . . . .	573
23.5.2 reduce() for Pearson Correlation . . . . .	574
23.5.3 reduce() for Pearson Correlation . . . . .	574
23.6 Hadoop Implementation for Pearson Correlation . . . . .	575
23.7 Pearson Correlation using Spark/Hadoop . . . . .	576
23.7.1 Input . . . . .	577
23.7.2 Output . . . . .	578
23.7.3 Spark Solution . . . . .	578
23.7.4 Spark Solution: High-Level Steps . . . . .	580
23.7.5 STEP-0: import required classes and interfaces . . . . .	582
23.7.6 Method smaller() . . . . .	582
23.7.7 MutableDouble Class . . . . .	583
23.7.8 Method toMap() . . . . .	584
23.7.9 Method toListOfString() . . . . .	584
23.7.10 Method readBiosets() . . . . .	586
23.7.11 STEP-1: handle input parameters . . . . .	587
23.7.12 STEP-2: create a Spark context object . . . . .	587
23.7.13 STEP-3: create list of input files/biomarkers . . . . .	588
23.7.14 STEP-4: broadcast "reference" as global shared object	589
23.7.15 STEP-5: read all biomarkers from HDFS and create the first RDD . . . . .	589
23.7.16 STEP-6: filter biomarkers by reference . . . . .	590
23.7.17 STEP-7: create (Gene-ID, (Patient-ID, Gene-Value) pairs . . . . .	591
23.7.18 STEP-8: group by gene . . . . .	593
23.7.19 STEP-9: create Cartesian product of all genes . . . . .	593
23.7.20 STEP-10: filter redundant pairs of genes . . . . .	594
23.7.21 STEP-11: calculate Pearson Correlation and p-value .	595
23.7.22 Pearson Class . . . . .	597

23.7.23	Test Pearson Class . . . . .	598
23.7.24	Pearson Correlation Using R . . . . .	599
23.7.25	YARN Script to Run Spark Program . . . . .	600
23.8	Spearman Correlation . . . . .	600
23.8.1	Spearman Correlation Wrapper Class . . . . .	601
23.8.2	Test Spearman Correlation Wrapper Class . . . . .	601
<b>24</b>	<b>DNA Base Count</b>	<b>603</b>
24.1	Introduction . . . . .	603
24.2	FASTA Format . . . . .	604
24.2.1	FASTA Format Example . . . . .	605
24.3	FASTQ Format . . . . .	605
24.3.1	FASTQ Format Example . . . . .	606
24.4	MapReduce Solution: FASTA Format . . . . .	606
24.4.1	Reading FASTA Files . . . . .	606
24.4.2	MapReduce Solution: map() . . . . .	606
24.4.3	MapReduce Solution: reduce() . . . . .	608
24.5	Hadoop Implementation: FASTA Format . . . . .	609
24.5.1	Hadoop Sample Run . . . . .	609
24.5.2	What If 1 . . . . .	610
24.5.3	What If 2 . . . . .	612
24.6	MapReduce Solution: FASTQ Format . . . . .	615
24.6.1	MapReduce Solution: map() . . . . .	615
24.6.2	MapReduce Solution: reduce() . . . . .	617
24.7	Hadoop Implementation . . . . .	617
24.7.1	Sample Run of Hadoop Implementation . . . . .	618
24.7.2	Reading FASTQ Files . . . . .	619
<b>25</b>	<b>RNA-Sequencing</b>	<b>621</b>
25.1	Introduction . . . . .	621
25.2	Data Size and Format . . . . .	622
25.3	MapReduce Solution . . . . .	622
25.3.1	Input data validation . . . . .	622
25.4	MapReduce Algorithms for RNA-Sequencing . . . . .	624
25.4.1	STEP-1: MapReduce Tophat mapping . . . . .	628
25.4.2	STEP-2: MapReduce Calling Cuffdiff . . . . .	633

<b>26 Gene Aggregation</b>	<b>636</b>
26.1 Introduction . . . . .	636
26.2 Input . . . . .	637
26.3 Output . . . . .	638
26.4 MapReduce Solution . . . . .	638
26.4.1 Mapper: Filter by Individual . . . . .	640
26.4.2 Reducer: Filter by Individual . . . . .	642
26.4.3 Mapper: Filter by Average . . . . .	642
26.4.4 Reducer: Filter by Average . . . . .	643
26.5 Computing Gene Aggregation . . . . .	644
26.6 Hadoop Implementation . . . . .	646
26.7 Analysis of Output . . . . .	650
26.8 Gene Aggregation in Spark . . . . .	654
26.9 Gene Aggregation in Spark: Filter by Individual . . . . .	654
26.9.1 High Level Solution . . . . .	656
26.9.2 High Level Solution . . . . .	656
26.9.3 STEP-1: handle input parameters . . . . .	657
26.9.4 STEP-2: Create a Spark Context Object . . . . .	657
26.9.5 STEP-3: Broadcast Shard Variables . . . . .	658
26.9.6 STEP-4: Create a JavaRDD For Biosets . . . . .	658
26.9.7 STEP-5: Map Biosets into JavaPairRDD(K,V) . . . . .	659
26.9.8 STEP-6: filter out the redundant RDD elements . . . . .	660
26.9.9 STEP-7: reduce by Key and sum up the frequency count	661
26.9.10 STEP-8: prepare the final output . . . . .	661
26.9.11 Utillity Functions . . . . .	661
26.9.12 Running Spark on YARN . . . . .	663
26.10 Gene Aggregation in Spark: Filter by Average . . . . .	664
26.10.1 STEP-0: import required classes and interfaces . . . . .	666
26.10.2 STEP-1: handle input parameters . . . . .	666
26.10.3 STEP-2: create a Java Spark context object . . . . .	667
26.10.4 STEP-3: share global variables in all cluster nodes . . . . .	668
26.10.5 STEP-4: read all bioset records and create an RDD . . . . .	668
26.10.6 STEP-5: map bioset records and create JavaPairRDD(K, V) . . . . .	669
26.10.7 STEP-6: filter redundant records created by STEP-5 . . . . .	670
26.10.8 STEP-7: group biosets by geneID and referenceType . . . . .	670
26.10.9 STEP-8: prepare the final desired output . . . . .	671
26.10.10 STEP-9: emit the final output . . . . .	672

26.10.11 <code>toList()</code> Method . . . . .	672
26.10.12 <code>readInputFiles()</code> Method . . . . .	673
26.10.13 <code>buildPatientsMap()</code> Method . . . . .	673
26.10.14 <code>buildPatientsMap()</code> Method . . . . .	674
26.10.15Running Spark on YARN . . . . .	675
<b>27 Linear Regression</b>	<b>677</b>
27.1 Introduction . . . . .	677
27.2 Simple Facts about Linear Regression . . . . .	678
27.3 Simple Example . . . . .	679
27.4 Problem Statement . . . . .	680
27.5 Input Data . . . . .	682
27.6 Expected Output . . . . .	682
27.7 MapReduce Solution using Apache Commons SimpleRegression	683
27.8 Hadoop Implementation using Apache Commons SimpleRe- gression . . . . .	686
27.9 MapReduce Solution using R's Linear Model . . . . .	687
27.9.1 MapReduce Solution using R's Linear Model: Phase 1	689
27.9.2 MapReduce Solution using R's Linear Model: Phase 2	693
27.9.3 Hadoop Implementation using using R's Linear Model	695
<b>28 MapReduce and Monoids</b>	<b>696</b>
28.1 Introduction . . . . .	696
28.2 Definition of Monoid . . . . .	698
28.2.1 How to form a Monoid? . . . . .	699
28.3 Monoidic and Non-Monoidic Examples . . . . .	700
28.3.1 Subtraction over Set of Integers . . . . .	700
28.3.2 Subtraction over Set of Integers . . . . .	700
28.3.3 Addition over Set of Integers . . . . .	700
28.3.4 Multiplication over Set of Integers . . . . .	701
28.3.5 Mean over Set of Integers . . . . .	701
28.3.6 Non-Commutative Example . . . . .	701
28.3.7 Median over Set of Integers . . . . .	701
28.3.8 Concatenation over Lists . . . . .	702
28.3.9 Union/Intersection over Integers . . . . .	702
28.3.10 Functional Example . . . . .	702
28.3.11 Matrix Example . . . . .	703
28.4 MapReduce Example: Not a Monoid . . . . .	704

28.5	MapReduce Example: Monoid . . . . .	705
28.6	Hadoop Implementation of Monodized MapReduce . . . . .	707
28.7	Sample Run of Monodized Hadoop/MapReduce . . . . .	708
28.7.1	Create Input File (as a SequenceFile) . . . . .	708
28.7.2	Create HDFS Input and Output Directories . . . . .	709
28.7.3	Copy Input File to HDFS and Verify . . . . .	709
28.7.4	Prepare a shell script to run your MapReduce job . . .	710
28.7.5	Run MapReduce Job . . . . .	711
28.7.6	View Hadoop Output . . . . .	712
28.8	Conclusion on Using Monoids . . . . .	712
28.9	Functors and Monoids . . . . .	713
<b>29</b>	<b>The Small Files Problem</b>	<b>716</b>
29.1	Introduction . . . . .	716
29.2	Solution to The Small Files Problem . . . . .	717
29.2.1	Input Data . . . . .	720
29.3	Solution With SmallFilesConsolidator . . . . .	721
29.3.1	Java Source Files . . . . .	721
29.3.2	Sample Run . . . . .	722
29.4	Solution Without SmallFilesConsolidator . . . . .	724
29.4.1	Java Source Files . . . . .	724
29.4.2	Sample Run . . . . .	724
<b>30</b>	<b>Huge Cache for MapReduce</b>	<b>726</b>
30.1	Introduction . . . . .	726
30.2	Implementation Options . . . . .	727
30.3	Fromalizing the Cache Problem . . . . .	729
30.4	Elegant Scalable Solution . . . . .	729
30.5	Implementation of Elegant Scalable Solution . . . . .	734
30.5.1	Use of LRU Map . . . . .	734
30.5.2	Test LRU Map . . . . .	735
30.5.3	Use of MapDB . . . . .	737
30.5.4	Test of MapDB: put() and get() . . . . .	740
30.6	MapReduce Using LRU-Map-Cache . . . . .	741
30.7	CacheManager Definition . . . . .	742
30.7.1	CacheManager Initilization . . . . .	743
30.7.2	CacheManager Closing . . . . .	743
30.7.3	CacheManager Usage . . . . .	744

<b>31 Bloom Filter</b>	<b>745</b>
31.1 Introduction . . . . .	745
31.2 A Simple Bloom Filter Example . . . . .	748
31.3 Bloom Filter in Guava Library . . . . .	749
31.4 Using Bloom Filter in MapReduce . . . . .	751

## Appendices

<b>A Bioset</b>	<b>754</b>
A.1 Introduction . . . . .	754
<b>B Spark RDDs</b>	<b>756</b>
B.1 Introduction . . . . .	756
B.2 What is a TupleN? . . . . .	757
B.3 What is an RDD . . . . .	758
B.4 How to Create RDDs . . . . .	759
B.5 Create RDDs by Collection Objects . . . . .	759
B.6 Collect Elements of an RDD . . . . .	760
B.7 Transform RDD into New RDD . . . . .	761
B.8 Create RDDs by Reading Files . . . . .	761
B.9 Grouping By Key . . . . .	762
B.10 Map Values . . . . .	763
B.11 Reducing By Key . . . . .	764
B.12 Filtering an RDD . . . . .	765
B.13 Saving RDD as HDFS Text File . . . . .	766
B.14 Saving RDD as HDFS Sequence File . . . . .	767
B.15 Reading RDD from HDFS Sequence File . . . . .	768
B.16 Counting RDD . . . . .	768
B.17 Spark RDD Examples in Scala . . . . .	769

# List of Figures

1	Simple View of MapReduce Process . . . . .	xxv
2	Relationship of Spark and Hadoop V1 . . . . .	xxviii
3	Relationship of Spark and Hadoop V2 . . . . .	xxix
1.1	Secondary Sorting Keys . . . . .	6
1.2	Secondary Sorting Data Flow . . . . .	9
2.1	Secondary Sorting Keys . . . . .	34
2.2	Secondary Sorting: Composite and Natural Keys . . . . .	39
3.1	Top-10 MapReduce Algorithm: for Unique Keys . . . . .	51
3.2	Top-10 MapReduce Algorithm: for Non-Unique Keys . . . . .	74
4.1	Left Out Join Illustration . . . . .	86
4.2	Left Outer Join Data Flow – Phase I . . . . .	90
4.3	Left Outer Join Data Flow – Phase II . . . . .	91
4.4	Union Data Flow . . . . .	99
6.1	Composite and Natural Keys . . . . .	153
7.1	MapReduce Algorithm for Market Basket Analysis . . . . .	164
7.2	Generating Association Rules . . . . .	177
7.3	Generating Association Rules: MR Phase I . . . . .	179
7.4	Generating Association Rules: MR Phase I . . . . .	180
9.1	Frequently Bought Together . . . . .	226
9.2	Friendship Graph . . . . .	233

11.1	Markov Work Flow . . . . .	289
11.2	Composite Key for Secondary Sorting . . . . .	297
12.1	Raw Data . . . . .	309
12.2	Clustered Data . . . . .	310
13.1	kNN Classification with 4 Classes {C1, C2, C3, C4} . . . . .	330
13.2	kNN Implementation in Spark . . . . .	336
14.1	Naive Bayes: Training Phase . . . . .	353
14.2	Naive Bayes: Classification . . . . .	354
14.3	Naive Bayes: Training Phase . . . . .	374
16.1	Graph Triangles . . . . .	400
17.1	K-mer MapReduce WorkFlow . . . . .	430
18.1	High-Level View of DNA Sequencing . . . . .	444
18.2	DNA Sequencing Pipeline . . . . .	445
18.3	FreeMarker Template Engine . . . . .	449
18.4	3 Steps MapReduce Solution (Steps 1 and 2) . . . . .	450
18.5	3 Steps MapReduce Solution (Step 3) . . . . .	451
18.6	DNA-Sequencing Data Flow . . . . .	452
18.7	DNA-Sequencing: Alignment Workflow . . . . .	453
18.8	DNA-Sequencing: Recalibration . . . . .	462
21.1	Mutation Example . . . . .	508
21.2	Allelic Frequency WorkFlow . . . . .	511
21.3	p-value for Allelic Frequency . . . . .	531
22.1	Ttest Spark WorkFlow . . . . .	552
23.1	Positive Correlation . . . . .	565
23.2	Negative Correlation . . . . .	566
23.3	No Correlation . . . . .	570
23.4	Pearson Correlation of All-vs-All . . . . .	581
25.1	RNA-Seq WorkFlow . . . . .	623
25.2	RNA-Seq Data Structures . . . . .	626
27.1	Linear Regression by R . . . . .	681

27.2 Linear Regression Model . . . . .	688
30.1 Hige Cache Partition Data Structure . . . . .	733

# Preface

## 0.1 Introduction

With the development of massive search engines (such as Google and Yahoo), genomic analysis (in DNA-Sequencing, RNA-Sequencing, and biomarker analysis), and social networks (such as Facebook and Twitter), data volumes generated and processed cross the peta-bytes scale threshold. To solve these massive computational requirements, we need efficient, scalable, and parallel algorithms. One such framework to tackle these problems is the MapReduce paradigm. The term MapReduce (originated from functional programming) was introduced by Google in a paper called "MapReduce: Simplified Data Processing on Large Clusters." Google's MapReduce [8] implementation is a proprietary solution and has not been released to the public yet.

A simple view of MapReduce process is illustrated below ( 1).

Input data is partitioned into small chunks (here we have 5 input partitions – called chunks), each chunk is sent to a mapper. Each mapper may generate any number of  $(K, V)$  pairs. In this simple example, all mappers only generate 2 keys:  $\{K_1, K_2\}$ . When all mappers are completed, Keys are sorted, shuffled, and grouped and sent to reducers. Finally reducers generate desired outputs. For this example, we have two reducers identified by  $\{K_1, K_2\}$  keys. Once all mappers are completed, then reducers start their execution process. Each reducer may create (as an output) any (zero or more) number of (key, value) pairs.

Simply put, MapReduce is about scalability. Using the MapReduce paradigm,

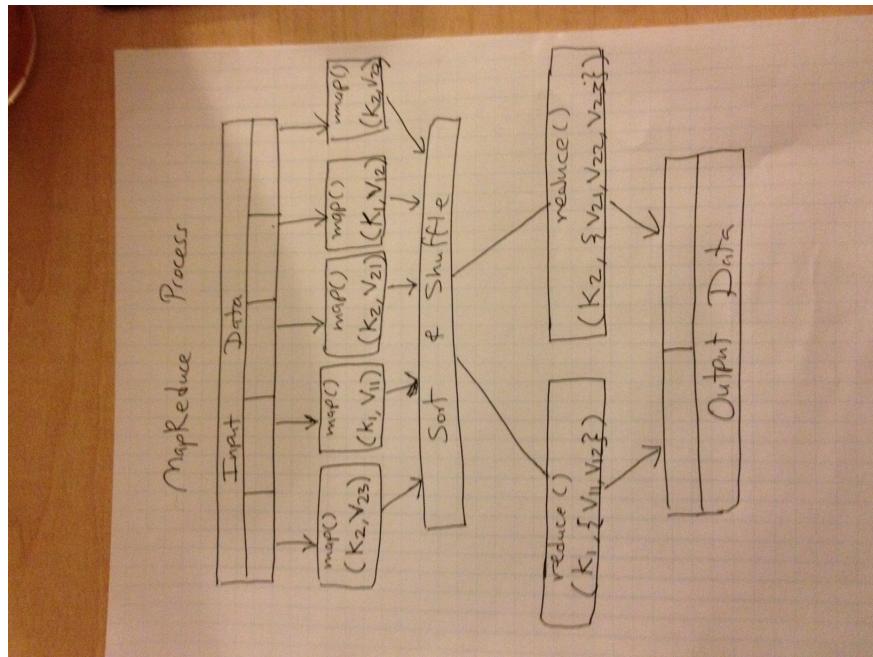


Figure 1: Simple View of MapReduce Process

you focus on writing two functions: `map()` – filter and aggregate data – and `reduce()` – reduce, group, and summarize by keys generated by `map()`. When writing your `map()` and `reduce()` functions, your solution has to be scalable. For example, if you are utilizing any data structure (such as `List`, `Array`, or `HashMap`), which will not easily fit into the memory of a commodity server, then your solution is not scalable. Note that your `map()` and `reduce()` functions will be executing in basic commodity servers, which might have 16GB or 32GB of RAM at most. Scalability is therefore the key and heart of MapReduce. If your MapReduce solution does not "scale out" well, then you should not call it a MapReduce solution. Here, when we talk about scalability, we mean "scaling out" (the term "scale out" means to add more commodity nodes to a system). MapReduce is mainly about "scaling out" (as opposed to "scale up" which means to add resources – such as memory and CPU – to a single node). For example, if DNA-Sequencing<sup>1</sup> takes 60 hours with three servers, then by "scaling out" the solution might produce the same DNA-Sequencing with 50 similar servers in less than 2 hours.

This book provides essential MapReduce algorithms in the following areas and chapters are organized accordingly:

1. **Basic Design Patterns**
2. **Data Mining and Machine Learning**
3. **Bioinformatics, Genomics, and Statistics**
4. **Optimization Techniques**

## 0.2 Relationship of Spark and Hadoop

In this book, most of the MapReduce algorithms are presented in a cookbook (compiled, complete and working solutions) format and implemented

---

<sup>1</sup>In a nutshell, DNA-Sequencing means reading a patient's blood sample (as input) and generating variants (output) for a DNA sample. Formally, "DNA sequencing is the process of determining the precise order of nucleotides within a DNA molecule. It includes any method or technology that is used to determine the order of the four bases – adenine (A), guanine (G), cytosine (C), and thymine (T) – in a strand of DNA." (source:[http://en.wikipedia.org/wiki/DNA\\_sequencing](http://en.wikipedia.org/wiki/DNA_sequencing))

in Java/MapReduce/Hadoop<sup>2</sup> and or Java/Spark/Hadoop<sup>3</sup>. Both Hadoop and Spark frameworks are open-source and enable us to perform huge volume of computations and data processing in a distributed environments. These frameworks enable scaling up by providing "scale out" methodology. These frameworks can be set up to run intensive computations in MapReduce paradigm on thousands of servers. Spark's API has a higher level abstraction than Hadoop's API; and because of this, we are able to express Spark solutions in a single Java driver class.

Hadoop and Spark are two different MapReduce software frameworks. Spark programs may run<sup>4</sup> with or without Hadoop and Spark may use HDFS<sup>5</sup> or other persistent mediums for Input/Output. Relationship of Spark and Hadoop (V1 and V2) are illustrated below in Figure 2 and Figure 3. This relationship shows that there are many ways to run MapReduce and Spark using HDFS. Note that Spark and Hadoop are two different software frameworks and Spark can run with or without Hadoop. In this book, I will use the following keywords:

- MapReduce refers to a general map/reduce framework paradigm
- MapReduce/Hadoop refers to a specific implementation of MapReduce framework using Hadoop
- Spark/Hadoop refers to a specific implementation of MapReduce framework using Spark:
  - Spark can run without Hadoop using "Spark Cluster"
  - Spark can run with Hadoop using Hadoop's YARN or MapReduce

Hadoop and Spark provide the following rich features for big data processing:

- **Reliable**: it is fault-tolerant (any node can go down without losing the result of desired computation)
- **Scalable**: large clusters of servers

---

<sup>2</sup><http://hadoop.apache.org/>

<sup>3</sup><http://spark.apache.org/>

<sup>4</sup>For details on Spark Programming Guide, refer to <http://spark.apache.org/docs/latest/programming-guide.html>

<sup>5</sup>HDFS = Hadoop's Distributed File System

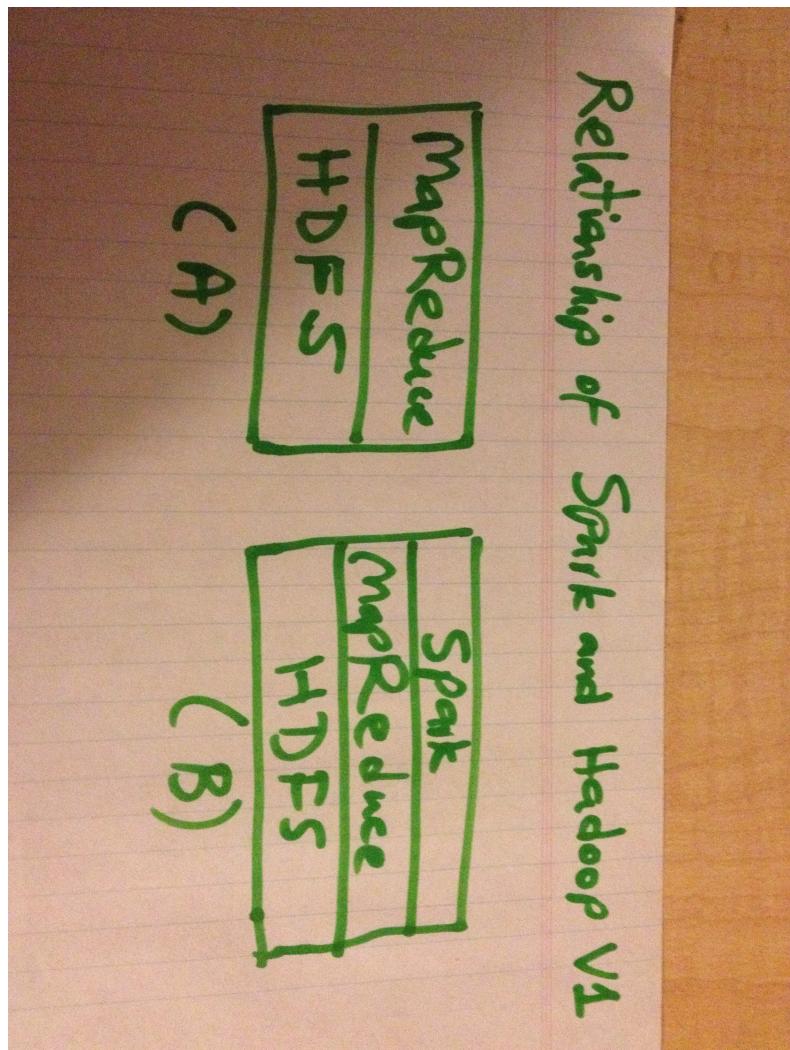


Figure 2: Relationship of Spark and Hadoop V1

## Relationship of Spark and Hadoop V2

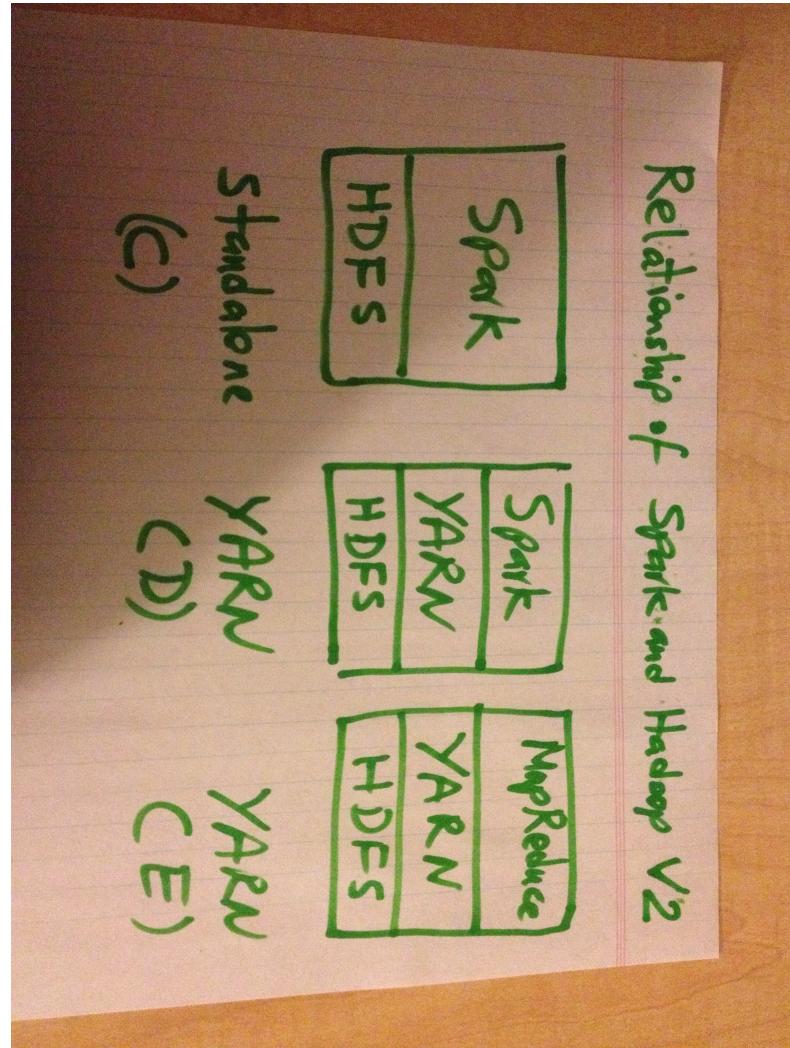


Figure 3: Relationship of Spark and Hadoop V2

- **Distributed**: input data and processing is distributed (supports big data from ground up)
- **Parallel**: computations are executed on cluster of nodes in parallel

Hadoop is designed mainly for batch processing, while with enough memory/RAM, Spark may be used for near real-time processing. To understand basic usage of Spark RDDs, you may reference Appendix B.

Using this book, you will learn step-by-step the algorithms and tools you need to build MapReduce applications with Hadoop. Apache Hadoop is a revolutionary technology which fundamentally alters the way we approach everything we do with big data. MapReduce/Hadoop has become the programming model of choice for processing large data sets (such as log data, genome sequences, statistical applications, and social graphs). MapReduce can be applied for any application that does not require tightly coupled parallel processing. Keep in mind that Hadoop is designed for MapReduce batch processing and is not an ideal solution for real-time processing. Do not expect to get your answers from Hadoop in 2 to 5 seconds. The smallest jobs might take 20+ seconds – Spark<sup>6</sup> is a subproject of Hadoop and is well-suited for real-time processing, but will require more RAM). It is very possible to run a job (such as Biomarker Analysis or Cox Regression), which processes 200 million records in 25 to 35 seconds by just using a cluster of 100 nodes. Typically, Hadoop jobs have a latency of 15 to 25 seconds; this depends on the size and configuration of the Hadoop cluster. Also note that Hadoop is primarily designed for batch jobs and not for realtime processing.

An implementation of MapReduce (such as Hadoop) runs on a large cluster of commodity machines and is highly scalable. For example, a typical MapReduce computation processes many peta-bytes or tera-bytes of data on hundreds/thousands of machines. MapReduce can easily be used by programmers since MapReduce framework implementation hides messy details of parallelization, fault-tolerance, data distribution and load balancing. Instead, a programmer focuses on writing two key functions: `map()` and `reduce()`.

The following are some of the major applications of MapReduce/Hadoop:

---

<sup>6</sup>Apache Spark is a fast and general engine for large-scale data processing. (source: <http://spark.apache.org/>)

- Query log processing
- Crawling, indexing, and search
- Analytics, Text processing, and sentiment analysis
- Machine learning (such as Markov chains and Bayes algorithm)
- Recommendation systems
- Document Clustering and Classification
- Bioinformatics (alignment, re-calibration, germline ingestion, and DNA/RNA-Sequencing)
- Genome analysis (biomarker analysis, regression algorithms such as linear and Cox)

### 0.3 What is MapReduce?

MapReduce [8] is a software framework (or paradigm) for processing large (giga, tera, or peta bytes) data sets in a parallel and distributed fashion over a set of machines (typically server computers). There are many ways to express MapReduce (in this book, our primary focus will be MapReduce/Hadoop and we will show how to accomplish MapReduce in Hadoop/Spark). MapReduce is a programming model for expressing distributed computations on massive amounts of data and an execution framework for large-scale data processing on clusters of commodity<sup>7</sup> servers. The core concept behind MapReduce is mapping your input data set into a collection of `<key, value>` pairs, and then reducing over all pairs with the same `key`. Even though the overall concept is simple, it is actually quite expressive and powerful when you consider that:

- Almost all data can be mapped into `<key, value>` pairs

---

<sup>7</sup>”Commodity computing (or Commodity cluster computing) is to use large numbers of already available computing components for parallel computing to get the greatest amount of useful computation at a low cost. It is computing done in commodity computers as opposed to high-cost super microcomputers or boutique computers.” (source:[http://en.wikipedia.org/wiki/Commodity\\_server](http://en.wikipedia.org/wiki/Commodity_server))

- Your keys and values may be of any type: strings, integers, FASTQ (for DNA-Sequencing), and user-defined custom types and, of course, `<key, value>` pairs themselves.

MapReduce can be used to scale computations over a set of commodity servers. How does MapReduce scale over a set of servers? The key to how MapReduce works is to take input as, conceptually, a list of records (each single record can be one or more lines of data). Then input records are split and passed to the many servers in the cluster to be consumed by the `map()` function. The result of the `map()` computation is a list of `<key, value>` pairs. Then the `reduce()` takes each set of values that has the same key and combines them into a single value (or set of values). So the `map()` takes a set of data chunks and produces `<key, value>` pairs and `reduce()` merges output of data generated by `map()`, so that instead of a set of `<key, value>` pair sets, you get your desired result.

## Simple Explanation of MapReduce

What is a very simple explanation of MapReduce? Let's say that we want to count the number of books in a library, which has 1000 shelves.

- Goal: count the number of books in the library.
- MapReduce solution:
  - **Map:** Hire 1000 workers, each worker counts one shelf.
  - **Reduce:** All workers get together and add up individual counts.

One of the major benefits of MapReduce is its "shared-nothing" data-processing platform. This means that all mappers can work independently, and when mappers complete their job, reducers start to work independently (no data or critical-region is shared among mappers or reducers – having a critical-region will slow distributed computing). This "shared-nothing" paradigm enables us to write `map()` and `reduce()` functions in an easy manner and improves parallelism effectively and effortlessly. Even though, MapReduce frameworks (such as Hadoop and Spark) is built on "shared-nothing" paradigm, but they do support sharing immutable data structures among all cluster nodes. In Hadoop, you may pass these values by Hadoop's

Configuration object to mappers and reducers and in Spark, you may share data structures among mappers and reducers by using `Broadcast` objects.

## 0.4 Why use MapReduce?

The simple answer is to "scale out" by adding more commodity servers (to scale horizontally or "scale out" means to add more nodes to a system, such as adding a new computer to a distributed software application; an example might be scaling out from one Web server system to five). MapReduce works on the promise of "scale out" (the opposite is "scale up", which means that to add more resources to a single node in a system – this can vary in cost and at some point you will not be able to add more resources due to cost and software or hardware limits). Many times, there are promising main memory-based algorithms available for solving data problems, but they lack scalability as the main memory is a bottleneck. For example, in DNA-Seq analysis, you might need over 512GB of RAM, which is very costly and not scalable.

If you need to increase your computational power, you'll need to distribute it across more than one machine. For example, to do DNA-Sequencing of 500GB sample data, it will take over 4 days by one server to just do the alignment phase; using 60 servers with MapReduce can cut this time to less than two hours. To process large volumes of data, you must be able to split-up the data in chunks for processing, which are then recombined later. MapReduce/Hadoop enables you to increase your computational power by just writing two functions: `map()` and `reduce()`. Now for sure we can say that data analytics have a powerful new tool with the MapReduce paradigm, which has recently surged in popularity as open source solutions such as Hadoop.

In a nutshell, MapReduce provides the following benefits:

- **Programming model + infrastructure**
- **Write programs that run on hundreds/thousands of machines**
- **Automatic parallelization and distribution**
- **Fault-tolerance (if a server dies, job will be completed)**
- **Program/job scheduling, status and monitoring**

Hadoop is the de facto standard for implementation of MapReduce applications. Hadoop is comprised of one (or more) master nodes and any number of slave nodes. Hadoop simplifies distributed applications by saying that "the data center is the computer," and by providing "`map`" and "`reduce`" functions (defined by programmer) for application developers/programmers to utilize data centers.

These two functions can be defined as (in informal presentation of `map()` and `reduce()` functions, the convention [...] is used throughout this book to denote a list):

- `map()` function: The master node takes the input, partitions it up into smaller data chunks, and distributes them to worker (slave) nodes. The worker nodes apply the same transformation function to each data chunk then pass the answers back to the master node. In MapReduce, the programmer defines a mapper with the following signature:

`map()`:  $(\text{key}_1, \text{value}_1) \rightarrow [(\text{key}_2, \text{value}_2)]$

- `reduce()` function: The master node shuffles and clusters the received results based on unique (key, values) then – through another redistribution to the workers/slaves – these values are combined through another type of transformation function. In MapReduce, the programmer defines a reducer with the following signature:

`reduce()`:  $(\text{key}_2, [\text{value}_2]) \rightarrow [(\text{key}_3, \text{value}_3)]$

Hadoop implements paradigm efficiently and is quite simple to learn; it is a powerful tool for processing large amounts of data in the range of TeraBytes. But does it provide the right level of abstraction for distributed computing.

## 0.5 What Is in This Book?

Each chapter of this book presents a problem and solves it by a set of MapReduce algorithms. MapReduce algorithms/solutions are a complete recipe (including the MapReduce driver, mapper, combiner, and reducer programs). You can use the code directly in your projects (although sometimes you may need to cut and paste the sections you need). This book is not about theory of MapReduce framework, but practical algorithms and examples using MapReduce/Hadoop on Big Data.

This book provides solid algorithms and guidelines for using MapReduce/Hadoop to solve tough big data problems, such as how to

- Market Basket Analysis for large set of transactions
- Data Mining algorithms (K-menas, KNN, and Naive Bayes)
- Process huge genome data for DNA-seq and RNA-Seq (data from 1000 Genomes project)
- Apply Naive Bayes' Theorem and Markov Chains for data and market prediction
- Recommendation algorithms and Pairwise document similarity
- Linear regression, Cox Regression, and Pearson Correlation for a big set of data
- Allelic-frequency and Mining DNA
- Social network analysis (recommendation systems, counting triangles, sentiment analysis)

You may cut and paste provided solutions from this book to build your own MapReduce applications and solutions using Hadoop. All the solutions have been compiled and tested using Java (JDK6) and Hadoop 1.2.1. This book is ideal for anyone who knows some Java (can read/write basic Java programs) and wants to be writing and deploying MapReduce algorithms using Java/Hadoop. MapReduce [14] has been discussed in detail in an excellent manner by Jimmy Lin and Chris Dyer [14]. The goal of this book is to provide concrete MapReduce algorithms and solutions using Hadoop. Also, this book will not discuss Hadoop in detail; Tom White's excellent book [28] addresses detailed Hadoop very well.

This book will not cover how to install Hadoop; I am going to assume you already have it installed. Also, any Hadoop commands are executed relative to the directory where Hadoop is installed (the `$HADOOP_HOME` environment variable). This book is explicitly about MapReduce/Hadoop algorithms and programming. We do, for example, discuss APIs, command-line invocations for running jobs, and provide complete working programs (including driver, mapper, combiner, and reducer).

Why do we use Hadoop? Hadoop, is the industry-standard; it is the very popular, scalable, affordable (apache license) and flexible solution that helps address myriad Big Data issues dealing with colossal data, complex in nature and from multiple sources. However, Hadoop has its own set of limitations and bottlenecks, such as batch-processing, data security, and so on.

## 0.6 What Is the Focus of This Book?

The focus of this book is to embrace MapReduce paradigm and provide concrete problems that can be solved using MapReduce/Hadoop algorithms. For each problem presented, we will detail the `map()`, `combine()`, and `reduce()` functions and finally provide a complete solution, which has:

- A client (calls the driver with proper input and output parameters)
- A driver (identifies, `map()` and `reduce()` functions, and identifies input and output)
- A mapper class, which implements the `map()` function
- A combiner class (when possible), which implements the `combine()` function. We will discuss when it is possible to use a combiner.
- A reducer class, which implements the `reduce()` function

The goal of this book is to provide step-by-step instructions for using Hadoop as a solution for MapReduce algorithms. Another goal is to show how an output of one MapReduce job can be used as an input to another MapReduce job (this is called chaining or pipelining MapReduce jobs).

## 0.7 What are Core Concepts of MapReduce/Hadoop?

- Input/output data are (key, value) pairs.
  - Typically, keys are integers, long, string, ...
  - Value can be almost any data type: string, integer, long, sentence, special format data

- Partition data over commodity nodes filling racks in a data center.
- Software handles failures, restarts, etc. This is an important feature of Hadoop (called fault-tolerance)
- MapReduce/Hadoop provides fine-grain fault tolerance for large jobs; failure in the middle of a multi-day/multi-hour execution does not require restarting the job from scratch.
- Basic examples: Word Count, Inverted index, Log Processing, Data Mining, Index Creation, Linear Regression, ...

## 0.8 Is MapReduce for Everything?

The simple answer is no. When we have big data, if we can partition data and each partition can be processed independently, then its possible to think about MapReduce algorithms. For example, graph algorithms do not work very well with MapReduce due to an iterative approach of graph algorithms. But if you are grouping or aggregating over a lot of data, then MapReduce paradigm works pretty well. To process graphs using MapReduce, you should take a look at the Apache Giraph<sup>8</sup> project.

### When is MapReduce appropriate?

- When you have to handle lots of input data (e.g., aggregate or compute statistics over big amounts of data)
- When you need to take advantage of parallel and distributed computing, data storage, and data locality
- When you can do many tasks independently without synchronization
- When you can take advantage of sorting/shuffling
- When you need fault tolerance and you can not afford job failures
- When there is lots of input/output data
- When synchronization is not required

---

<sup>8</sup>Apache Giraph is an iterative graph processing system built for high scalability.  
Source: <http://giraph.apache.org/>

## 0.9 What is not MapReduce

- MapReduce is not a programming language, but it is a framework to develop distributed application programming using Java, Scala, and other programming languages
- MapReduce's distributed file system is not a replacement for a relational database management systems (such as MySQL or Oracle). Typically, the input to MapReduce is usually plain text files (a mapper input record can be one or many lines).
- Online tool: The process starts in batches, might run for hours (DNA-Sequencing can take 24 hours on a cluster of five servers)
- A programming language: Any programming language (such as Java, Scala, Python, PERL, ...) can be used to create Map and Reduce steps (depending on the framework used)
- MapReduce is not a solution for all software problems

## 0.10 Who Is This Book For?

This book is for software engineers, software architects, data scientists, and application developers who know the basics of Java and want to develop MapReduce algorithms (in Data Mining, Machine Learning, Bioinformatics, Genomics, and Statistics) and solutions using Hadoop. I also assume you know the basics of the Java programming language (writing a class, defining a new class from an existing class, using basic control structures such as `while-loop`, `if-then-else`, and so on).

This book, MapReduce Algorithms, is targeted for the following readers:

1. Data science engineers and professionals/engineers who want to do analytics (classification, regression algorithms) on big data. The book shows basic steps, in the format of a cookbook, on how to apply classification and regression algorithms using big data. The book details `map()` and `reduce()` functions by showing how it is applied to real data. This book shows where to apply basic design patterns in solving MapReduce problems. These MapReduce algorithms can be easily adapted across the professions by some minor changes (for example

by changing input format). All solutions have been implemented in Apache Hadoop so that these can be adapted in real-world situations.

2. Software engineers and software architects who want to design machine learning algorithms such as Naive Bayes' and Markov Chain algorithms. The book shows how build the model and then apply them for a new data using MapReduce design patterns
3. Software engineers and software architects who want to do data mining (such as K-Means Clustering and K-Nearest-Neighbors) algorithms using MapReduce. Detailed Examples are given to guide professional to implement similar algorithms.
4. Data science engineers who want to apply MapReduce algorithms to clinical and biological data (such as DNA-Seq and RNA-Seq). The book clearly shows practical algorithms suitable for bioinformaticians and clinicians. This book presents the most relevant regression/analytical algorithms used for different biological data types. Majority of these algorithms have been deployed in real-world production systems.
5. Software architects who want to apply the most important optimizations in a MapReduce/distributed environment.

## 0.11 What Software Is Used in This Book?

When developing solutions and examples for this book, I used the following software and programming environments:

Software and Version	
Software	Version
Java programming language (JDK7)	1.7.0_67
Operating system: Linux CentOS	6.3
Operating system: Mac OS X	10.9
Apache Hadoop	2.5.0
Apache Spark	1.0.2
Eclipse IDE	luna

All programs in this book were tested with JDK7, Hadoop 2.4.1/2.5.0, and Spark 1.0.2/1.1.0. Examples are given in mixed operating system environments (Linux and MacBook Pro). For all examples and solutions, I engaged basic text editors (such as `vi`, `vim`, and `TextWrangler`) and compiled them using the Java command-line compiler (`javac`).

## 0.12 Using Code Examples

This book is here to help you get your job done using MapReduce/Hadoop. You may use the code in this book in your programs and documentation. You do not need to contact us for permission. All code and examples presented in this book are "open-source"; you may use it in any way you wish.

### Book Prerequisites

This book assumes you have a basic understanding of Java and Hadoop's HDFS concepts. If you need to become familiar with the Hadoop, then the following books will offer you the background information you will need:

- Hadoop, The Definitive Guide by Tom White (publisher: O'Reilly Media, Inc.)
- Hadoop in Action by Chuck Lam (publisher: Manning Publications)
- Hadoop in Practice by Alex Holmes (publisher: Manning Publications)

## 0.13 Where NOT to use MapReduce?

There are few scenarios[9] where MapReduce programming model cannot be employed. If the computation of a value depends on previously computed values, then MapReduce cannot be used. One good example is the Fibonacci series where each value is summation of the previous two values. i.e.,

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

Also, if the data set is small enough to be computed on a single machine, then it is better to do it as a single `reduce(map(data))` operation rather than going through the entire map reduce process.

## When is MapReduce less appropriate?

- synchronization is required to access shared data
- all of your input data fits in memory
- one operation depends on other operations
- basic computations are processor-intensive

## 0.14 Chapters in This Book?

This book is organized in 5 sections:

### Section-1: Basic Design Patterns

- Chapter 1 introduces "secondary sort" design pattern
- Chapter 2 presents "secondary sort" design pattern and provide detail implementation of the algorithm in MapReduce/Hadoop.
- Chapter 3 implements the TOP-10 design pattern in MapReduce/Hadoop and Spark/Hadoop.
- Chapter 4 presents the "Left-Join" algorithm and provides its implementation in Mapreduce/Hadoop.
- Chapter 5 implements the "Order-Inversion" algorithm.
- Chapter 6 presents the "Moving-Average" algorithm.

### Section-2: Data Mining and Machine Learning

- Chapter 7 Market-Basket-Analysis
- Chapter 8 Common-Friends
- Chapter 9 Recommendation Engines using MapReduce
- Chapter 10 Content-based Recommendation
- Chapter 11 Computing Markov Models
- Chapter 12 K-means Clustering
- Chapter 13 KNN
- Chapter 14 Naive Bayes

- Chapter 15 Sentiment Analysis
- Chapter 16 Counting Triangles in Social Networks

### **Section-3: Bioinformatics, Genomics, and Statistics**

- Chapter 17 K-mer Counting
- Chapter 18 DNA-Sequencing
- Chapter 19 Cox Regression
- Chapter 20 Cochran-Armitage Test for Trend
- Chapter 21 Allelic Frequency
- Chapter 22 The T-Test
- Chapter 23 Computing Pearson Correlation
- Chapter 24 DNA Base Count
- Chapter 25 RNA-Sequencing
- Chapter 26 Gene Aggregation
- Chapter 27 Linear Regression

### **Section-4: Optimization Techniques**

- Chapter 28 MapReduce and Monoids
- Chapter 29 The Small Files Problem
- Chapter 30 Huge Cache for MapReduce
- Chapter 31 Bloom Filter

### **Section-5: Appendices**

- Appendix A: Biosets
- Appendix B: Spark RDDs

## **0.15 Online Resources**

There are two web sites that accompanies this book:

- <https://github.com/mahmoudparsian/data-algorithms-book/>

In this GitHub site, you will find links to the source code (organized by chapters), some shell scripts, sample input files for testing, and some extra content that isn't in the book, including a couple of chapters.

- <http://mapreduce4hackers.com/>

In this site, you will find links to the extra source files (not mentioned in book) plus some extra content that isn't in the book. Expect more coverage of MapReduce/Hadoop/Spark topics as time progresses.

## 0.16 Comments and Questions for This Book?

I am always interested in your feedback and comments regarding the problems and solutions described in this book. Please e-mail comments and questions for this book to [mahmoud.parsian@yahoo.com](mailto:mahmoud.parsian@yahoo.com). You can also find me at <http://www.mapreduce4hackers.com>.

Mahmoud Parsian  
Sunnyvale, California  
September 12, 2014

# Secondary Sort: Introduction

## 1.1 What is a Secondary Sort Problem?

The goal of this chapter is to implement "secondary sort" design-pattern by MapReduce/Hadoop and Spark/Hadoop. In software design and programming, a design pattern is a reusable algorithm (typically, a design pattern is not presented in a specific programming language – but can be implemented by many programming languages) to a commonly occurring problem.

MapReduce framework automatically sorts the keys generated by mappers. This means that, before starting reducers all intermediate  $(key, value)$  pairs generated by mappers must be sorted by *key* (and not by *value*). Values passed to each reducer are not sorted at all and they can be in any order. What we know is that MapReduce sorts input to reducers by *key* and values may be arbitrarily ordered. What if we want to sort reducer's values also? MapReduce/Hadoop and Spark/Hadoop do not sort values for a reducer. For example, for some applications (such as time series data), you want your data to be sorted. Secondary Sort design pattern enable us to sort redcer's values.

First we focus on MapReduce/Hadoop solution. Let's look at the MapReduce paradigm and then explain the concept of the Secondary Sort:

```
map(key1, value1) → list(key2, value2)
reduce(key2, list(value2)) → list(key3, value3)
```

First, the `map()` function receives a key-value pair input,  $(\text{key}_1, \text{value}_1)$ . Then it outputs another (any number of them) key-value pair,  $(\text{key}_2, \text{value}_2)$ . Second, the `reduce()` function receives as input another key-value pair,  $(\text{key}_2, \text{list}(\text{value}_2))$ , and outputs (any number of them)  $(\text{key}_3, \text{value}_3)$ .

Now consider the following key-value pair  $(\text{key}_2, \text{list}(\text{value}_2))$  as an input for a reducer:

$$\text{list}(\text{value}_2) = (V_1, V_2, \dots, V_n)$$

where there is no ordering between reducer values  $(V_1, V_2, \dots, V_n)$ .

The goal of the "secondary sort" is to give *some ordering* for the values received by a reducer. So, once we apply/inject "secondary sort" to our MapReduce paradigm, then we will have:

$$\text{list}(\text{value}_2) = (S_1, S_2, \dots, S_n)$$

where

- $S_1 < S_2 < \dots < S_n$  (ascending order) or
- $S_1 > S_2 > \dots > S_n$  (descending order)
- $S_i \in \{V_1, V_2, \dots, V_n\}$

We will show by a concrete example how to achieve secondary sorting in ascending or descending order. What is "secondary sorting" problem? "Secondary Sorting Problem" is the problem of sorting values associated with a key in the reduce phase. Sometimes, this is called "value-to-key conversion." The "secondary sorting" technique will enable us to sort the values (in ascending or descending order) passed to each reducer.

Apache Hadoop and Spark, which are popular MapReduce paradigms, do not sort the values passed to reducers (values can be in any arbitrary order). On the other hand, it is believed that the Google's MapReduce implementation provides an option for sorting the values passed to each reducer. Note, that since we do not have access to Google's MapReduce implementation, we just rely on Google's documentation for this claim.

Consider the example of temperature data from a scientific experiment. A dump of the temperature data might look something like the following (columns are `year`, `month`, `day`, and the `temperature` for that day).

```
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
```

Suppose we want to output the temperature for every "Year-Month" with temperature values sorted (ascending). Essentially, we want the reducers values iterator to be sorted. Therefore, we want to generate something like this output (the first column is Year-Month and the second column is the sorted temperatures):

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

## 1.2 Solutions to Secondary Sort Problem

There are at least two possible approaches for sorting the reducers values.

1. The first approach involves having the reducer read and buffer all of the values for a given *key* (in an array data structure, for example) Then do an in-reducer sort on the values. This approach will not scale: since the reducer will be receiving all values for a given *key*, this approach could possibly cause the reducer to run out of memory (`java.lang.OutOfMemoryError`). This approach can work well if the number of values is small enough, which will not cause out-of-memory error.

2. The second approach involves using MapReduce framework for sorting the reducers values (does not require in-reducer sorting of values passed to the reducer). This approach involves "creating a composite key by adding a part of, or the entire value to, the natural key to achieve your sorting objectives." For the details on this approach, see javacodegeeks<sup>1</sup>. The second approach is scaleable and you will not see out-of-memory errors. Here, we basically offload the sorting to the MapReduce framework (sorting is a paramount feature of MapReduce/Hadoop framework)..

This is the summary of second approach:

- (a) Use "Value-to-Key Conversion" design pattern: form composite intermediate key,  $(k, v_1)$ , where  $v_1$  is the secondary key. Here,  $k$  is called a natural key. How to inject value (i.e.,  $v_1$ ) into a reducer key? Just create a composite key (for details, see the `DateTemperaturePair` class). In our example,  $v_1$  is the `temperature` data.
- (b) Let MapReduce execution framework do the sorting (rather than sorting in memory, let the framework do the sorting by using the cluster nodes)
- (c) Preserve state across multiple  $(key, value)$  pairs to handle processing; this can be achieved by having proper mapper output partitioners (for example, we partition mapper's output by the natural key)

## Implementation details

To implement the "secondary sort" feature, we do need additional plug-in Java classes. We have to tell to MapReduce/Hadoop framework:

1. how to sort reducer keys
2. how to partition keys passed to reducers (custom partitioner)
3. how to group data reached to reducers

---

<sup>1</sup><http://www.javacodegeeks.com/2013/01/mapreduce-algorithms-secondary-sorting.html>

### 1.2.1 Sort Order of Intermediate Keys

To accomplish "secondary sorting," we do need to take control of the "sort order of intermediate keys" and the control order in which reducers process keys. First, we inject value (`temperature` data) to the composite key and then provide control on sort order of intermediate keys. The relationships of natural key, composite key, and (key, value) pairs are depicted below:

In this section, we will implement the "secondary sort" (or so called "value to key") approach. What value should we add to the natural key? The answer is the `temperature` data field (because we want the reducer's values to be sorted by `temperature`). So, we have to mention how `DateTemperaturePair` objects are sorted by the `compareTo()` method. You need to define a proper data structure for holding your key and value, while also providing the sort order of intermediate keys. In Hadoop, to persist custom data types (such as `DateTemperaturePair`), they have to implement the `Writable` interface; and if we are going to compare custom data types, then they have to implement an additional interface called `Comparable`.

**Listing 1.1:** DateTemperaturePair Class

```
1 public class DateTemperaturePair
2     implements Writable, WritableComparable<DateTemperaturePair> {
3
4     private Text yearMonth = new Text(); // natural key
5     private Text day = new Text();
6     private IntWritable temperature = new IntWritable(); // secondary key
7
8     ...
9
10    @Override
11    /**
12     * This comparator controls the sort order of the keys.
13     */
14    public int compareTo(DateTemperaturePair pair) {
15        int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
16        if (compareValue == 0) {
17            compareValue = temperature.compareTo(pair.getTemperature());
18        }
19        //return compareValue;      // sort ascending
20        return -1*compareValue;   // sort descending
21    }
22    ...
23 }
```

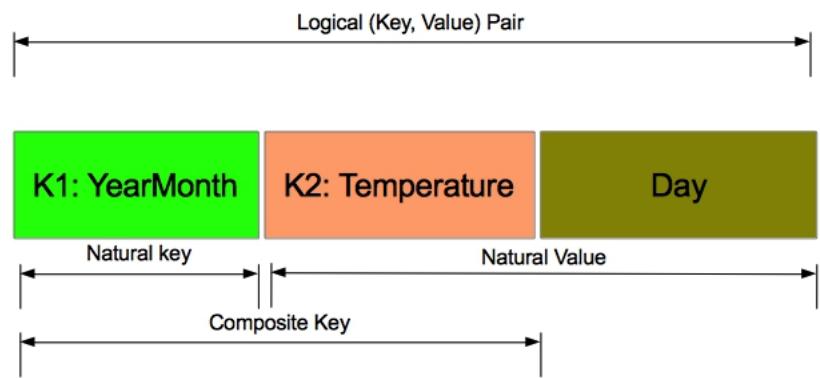


Figure 1.1: Secondary Sorting Keys

### 1.2.1.1 Partition code

In a nutshell, partitioner decides which mapper output goes to which reducer based on mapper output key. We do need a plug-in partitioner to control which reducer processes which keys. We do need to write a custom partitioner to ensure that all the data with same key (the natural key not including the composite key with the value – value is the `temperature` data field) is sent to the same reducer and a custom Comparator so that the natural key (composition of year and month) groups the data once it arrives at the reducer.

**Listing 1.2:** DateTemperaturePartitioner Class

```
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         // make sure that partitions are non-negative
12         return Math.abs(pair.getYearMonth().hashCode() % numberofPartitions);
13     }
14 }
```

How do we inject the partitioner code? Hadoop provides a plug-in architecture for custom code. This is how we do it inside the driver class (which submits the MapReduce job to Hadoop):

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

### 1.2.1.2 Grouping Comparator

We define the comparator (`DateTemperatureGroupingComparator` class ) that controls which keys are grouped together for a single call to `Reducer.reduce()` function.

**Listing 1.3:** DateTemperatureGroupingComparator Class

```

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     * This comparator controls which keys are grouped
14     * together into a single call to the reduce method
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}

```

---

How do we inject the "grouping comparator" code? Hadoop provides a plug-in architecture for "grouping comparator" code. This is how we do it inside the driver class (which submits the MapReduce job to Hadoop):

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

## 1.3 Data Flow Using Plug-in Classes

To understand the map(), reduce(), and custom plug-in classes, the data flow for portion of input is illustrated below.

The mappers create (K,V) pairs, where K is a composite key of (year,month,temperature) and V is a temperature. Note that (year,month) part of the composite key is called a natural key. The partitioner plug-in class enables us to send all natural keys to the same reducer and "grouping comparator" plug-in class enable temperature's to arrive sorted to reducers. The "secondary sort" design pattern uses Mapreduce's framework for sorting the reducer's values rather than collecting all and then sort in memory. The "secondary sort" design pattern enable us to "scale out" no matter how many reducer values we want to sort.

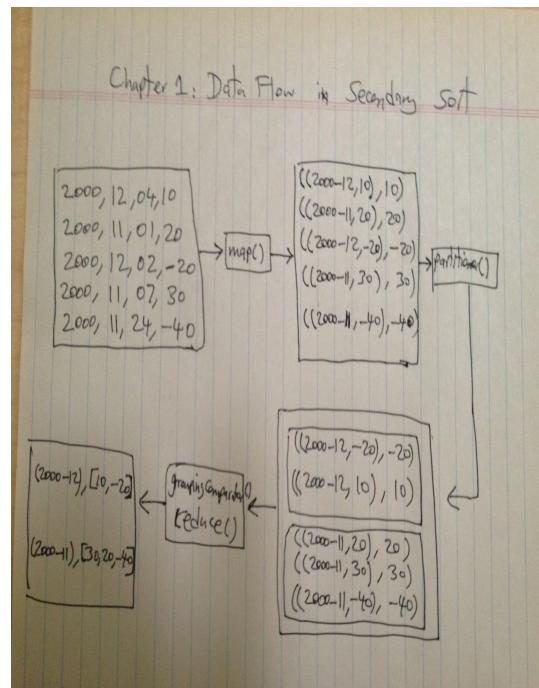


Figure 1.2: Secondary Sorting Data Flow

## 1.4 Mapreduce/Hadoop Solution

This section provides a complete MapReduce implementation of "secondary sort" problem by using Hadoop framework.

### 1.4.1 Input

Input will be a set of files, which will have the following format:

Format:

<year><,><month><,><day><,><temperature>

Example-1:

2012, 01, 01, 35

Example-2:

2011, 12, 23, -4

### 1.4.2 Expected Output

Input will be a set of files, which will have the following format:

Format:

<year>-<month>: <temperature1>,<temperature2>, ...

Example:

2012-01: 5, 10, 35, 45, ...

2001-11: 40, 46, 47, 48, ...

2005-08: 38, 50, 52, 70, ...

### 1.4.3 map() function

The map() function parses and tokenizes the input and then injects the value (value of temperature) into the reducer key.

#### **Listing 1.4: map() for Secondary Sorting**

```
1 /**
2 * @param key is generated by Hadoop (ignored here)
3 * @param value has this format: "YYYY,MM,DD,temperature"
4 */
5 map(key, value) {
6     String[] tokens = value.split(",");
7     // YYYY = tokens[0]
8     // MM = tokens[1]
9     // DD = tokens[2]
10    // temperature = tokens[3]
11    String yearMonth = tokens[0] + tokens[1];
12    String day = tokens[2];
13    int temperature = Integer.parseInt(tokens[3]);
14    // prepare reducer key
15    DateTemperaturePair reducerKey = new DateTemperaturePair();
16    reducerKey.setYearMonth(yearMonth);
17    reducerKey.setDay(day);
18    reducerKey.setTemperature(temperature); // inject value into key
19    // send it to reducer
20    emit(reducerKey, temperature);
21 }
```

#### **1.4.4 reduce() function**

#### **Listing 1.5: reduce() for Secondary Sorting**

```
1 /**
2 * @param key is a DateTemperaturePair object
3 * @param value is a list of temperature
4 */
5 reduce(key, value) {
6     StringBuilder sortedTemperatureList = new StringBuilder();
7     for (Integer temperature : value) {
8         sortedTemperatureList.append(temperature);
9         sortedTemperatureList.append(",");
10    }
11    emit(key, sortedTemperatureList);
12 }
```

#### **1.4.5 Hadoop Implementation**

The following classes are used to solve the problem.

<i>Class Name</i>	<i>Class Description</i>
SecondarySortDriver.java	The driver class, defines input/output and registers plug-in classes.
SecondarySortMapper.java	Defines the map() function
SecondarySortReducer.java	Defines the reduce() function
DateTemperatureGroupingComparator.java	Defines how keys will be grouped together
DateTemperaturePair.java	Defines pair of date and temperature as a Java object
DateTemperaturePartitioner.java	Defines custom partitioner

How is the value injected into the key? The first comparator (`DateTemperaturePair.compare()` method) controls the sort order of the keys and the second comparator (`DateTemperatureGroupingComparator.compare()` method) controls which keys are grouped together into a single call to the reduce method. The combination of these two allows you to set up jobs that act like you've defined an order of the values.

The `SecondarySortDriver` is the driver class, which registers the custom plug-in classes (`DateTemperaturePartitioner` and `DateTemperatureGroupingComparator` classes) with the Mapreduce/Hadoop framework. This class is presentd below:

#### **Listing 1.6:** SecondarySortDriver Class

```

1 public class SecondarySortDriver extends Configured implements Tool {
2
3     public int run(String[] args) throws Exception {
4         Configuration conf = getConf();
5         Job job = new Job(conf);
6         job.setJarByClass(SecondarySortDriver.class);
7         job.setJobName("SecondarySortDriver");
8
9         // args[0] = input directory
10        // args[1] = output directory
11        FileInputFormat.setInputPaths(job, new Path(args[0]));
12        FileOutputFormat.setOutputPath(job, new Path(args[1]));
13
14        job.setOutputKeyClass(TemperaturePair.class);
15        job.setOutputValueClass(NullWritable.class);
16
17        job.setMapperClass(SecondarySortingTemperatureMapper.class);
18        job.setReducerClass(SecondarySortingTemperatureReducer.class);
19        job.setPartitionerClass(TemperaturePartitioner.class);
20        job.setGroupingComparatorClass(YearMonthGroupingComparator.class);
21
22        boolean status = job.waitForCompletion(true);

```

```

23         theLogger.info("run(): status="+status);
24         return status ? 0 : 1;
25     }
26
27     /**
28      * The main driver for word count map/reduce program.
29      * Invoke this method to submit the map/reduce job.
30      * @throws Exception When there is communication problems with the job tracker.
31     */
32     public static void main(String[] args) throws Exception {
33         // Make sure there are exactly 2 parameters
34         if (args.length != 2) {
35             throw new IllegalArgumentException("Usage: SecondarySortDriver <input-dir> <output-dir>");
36         }
37
38         //String inputDir = args[0];
39         //String outputDir = args[1];
40         int returnStatus = ToolRunner.run(new SecondarySortDriver(), args);
41         System.exit(returnStatus);
42     }
43
44 }
```

---

## 1.4.6 Sample Run of Hadoop Implementation

### 1.4.6.1 Input

```
# 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
```

### 1.4.6.2 HDFS Input

```
1 # hadoop fs -mkdir /secondary_sort
2 # hadoop fs -mkdir /secondary_sort/input
3 # hadoop fs -mkdir /secondary_sort/output
4 # hadoop fs -put sample_input.txt /secondary_sort/input/
5 # hadoop fs -ls /secondary_sort/input/
6 Found 1 items
7 -rw-r--r-- 1 mahmoud staff 128 2013-02-26 15:14 /secondary_sort/input/sample_input.txt
```

### 1.4.7 Sample Run

#### 1.4.7.1 The Script

```
1 # cat run.sh
2#!/bin/bash
3
4 export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
5 export HADOOP_HOME_WARN_SUPPRESS=true
6
7 export JAVA_HOME='/usr/libexec/java_home'
8 echo "JAVA_HOME=$JAVA_HOME"
9
10 PATH=.::/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
11 PATH=$PATH:$HADOOP_HOME/bin
12 PATH=$PATH:$JAVA_HOME/bin
13 export PATH
14
15 CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
16 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.0.3.jar
17 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.0.3.jar
18 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.0.3.jar
19 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.0.3.jar
20 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.0.3.jar
21 CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/log4j-1.2.15.jar
22 CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-cli-1.2.jar
23 CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-logging-1.1.1.jar
24
25 export JAR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/secondary_sort/secondary_sort.jar
26 export CLASSPATH=$CLASSPATH:$JAR
27 export HADOOP_CLASSPATH=$CLASSPATH
28
29 javac *.java
30 jar cvf $JAR *.class
31
32 $HADOOP_HOME/bin/hadoop fs -rmr /secondary_sort/output
33 $ INPUT=/secondary_sort/input
34 $ OUTPUT=/secondary_sort/output
35 $HADOOP_HOME/bin/hadoop jar $JAR SecondarySortDriver $INPUT $OUTPUT
```

#### 1.4.7.2 Log of Sample Run

```
# ./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
```

#### 1.4.7.3 Inspecting Output

```
# 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
```

## 1.5 What If Sorting Ascending or Descending

You can easily control the sorting order of the values (Ascending or Descending) by the `DateTemperaturePair.compareTo()` method. This is how you can do it:

```
public int compareTo(DateTemperaturePair pair) {
    int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
```

```

        if (compareValue == 0) {
            compareValue = temperature.compareTo(pair.getTemperature());
        }
        //return compareValue;      // sort ascending
        return -1*compareValue;   // sort descending
    }
}

```

## 1.6 Spark Solution To Secondary Sorting

To solve a "secondary sorting" in Spark<sup>2</sup>, we have at least two options and we will present a solution to each option.

- OPTION-1: read and buffer all of the values for a given *key* in an array of list data structure and then do an in-reducer sort on the values. This solution works if you have small set of values (which will fit in memory) per reducer key.
- OPTION-2: use Spark framework for sorting the reducers values (does not require in-reducer sorting of values passed to the reducer). This approach involves "creating a composite key by adding a part of, or the entire value to, the natural key to achieve your sorting objectives." This option always scales (because you are not limited by memory of a commodity server).

### 1.6.1 Time-Series as Input

To demonstrate "secondary sorting", let's use a time-series data:

name	time	value
x	2	9
y	2	5
x	1	3
y	1	7
y	3	1

---

<sup>2</sup>Apache Spark is a fast and general engine for large-scale data processing. (source: <http://spark.apache.org/>)

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

### 1.6.2 Expected Output

Our expected output is illustrated below. As you observe, the values of reducers are grouped by "name" and sorted by "time".

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]	

### 1.6.3 Option-1: Secondary Sorting in Memory

Since Spark has a very powerfull and high-level API, we will present the entire solution in a single Java class. Spark API is built by the basic abstraction concept of RDD (Resilient Distributed Dataset). To fully utilize Spark's API, we have to understand RDD's. An `RDD<T>` (or type T) object represents an immutable, partitioned collection of elements (or type T) that can be operated on in parallel. The `RDD<T>` class contains the basic MapReduce operations available on all RDDs, such as `map`, `filter`, and `persist`. While `JavaPairRDD<K,V>` class contains the basic MapReduce operations such as `mapToPair`, `flatMapToPair`, and `groupByKey`. In addition, Spark's `PairRDDFunctions` contains operations available only on RDDs of (key, value) pairs, such as `reduce`, `groupByKey` and `join`. For details on

RDDs, see Spark's API<sup>3</sup>. Therefore, `JavaRDD<T>` is a list of objects of type `T` and `JavaPairRDD<K,V>` is a list of objects of type `Tuple2<K,V>` (each tuple represents a (key,value) pair).

The Spark-based algorithm is listed below: even though we have 9 steps, but most of them are trivial and some are provided for debugging purposes only.

**STEP-0** : Import required Java/Spark classes. The main Java classes for MapReduce are given in the `org.apache.spark.api.java` package. This package includes the following classes and interfaces:

- `JavaRDDLike` (interface)
- `JavaDoubleRDD`
- `JavaPairRDD`
- `JavaRDD`
- `JavaSparkContext`
- `StorageLevels`

**STEP-1** : Pass Sark master and input data as arguments and validate.  
Spark master URL is specified as: `spark://<spark-master-server>:7077`.

**STEP-2** : Connect to the Sark master by creating `JavaSparkContext` object, which is a factory class for creating new RDDs.

**STEP-3** : Use the context object (created in STEP-2) and create an RDD for input file, the resulting RDD will be `JavaRDD<String>`. Each element of this RDD will be a record of time series data: `<name><,><time><,><value>`.

**STEP-4** : Next we want to create (key, value) pairs from `JavaRDD<String>`, where key is the `name` and value is a pair of (`time`, `value`). The resulting RDD will be `JavaPairRDD<String, Tuple2<Integer, Integer>>`.

**STEP-5** : To validate STEP-4, we collect all values from `JavaPairRDD<>` and print it.

---

<sup>3</sup><http://spark.apache.org/docs/1.0.0/api/java/org/apache/spark/rdd/RDD.html>

**STEP-6**: We group JavaPairRDD<> elements by the key (`name`). To accomplish this, we use the `groupByKey()` method. The result will be the RDD:

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

Note that the resulting list (`Iterable<Tuple2<Integer, Integer>>`) is unsorted.

**STEP-7**: To validate STEP-6, we collect all values from `JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>` and print it.

**STEP-8**: Sort the reducer's values and this will give us the final output. This is accomplished by writing a custom `mapValues()` method. We just sort the values (key remains the same).

**STEP-9**: To validate the final result, we collect all values from sorted `JavaPairRDD<>` and print it.

A solution for Option-1 is implemented by a single class: `SecondarySorting`. All steps (STEP-0 – STEP-9) are listed inside the class definition, which will be presented in the following sections. Typically, a Spark application program consists of a driver program that runs the user's `main()` function and executes various parallel operations on a cluster. Parallel operations will be achieved by using an extensive use of RDDs. In a nutshell, an RDD is a fundamental Spark abstraction of data (an immutable data structure) and is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel.

#### **Listing 1.7: SecondarySort Class Overall Structure**

```
1 // STEP-0: import required Java/Spark classes.
2 public class SecondarySort {
3     public static void main(String[] args) throws Exception {
4         // STEP-1: read input parameters and validate them
5         // STEP-2: Connect to the Spark master by creating JavaSparkContext object
6         // STEP-3: Use ctx to create JavaRDD<String>
7         // STEP-4: create (key, value) pairs from JavaRDD<String> where
8         //         key is the {name} and value is a pair of (time, value).
9         // STEP-5: validate STEP-4, we collect all values from JavaPairRDD<> and print it.
10        // STEP-6: We group JavaPairRDD<> elements by the key ({name}).
11        // STEP-7: validate STEP-6, we collect all values from JavaPairRDD<> and print it.
```

```

12     // STEP-8: Sort the reducer's values and this will give us the final output.
13     // STEP-9: validate STEP-8, we collect all values from JavaPairRDD<> and print it.
14     System.exit(0);
15 }
16 }
```

### 1.6.3.1 STEP-0: Import Required Classes

The main Spark package for Java API is `org.apache.spark.api.java`, which includes `JavaRDD`, `JavaPairRDD`, and `JavaSparkContext` classes. The `JavaSparkContext` is a factory class for creating new RDDs (such as `JavaRDD` and `JavaPairRDD`).

#### **Listing 1.8:** STEP-0: Import Required Classes

```

1 // STEP-0: import required Java/Spark classes.
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.Function2;
8 import org.apache.spark.api.java.function.PairFunction;
9
10 import java.util.List;
11 import java.util.ArrayList;
12 import java.util.Map;
13 import java.util.Collections;
14 import java.util.Comparator;
```

### 1.6.3.2 STEP-1: Read Input Parameters

This step reads a spark master URL and HDFS input file. The examples for this step can be:

- Spark master URL: `spark://<spark-master-node>:7077`
- HDFS file: `/dir1/dir2/myfile.txt`

#### **Listing 1.9:** STEP-1: Read Input Parameters

```

1 // STEP-1: read input parameters and validate them
2 if (args.length < 2) {
3     System.err.println("Usage: Top10 <master> <file>");
4     System.exit(1);
5 }
```

```
6  
7     System.out.println("args[0]: <master>="+args[0]);  
8     System.out.println("args[1]: <file>="+args[1]);
```

---

### 1.6.3.3 STEP-2: Connect to the Sark Master

To do some work with RDDs, first you need to create a `JavaSparkContext` object, which is a factory class for creating `JavaRDD` and `JavaPairRDD` objects. It is also possible to create a `JavaSparkContext` object by injecting `SparkConf` object to `JavaSparkContext`'s class constructor. This approach is useful when you read your cluster configurations from an XML file. In a nutshell, `JavaSparkContext` object has the following responsibilities:

- Initializes the application driver
- Registers the application driver to the cluster manager (if you are using Spark cluster, then this will be Spark master, and if you are using YARN, then that will be YARN's resource manager)
- Obtain a list of executors for executing your application driver

#### **Listing 1.10:** STEP-2: Connect to the Sark Master

```
1 // STEP-2: Connect to the Sark master by creating JavaSparkContext object  
2 final JavaSparkContext ctx = new JavaSparkContext(  
3     args[0],  
4     "Top10",  
5     System.getenv("SPARK_HOME"),  
6     System.getenv("SPARK_EXAMPLES_JAR"));
```

---

### 1.6.3.4 STEP-3: Use JavaSparkContext to create JavaRDD

This step reads an HDFS file and creates a `JavaRDD<String>` (represents a set of records – each record is a `String` object). By definition, Spark's RDDs are immutable (cannot be altered or modified). Note that Spark's RDDs are the basic abstraction for parallel execution.

#### **Listing 1.11:** STEP-3: Create JavaRDD

```

1 // STEP-3: Use ctx to create JavaRDD<String>
2 // input record format: <name>,><time>,><value>
3 JavaRDD<String> lines = ctx.textFile(args[1], 1);

```

### 1.6.3.5 STEP-4: Create (key, value) pairs from JavaRDD

This step implements a mapper. Each record (from `JavaRDD<String>` as `<name>,><time>,><value>`) is converted to a (key, value) where key is a name and value is a `Tuple2(time, value)`.

**Listing 1.12:** STEP-4: create (key, value) pairs from JavaRDD

```

1 // STEP-4: create (key, value) pairs from JavaRDD<String> where
2 // key is the {name} and value is a pair of (time, value).
3 // The resulting RDD will be JavaPairRDD<String, Tuple2<Integer, Integer>>.
4 // convert each record into Tuple2(name, time, value)
5 // PairFunction<T, K, V>
6 //   T => Tuple2(K, V) where T is input (as String),
7 //   K=String
8 //   V=Tuple2<Integer, Integer>
9 JavaPairRDD<String, Tuple2<Integer, Integer>> pairs =
10    //          T      K      V
11    lines.map(new PairFunction<String, String, Tuple2<Integer, Integer>>() {
12      public Tuple2<String, Tuple2<Integer, Integer>> call(String s) {
13        String[] tokens = s.split(","); // x,2,5
14        System.out.println(tokens[0] + "," + tokens[1] + "," + tokens[2]);
15        Integer time = new Integer(tokens[1]);
16        Integer value = new Integer(tokens[2]);
17        Tuple2<Integer, Integer> timevalue = new Tuple2<Integer, Integer>(time, value);
18        return new Tuple2<String, Tuple2<Integer, Integer>>(tokens[0], timevalue);
19      }
20    });

```

### 1.6.3.6 STEP-5: Validate STEP-4

To debug and validate your steps in Spark, you may use `JavaRDD.collect()` and `JavaPairRDD.collect()`.

**Listing 1.13:** STEP-5: Validate STEP-4

```

1 // STEP-5: validate STEP-4, we collect all values from JavaPairRDD<> and print it.
2 List<Tuple2<String, Tuple2<Integer, Integer>>> output = pairs.collect();
3 for (Tuple2 t : output) {
4   Tuple2<Integer, Integer> timevalue = (Tuple2<Integer, Integer>) t._2;
5   System.out.println(t._1 + "," + timevalue._1 + "," + timevalue._2);
6 }

```

### 1.6.3.7 STEP-6: Group JavaPairRDD elements by the key (name)

We implement the reducer operation by `groupByKey()`. As you observe, it is much easier to implement the reducer by Spark than MapReduce/Hadoop.

#### Listing 1.14: STEP-6: Group JavaPairRDD Elements

```
1 // STEP-6: We group JavaPairRDD<> elements by the key ({name}).
2 JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> groups = pairs.groupByKey();
```

### 1.6.3.8 STEP-7: Validate STEP-6

#### Listing 1.15: STEP-7: Validate STEP-6

```
1 // STEP-7: validate STEP-6, we collect all values from JavaPairRDD<> and print it.
2 System.out.println("====DEBUG1====");
3 List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>> output2 = groups.collect();
4 for (Tuple2<String, Iterable<Tuple2<Integer, Integer>> t : output2) {
5     Iterable<Tuple2<Integer, Integer>> list = t._2;
6     System.out.println(t._1);
7     for (Tuple2<Integer, Integer> t2 : list) {
8         System.out.println(t2._1 + "," + t2._2);
9     }
10    System.out.println("=====");
11 }
```

The output of this step is presented below. As you can observe, the reducer values are not sorted.

```
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
=====

```

#### 1.6.3.9 STEP-8: Sort the Reducer's Values in Memory

This step uses another powerful Spark method `mapValues()` to just sort the values generated by reducers. The `mapValues()` method enable us to convert  $(K, V_1)$  into  $(K, V_2)$ . One important note about Spark's RDD is that they are immutable and can not be altered/updated by any means. For example, in STEP-8, to sort our values, we have to copy them into another list before sorting. Immutability applies to RDD itself and its elements.

##### **Listing 1.16:** STEP-8: Sort the Reducer's Values in Memory

```

1 //STEP-8: Sort the reducer's values and this will give us the final output.
2 // OPTION-1: worked
3 // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
4 // Pass each value in the key-value pair RDD through a map function
5 // without changing the keys;
6 // this also retains the original RDD's partitioning.
7 JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> sorted =
8     groups.mapValues(new Function<Iterable<Tuple2<Integer, Integer>>, // input
9                     Iterable<Tuple2<Integer, Integer>> // output
10                    >() {
11     public Iterable<Tuple2<Integer, Integer>> call(Iterable<Tuple2<Integer, Integer>> s) {
12         List<Tuple2<Integer, Integer>> newList = new ArrayList<Tuple2<Integer, Integer>>(s);
13         Collections.sort(newList, new TupleComparator());
14         return newList;
15     }
16 });

```

#### 1.6.3.10 STEP-9: Output Final Result

The `collect()` method collects all RDD's elements into a `java.util.List` object. Then we iterate through the `List` to get all final elements.

### **Listing 1.17: STEP-9: Output Final Result**

```
1 // STEP-9: validate STEP-8, we collect all values from JavaPairRDD<> and print it.
2 System.out.println("==DEBUG2==");
3 List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>> output3 = sorted.collect();
4 for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output3) {
5     Iterable<Tuple2<Integer, Integer>> list = t._2;
6     System.out.println(t._1);
7     for (Tuple2<Integer, Integer> t2 : list) {
8         System.out.println(t2._1 + "," + t2._2);
9     }
10    System.out.println("=====");
11 }
```

## **1.6.4 Spark Sample Run**

### **1.6.4.1 HDFS Input**

```
hadoop@hnode01319:~# hadoop fs -cat /mp/timeseries.txt
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
```

### **1.6.4.2 Spark Run**

As far as Spark/Hadoop is concerned: you can run a Spark application in three different modes:<sup>4</sup>

---

<sup>4</sup>For details, see: <http://spark.apache.org/docs/1.0.0/running-on-yarn.html>

- **Standalone mode**, which is the default setup. You start Spark master on a master node and a ”worker” on every slave node and submit you Spark application to the Spark master.
- **YARN client mode**: in this mode, you do not start a spark master or worker nodes. In this mode, you submit the Spark application to YARN, which runs the Spark driver in the client Spark process that submits the application.
- **YARN cluster mode**: in this mode, you do not start a spark master or worker nodes. In this mode, you submit the Spark application to YARN, which runs the Spark driver in the ApplicationMaster in YARN.

Below, we will show how to submit the ”secondary sort” application in Standalone and YARN client modes.

#### 1.6.4.3 Running Spark Standalone Mode

##### HDFS Input

```

1 # hadoop fs -cat /mp/timeseries.txt
2 x,2,9
3 y,2,5
4 x,1,3
5 y,1,7
6 y,3,1
7 x,3,6
8 z,1,4
9 z,2,8
10 z,3,7
11 z,4,0
12 p,2,6
13 p,4,7
14 p,1,9
15 p,6,0
16 p,7,3

```

##### The Script

```

1 # cat run_secondarysorting.sh
2 source /home/hadoop/conf/env_2.3.0.sh
3 export SPARK_HOME=/home/hadoop/spark-1.0.0
4 source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
5 source $SPARK_HOME/conf/spark-env.sh
6
7 CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop

```

```

8
9 jars='find $SPARK_HOME -name *.jar'
10 for j in $jars ; do
11     CLASSPATH=$CLASSPATH:$j
12 done
13 # app jar:
14 export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
15 export SPARK_CLASSPATH=$CLASSPATH
16 export SPARK_MASTER=spark://hnode01319.nextbiosystem.net:7077
17 INPUT=/mp/timeseries.txt
18 OPTIONS="--Dspark.master=$SPARK_MASTER"
19 $JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS SecondarySort $SPARK_MASTER $INPUT

```

## Log of the Run

```

1 # ./run_secondarysorting.sh
2 args[0]: <master>=spark://hnode01319.nextbiosystem.net:7077
3 args[1]: <file>=/mp/timeseries.txt
4 ...
5 === DEBUG STEP-4 ===
6 14/06/04 08:42:50 INFO mapred.FileInputFormat: Total input paths to process : 1
7 14/06/04 08:42:50 INFO spark.SparkContext: Starting job: collect at SecondarySort.java:85
8 ...
9 14/06/04 08:42:50 INFO scheduler.DAGScheduler: Submitting Stage 0
10    (MappedRDD[2] at mapToPair at SecondarySort.java:75), which has no missing parents
11 14/06/04 08:42:50 INFO scheduler.DAGScheduler: Submitting 1 missing tasks from Stage 0
12    (MappedRDD[2] at mapToPair at SecondarySort.java:75)
13 ...
14 14/06/04 08:42:54 INFO scheduler.DAGScheduler: Stage 0 (collect at SecondarySort.java:85)
15    finished in 4.058 s
16 14/06/04 08:42:54 INFO spark.SparkContext: Job finished: collect at SecondarySort.java:85,
17    took 4.153418069 s
18 x,2,2
19 y,2,2
20 x,1,1
21 y,1,1
22 y,3,3
23 x,3,3
24 z,1,1
25 z,2,2
26 z,3,3
27 z,4,4
28 p,2,2
29 p,4,4
30 p,1,1
31 p,6,6
32 p,7,7
33 === DEBUG STEP-6 ===
34 14/06/04 08:42:54 INFO spark.SparkContext: Starting job: collect
35    at SecondarySort.java:96
36 14/06/04 08:42:54 INFO scheduler.DAGScheduler: Registering RDD 2
37    (mapToPair at SecondarySort.java:75)
38 ...
39 14/06/04 08:42:55 INFO scheduler.DAGScheduler: Stage 1

```

```

40      (collect at SecondarySort.java:96) finished in 0.273 s
41 14/06/04 08:42:55 INFO spark.SparkContext: Job finished:
42      collect at SecondarySort.java:96, took 1.587001929 s
43
44 z
45 1,4
46 2,8
47 3,7
48 4,0
49 =====
50 p
51 2,6
52 4,7
53 1,9
54 6,0
55 7,3
56 =====
57 x
58 2,9
59 1,3
60 3,6
61 =====
62 y
63 2,5
64 1,7
65 3,1
66 =====
67 === DEBUG STEP-8 ===
68 14/06/04 08:42:55 INFO spark.SparkContext: Starting job: collect
69      at SecondarySort.java:158
70 ...
71 14/06/04 08:42:55 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0,
72      whose tasks have all completed, from pool
73 14/06/04 08:42:55 INFO spark.SparkContext: Job finished: collect at
74      SecondarySort.java:158, took 0.074271723 s
75 z
76 1,4
77 2,8
78 3,7
79 4,0
80 =====
81 p
82 1,9
83 2,6
84 4,7
85 6,0
86 7,3
87 =====
88 x
89 1,3
90 2,9
91 3,6
92 =====
93 y
94 1,7
95 2,5
96 3,1
97 =====

```

Typically, you save the final result to HDFS. This can be accomplished by adding the following line of code after creating your "sorted" RDD:

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

Then you may view the output as:

```
1 # hadoop fs -ls /mp/output/
2 Found 2 items
3 -rw-r--r-- 3 hadoop root,hadoop 0 2014-06-04 10:49 /mp/output/_SUCCESS
4 -rw-r--r-- 3 hadoop root,hadoop 125 2014-06-04 10:49 /mp/output/part-00000
5
6
7 # hadoop fs -cat /mp/output/part-00000
8 (z,[(1,4), (2,8), (3,7), (4,0)])
9 (p,[(1,9), (2,6), (4,7), (6,0), (7,3)])
10 (x,[(1,3), (2,9), (3,6)])
11 (y,[(1,7), (2,5), (3,1)])
12
```

## Running Spark YARN Cluster Mode

The Script to submit our Spark application YARN cluster mode is given below:

```
1 # cat run_secondarysorting_yarn.sh
2 #!/bin/bash
3 export HADOOP_HOME=/usr/local/hadoop/hadoop-2.3.0
4 export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
5 export JAVA_HOME=/usr/java/jdk7
6 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
7 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
8 export SPARK_HOME=/home/hadoop/spark-1.0.0
9 export MY_JAR=/home/hadoop/spark_mahmoud_examples/mp.jar
10 export SPARK_MASTER=spark://hnnode01319.nextbiosystem.net:7077
11
12 $SPARK_HOME/bin/spark-submit \
13   --class SecondarySort \
14   --master yarn-cluster \
15   --executor-memory 2G \
16   --num-executors 10 \
17   $MY_JAR \
18   $SPARK_MASTER \
19   /mp/timeseries.txt
20
```

### 1.6.5 Option-2: Secondary Sorting using Framework

In solution for Option-1, we sorted reducers values in memory (using Java's `Collections.sort()` method), which might not scale if the reducers values will not fit in a commodity server's memory. We implemented Option-2 for MapReduce/Hadoop framework. we cannot achieve this in current Spark (Spark-1.0.0) framework, because current Sparks Shuffle is based on hash, which is different from MapReduces sort-based shuffle, so you should implement sorting explicitly using RDD operator. If we had a partitioner by a natural key (by name), which preserved the order of RDD, then that would be a viable solution: for example, if we sort by (name, time), we would get:

```
(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)
```

There is a partitioner (represented as an abstract class `org.apache.spark.Partitioner`), but it does not preserve the order of original RDD elements. Therefore, Option-2 cannot be implemented by the current version of Spark (1.0.0).

# Chapter 2

## Secondary Sorting: Detailed Example

### 2.1 Introduction

MapReduce framework sorts input to reducers by key, but values of reducers are arbitrarily ordered. This means that if all mappers generated the following (key-value) pairs for `key = K`:

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

Then all these values  $\{V_1, V_2, \dots, V_n\}$  will be processed by a single reducer (for `key = K`), but there is no order (ascending or descending) between  $V_i$ 's. Secondary sorting is a design pattern which will put some kind of ordering (such as "ascending sort" or "descending sort") among the values  $V_i$ 's. How do we accomplish this? That is we want to have some order between the reducer values:

$$S_1 \leq S_2 \leq \dots \leq S_n$$

or

$$S_1 \geq S_2 \geq \dots \geq S_n$$

where  $S_i \in \{V_1, V_2, \dots, V_n\}$  for  $i = \{1, 2, \dots, n\}$ . Note that each  $V_i$  might be a simple data type such as String or Integer or a tuple (more than a single value – a composite object).

There are two ways to have sorted values for reducer values:

- **Solution-1:** Buffer reducer values in memory, then sort. If the number of reducer values are small enough, so they can fit in memory (per

reducer), then this solution will work. But if the the number of reducer values are high, then these values might not fit in memory (not a preferable solution). Implementation of this solution is trivial and will not be discussed in this chapter.

- **Solution-2:** Use ”secondary sorting” design pattern of MapReduce framework and reducer values will arrive sorted to a reducer (no need to sort values in memory). This technique uses the shuffle and sort technique of MapReduce framework to perform sorting of reducer values. This technique is preferable to Solution-1 because you do not depend on the memory for sorting (and if you have too many values, then Solution-1 might not be a viable option). The rest of this chapter will focus on presenting Solution-2. We present implementation of Solution-2 in

Hadoop by using

- Old Hadoop API (using `org.apache.hadoop.mapred.JobConf` and `org.apache.hadoop.mapred.*`); I intentionally included Hadoop’s old API if in case you are using an old API and have not migrated to new Hadoop API.
- New Hadoop API (using `org.apache.hadoop.mapreduce.Job` and `org.apache.hadoop.mapreduce.lib.*`)

## 2.2 Secondary Sorting Technique

Let’s have the following values for `key = K`:

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

and further assume that each  $V_i$  is a tuple of  $m$  attributes as:

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

where we want to sort reducer’s tuple values by  $a_{i1}$ . We will denote  $(a_{i2}, \dots, a_{im})$  (the remaining attributes) by  $r$ . Therefore, we can express reducer values as:

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

To sort the reducer values by  $a_i$ , we create a composite key:  $(K, a_i)$ . Our new mappers will emit the following (key, value) pairs for `key = K`.

Key	value
$(K, a_1)$	$(a_1, r_1)$
$(K, a_2)$	$(a_2, r_2)$
...	...
$(K, a_n)$	$(a_n, r_n)$

So the "composite key" is  $(K, a_i)$  and the "natural key" is  $K$ . Defining the composite key (by adding the attribute  $a_i$  to the "natural key" where the values will be sorted on) enables us to sort the reducer values by the MapReduce framework, but when we want to partition keys, we will partition it by the "natural key" (as  $K$ ). Below "composite key" and "natural key" are presented visually in [2.2](#).

Since we defined a "composite key" (composed of "natural key" (as  $K$ ) and an attribute (as  $a_i$ ) where the reducer values will be sorted on), we have to tell the MapReduce framework how to sort the keys by using a "composite key" (comprised of two fields:  $K$  and  $a_i$ ): for this we need to define a plug-in sort class, `CompositeKeyComparator`, which will be sorting the Composite Keys. This is how you plug-in this comparator class to a MapReduce framework:

#### Listing 2.1: Plug-in Comparator Class

```

1 import org.apache.hadoop.mapred.JobConf;
2 ...
3 JobConf conf = new JobConf(getConf(), <your-mapreduce-driver-class>.class);
4 ...
5 // map() creates (key-value) pairs of
6 // of (CompositeKey, NaturalValue)
7 conf.setMapOutputKeyClass(CompositeKey.class);
8 conf.setMapOutputValueClass(NaturalValue.class);
9 ...
10 // Plug-in Comparator Class:
11 // how CompositeKey objects will be sorted
12 conf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
```

The `CompositeKeyComparator` class is telling to the MapReduce framework how to sort the composite keys (comprised of two fields:  $K$  and  $a_i$ ). The implementation is provided below, which compares two `WritableComparables` objects (representing a `CompositeKey` object).

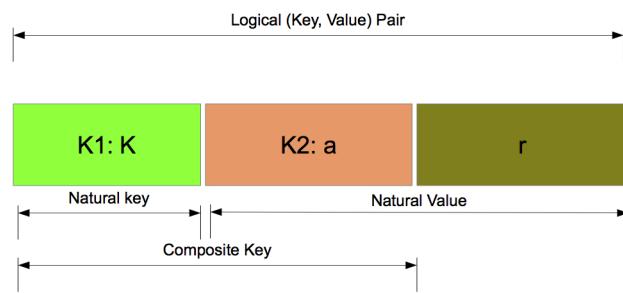


Figure 2.1: Secondary Sorting Keys

### **Listing 2.2:** Comparator Class: 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        // compare ck1 with ck2 and return
16        // 0, if ck1 and ck2 are identical
17        // 1, if ck1 > ck2
18        // -1, if ck1 < ck2
19
20        // detail of implementation is provided in sub sections
21    }
22}
```

The next piece of plug-in class is a "natural key partitioner" class (let's call this `NaturalKeyPartitioner` class), which will implement the `org.apache.hadoop.mapred.Partitioner` interface. This is how we plug-in the class to the MapReduce framework:

### **Listing 2.3:** Plug in Natural Key Partitioner

```
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);
```

Next, we define the Natural Key Partitioner class:

### **Listing 2.4:** Natural Key Partitioner Class Definition

```
1 import org.apache.hadoop.mapred.JobConf;
2 import org.apache.hadoop.mapred.Partitioner;
3
4 /**
5  * NaturalKeyPartitioner partitions the data output from the
6  * map phase before it is sent through the shuffle phase.
7  *
8  * getPartition() partitions data generated by mappers;
9  * This function should partition data by the "natural key".
10 *
```

```

11  */
12 public class NaturalKeyPartitioner implements
13     Partitioner<CompositeKey, NaturalValue> {
14
15     @Override
16     public int getPartition(CompositeKey key,
17                           NaturalValue value,
18                           int numberOfPartitions) {
19         return <number-based-on-composite-key> % numberOfPartitions;
20     }
21
22     @Override
23     public void configure(JobConf arg) {
24     }
25 }
```

The last piece to plugin is `NaturalKeyGroupingComparator`, which considers the natural key. This class just compares two natural keys. This is how you plug-in the class to the MapReduce framework:

#### **Listing 2.5:** Natural Key Grouping Comparator Plug-in

```

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);
```

This is how you define the `NaturalKeyGroupingComparator` class:

#### **Listing 2.6:** Natural Key Grouping Comparator Class

```

1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3
4 /**
5 *
6 * NaturalKeyGroupingComparator
7 *
8 * This class is used during Hadoop's shuffle phase to group
9 * composite Key's by the first part (natural) of their key.
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 }
```

---

## 2.3 Complete Example of Secondary Sorting

### 2.3.1 Problem Statement

Consider the following data:

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

and assume that we want to generate the following output data per stock-symbol:

Stock-Symbol: (Date<sub>1</sub>, Price<sub>1</sub>)(Date<sub>2</sub>, Price<sub>2</sub>)...(Date<sub>n</sub>, Price<sub>n</sub>)

where

$$\text{Date}_1 \leq \text{Date}_2 \leq \dots \leq \text{Date}_n$$

That is we want the reducer values to be sorted by the date of closed price. This can be accomplished by "secondary sorting".

### 2.3.2 Input Format

We assume that input data is in CSV format:

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

for example:

```
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
```

### 2.3.3 Output Format

We want our output to be sorted by "date of closed price": for our sample input, our desired output is listed below:

```
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)
```

### 2.3.4 Composite Key

The "natural key" is the stock symbol and the "composite key" is a pair of (Stock-Symbol, Date). The Date field has to be part of our "composite key" because we want reducer values to be sorted by Date. The "natural key" and "composite key" are illustrated below:

Therefore we can define the "composite key" as `CompositeKey` class and its associated comparator class as `CompositeKeyComparator` (how to sort objects of `CompositeKey`):

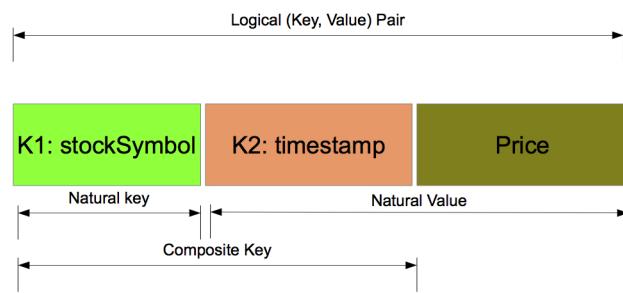


Figure 2.2: Secondary Sorting: Composite and Natural Keys

#### 2.3.4.1 Composite Key Definition

The Composite Key Definition is implemented as a `CompositeKey` class, which implements the `WritableComparable<CompositeKey>` interface<sup>1</sup>

**Listing 2.7:** Composite Key Definition

```
1 import java.io.DataInput;
2 import java.io.DataOutput;
3 import java.io.IOException;
4 import org.apache.hadoop.io.WritableComparable;
5 import org.apache.hadoop.io.WritableComparator;
6
7 /**
8 *
9 * CompositeKey: represents a pair of
10 * (String stockSymbol, long timestamp).
11 * Note that timestamp represents the Date.
12 *
13 *
14 * We do a primary grouping pass on the stockSymbol
15 * field to get all of the data of one type together,
16 * and then our "secondary sort" during the shuffle
17 * phase uses the timestamp long member to sort the
18 * data points so that they arrive at the reducer
19 * partitioned and in sorted order (by date).
20 *
21 */
22 public class CompositeKey implements WritableComparable<CompositeKey> {
23     // natural key is (stockSymbol)
24     // composite key is a pair (stockSymbol, timestamp)
25     private String stockSymbol; // stock symbol
26     private long timestamp; // date
27
28     public CompositeKey(String stockSymbol, long timestamp) {
29         set(stockSymbol, timestamp);
30     }
31
32     public CompositeKey() {
33     }
34
35     public void set(String stockSymbol, long timestamp) {
36         this.stockSymbol = stockSymbol;
37         this.timestamp = timestamp;
38     }
39
40     public String getStockSymbol() {
41         return this.stockSymbol;
42     }
43 }
```

---

<sup>1</sup>`WritableComparable`(s) can be compared to each other, typically via `Comparator`(s). Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.

```

44     public long getTimestamp() {
45         return this.timestamp;
46     }
47
48     @Override
49     public void readFields(DataInput in) throws IOException {
50         this.stockSymbol = in.readUTF();
51         this.timestamp = in.readLong();
52     }
53
54     @Override
55     public void write(DataOutput out) throws IOException {
56         out.writeUTF(this.stockSymbol);
57         out.writeLong(this.timestamp);
58     }
59
60     @Override
61     public int compareTo(CompositeKey other) {
62         if (this.stockSymbol.compareTo(other.stockSymbol) != 0) {
63             return this.stockSymbol.compareTo(other.stockSymbol);
64         }
65         else if (this.timestamp != other.timestamp) {
66             return timestamp < other.timestamp ? -1 : 1;
67         }
68         else {
69             return 0;
70         }
71     }
72 }
73 }
```

---

#### 2.3.4.2 Composite Key Comparator Definition

Composite Key Comparator Definition is implemented by the `CompositeKeyComparator` class which compares two `CompositeKey` objects by implementaing the `compare()` method. The `compare()` method returns 0 if they are identical, returns `-1` if the first composite key is smaller than the second one, otherwise returns `+1`.

**Listing 2.8:** Composite Key Comparator Definition

```

1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3
4 /**
5  * CompositeKeyComparator
6  *
7  * The purpose of this class is to enable comparison of two CompositeKey(s).
8  */
9 */
```

```

10 public class CompositeKeyComparator extends WritableComparator {
11
12     protected CompositeKeyComparator() {
13         super(CompositeKey.class, true);
14     }
15
16     @Override
17     public int compare(WritableComparable wc1, WritableComparable wc2) {
18         CompositeKey ck1 = (CompositeKey) wc1;
19         CompositeKey ck2 = (CompositeKey) wc2;
20
21         int comparison = ck1.getStockSymbol().compareTo(ck2.getStockSymbol());
22         if (comparison == 0) {
23             // stock symbols are equal here
24             if (ck1.getTimestamp() == ck2.getTimestamp()) {
25                 return 0;
26             }
27             else if (ck1.getTimestamp() < ck2.getTimestamp()) {
28                 return -1;
29             }
30             else {
31                 return 1;
32             }
33         }
34         else {
35             return comparison;
36         }
37     }
38 }

```

---

## 2.3.5 Sample Run

### 2.3.5.1 Implementation Classes using Old Hadoop API

<i>Class name</i>	<i>Description</i>
CompositeKey	Defines a composite key
CompositeKeyComparator	Implements sorting composite keys
DateUtil	Defines some useful date handling methods
HadoopUtil	Defines some utility functions
NaturalKeyGroupingComparator	Defines how natural keys will be grouped together
NaturalKeyPartitioner	Implements how natural keys will be partitioned
NaturalValue	Defines a natural value
SecondarySortDriver	Submits a job to Hadoop
SecondarySortMapper	Defines map()
SecondarySortReducer	Defines reduce()

### 2.3.5.2 Input

```
# 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
```

### 2.3.5.3 Running MapReduce Job

```
1 # ./run.sh
2 ...
3 ...
4 13/12/12 21:13:20 INFO mapred.FileInputFormat: Total input paths to process : 1
5 13/12/12 21:13:21 INFO mapred.JobClient: Running job: job_201312122109_0002
6 13/12/12 21:13:22 INFO mapred.JobClient: map 0% reduce 0%
7 13/12/12 21:13:29 INFO mapred.JobClient: map 23% reduce 0%
8 ...
9 13/12/12 21:14:24 INFO mapred.JobClient: map 100% reduce 93%
10 13/12/12 21:14:25 INFO mapred.JobClient: map 100% reduce 100%
11 13/12/12 21:14:26 INFO mapred.JobClient: Job complete: job_201312122109_0002
12 ...
13 13/12/12 21:14:26 INFO mapred.JobClient: Map-Reduce Framework
14 13/12/12 21:14:26 INFO mapred.JobClient: Map output materialized bytes=3143
15 13/12/12 21:14:26 INFO mapred.JobClient: Map input records=8
16 13/12/12 21:14:26 INFO mapred.JobClient: Reduce shuffle bytes=3143
17 13/12/12 21:14:26 INFO mapred.JobClient: Spilled Records=16
18 13/12/12 21:14:26 INFO mapred.JobClient: Map output bytes=238
19 13/12/12 21:14:26 INFO mapred.JobClient: Total committed heap usage (bytes)=4730576896
20 13/12/12 21:14:26 INFO mapred.JobClient: Map input bytes=185
21 13/12/12 21:14:26 INFO mapred.JobClient: Combine input records=0
22 13/12/12 21:14:26 INFO mapred.JobClient: SPLIT_RAW_BYTES=2520
23 13/12/12 21:14:26 INFO mapred.JobClient: Reduce input records=8
24 13/12/12 21:14:26 INFO mapred.JobClient: Reduce input groups=3
25 13/12/12 21:14:26 INFO mapred.JobClient: Combine output records=0
26 13/12/12 21:14:26 INFO mapred.JobClient: Reduce output records=3
27 13/12/12 21:14:26 INFO mapred.JobClient: Map output records=8
```

### 2.3.5.4 Output

```

1 # hadoop fs -ls /secondary_sort_chapter/output/
2 Found 12 items
3 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:14 /secondary_sort_chapter/output/_SUCCESS
4 drwxr-xr-x - mahmoud staff 0 2013-12-12 21:13 /secondary_sort_chapter/output/_logs
5 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:13 /secondary_sort_chapter/output/part-00000
6 -rw-r--r-- 1 mahmoud staff 66 2013-12-12 21:13 /secondary_sort_chapter/output/part-00001
7 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:13 /secondary_sort_chapter/output/part-00002
8 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:13 /secondary_sort_chapter/output/part-00003
9 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:14 /secondary_sort_chapter/output/part-00004
10 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:14 /secondary_sort_chapter/output/part-00005
11 -rw-r--r-- 1 mahmoud staff 61 2013-12-12 21:14 /secondary_sort_chapter/output/part-00006
12 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:14 /secondary_sort_chapter/output/part-00007
13 -rw-r--r-- 1 mahmoud staff 0 2013-12-12 21:14 /secondary_sort_chapter/output/part-00008
14 -rw-r--r-- 1 mahmoud staff 43 2013-12-12 21:14 /secondary_sort_chapter/output/part-00009
15
16
17 # hadoop fs -cat /secondary_sort_chapter/output/part*
18 GOOG      (2013-12-05,1057.34)(2013-12-06,1069.87)(2013-12-09,1078.14)
19 ILMN      (2013-12-05,97.65)(2013-12-06,99.25)(2013-12-09,101.33)
20 IBM       (2013-12-06,177.67)(2013-12-09,177.46)
21

```

## 2.4 Secondary Sort using New Hadoop API

### 2.4.0.5 Implementation Classes using New API

<i>Class name</i>	<i>Description</i>
CompositeKey	Defines a composite key
CompositeKeyComparator	Implements sorting composite keys
DateUtil	Defines some useful date handling methods
HadoopUtil	Defines some utility functions
NaturalKeyGroupingComparator	Defines how natural keys will be grouped together
NaturalKeyPartitioner	Implements how natural keys will be partitioned
NaturalValue	Defines a natural value
SecondarySortDriver	Submits a job to Hadoop
SecondarySortMapper	Defines map()
SecondarySortReducer	Defines reduce()

#### 2.4.0.6 Input

```
# 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
```

#### 2.4.0.7 Running MapReduce Job

```
1 # ./run.sh
2 ...
3 ...
4 13/12/14 21:18:25 INFO input.FileInputFormat: Total input paths to process : 1
5 ...
6 13/12/14 21:18:25 INFO mapred.JobClient: Running job: job_201312142112_0002
7 13/12/14 21:18:26 INFO mapred.JobClient: map 0% reduce 0%
8 13/12/14 21:18:31 INFO mapred.JobClient: map 100% reduce 0%
9 13/12/14 21:18:39 INFO mapred.JobClient: map 100% reduce 10%
10 ...
11 13/12/14 21:19:14 INFO mapred.JobClient: map 100% reduce 83%
12 13/12/14 21:19:15 INFO mapred.JobClient: map 100% reduce 100%
13 13/12/14 21:19:16 INFO mapred.JobClient: Job complete: job_201312142112_0002
14 ...
15 13/12/14 21:19:16 INFO mapred.JobClient: Map-Reduce Framework
16 13/12/14 21:19:16 INFO mapred.JobClient: Map output materialized bytes=314
17 13/12/14 21:19:16 INFO mapred.JobClient: Map input records=8
18 13/12/14 21:19:16 INFO mapred.JobClient: Reduce shuffle bytes=314
19 13/12/14 21:19:16 INFO mapred.JobClient: Spilled Records=16
20 13/12/14 21:19:16 INFO mapred.JobClient: Map output bytes=238
21 13/12/14 21:19:16 INFO mapred.JobClient: Total committed heap usage (bytes)=1034620928
22 13/12/14 21:19:16 INFO mapred.JobClient: Combine input records=0
23 13/12/14 21:19:16 INFO mapred.JobClient: SPLIT_RAW_BYTES=140
24 13/12/14 21:19:16 INFO mapred.JobClient: Reduce input records=8
25 13/12/14 21:19:16 INFO mapred.JobClient: Reduce input groups=3
26 13/12/14 21:19:16 INFO mapred.JobClient: Combine output records=0
27 13/12/14 21:19:16 INFO mapred.JobClient: Reduce output records=3
28 13/12/14 21:19:16 INFO mapred.JobClient: Map output records=8
29
```

#### 2.4.0.8 Output of MapReduce Job

```
# 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)
```

# Chapter 3

## Top 10 List

### 3.1 Introduction

Given a set of (**key-as-string**, **value-as-integer**) pairs, then finding a Top-N ( where  $N > 0$  ) list is a "design pattern" (a "design pattern" is a language-independent reusable solution to a common problem, which enable us to produce reusable code). For example, let **key-as-string** be a URL and **value-as-integer** be the number of times that URL is visited, then you might ask: what are the top-10 URLs for last week? This kind of a question is common for this type of (key, value) pairs. For details on the "top-10 list" design pattern refer to MapReduce Design Patterns book by Donald Miner[16]. Finding "top-10 list" falls into "Filtering Pattern" (you filter out data and find top-10 list).

This chapter provides a complete MapReduce solution for Top-10 design pattern and its associated implementations with Apache Hadoop (using classic MapReduce's `map()` and `reduce()` functions) and Apache Spark (using RDD's – Resilient Distributed Datasets). For this chapter we present 3 solutions:

- First Top-10 Solution in MapReduce/Hadoop. The assumption is that all input keys are unique. For a given input  $\{(K, V)\}$  all K's are unique.
- Second Top-10 Solution in Spark/Hadoop. The assumption is that all input keys are unique. For a given input  $\{(K, V)\}$  all K's are unique.
- Third Top-10 Solution in Spark/Hadoop. The assumption is that all

input keys are not unique. For a given input  $\{(K, V)\}$  all K's are not unique.

Our MapReduce solutions will generalize the "top 10 list" and will be able to find "top  $N$  list" (for  $N > 1$ ). For example, we will be able to find "top 10 cats", "top 50 most visited web sites", or "top 100 search queries of a search engine".

The question for "top  $N$  list" can be like:

1. What are the "top 10 list" of cats?
2. What are the "top 100 list" of web sites visited last month?
3. What are the "top 20 list" of search queries from Google or Yahoo in the last week?
4. What are the "top 10 list" of liked items from Facebook yesterday?
5. What are the "top 5 list" of cartoons of all time?

## 3.2 Top-N Formalized

Let  $N > 0$ , Let  $L$  be a `List<Tuple2<T, Integer>>`, where  $T$  can be any type,  $L.size() = S$ ,  $S > N$ , and elements of  $L$  be:

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

where  $K_i$  has a type of  $T$  and  $V_i$  is an `Integer` type (this is the frequency of  $K_i$ ). Let `sort(L)` be (sort is done by using frequency as a key):

$$\{(A_j, B_j), 1 \leq j \leq S, B_1 \geq B_2 \geq \dots \geq B_S\}$$

where  $(A_j, B_j) \in L$ . Then `top-N` of  $L$  is defined as:

$$\text{top-N}(L) = \{(A_j, B_j), 1 \leq j \leq N, B_1 \geq B_2 \geq \dots \geq B_N\}$$

The easy way to implement `top-N` in Java is to use `SortedMap` and `TreeMap` data structures and then keep adding all elements of  $L$  to `topN`, but make sure to remove the first element (an element with the smallest frequency) of `topN` if `topN.size() > N`. The following code segment (POJO-like) illustrates `top-N` algorithm for list  $L$ :

### Listing 3.1: Top-N Algorithm

```
1 import scala.Tuple2;
2 import java.util.List;
3 import java.util.TreeMap;
4 import java.util.SortedMap;
5 import <your-package>.T;
6 ...
7 static SortedMap<Integer,T> topN(List<Tuple2<T,Integer>> L, int N) {
8     SortedMap<Integer, T> topN = new TreeMap< Integer, T>();
9     for (Tuple2<T,Integer> element : L) {
10         // element._1 is a type T
11         // element._2 is the frequency of type Integer
12         topN.put(element._2, element._1);
13         // keep only top N
14         if (topN.size() > N) {
15             // remove element with the smallest frequency
16             topN.remove(topN.firstKey());
17         }
18     }
19     return topN;
20 }
```

## 3.3 MapReduce Solution

Let `cats` be a relation of 3 attributes: (`cat_id`, `cat_name`, `cat_weight`) and assume we have billions of cats (big data).

cats Table	
Attribute Name	Attribute Type
<code>cat_id</code>	String
<code>cat_name</code>	String
<code>cat_weight</code>	Double

Let  $N > 0$  and suppose we want to find "top  $N$  list" of cats (based on `cat_weight`). Before we delve into MapReduce solution, let's see how we can express "top 10 list" of cats in SQL:

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

The solution in SQL is very straightforward and will require sorting the whole *cats* table. The question is why not use a relational database and SQL

for this. The short answer is most often our big data is not as structured as relational databases and tables and quite often we do need to parse semi-structured data such as log files or other type of data to make this happen. And when your size of data is huge in relational databases, they become non-responsive and do not scale well.

The MapReduce solution is pretty straightforward: each mapper will find a local "top  $N$  list" (for  $N > 1$ ) and then will pass it to a SINGLE reducer. Then the single reducer will find the final "top  $N$  list" from all local "top  $N$  list" passed from mappers. In general, in most of the MapReduce algorithms, having a single reducer is problematic and will cause a performance bottleneck (the reason for bottleneck is that one reducer in one server receives all data – which can be very big data volume – and all other cluster nodes do nothing, so the entire pressure and load will be on a single node, which can cause performance bottlenecks). Here, our single reducer will not cause a performance problem. Here is how: let's assume that we will have 1000 mappers, then each mapper will only generate 10 (key, value) pairs. Therefore, our single reducer will only get 10,000 ( $10 \times 1000$ ) records (which is not that much data at all to cause performance bottleneck!).

The Top-10 algorithm is presented below. Input is partitioned into smaller chunks and each chunk is sent to a mapper. Each mapper creates a local top-10 list and then emits the local top-10 to be sent to reducers. In emitting mappers output, we use a single reducer key so that all mappers output will be consumed by a single reducer.

To parameterize the "top  $N$  list", we just need to pass the  $N$  from the driver (which launched the MapReduce job) to *map()* and *reduce()* functions by using MapReduce Configuration<sup>1</sup> object. The driver sets "top.n" parameter and *map()* and *reduce()* read that parameter in their *setup()* functions.

Here we will focus on finding "top  $N$  list" of cats. The mapper class will have the following structure:

Listing 3.1: Mapper Class Outline

```
// imports ...
```

---

<sup>1</sup> org.apache.hadoop.conf.Configuration

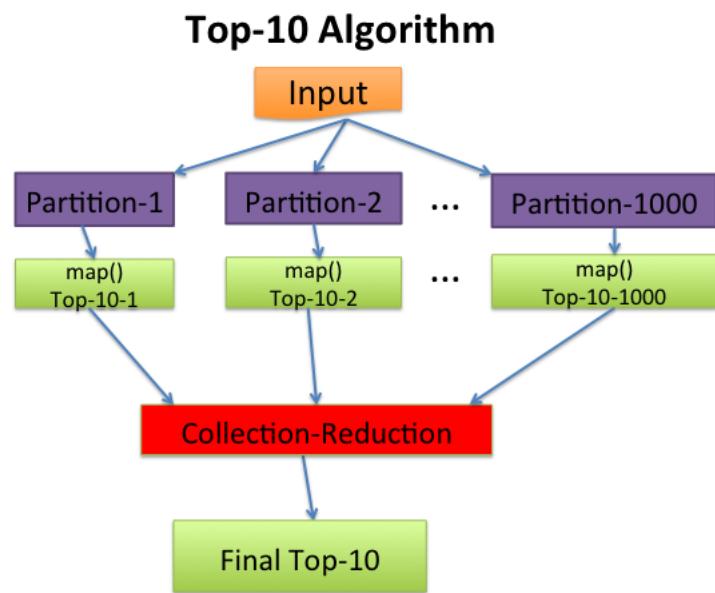


Figure 3.1: Top-10 MapReduce Algorithm: for Unique Keys

```

public class TopN_Mapper {
    // define data structures needed for finding local top-10
    private SortedMap<Double, Text> top10cats = new TreeMap<Double, Text>()
        ;
    private int N = 10; // default is top 10

    // setup() function will be executed once() per mapper
    setup(Context context) {
        ...
    }

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

    // cleanup() function will be executed once() per mapper
    cleanup(Context context) {
        ...
    }
}

```

---

Next, we define the setup() function:

Listing 3.2: setup() for Top N List

```

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

    /**
     * setup() function will be executed once() per mapper
     * Here we setup the "cats top N list" as top10cats
     */
    setup(Context context) {
        // "top.n" has to be set up by the driver of our job
        Configuration conf = context.getConfiguration();
        N = conf.get("top.n");
    }
}

```

---

The `map()` function accepts a chunk of input and generates a local top-10 list. We are using different delimiters to optimize parsing input by mappers

and reducers (to avoid non-necessary String concatenations).

Listing 3.3: map() for Top N List

```
/**  
 * @param key is generated by MapReduce framework, ignored here  
 * @param value as a String has the format:  
 * <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);  
  
    // keep only top N  
    if (top10cats.size() > N) {  
        // remove an element with the smallest frequency  
        top10cats.remove(top10cats.firstKey());  
    }  
}
```

Each mapper accepts a partition of cats. After mapper finishes creating a top-10 list as `SortedMap<Double, Text>`, the `cleanup()` method emits the top-10 list. Note that we use a single key as `NullWritable.get()`, which guarantees that all mappers's output will be consumed by a single reducer.

Listing 3.4: cleanup() for Top N List

```
/**  
 * cleanup() function will be executed once at the end of each mapper  
 * Here er setup the cats top N list as top10cats  
 */  
cleanup(Context context) {  
    // now we emit top N from this mapper  
    for (String catAttributes : top10cats.values() ) {  
        context.write(NullWritable.get(), catAttributes);  
    }  
}
```

The single reducer gets all local top-10 lists and create a single final top-10 list.

Listing 3.5: reduce() for Top N List

```
/**  
 * @param key is null (single reducer)  
 * @param values as a List of String and each element  
 * of the list has the format <cat-weight>,<cat-id>,<cat-name>  
 */  
reduce(key, values) {  
    SortedMap<Double, Text> finaltop10 = new TreeMap< Double, Text>();  
  
    // aggregate all local top-10 lists  
    for (Text catRecord : values) {  
        String[] tokens = catRecord.split(",");  
        Double weight = Double.parseDouble(tokens[0]);  
        finaltop10.put(weight, value);  
  
        if (finaltop10.size() > N) {  
            // remove an element with the smallest frequency  
            finaltop10.remove(finaltop10.firstKey());  
        }  
    }  
  
    // emit final top-10 list  
    for (Text text : finaltop10.values()) {  
        context.write(NullWritable.get(), text);  
    }  
}
```

## 3.4 Implementation in Hadoop

The MapReduce/Hadoop implementation is comprised of the following classes:

Implementation Classes in Hadoop	
Class Name	Class Description
TopN_Driver	Driver to submit job
TopN_Mapper	Defines map()
TopN_Reducer	Defines reduce()

The **TopN\_Driver** class reads  $N$  (for Top-N) from a command line and sets it in the Hadoop's Configuration object to be read by the `map()` function.

### 3.4.1 Input

```
# 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
```

### 3.4.2 Sample Run 1: find top 10 list

```
# cat run.sh
#!/bin/bash
export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true

export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.:$/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
PATH=$PATH:$HADOOP_HOME/bin
PATH=$PATH:$JAVA_HOME/bin
export PATH

CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.0.3.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.0.3.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.0.3.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.0.3.jar
```

```

CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.0.3.jar

CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/log4j-1.2.15.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-cli-1.2.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/lib/commons-logging-1.1.1.jar
JAR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/top10list/top10list.jar
export CLASSPATH=$CLASSPATH:$JAR
export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class

$ INPUT="/top10list/input"
$ OUTPUT="/top10list/output"
$ HADOOP_HOME/bin/hadoop fs -rmr $OUTPUT
$ HADOOP_HOME/bin/hadoop jar $JAR TopN_Driver $INPUT $OUTPUT

# ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
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

```

### 3.4.3 Output

```

# 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

```

### 3.4.4 Sample Run 2: find top 5 list

The default returns "top 10 list", to get "top 5 list", we just need to pass another parameter: this is how we can invoke our program:

```

$ INPUT="/top10list/input"
$ OUTPUT="/top10list/output"
$ $HADOOP_HOME/bin/hadoop jar $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

```

## 3.5 Bottom 10

In previous sections, we showed that how to find the "top 10" list. To find the "bottom 10", we just need to change one line of code:

Replace the following

```
// find top-10
if (top10cats.size() > N) {
    // remove the element with the smallest frequency
    top10cats.remove(top10cats.firstKey());
}
```

With

```
// find bottom-10
if (top10cats.size() > N) {
    // remove the element with the largest frequency
    top10cats.remove(top10cats.lastKey());
}
```

## 3.6 Spark Implementation: Unique Keys

For this implementation, we assume that for all given input (K, V) pairs, K's are unique. For Top-10 design pattern, we provide Spark/Hadoop<sup>2</sup> implementation. Spark has a higher-level abstraction than classical MapReduce/Hadoop. Spark provides a rich set of functional programming API, which makes MapReduce programming easy. Spark can read/write data from local file system as well as Hadoop's HDFS. Spark uses the Hadoop-client library to read/write from/to HDFS and other Hadoop-supported storage systems. Typically, Spark programs can run faster than equivalent MapReduce/Hadoop ones if you provide more RAM to the cluster nodes. Spark provides `StorageLevel` class, which has flags for controlling the storage of an RDD. Some of these flags are:

---

<sup>2</sup>Apache Spark is a fast and general-purpose cluster computing system. (source: <http://spark.apache.org/>)

- MEMORY\_ONLY (use only memory for RDDs)
- DISK\_ONLY (use only hard disk for RDDs)
- MEMORY\_AND\_DISK (use combination of memory and disk for RDDs).

Here, I am assuming that you have a ready Spark cluster running on top of Hadoop (or you may run Spark on YARN without starting a Spark cluster). To understand how to install Spark/Hadoop, you may look into:

- Apache Spark web site: <http://spark.apache.org/>
- Fast Data Processing with Spark book by Holden Karau (Packt Publishing)

### 3.6.1 Introduction

To understand Spark, we need to understand the concept of RDD (Resilient Distributed Dataset). RDD is the basic abstraction in Spark, which represents an immutable, partitioned collection of elements that can be operated on in parallel. Rather than working with different types of input/output, you work with an RDD, which can represent typed input/output. For example, the following code snippet presents two RDD's (`lines` and `words`):

```

1 // connect to Spark cluster by creating a JavaSparkContext object
2 JavaSparkContext ctx = new JavaSparkContext(...);
3
4 JavaRDD<String> lines = ctx.textFile(args[1], 1);
5
6 JavaRDD<String> words = lines.flatMap(new FlatMapFunction<String, String>() {
7     public Iterable<String> call(String s) {
8         return Arrays.asList(s.split(" "));
9     }
10});
```

The following table explains the code:

Spark RDD's	
Line(s)	Description
1	creates a Java Spark Context as <code>JavaSparkContext</code> , represents the connection to a Spark cluster, and can be used to create RDDs, accumulators and broadcast variables on that cluster.
3	Creates a new <code>JavaRDD&lt;String&gt;</code> , which represents a text file as a set of lines (each line/record is a <code>String</code> object)
5-9	We create a new RDD (from an existing RDD) as <code>JavaRDD&lt;String&gt;</code> , which represents a tokenized set of words. <code>FlatMapFunction&lt;T, R&gt;</code> has a function type of <code>T =&gt; Iterable&lt;R&gt;</code>

Spark is very powerfull in creating new RDD's from existing ones. For example, below, we use `lines` to create a new RDD as `JavaPairRDD<Integer, String>` as `pairs`.

```

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

```

Each item in `JavaPairRDD<String, Integer>` represents a `Tuple2<String, Integer>`. Here we assume that each input record has two tokens: `<String><, ><Integer>`.

### 3.6.2 What is an RDD?

In Spark, an RDD (Resilient Distributed Dataset) is the basic data abstraction. RDDs are used to represent a set of immutable objects in Spark. For example, to represent a set of `Strings`, we can use `JavaRDD<String>` and to represent (`key-as-string, value-as-integer`) pairs, we can use `JavaPairRDD<String, Integer>`. RDDs enable MapReduce operations (such as `map` and `reduceByKey`) to run in parallel (parallelism is achieved by partitioning RDDs). Spark's API enables us to implement custom RDDs. For details on RDDs, refer to <http://spark.apache.org/docs/>.

### 3.6.3 Spark's Function Classes

Java does not support anonymous or first-class functions (some of these are implemented in JDK8), so functions must be implemented by extending

ing the `Function`<sup>3</sup>, `Function2`<sup>4</sup>, etc. classes. Spark's RDD methods like `collect()` and `countByKey()` return Java collections data types, such as `java.util.List` and `java.util.Map`. To represent (key-value) pairs, you may use `scala.Tuple2`, which is a Java class and (key, value) can be created by using `new Tuple2<K, V>(key, value)`.

The main power of Spark is in Function Classes (represented in the following table) used by Java API, where each class has a single abstract method, `call()`, that must be implemented by a programmer.

Spark Java Class	Function Type
<code>Function&lt;T, R&gt;</code>	<code>T =&gt; R</code>
<code>DoubleFunction&lt;T&gt;</code>	<code>T =&gt; Double</code>
<code>PairFunction&lt;T, K, V&gt;</code>	<code>T =&gt; Tuple2&lt;K, V&gt;</code>
<code>FlatMapFunction&lt;T, R&gt;</code>	<code>T =&gt; Iterable&lt;R&gt;</code>
<code>DoubleFlatMapFunction&lt;T&gt;</code>	<code>T =&gt; Iterable&lt;Double&gt;</code>
<code>PairFlatMapFunction&lt;T, K, V&gt;</code>	<code>T =&gt; Iterable&lt;Tuple2&lt;K, V&gt;&gt;</code>
<code>Function2&lt;T1, T2, R&gt;</code>	<code>T1, T2 =&gt; R (function of two arguments)</code>

### 3.6.4 Spark Solution for Top-10 Pattern

In Spark, you may write your whole job of processing big data in a single driver. This is possible, since Spark offers a rich high-level abstraction on MapReduce paradigm. Before presenting Top-10 algorithm in Spark, let's review the Top-10 algorithm. Let's assume that our input records will have the following format

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

and the goal is to find Top-10 list for a given input. First, we will partition input into segments (let's say, we partition our input into 1000 mappers – each mapper will work on one segment of the partition independently):

```
1 class mapper :
2     setup(): initialize top10 SortedMap<Integer, String>
3
4     map(key, inputrecord):
```

<sup>3</sup>`org.apache.spark.api.java.function.Function`

<sup>4</sup>`org.apache.spark.api.java.function.Function2`

```

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>

```

Since Spark does not support classic `setup()` and `cleanup()` functions in the mapper or reducer, we will utilize `JavaPairRDD.mapFunctions()` method to achieve the same functionality as MapReduce/Hadoop `setup()` and `cleanup()` methods. By using Spark's `JavaPairRDD.mapFunctions()` we achieve the same functionality and having the "setup()" code at the top of the `mapPartitions()` and "cleanup()" at the bottom.

The job of a reducer is almost similar to the mapper: it finds the top-10 from a given set of all top-10 generated by mappers. The reducer will get a collection of `SortedMap<Integer, String>` (as an input) and will create a single final `SortedMap<Integer, String>` as an output.

```

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

```

### 3.6.5 Complete Spark Solution for Top-10 Pattern

First, I will present all main steps and then will expand each step. We use a single Java class (represented as `Top10.java`) for solving Top-10 in Spark:

**Listing 3.2:** Top-10 Program in Spark

```

1 // STEP-0: import required classes
2 public class Top10 {
3     public static void main(String[] args) throws Exception {
4         // STEP-1: make sure we have two input parameters
5         // STEP-2: create a connection to the Spark master
6         // STEP-3: read input file from HDFS
7         // STEP-4: create set of Tuple2<Integer, String>
8         // STEP-5: create a local top-10 for each input partition
9         // STEP-6: collect all local top-10's and create a final top-10 list
10        // STEP-7: output the final top-10 list
11        System.exit(0);
12    }
13 }

```

---

### 3.6.5.1 Top10 class: STEP-0

The following section shows the classes, which needs to be imported.

**Listing 3.3:** Top-10 Program in Spark: import required classes

```

1 // STEP-0: import required classes
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;

```

---

### 3.6.5.2 Top10 class: STEP-1

In this step, we make sure we have two input parameters: Spark-Master and HDFS Input File. Sample example for these two input parameters will be

```

args[0]: spark://myserver.mycompany.com:7077
args[1]: /top10/input/top10data.txt

```

#### **Listing 3.4:** Top-10 Program in Spark: handle input parameters

```
1 // STEP-1: make sure we have two input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: Top10 <spark-master> <hdfs-file>");
4     System.exit(1);
5 }
6 System.out.println("args[0]: <spark-master>=" + args[0]);
7 System.out.println("args[1]: <hdfs-file>=" + args[1]);
```

#### **3.6.5.3 Top10 class: STEP-2**

In this step, we create a connection object (as JavaSparkContext) to a Spark-Master. We will use the ctx object to create RDD's.

#### **Listing 3.5:** Top-10 Program in Spark: connect to Spark Master

```
1 // STEP-2: create a connection to the Spark master
2 JavaSparkContext ctx = new JavaSparkContext(
3         args[0], // <spark-master>
4         "Top10",
5         System.getenv("SPARK_HOME"),
6         System.getenv("SPARK_EXAMPLES_JAR")
7     );
```

#### **3.6.5.4 Top10 class: STEP-3**

In this step, we read an file file from HDFS and create an RDD of JavaRDD<String>.

#### **Listing 3.6:** Top-10 Program in Spark: read input file from HDFS

```
1 // STEP-3: read input file from HDFS
2 // input record format: <string-key>,<><integer-value>
3 JavaRDD<String> lines = ctx.textFile(args[1], 1);
```

#### **3.6.5.5 Top10 class: STEP-4**

In this step, we create a new RDD (JavaPairRDD<Integer, String>) from an existing RDD (JavaRDD<String>). The PairFunction class has 3 arguments: the first argument (T) is an input and the last two parameters (K, V) are output:

- Spark Java Class: PairFunction<T, K, V>

- Function Type:  $T \Rightarrow \text{Tuple2}\langle K, V \rangle$

**Listing 3.7:** Top-10 Program in Spark: create set of Tuple2

```

1 // STEP-4: create set of 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});
```

### 3.6.5.6 Top10 class: STEP-5

In this step, again, we create a new RDD ( $\text{JavaRDD}\langle \text{SortedMap}\langle \text{Integer}, \text{String} \rangle \rangle$ ) from an existing RDD ( $\text{JavaPairRDD}\langle \text{String}, \text{Integer} \rangle$ ). The `mapPartitions()` is a very powerful method and enable us to achieve the semantic of classic MapReduce/Hadoop methods (`setup()`, `map()`, and `cleanup()`). In this step, we create a local top-10 list from each partition.

**Listing 3.8:** Top-10 Program in Spark: create a local top-10 for each input partition

```

1 // STEP-5: create a local top-10 for each input partition
2 JavaRDD<SortedMap<Integer, String>> partitions = pairs.mapPartitions(
3     new FlatMapFunction<Iterator<Tuple2<String, Integer>>, SortedMap<Integer, String>>() {
4         @Override
5         public Iterable<SortedMap<Integer, String>> call(Iterator<Tuple2<String, Integer>> iter) {
6             SortedMap<Integer, String> top10 = new TreeMap<Integer, String>();
7             while (iter.hasNext()) {
8                 Tuple2<String, Integer> tuple = iter.next();
9                 top10.put(tuple._2, tuple._1);
10                // keep only top N
11                if (top10.size() > 10) {
12                    // remove the element with the smallest frequency
13                    top10.remove(top10.firstKey());
14                }
15            }
16            return Collections.singletonList(top10);
17        }
18   });
```

How did we achieve the implementation of classic MapReduce/Hadoop methods (`setup()`, `map()`, and `cleanup()`). Line 6 accomplishes the `setup()` equivalent (creates and initializes a local top10 as `SortedMap<Integer, String>`). Lines 7-14 implements the `map()` function, and finally the `cleanup()` method is implemented by Line 16 (by returning the result of a local top-10).

### 3.6.5.7 Top10 class: STEP-6

This step iterates over all local top-10 lists created per partition and creates a single final top-10 list. To get all local top-10 lists, we use the `collect()` method and then iterate over all local top-10 lists.

**Listing 3.9:** Top-10 Program in Spark: create the final top-10 list

```

1 // STEP-6: collect all local top-10's and create a final top-10 list
2 SortedMap<Integer, String> finaltop10 = new TreeMap<Integer, String>();
3 List<SortedMap<Integer, String>> alltop10 = partitions.collect();
4 for (SortedMap<Integer, String> localtop10 : alltop10) {
5     // weight = tuple._1
6     // catname = tuple._2
7     for (Map.Entry<Integer, String> entry : localtop10.entrySet()) {
8         // System.out.println(entry.getKey() + " -- " + entry.getValue());
9         finaltop10.put(entry.getKey(), entry.getValue());
10        // keep only top 10
11        if (finaltop10.size() > 10) {
12            // remove the element with the smallest frequency
13            finaltop10.remove(finaltop10.firstKey());
14        }
15    }
16 }
```

### 3.6.5.8 Top10 class: STEP-7

This step emits the final top-10 list by iterating over the `SortedMap<Integer, String>`.

**Listing 3.10:** Top-10 Program in Spark: emit the final top-10 list

```

1 // STEP-7: output the final top-10 list
2 System.out.println("== top-10 list ==");
3 for (Map.Entry<Integer, String> entry : finaltop10.entrySet()) {
4     System.out.println(entry.getKey() + " -- " + entry.getValue());
5 }
```

### 3.6.6 Input

For our sample run, I have created a sample input file:

```
# hadoop fs -ls /top10/input/top10data.txt
Found 1 items
-rw-r--r-- 3 hadoop,hadoop 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
```

### 3.6.7 Sample Run : find top-10 list

This sample is run on a 3-node cluster: sparkserver100, sparkserver200, sparkserver300, where sparkserver100 is the master Spark node.

#### 3.6.7.1 Sample Run : The Script to run Top-10

```
# cat run_top10.sh
source /home/hadoop/conf/env_2.3.0.sh
export SPARK_HOME=/home/hadoop/spark-1.0.0
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
source $SPARK_HOME/conf/spark-env.sh
```

```

# system jars:
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
jars='find $SPARK_HOME -name *.jar'
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
export SPARK_CLASSPATH=$CLASSPATH
export SPARK_MASTER=spark://sparkserver100:7077
OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER"
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS Top10 $SPARK_MASTER /top10/top10data.txt

```

### 3.6.7.2 Top-10 Run with Spark Cluster

Output id formatted to fit the page style.

```

1  # ./run_top10.sh
2  hadoop@hnode01319:~/spark_mahmoud_examples# ./run_top10.sh
3  args[0]: <master>=spark://myserver100:7077
4  args[1]: <file>/top10/top10data.txt
5  ...
6
7  14/06/03 22:42:24 INFO scheduler.DAGScheduler: Completed ResultTask(0, 0)
8  14/06/03 22:42:24 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0,
9  whose tasks have all completed, from pool
10 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Stage 0 (collect at Top10.java:69)
11  finished in 4.418 s
12 14/06/03 22:42:24 INFO spark.SparkContext: Job finished: collect at Top10.java:69,
13  took 4.5214644436 s
14  key= cat10    value= 10
15  key= cat1     value= 1
16  key= cat10000 value= 10000
17  key= cat400   value= 400
18  key= cat500   value= 500
19  key= cat2     value= 2
20  key= cat4     value= 4
21  key= cat6     value= 6
22  key= cat1200  value= 1200
23  key= cat10    value= 10
24  key= cat11    value= 11
25  key= cat12    value= 12
26  key= cat13    value= 13
27  key= cat50    value= 50
28  key= cat51    value= 51
29  key= cat45    value= 45
30  key= cat46    value= 46
31  key= cat200   value= 200
32  key= cat234   value= 234
33 14/06/03 22:42:24 INFO spark.SparkContext: Starting job: collect at Top10.java:116
34 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Got job 1 (collect at Top10.java:116)
35  with 1 output partitions (allowLocal=false)
36 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Final stage: Stage 1(collect at Top10.java:116)
37 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Parents of final stage: List()
38 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Missing parents: List()
39 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Submitting Stage 1
40  (MapPartitionsRDD[3] at mapPartitions at Top10.java:94), which has no missing parents

```

```

41 14/06/03 22:42:24 INFO scheduler.DAGScheduler: Submitting 1 missing tasks from Stage 1
42   (MapPartitionsRDD[3] at mapPartitions at Top10.java:94)
43 ...
44 14/06/03 22:42:25 INFO scheduler.DAGScheduler: Stage 1
45   (collect at Top10.java:116) finished in 1.134 s
46 14/06/03 22:42:25 INFO scheduler.TaskSchedulerImpl:
47   Removed TaskSet 1.0, whose tasks have all completed, from pool
48 14/06/03 22:42:25 INFO spark.SparkContext: Job finished:
49   collect at Top10.java:116, took 1.15125893 s
50 45--cat45
51 46--cat46
52 50--cat50
53 51--cat51
54 200--cat200
55 234--cat234
56 400--cat400
57 500--cat500
58 1200--cat1200
59 10000--cat10000
60

```

## 3.7 What If for Top-N

In MapReduce/Hadoop implementation, we were able to parameterize N, where we could find Top-10, Top-20, or Top-100. How do we parameterize N in Spark. It is easy to do this in Spark. We can make N as a global shared data and then be able to access it from any cluster node. This is how we do it:

```

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 JavaContextObject context = <create-acontext-object>;
7 ...
8 final Broadcast<Integer> broadcastTopN = context.broadcast(topN);
9
10 RDD.map() {
11   ...
12   final int topN = broadcastTopN.value();
13   // use topN
14   ...
15 }
16
17 RDD.groupBy() {
18   ...
19   final int topN = broadcastTopN.value();
20   // use topN

```

```
21     ...
22 }
```

### 3.7.1 Shared Data Structures Definition and Usage

- Lines 1-2: import required classes. The `Broadcast` class enable us to define global shared data structures and then read them from any cluster node within mappers, reducers, and transformers. The general format to define a shared data structure of type T is:

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

After a data structure (`broadcastT` is broadcasted), then it may be read from any cluster node within mappers, reducers, and transformers.

- Line 4: define your top-N as top-10, top-20, or top-100
- Line 6: create an instance of `JavaContextObject`
- Line 8: define a global shared data structure for `topN` (which can be any value)
- Line 12, 19: read and use a global shared data structure for `topN` (from any cluster node). The general format to read a shared data structure of type T is:

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

## 3.8 What If for Bottom-N

What if we want to find Top-N or Bottom-N. This can be achieved by another parameter, which can be shared among all cluster nodes. For this, we introduce another shared variable, called `direction`, which can be in {"`top`", "`bottom`"}. This is how we do it:

```

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 JavaContextObject context = <create-acontext-object>;
7 ...
8 String direction = "top"; // or "bottom"
9 ...
10 final Broadcast<Integer> broadcastN = context.broadcast(N);
11 final Broadcast<String> broadcastDirection = context.broadcast(direction);

```

Now, based on the value of `broadcastDirection`, we will either remove the first entry (when direction is equal to "top") or last entry (when direction is equal to "bottom"): this has to be done consistently to all code.

```

1 String direction = broadcastDirection.value();
2 if (direction.equals("top")) {
3     // remove element with the smallest frequency
4     finalN.remove(finalN.firstKey());
5 }
6 else {
7     // direction.equals("bottom")
8     // remove element with the largest frequency
9     finalN.remove(finalN.lastKey());
10 }

```

## 3.9 Spark Implementation : Non-Unique Keys

For this implementation, we assume that for all given (K, V) pairs, K's are not unique. Since our K's are not unique, we have to do some extra steps to make sure that our keys are unique before applying the top-10 algorithm. To understand the solution for non-unique keys, I provide a simple example. Let's assume that we want to find top-10 visited URLs for a web site. Further assume that we have 3 web servers (web-server-1, web-server-2, and web-server-3) and each web server has collected URLs in this form:

(URL, count)

To further understand the non-unique keys concept, let's assume that we have only 7 URLs : {A, B, C, D, E, F, G} and the following are tallies of URLs generated per web server:

(K,V) Per Web Server		
Web Server 1	Web Server 2	Web Server 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)

Let's assume that we want to get top-2 of all visited URLs. If we get local top-2 per each web server and then get the top-2 of all three local top-2's, the result will not be correct. The reason is that URLs are not unique among all web servers. To make a correct solution, first we create unique URLs from all input and then we partition unique URLs into  $M > 0$  partitions. Next we get the local top-2 per partition and finally we perform final top-2 amng all local top-2's. For our example, the generated unique URLs will be:

Aggregated/Reduced (K,V) Pairs
(A, 5)
(B, 5)
(C, 7)
(D, 4)
(E, 3)
(F, 2)
(G, 6)

Now assume that we partition all unique URLs into two partitions:

(K,V) Per Partition	
Partition-1	Partition-2
(A, 5)	(D, 4)
(B, 4)	(E, 3)
(C, 7)	(F, 2)
	(G, 6)

To find Top-2 of all data:

```
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) }
```

So the main point is that before finding Top-N of any set of (K,V) pairs, we have to make sure that all K's are unique.

For Top-10 design pattern, we provide Spark/Hadoop implementation with the assumption that all K's are non-unique. Our top-N algorithm (only main steps) is presented below:

- MAIN-STEP-1: Make all your K's unique. To make K's unique, we will map our input into `JavaPairRDD<K, V>` and then `reduceByKey()`: this will provide us unique K's.
- MAIN-STEP-2: Partition all unique (K, V) pairs into M partitions.
- MAIN-STEP-3: Find Top-N for each partition (call this a local top-N)
- MAIN-STEP-4: Find Top-N from all local top-N's

The Top-10 algorithm for non-unique keys is presented below.

### 3.9.1 Complete Spark Solution for Top-10 Pattern

First, I will present all high-level steps and then will expand each step. We use a single Java class (represented as `Top10NonUnique.java`) for solving Top-10 in Spark. Note that this solution is a general solution and it really does not matter if all key's are unique or not. One of the steps for this solution is to make sure that all keys's are unique.

**Listing 3.11:** Top10NonUnique Program in Spark

```
1 package org.dataalgorithms.chap03;
2 // STEP-0: import required classes and interfaces
3 /**
4 * Assumption: for all input (K, V), K's are non-unique.
5 * This class implements Top-N design pattern for N > 0.
6 * The main assumption is that for all input (K, V)'s, K's
```

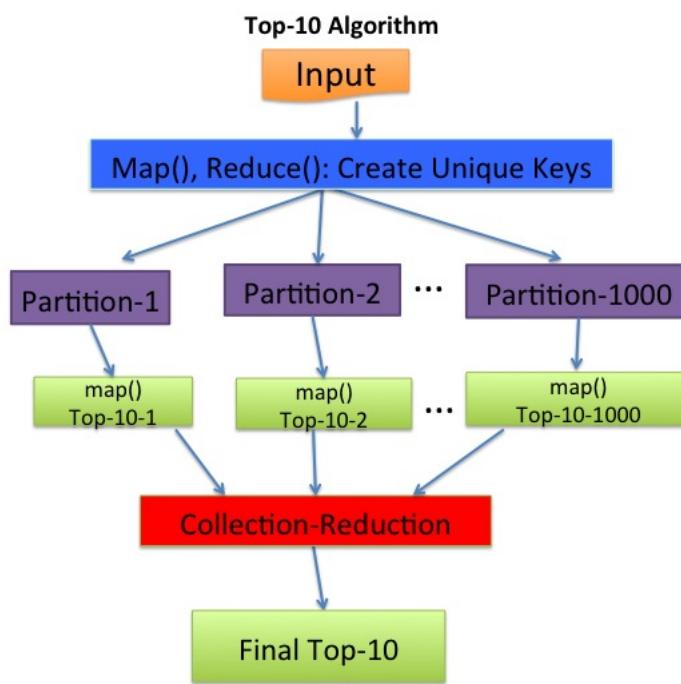


Figure 3.2: Top-10 MapReduce Algorithm: for Non-Unique Keys

```

7 * are non-unique. It means that you may find entries like
8 * (A, 2), ..., (A, 5),.... If we find duplicate K's, then
9 * we will sum up the values for them and then create a unique
10 * K. If we have (A, 2) and (A, 5) then a unique entry will
11 * be created as (A, 7).
12 *
13 * This class may be used to find bottom-N as well (by
14 * just keeping N-smallest elements in the set.
15 *
16 * Top-10 Design Pattern: Top Ten Structure
17 *
18 * 1. map(input) => (K, V)
19 * 2. reduce(K, List<V1, V2, ..., Vn>) => (K, V),
20 * where V = V1+V2+...+Vn; now all K's are unique
21 * 3. partition (K,V)'s into P partitions
22 * 4. Find top-N for each partition (we call this a local Top-N)
23 * 5. Find Top-N from all local Top-N's
24 *
25 * @author Mahmoud Parsian
26 *
27 */
28 public class Top10NonUnique {
29     public static void main(String[] args) throws Exception {
30         // STEP-1: handle input parameters
31         // STEP-2: create a Java Spark Context object
32         // STEP-3: broadcast the topN to all cluster nodes
33         // STEP-4: create an RDD from input
34         // STEP-5: partition RDD
35         // STEP-6: map input (T) into (K,V) pair
36         // STEP-7: reduce frequent K's
37         // STEP-8: create a local top-N
38         // STEP-9: find a final top-N
39         // STEP-10: emit final top-N
40         System.exit(0);
41     }
42 }
```

---

### 3.9.1.1 Input

Sample input is provided. We use this HDFS input to print out output for each step. As you can observe, the keys are non-unique. Before applying the top-10 algorithm, unique keys will be generated.

```

1 # hadoop fs -ls /top10/top10input/
2 Found 3 items
3 -rw-r--r-- 3 hadoop hadoop 24 2014-08-31 12:50 /top10/top10input/file1.txt
4 -rw-r--r-- 3 hadoop hadoop 24 2014-08-31 12:50 /top10/top10input/file2.txt
5 -rw-r--r-- 3 hadoop hadoop 28 2014-08-31 12:50 /top10/top10input/file3.txt
6
7 # hadoop fs -cat /top10/top10input/file1.txt
8 A,2
9 B,2
```

```

10 | C,3
11 | D,2
12 | E,1
13 | G,2
14 |
15 | # hadoop fs -cat /top10/top10input/file2.txt
16 | A,1
17 | B,1
18 | C,3
19 | E,1
20 | F,1
21 | G,2
22 |
23 | # hadoop fs -cat /top10/top10input/file3.txt
24 | A,2
25 | B,2
26 | C,1
27 | D,2
28 | E,1
29 | F,1
30 | G,2

```

**Listing 3.12:** STEP-0: import required classes and interfaces

```

1 // STEP-0: import required classes and interfaces
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;

```

### 3.9.1.2 STEP-1: handle input parameters

**Listing 3.13:** STEP-1: handle input parameters

```

1 // STEP-1: handle input parameters
2 if (args.length < 3) {
3     // Spark master URL:
4     //   format: spark://<spark-master-host-name>:7077
5     //   example: spark://myserver00:7077
6     System.out.println("Usage: Top10 <spark-master-URL> <input-path> <topN>");
7     System.exit(1);
8 }
9 System.out.println("args[0]: <spark-master-URL>=" + args[0]);
10 System.out.println("args[1]: <input-path>=" + args[1]);
11 System.out.println("args[2]: <topN>=" + args[2]);
12 final int N = Integer.parseInt(args[2]);

```

---

### 3.9.1.3 STEP-2: create a Java Spark Context object

**Listing 3.14:** STEP-2: create a Java Spark Context object

```

1 // STEP-2: create a Java Spark Context object
2 // args[0] denotes a Spark Master URL
3 JavaSparkContext ctx = SparkUtil.createJavaSparkContext(args[0], "Top10");

```

---

### 3.9.1.4 STEP-3: broadcast the topN to all cluster nodes

To broadcast or share objects and data structures among all cluster nodes, you may use Spark's **Broadcast** class.

**Listing 3.15:** STEP-3: broadcast the topN to all cluster nodes

```

1 // STEP-3: broadcast the topN to all cluster nodes
2 final Broadcast<Integer> topN = ctx.broadcast(N);
3 // now topN is available to be read from all cluster nodes

```

---

### 3.9.1.5 STEP-4: create an RDD from input

Input data is read from HDFS and the first RDD is created.

**Listing 3.16:** STEP-4: create an RDD from input

```

1 // STEP-4: create an RDD from input
2 //   input record format:
3 //     <string-key>,><integer-value-count>
4 JavaRDD<String> lines = ctx.textFile(args[1], 1);
5 lines.saveAsTextFile("/output/1");

```

---

To debug STEP-4, the first RDD is printed out:

```
1 # hadoop fs -cat /output/1/part*
2 A,2
3 B,2
4 C,3
5 D,2
6 E,1
7 G,2
8 A,1
9 B,1
10 C,3
11 E,1
12 F,1
13 G,2
14 A,2
15 B,2
16 C,1
17 D,2
18 E,1
19 F,1
20 G,2
```

### 3.9.1.6 STEP-5: partition RDD

Partitioning RDD is an art and science. What is the right number of partitions? There is no magic bullet formula for calculating the number of partitions. This does depend on the number of cluster nodes, the number of cores per server, and the size of RAM available. My experience indicate that you need to set this by trial and experience.

#### **Listing 3.17: STEP-5: partition RDD**

```
1 // STEP-5: partition RDD
2 // public JavaRDD<T> coalesce(int numPartitions)
3 // Return a new RDD that is reduced into numPartitions partitions.
4 JavaRDD<String> rdd = lines.coalesce(9);
```

---

### 3.9.1.7 STEP-6: map input(T) into (K,V) pair

This step does basic mapping: it converts every input record into a (K,V) pair, where K is a "key such as URL" and V is a "value such as count". This step will generate duplicate keys.

**Listing 3.18:** STEP-6: map input(T) into (K,V) pair

```
1 // STEP-6: map input(T) into (K,V) pair
2 // PairFunction<T, K, V>
3 // T => Tuple2<K, V>
4 JavaPairRDD<String, Integer> kv = rdd.mapToPair(new PairFunction<String, String, Integer>() {
5     public Tuple2<String, Integer> call(String s) {
6         String[] tokens = s.split(","); // url,789
7         return new Tuple2<String, Integer>(tokens[0], Integer.parseInt(tokens[1]));
8     }
9 });
10 kv.saveAsTextFile("/output/2");
```

```
1 # hadoop fs -cat /output/2/part*
2 (A,2)
3 (B,2)
4 (C,3)
5 (D,2)
6 (E,1)
7 (G,2)
8 (A,1)
9 (B,1)
10 (C,3)
11 (E,1)
12 (F,1)
13 (G,2)
14 (A,2)
15 (B,2)
16 (C,1)
17 (D,2)
18 (E,1)
19 (F,1)
20 (G,2)
```

### 3.9.1.8 STEP-7: reduce frequent Keys

Previos step (STEP-6) generated duplicate keys. This step creates unique keys and aggregates the associated values. For example, if for a key K we have:

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

Then, these will be aggregated/reduced into a (K,V) where

$$V = (V_1 + V_2 + \dots + V_n)$$

The reducer is implemented by `JavaPairRDD.reduceByKey()`.

### Listing 3.19: STEP-7: reduce frequent Keys

```
1 // STEP-7: reduce frequent K's
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");
```

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

### 3.9.1.9 STEP-8: create a local top-N

The goal of this step is to create a local top-10 per partition.

### Listing 3.20: STEP-8: create a local top-N

```
1 // STEP-8: create a local top-N
2 JavaRDD<SortedMap<Integer, String>> partitions = uniqueKeys.mapPartitions(
3     new FlatMapFunction<Iterator<Tuple2<String, Integer>>, SortedMap<Integer, String>>() {
4         @Override
5         public Iterable<SortedMap<Integer, String>> call(Iterator<Tuple2<String, Integer>> iter) {
6             final int N = topN.value();
7             SortedMap<Integer, String> localTopN = new TreeMap<Integer, String>();
8             while (iter.hasNext()) {
9                 Tuple2<String, Integer> tuple = iter.next();
10                localTopN.put(tuple._2, tuple._1);
11                // keep only top N
12                if (localTopN.size() > N) {
13                    // remove element with the smallest frequency
14                    localTopN.remove(localTopN.firstKey());
15                }
16            }
17            return Collections.singletonList(localTopN);
18        }
19    });
20 partitions.saveAsTextFile("/output/4");
```

```
1
2 hadoop@hnode01319:~# hadoop fs -ls /output/4/
```

```

3 | Found 4 items
4 | -rw-r--r-- 3 hadoop hadoop      0 2014-08-31 13:11 /output/4/_SUCCESS
5 | -rw-r--r-- 3 hadoop hadoop    11 2014-08-31 13:11 /output/4/part-00000
6 | -rw-r--r-- 3 hadoop hadoop    11 2014-08-31 13:11 /output/4/part-00001
7 | -rw-r--r-- 3 hadoop hadoop    11 2014-08-31 13:11 /output/4/part-00002
8 |
9 | # hadoop fs -cat /output/4/part-00000
10| {3=E, 5=B}
11|
12| # hadoop fs -cat /output/4/part-00001
13| {2=F, 7=C}
14|
15| # hadoop fs -cat /output/4/part-00002
16| {5=A, 6=G}
17|
18| # hadoop fs -cat /output/4/part*
19| {3=E, 5=B}
20| {2=F, 7=C}
21| {5=A, 6=G}

```

### 3.9.1.10 STEP-9: find a final top-N

This step accepts all local top-10's and generates the final top-10 list.

#### **Listing 3.21:** STEP-9: find a final top-N

```

1 // STEP-9: find a final 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         // keep only top N
10        if (finalTopN.size() > N) {
11            finalTopN.remove(finalTopN.firstKey());
12        }
13    }
14 }

```

### 3.9.1.11 STEP-10: emit final top-N

This step emits the final output on the standard output device.

#### **Listing 3.22:** STEP-10: emit final top-N

```

1 // STEP-10: emit final 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 }

```

---

```

1 ---- Top-N List ---
2 -----
3 key=G  value=6
4 key=C  value=7

```

### 3.9.1.12 Shell Script To Run Spark Program

**Listing 3.23:** Shell Script To Run Spark Program

```

1 # cat ./run_top10_nonunique.sh
2 #!/bin/bash
3 source /home/hadoop/conf/env_2.5.0.sh
4 export SPARK_HOME=/home/hadoop/spark-1.0.2
5 source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
6 source $SPARK_HOME/conf/spark-env.sh
7
8 # system jars:
9 CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
10 jars='find $SPARK_HOME -name \'*.jar\''
11 for j in $jars ; do
12     CLASSPATH=$CLASSPATH:$j
13 done
14
15 # app jar:
16 export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
17 export SPARK_CLASSPATH=$CLASSPATH
18 export SPARK_MASTER=spark://myserver100:7077
19 OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER"
20 export INPUT=/top10/top10input
21 export topN=2
22 $JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS Top10NonUnique $SPARK_MASTER $INPUT $topN

```

---

### 3.9.1.13 Sample Run

Output is trimmed and edited to fit the page. For this run, the Spark cluster had 3 server nodes: {myserver100, myserver200, myserver300}.

### Listing 3.24: Sample Run

```
1 # ./run_top10_nonunique.sh
2 args[0]: <spark-master-URL>=spark://myserver100:7077
3 args[1]: <input-path>/top10/top10input
4 args[2]: <topN>=2
5 ...
6 INFO : Total input paths to process : 3
7 INFO : Starting job: saveAsTextFile at Top10NonUnique.java:73
8 INFO : Got job 0 (saveAsTextfile at Top10NonUnique.java:73) with
9   3 output partitions (allowLocal=false)
10 INFO : Final stage: Stage 0(saveAsTextFile at Top10NonUnique.java:73)
11 ...
12 INFO : Starting task 0.0:0 as TID 0 on executor 4: myserver100 (PROCESS_LOCAL)
13 INFO : Serialized task 0.0:0 as 13682 bytes in 4 ms
14 ...
15 INFO : Submitting Stage 1 (MappedRDD[5] at saveAsTextFile at Top10NonUnique.java:89),
16   which has no missing parents
17 INFO : Submitting 3 missing tasks from Stage 1 (MappedRDD[5] at saveAsTextFile at
18   Top10NonUnique.java:89)
19 INFO : Adding task set 1.0 with 3 tasks
20 INFO : Starting task 1.0:0 as TID 3 on executor 7: myserver300 (NODE_LOCAL)
21 INFO : Serialized task 1.0:0 as 14119 bytes in 1 ms
22 INFO : Starting task 1.0:2 as TID 4 on executor 5: myserver200 (NODE_LOCAL)
23 INFO : Serialized task 1.0:2 as 14119 bytes in 0 ms
24 INFO : Starting task 1.0:1 as TID 5 on executor 3: myserver100 (NODE_LOCAL)
25 INFO : Serialized task 1.0:1 as 14119 bytes in 1 ms
26 INFO : Completed ResultTask(1, 0)
27 ...
28 INFO : Submitting Stage 3 (MapPartitionsRDD[6] at reduceByKey at Top10NonUnique.java:92),
29   which has no missing parents
30 INFO : Submitting 3 missing tasks from Stage 3 (MapPartitionsRDD[6] at reduceByKey
31   at Top10NonUnique.java:92)
32 INFO : Adding task set 3.0 with 3 tasks
33 INFO : Starting task 3.0:0 as TID 6 on executor 1: myserver300 (NODE_LOCAL)
34 INFO : Serialized task 3.0:0 as 2537 bytes in 1 ms
35 INFO : Starting task 3.0:1 as TID 7 on executor 4: myserver100 (NODE_LOCAL)
36 INFO : Serialized task 3.0:1 as 2537 bytes in 1 ms
37 INFO : Starting task 3.0:2 as TID 8 on executor 8: myserver200 (NODE_LOCAL)
38 INFO : Serialized task 3.0:2 as 2537 bytes in 0 ms
39 INFO : Finished TID 7 in 109 ms on myserver100 (progress: 1/3)
40 ...
41 INFO : Completed ResultTask(4, 0)
42 INFO : Finished TID 12 in 244 ms on myserver100 (progress: 1/3)
43 INFO : Completed ResultTask(4, 2)
44 INFO : Finished TID 14 in 304 ms on myserver200 (progress: 2/3)
45 INFO : Completed ResultTask(4, 1)
46 INFO : Finished TID 13 in 362 ms on myserver100 (progress: 3/3)
47 ...
48 INFO : Adding task set 6.0 with 3 tasks
49 ...
50 INFO : Completed ResultTask(6, 1)
51 INFO : Finished TID 16 in 60 ms on myserver300 (progress: 3/3)
52 INFO : Removed TaskSet 6.0, whose tasks have all completed, from pool
53 INFO : Stage 6 (collect at Top10NonUnique.java:121) finished in 0.062 s
54 INFO : Job finished: collect at Top10NonUnique.java:121, took 0.076470196 s
55 --- Top-N List ---
56 -----
57 key=G  value=6
58 key=C  value=7
```

# Left Outer Join in MapReduce

## 4.1 Introduction

The purpose of the is chapter is to show how to implement a **Left Outer Join** in MapReduce environment. To solve **Left Outer Join**, we provide three distinct implementations in MapReduce/Hadoop and Spark/Hadoop:

- MapReduce/Hadoop solution using `map()` and `reduce()` functions
- Spark/Hadoop solution without using built-in `JavaPairRDD.leftOuterJoin()`
- Spark/Hadoop solution using built-in `JavaPairRDD.leftOuterJoin()`

Consider a company such as Amazon, which has over 200 millions of users and possibly can do hundreds of millions of transactions per day. To show the concept of **Left Outer Join**, assume we have two types of data: users and transactions: where users' data has user's "location" (say `location_id`) information and transactions has "user" (say `user_id`) information, but transactions do not have direct information about user's locations. Given users and transactions:

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

the goal is to find the number of unique locations in which each product has been sold.

What is a **Left Outer Join**? Let  $T_1$  (be a left table) and  $T_2$  (be a right table) be two relations defined as (where  $t_1$  are attributes of  $T_1$  and  $t_2$  are attributes of  $T_2$ ):

$$\begin{aligned} T_1 &= (K, t_1) \\ T_2 &= (K, t_2) \end{aligned}$$

Then the result of a left outer join for relations  $T_1$  and  $T_2$  on join key of  $K$  contains all records of the "left" table ( $T_1$ ), even if the join-condition does not find any matching record in the "right" table ( $T_2$ ). If the ON clause with key  $K$  matches 0 (zero) records in  $T_2$  (for a given record in  $T_1$ ), the join will still return a row in the result (for that record)— but with NULL in each column from  $T_2$ . A left outer join returns all the values from an inner join plus all values in the left table that do not match to the right table. Formally, we can express this as:

$$\begin{aligned} \text{LeftOuterJoin}(T_1, T_2, K) = & \{(k, t_1, t_2) \text{ where } k \in T_1.K \text{ and } k \in T_2.K\} \\ & \cup \\ & \{(k, t_1, \text{null}) \text{ where } k \in T_1.K \text{ and } k \notin T_2.K\} \end{aligned}$$

In SQL, we can express **Left Outer Join** as (where  $K$  is the column where  $T_1$  and  $T_2$  are joined on):

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

**Left Outer Join** can be visually expressed as (the colored part is included and the part with white color is excluded):

Consider the following values for our users and transactions (note that these values are just examples to demonstrate the concept of **Left Outer Join** in MapReduce environment):

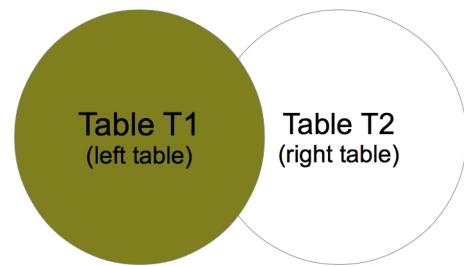


Figure 4.1: Left Out Join Illustration

Users Table	
user_id	location_id
u1	UT
u2	GA
u3	CA
u4	CA
u5	GA

Transactions Table				
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

Here are some SQL queries to answer our questions:

- find all products (and associated location) sold [Query-1]

```
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)
```

- find all products (and associated location counts) sold [Query-2]

```
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)
```

- find all products (and unique location counts) sold [Query-3]

```
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)
```

## 4.2 Implementation of Left Outer Join in MapReduce

The desired output we are looking is provided by SQL Query-3, which finds all number of distinct (unique) locations in which each product has been sold for given all transactions. We present solution for `Left Outer Join` in two steps:

- MapReduce Phase-1: find all products (and associated locations) sold. The answer to Phase-1 is using SQL Query-1.
- MapReduce Phase-2: find all products (and associated unique location counts) sold. The answer to Phase-2 is using SQL Query-3.

### 4.2.1 MapReduce Phase-1

This phase will perform "left outer join" operation with a MapReduce job, which will utilize two mappers (one for users and the other one for transactions) and reducer will emit `(Key,Value)` with `Key = product_id`, and `Value = location_id`. Using multiple mappers is enabled by the `MultipleInputs`<sup>1</sup> class (note that if we had a single mapper, then we would have used `Job.setMapperClass()`):

```
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);
```

---

<sup>1</sup> `org.apache.hadoop.mapreduce.lib.input.MultipleInputs`

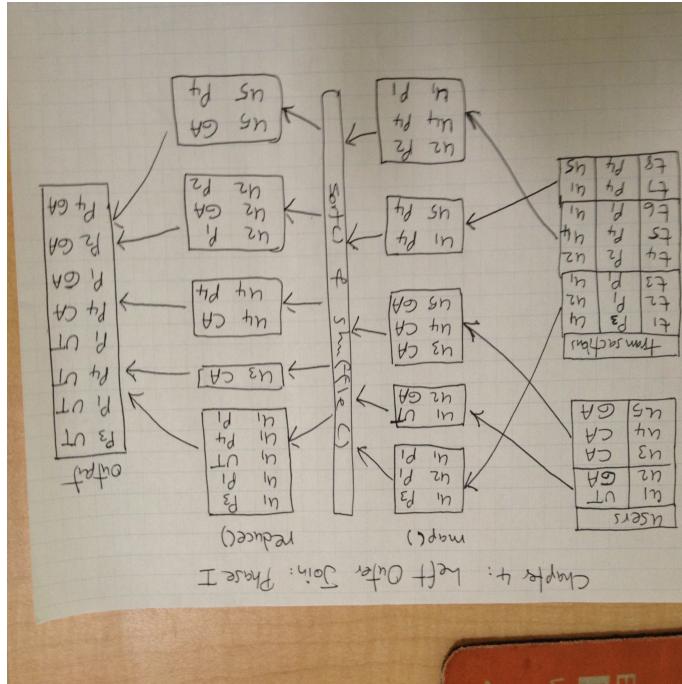


Figure 4.2: Left Outer Join Data Flow – Phase I

The following Figures illustrates the working MapReduce flow of Left Outer Join algorithm, which consists of two MapReduce jobs (labeled as Phase I and Phase II).

**Transaction Mapper [VERSION-1]** : the transaction map() reads (`transaction_id`, `product_id`, `user_id`, `quantity`, `amount`) and emits (`key=user_id`, `value=product_id`).

**User Mapper [VERSION-1]** : the user map() reads (`user_id`, `location_id`) and emits (`key=user_id`, `value=location_id`).

The reducer for Phase-1 [VERSION-1] , gets both user's `location_id` and `product_id` and emits (`key=product_id`, `value=location_id`). Now, the question is how the reducer will identify `location_id` from `product_id`? In Hadoop, the order of reducer values is undefined. Therefore, the reducer

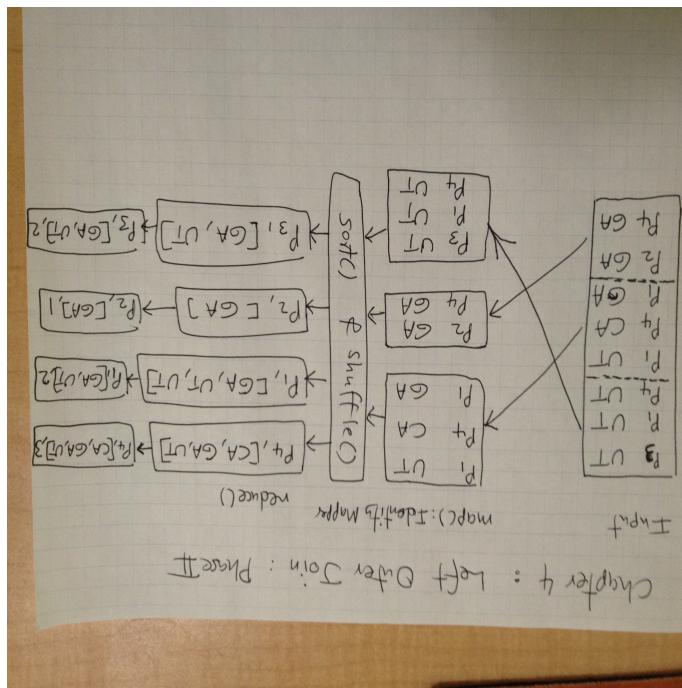


Figure 4.3: Left Outer Join Data Flow – Phase II

for a specific Key=`user_id` has no clue how to process the values. To remedy this problem we modify Transaction and User mappers/reducers (call it VERSION-2):

**Transaction Mapper [VERSION-2]**: the transaction map() reads (`transaction_id, product_id, user_id, quantity, amount`) and emits (key=Pair(`user_id, 2`), value=Pair("P", `product_id`)). By adding a number "2" to the reducer key, we will guarantee that `product_id`(s) arrive at the end. This will be accomplished by "secondary sorting" technique. We added "P" to the value to identify products. In Hadoop, for implementation of Pair(String, String), we will use the `PairOfStrings`<sup>2</sup> class.

#### 4.2.1.1 Transaction Mapper [VERSION-2]

**Listing 4.1:** Transaction Mapper [VERSION-2]

```

1 /**
2  * @param key is framework generated, ignored here
3  * @param value is the transaction_id<TAB>product_id<TAB>user_id<TAB>quantity<TAB>amount
4 */
5 map(key, value) {
6     String[] tokens = StringUtil.split(value, "\t");
7     String productID = tokens[1];
8     String userID = tokens[2];
9     outputKey = Pair(userID, 2);
10    outputValue = Pair("P", productID);
11    emit(outputKey, outputValue);
12 }
```

**User Mapper [VERSION-2]**: the user map() reads (`user_id, location_id`) and emits (key=Pair(`user_id, 1`), value=Pair("L", `location_id`)). By adding a number "1" to the reducer key, we will guarantee that `location_id`(s) arrive first. This will be accomplished by "secondary sorting" technique. We added "L" to the value to identify locations.

---

<sup>2</sup> `edu.umd.cloud9.io.pair.PairOfStrings` (which implements `WritableComparable<PairOfStrings>`)

#### 4.2.1.2 User Mapper [VERSION-2]

**Listing 4.2:** User Mapper [VERSION-2]

```
1 /**
2  * @param key is framework generated, ignored here
3  * @param value is the user_id<TAB>location_id
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); // make sure location shows before products
10    outputValue = Pair("L", locationID);
11    emit(outputKey, outputValue);
12 }
```

The reducer for Phase-1 [VERSION-2], gets both Pair("L", location\_id) and Pair("P", product\_id) and emits (key=product\_id, value=location\_id). Note that since  $1 < 2$ , it means that user's location\_id arrives first.

#### 4.2.1.3 The reducer for Phase-1 [VERSION-2]

**Listing 4.3:** The reducer for Phase-1 [VERSION-2]

```
1 /**
2  * @param key is user_id
3  * @param values is List<Pair<left, right>>, where
4  * values = List<{Pair<"L", locationID>, Pair<"P", productID1>, Pair<"P", productID2>, ...}>
5  * NOTE that, the Pair<"L", locationID> arrives before all product pairs.
6  * The first value is location, if it's not, then we don't have a user record,
7  * so we'll set the locationID as "undefined"
8 */
9 reduce(key, values) {
10    locationID = "undefined";
11    for (Pair<left, right> value: values) {
12        // the following if-stmt will be true
13        // once at the first iteration
14        if (value.left.equals("L")) {
15            locationID = value.right;
```

```

16         continue;
17     }
18
19     // here we have a product: value.left.equals("P")
20     productID = value.right;
21     emit(productID, locationID);
22 }
23 }
```

---

## 4.2.2 MapReduce Phase-2: Counting Unique Locations

This phase will use output of Phase-1 (which is a sequence of pairs of (product\_id, location\_id) and generates pairs of (product\_id, number\_of\_unique\_locations). The mapper for this phase is an identity mapper and the reducer will count the number of unique locations (by using a **Set** data structure) per product.

### 4.2.2.1 Mapper Phase-2: Counting Unique Locations

#### **Listing 4.4:** Mapper Phase-2: Counting Unique Locations

```

1 /**
2  * @param key is product_id
3  * @param value is location_id
4 */
5 map(key, value) {
6     emit(key, value);
7 }
```

---

### 4.2.2.2 Reducer Phase-2: Counting Unique Locations

#### **Listing 4.5:** Reducer Phase-2: Counting Unique Locations

```

1 /**
2  * @param key is product_id
3  * @param values is List<location_id>
4 */
5 reduce(key, values) {
```

```

6   Set<String> set = new HashSet<String>();
7   for (String locationID : values) {
8     set.add(locationID);
9   }
10
11  int uniqueLocationsCount = set.size();
12  emit(key, uniqueLocationsCount);
13 }

```

---

### 4.2.3 Implementation Classes in Hadoop

Implementation Classes in Hadoop		
Phase	Class Name	Class Description
Phase-1	LeftJoinDriver	Dirver to submit job for Phase-1
	LeftJoinReducer	Left Join Reducer
	LeftJoinTransactionMapper	Left Join transaction mapper
	LeftJoinUserMapper	Left Join user mapper
	SecondarySortPartitioner	How to partition natural keys
	SecondarySortGroupComparator	How to group by natural key
Phase-2	LocationCountDriver	Dirver to submit job for Phase-2
	LocationCountMapper	Define map() for location count
	LocationCountReducer	Define reduce() for location count

## 4.3 Sample Run

### 4.3.1 Input for Phase-1

```

# 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

```

### 4.3.2 run Phase-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 21:17:55 INFO mapred.JobClient: map 100% reduce 0%
13/12/29 21:18:03 INFO mapred.JobClient: map 100% reduce 6%
13/12/29 21:18:04 INFO mapred.JobClient: map 100% reduce 13%
...
13/12/29 21:18:39 INFO mapred.JobClient: map 100% reduce 86%
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 output materialized bytes=328
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

```

### 4.3.3 View Output of Phase-1 (Input of Phase-2)

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

### 4.3.4 Run Phase-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 21:19:36 INFO mapred.JobClient: map 50% reduce 0%
...
13/12/29 21:20:22 INFO mapred.JobClient: map 100% reduce 86%
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 output materialized bytes=664
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
```

#### 4.3.5 View Output of Phase-2

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

### 4.4 Spark Implementation

Since Spark provides a higher-level Java API than MapReduce/Hadoop API, we will present the whole solution in a single Java class (called `LeftOuterJoin`), which will include a series of `map()`, `groupBy()`, and `reduce()` functions. In MapReduce/Hadoop implementation we used `MultipleInputs` class to process two different types of input by two different mappers. Spark provides a much richer API for mappers and reducers. Without a special plug-in classes, you may have many different types of mappers (by using `map()`, `flatMap()`, and `flatMapToPair()` functions). In Spark, instead of using Hadoop's `MultipleInputs` class, we will use `JavaRDD.union()` function to return the union of two `JavaRDDs` (users RDD and transactions RDD), which will be merged to create a new RDD. The `JavaRDD.union()` function is defined as:

```
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).

You can only apply the `union()` function to `JavaRDD`'s of the same type (`T`). Therefore, we will create the same RDD type for users and transactions. This is how we do it:

### Left Outer Join RDD Flow

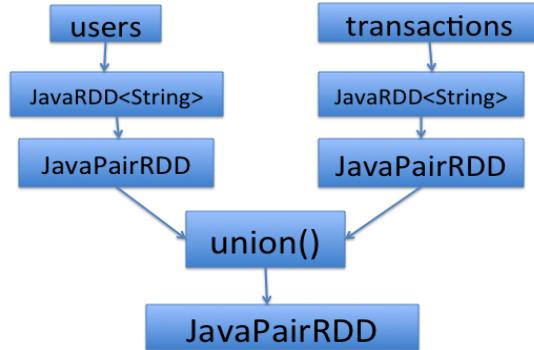


Figure 4.4: Union Data Flow

```
JavaPairRDD<String, Tuple2<String, String>> usersRDD = users.map(...);  
JavaPairRDD<String, Tuple2<String, String>> transactionsRDD = transactions.map(...)  
  
// here we perform a union() on usersRDD and transactionsRDD  
JavaPairRDD<String, Tuple2<String, String>> allRDD = transactionsRDD.union(usersRDD)
```

The `union()` workflow is expressed in the following Figure.

To refresh our memory, here are the data for users and transactions (these data will be used as input files for our sample run at the end of this chapter):

```
# 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
```

This is how the algorithm works: for users and transactions data we generate (here T2 refers to Tuple2):

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

Next, we create a union of these data:

```
all = transactions.union(users);
= { (userID1, T2("L", location)),
  (userID1, T2("P", P11)),
  (userID1, T2("P", P12)),
  ...
  (userID1, T2("P", P1n)),
  ...
}
```

where Pi is a productID

The next step is to group data by userID. This will generate:

```
{
  (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), ...>),
  ...
}
```

```

    }

where
Li is a locationID,
Pij is a productID.
```

#### 4.4.1 Spark Program

First, I will provide a high-level solution in 10 steps, and then we will dissect into each step with a proper working Spark code.

**Listing 4.6:** LeftOuterJoin High-Level Solution

```

1 // STEP-0: import required classes and interfaces
2 public class LeftOuterJoin {
3     public static void main(String[] args) throws Exception {
4         // STEP-1: Read Input Parameters
5         // STEP-2: Create JavaSparkContext object
6         // STEP-3: Create a JavaPairRDD for users
7         // STEP-4: Create a JavaPairRDD for transactions
8         // STEP-5: Create a union of RDD's created by STEP-3 and STEP-4
9         // STEP-6: Create a JavaPairRDD(userID, List<T2>) by calling groupBy()
10        // STEP-7: Create a productLocationsRDD as JavaPairRDD<String, String>
11        // STEP-8: Find all locations for a product,
12        //         result will be JavaPairRDD<String, List<String>>
13        // STEP-9: Finalize output by changing "value" from List<String>
14        //         to Tuple2<Set<String>, Integer>, where you have unique
15        //         set of locations and their count.
16        // STEP-10: Print the final result RDD
17        System.exit(0);
18    }
19 }
```

#### 4.4.2 STEP-0: Import Required Classes

We import required classes and interfaces from the JAR files provided by the binary distributions of Spark framework. Spark provides two Java packages (`org.apache.spark.api.java` and `org.apache.spark.api.java.function`) for creating and manipulating RDDs.

**Listing 4.7:** STEP-0: import required classes and interfaces

```

1 // STEP-0: import required classes and interfaces
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;

```

---

#### 4.4.3 STEP-1: Read Input Parameters

We read 3 input parameters: "spark master URL", users data and transactions data. Users and transaction data are provided as HDFS text files.

**Listing 4.8:** STEP-1: Read Input Parameters

```

1 if (args.length < 3) {
2     System.err.println("Usage: LeftOuterJoin <master> <users> <transactions>");
3     System.exit(1);
4 }
5
6 String sparkMaster = args[0]; // spark master URL
7 String usersInputFile = args[1]; // HDFS text file
8 String transactionsInputFile = args[2]; // HDFS text file
9 System.out.println("sparkMaster="+ sparkMaster);
10 System.out.println("users="+ usersInputFile);
11 System.out.println("transactions="+ transactionsInputFile);

```

---

The output of this step will be (myserver100 is the spark master server):

```

sparkMaster=spark://myserver100:7077
users=/data/leftouterjoins/users.txt
transactions=/data/leftouterjoins/transactions.txt

```

#### 4.4.4 STEP-2: Create JavaSparkContext Object

A JavaSparkContext object is created by using spark master URL. This object is used to create first RDD.

**Listing 4.9:** STEP-2: Create JavaSparkContext Object

```
1  JavaSparkContext ctx = new JavaSparkContext(
2      sparkMaster,
3      "MyJavaWordCount",
4      System.getenv("SPARK_HOME"),
5      System.getenv("SPARK_EXAMPLES_JAR"));
```

#### 4.4.5 STEP-3: Create a JavaPairRDD for Users

First, we create a users JavaRDD<String>, where RDD element is a single record of text file (representing userID and locationID). Next, we use JavaRDD<String>.mapToPair() function to create a new JavaPairRDD<String, Tuple2<String, S>>, where key is a userID and value is a Tuple2("L", location). Later on we will create Tuple2("P", product) pairs for transactions data. The tags "L" and "P" identify locations and products.

**Listing 4.10:** STEP-3: Create a JavaPairRDD for Users

```
1  JavaRDD<String> users = ctx.textFile(usersInputFile, 1);
2
3  // mapToPair
4  // <K2, V2> JavaPairRDD<K2, V2> mapToPair(PairFunction<T, K2, V2> f)
5  // Return a new RDD by applying a function to all elements of this RDD.
6  // PairFunction<T, K, V>
7  // T => Tuple2<K, V>
8  JavaPairRDD<String, Tuple2<String, String>> usersRDD =
9      //           T   K     V
10     users.mapToPair(new PairFunction<String, String, Tuple2<String, String>>() {
11         public Tuple2<String, Tuple2<String, String>> call(String s) {
12             String[] userRecord = s.split("\t");
13             Tuple2<String, String> location = new Tuple2<String, String>("L", userRecord[1]);
14             return new Tuple2<String, Tuple2<String, String>>(userRecord[0], location);
15         }
16     });

```

#### 4.4.6 STEP-4: Create a JavaPairRDD for Transactions

First, we create a transactions `JavaRDD<String>`, where RDD element is a single record of text file (representing transaction record). Next, we use `JavaRDD<String>.mapToPair()` function to create a new `JavaPairRDD<String,Tuple2<String, String>`, where key is a userID and value is a `Tuple2("P", product)`. In previous step, we created `Tuple2("L", location)` pairs for users. The tags "L" and "P" identify locations and products.

**Listing 4.11:** STEP-4: Create a JavaPairRDD for Transactions

```
1  JavaRDD<String> transactions = ctx.textFile(transactionsInputFile, 1);
2
3  // mapToPair
4  // <K2, V2> JavaPairRDD<K2, V2> mapToPair(PairFunction<T, K2, V2> f)
5  // Return a new RDD by applying a function to all elements of this RDD.
6  // PairFunction<T, K, V>
7  // T => Tuple2<K, V>
8  JavaPairRDD<String, Tuple2<String, String>> transactionsRDD =
9      //          T      K      V
10     transactions.mapToPair(new PairFunction<String, String, Tuple2<String, String>>() {
11         public Tuple2<String, Tuple2<String, String>> call(String s) {
12             String[] transactionRecord = s.split("\t");
13             Tuple2<String, String> product = new Tuple2<String, String>("P", transactionRecord[1]);
14             return new Tuple2<String, Tuple2<String, String>>(transactionRecord[2], product);
15         }
16     });

```

#### 4.4.7 STEP-5: Create a union of RDD's created by STEP-3 and STEP-4

This step creates union of two `JavaPairRDD<String, Tuple2<String, String>>`s. the `JavaPairRDD.union()` requires that both RDD's to have the same exact types.

**Listing 4.12:** STEP-5: Create a union of RDD's

```
1  // here we perform a union() on usersRDD and transactionsRDD
2  JavaPairRDD<String, Tuple2<String, String>> allRDD = transactionsRDD.union(usersRDD);
```

This step could have been implemented as (semantically equivalent – by changing the order of union parameters):

#### **Listing 4.13:** STEP-5: Create a union of RDD's

```
1 // here we perform a union() on usersRDD and transactionsRDD
2 JavaPairRDD<String, Tuple2<String, String>> allRDD = usersRDD.union(transactionsRDD);
```

So the result of union of two JavaPairRDDs will be the following (key, value) pairs:

```
{
  (userID, T2("L", location)),
  ...
  (userID, T2("P", product))
  ...
}
```

#### **4.4.8 STEP-6: Create a JavaPairRDD(userID, List(T2)) by calling groupBy()**

Next, we group data (created in STEP-5) by userID. This step is accomplished by JavaPairRDD.groupByKey().

#### **Listing 4.14:** STEP-6: Create a JavaPairRDD

```
1 // group allRDD by userID
2 JavaPairRDD<String, Iterable<Tuple2<String, String>>>
3   groupedRDD = allRDD.groupByKey();
4 // now the groupedRDD entries will be as:
5 // <userIDi, List[T2("L", location),
6 //                 T2("P", Pi1),
7 //                 T2("P", Pi2),
8 //                 T2("P", Pi3), ...
9 //                 ]
10 // >
```

The result of this step will be:

```
(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), ...]),  
...  
...
```

where  $P_{ij}$  is a productID.

#### 4.4.9 STEP-7: Create a productLocationsRDD as Java-PairRDD<String, String>

In this step, the userIDs are dropped from RDDs. For a given RDD element:

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

we create JavaPairRDD<String, String> as:

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

This step is accomplished by `JavaPairRDD.flatMapToPair()` function, which we implement a `PairFlatMapFunction.call()` method. The `PairFlatMapFunction` works as:

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

where in our example: T is an input and

we create (K, V) pairs as output:

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

Here is the complete implementation of `PairFlatMapFunction.call()` method:

#### Listing 4.15: STEP-7: Create a productLocationsRDD

```
1 // PairFlatMapFunction<T, K, V>
2 // T => Iterable<Tuple2<K, V>>
3 JavaPairRDD<String, String> productLocationsRDD =
4     groupedRDD.flatMap(new PairFlatMapFunction<
5         Tuple2<String, Iterable<Tuple2<String, String>>>, // T
6         String, // K
7         String>() {
8     public Iterable<Tuple2<String, String>>
9         call(Tuple2<String, Iterable<Tuple2<String, String>>> s) {
10        // String userID = s._1; // NOT Needed
11        Iterable<Tuple2<String, String>> pairs = s._2;
12        String location = "UNKNOWN";
13        List<String> products = new ArrayList<String>();
14        for (Tuple2<String, String> t2 : pairs) {
15            if (t2._1.equals("L")) {
16                location = t2._2;
17            }
18            else {
19                // t2._1.equals("P")
20                products.add(t2._2);
21            }
22        }
23
24        // now emit (K, V) pairs
25        List<Tuple2<String, String>> kvList =
26            new ArrayList<Tuple2<String, String>>();
27        for (String product : products) {
28            kvList.add(new Tuple2<String, String>(product, location));
29        }
30        // Note that edges must be reciprocal, that
31        // is every {source, destination} edge must have
32        // a corresponding {destination, source}.
33        return kvList;
34    }
35});
```

#### 4.4.10 STEP-8: Find all locations for a product

This step groups RDD pairs of (product, location) by grouping of products. We use `JavaPairRDD.groupByKey()` to accomplish this step. This step does some basic debugging too by calling `JavaPairRDD.collect()` function.

**Listing 4.16:** STEP-8: Find all locations for a product

```
1 // Find all locations for a product
2 JavaPairRDD<String, Iterable<String>> productByLocations =
3     productLocationsRDD.groupByKey();
4
5 // debug3
6 List<Tuple2<String, List<String>>> debug3 = productByLocations.collect();
7 System.out.println("--- debug3 begin ---");
8 for (Tuple2<String, Iterable<String>> t2 : debug3) {
9     System.out.println("debug3 t2._1="+t2._1);
10    System.out.println("debug3 t2._2="+t2._2);
11 }
12 System.out.println("--- debug3 end ---");
```

#### 4.4.11 STEP-9: Finalize output by changing "value"

STEP-8 produced a `JavaPairRDD<String, List<String>>` object, where key is product (as a String) and value is a `List<String>`, which a list of locations (but might have duplicates). To remove duplicate elements from a value, we use a `JavaPairRDD.mapValues()` function. We implement this function by converting a `List<String>` to a `Set<String>`. Note that the keys are not altered. Mapping values are implemented by a `Function(T, R).call()`, where T is an input (as `List<String>`) and R is an output (as `Tuple2<Set<String>, Integer>`).

**Listing 4.17:** STEP-9: Finalize output

```
1 JavaPairRDD<String, Tuple2<Set<String>, Integer>> productByUniqueLocations =
2     productByLocations.mapValues(
3         new Function< Iterable<String>,      // input
4             Tuple2<Set<String>, Integer>        // output
5         >() {
6             public Tuple2<Set<String>, Integer> call(Iterable<String> s) {
7                 Set<String> uniqueLocations = new HashSet<String>();
8                 for (String location : s) {
9                     uniqueLocations.add(location);
10                }
11                return new Tuple2<Set<String>, Integer>(uniqueLocations,
12                                              uniqueLocations.size());
13            }
14        });
15 
```

---

#### 4.4.12 STEP-10: Print the final result RDD

The final step emits the results by using `JavaPairRDD.collect()` method.

**Listing 4.18:** STEP-10: Print the final result RDD

```
1 // debug4
2 System.out.println("== Unique Locations and Counts ==");
3 List<Tuple2<String, Tuple2<Set<String>, Integer>>> debug4 =
4     productByUniqueLocations.collect();
5 System.out.println("--- debug4 begin ---");
6 for (Tuple2<String, Tuple2<Set<String>, Integer>> t2 : debug4) {
7     System.out.println("debug4 t2._1="+t2._1);
8     System.out.println("debug4 t2._2="+t2._2);
9 }
10 System.out.println("--- debug4 end ---");
```

---

#### 4.4.13 Running Spark Solution

##### 4.4.13.1 The Shell Script

```
# cat run_left_outer_join.sh
#!/bin/bash
source /home/hadoop/conf/env_2.3.0.sh
export SPARK_HOME=/home/hadoop/spark-1.0.0
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
source $SPARK_HOME/conf/spark-env.sh

# system jars:
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop

jars='find $SPARK_HOME -name \'*.jar\''
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
```

```

export SPARK_CLASSPATH=$CLASSPATH
export SPARK_MASTER=spark://hnnode01319.nextbiosystem.net:7077
#-Dsun.lang.ClassLoader.allowArraySyntax=true
USERS=/data/leftouterjoins/users.txt
TRANSACTIONS=/data/leftouterjoins/transactions.txt
OPTIONS="--Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER"
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS LeftOuterJoin $SPARK_MASTER $USERS $T

```

#### 4.4.13.2 Running The Shell Script

Log output is trimmed to fit the page.

```

# ./run_left_outer_join.sh
sparkMaster=spark://hnnode01319.nextbiosystem.net:7077
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

```

## 4.5 Running Spark on YARN

In this example, we show how to submit Spark's ApplicationMaster to Hadoop YARN's ResourceManager, and instruct Spark to run the `LeftOuterJoin` program. Further, we can instruct our Spark program to save our final result into an HDFS file. Saving file to HDFS is accomplished by adding the following line after creating `productByUniqueLocations` RDD. Below, `/left/output` is an HDFS output directory.

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

### 4.5.1 Script to Run Spark on YARN

The script to run Spark on YARN is given below:

```

# cat leftjoin.sh
export SPARK_HOME=/usr/local/spark
export SPARK_ASSEMBLY_JAR=$SPARK_HOME/assembly/target/scala-2.10/spark-assembly-1.0.0-SNAPSHOT-jar-with-dependencies.jar
export SPARK_LIBRARY_PATH=/usr/local/hadoop/lib/native
export JAVA_HOME=/usr/java/jdk6
export HADOOP_CONF_DIR=/usr/local/hadoop/conf
export YARN_CONF_DIR=/usr/local/hadoop/conf

# Submit Spark's ApplicationMaster to YARN's ResourceManager,
# and instruct Spark to run the LeftOuterJoin2 example
SPARK_JAR=$SPARK_ASSEMBLY_JAR \
    $SPARK_HOME/bin/spark-class org.apache.spark.deploy.yarn.Client \
    --jar $SPARK_HOME/tmp/mp.jar \
    --class LeftOuterJoin \
    --args yarn-standalone \
    --args /left/users.txt \
    --args /left/transactions.txt \
    --num-workers 3 \
    --master-memory 4g \
    --worker-memory 2g \
    --worker-cores 1

```

For details on Spark parameters (such as `num-workers` and `worker-memory`) and environment variables, you may refer to Spark Summit Slides<sup>3</sup>. Most of these parameters depend on the number of worker nodes, number of cores per cluster node, and amount of RAM available. Optimal setting of these parameters will require some trial and error and by trying different size of input data.

### 4.5.2 Running Script

Output is formatted to fit the page.

---

<sup>3</sup><http://spark-summit.org/wp-content/uploads/2013/10/Spark-Ops-Final.pptx>

```

# ./nleftjoin.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 (A
    number of NodeManagers: 13
14/05/28 16:49:31 INFO Client: Queue info ... queueName: default, queueCurrentCapa
    0.0, queueMaxCapacity: 1.0,
        queueApplicationCount = 0, queueChildQueueCount = 0
14/05/28 16:49:31 INFO Client: Max mem capability of a single resource in this cl
14/05/28 16:49:31 INFO Client: Preparing Local resources
14/05/28 16:49:31 INFO Client: Uploading file:/usr/local/spark/tmp/mp.jar to
    hdfs://myserver100:9000/user/hadoop/.sparkStaging/application_1401319796895_0008
14/05/28 16:49:32 INFO Client: Uploading file:
    /usr/local/spark/tmp/assembly/target/scala-2.10/spark-assembly-1.0.0-hadoop2.3.0
        to hdfs://myserver100:9000/user/hadoop/.sparkStaging/
            application_1401319796895_0008/spark-assembly-1.0.0-hadoop2.3.0.jar
14/05/28 16:49:33 INFO Client: Setting up the launch environment
14/05/28 16:49:33 INFO Client: Setting up container launch context
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/mp.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
14/05/28 16:49:33 INFO YarnClientImpl: Submitted application application_140131979
14/05/28 16:49:34 INFO Client: Application report from ASM:
    application identifier: application_1401319796895_0008
    appId: 8
    clientToAMToken: null
    appDiagnostics:
    appMasterHost: N/A
    appQueue: default
    appMasterRpcPort: -1
    appStartTime: 1401320973326
    yarnAppState: ACCEPTED
    distributedFinalState: UNDEFINED
    appTrackingUrl: http://myserver100:50030/proxy/application_1401319796895_0008/

```

```

appUser: hadoop
...
14/05/28 16:49:50 INFO Client: Application report from ASM:
  application identifier: application_1401319796895_0008
  appId: 8
  clientToAMToken: null
  appDiagnostics:
  appMasterHost: myserver400
  appQueue: default
  appMasterRpcPort: 0
  appStartTime: 1401320973326
  yarnAppState: FINISHED
  distributedFinalState: SUCCEEDED
  appTrackingUrl: http://myserver100:50030/proxy/application_1401319796895_0008/A
  appUser: hadoop

```

#### 4.5.3 Checking Expected Output

```

# 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))

```

## 4.6 Left Outer Join by Spark's leftOuterJoin()

This section solves the left outer join by using Spark's built in `JavaPairRDD.leftOuterJoin()` method (note that MapReduce/Hadoop does not offer higher level API such

as `leftOuterJoin()` method):

```
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)
    // Perform a left outer join of this and other. For each
    // element (k, v) in this, the resulting RDD will either
    // contain all pairs (k, (v, Some(w))) for w in other, or
    // the pair (k, (v, None)) if no elements in other have key k.
```

Using Sparks's `JavaPairRDD.leftOuterJoin()` method enable us

1. To avoid the costly `JavaPairRDD.union()` operation between `users` and `transactions`
2. To avoid introducing custom flags such as "L" for location and "P" for products.
3. To avoid extra RDD transformations to separate custom flags from each other

Using `JavaPairRDD.leftOuterJoin()` method enable us to produce the result efficiently. The `transactionsRDD` is the left table and `usersRDD` is the right table.

```
JavaPairRDD<String,String> usersRDD = ...;           // (K=userID, V=location)
JavaPairRDD<String,String> transactionsRDD = ...; // (K=userID, V=product)
// perform left outer join by built-in leftOuterJoin()
JavaPairRDD<String, Tuple2<String,Optional<String>>> joined =
    transactionsRDD.leftOuterJoin(usersRDD);
```

Now, the `joined` RDD contains:

```
(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)))
```

Since, we are only interested in the products and unique locations, in the next step, we ignore user ID's (the key). This can be accomplished by another `JavaPairRDD.mapToPair()` function. After ignoring user ID's, we generate:

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

which has the desired information to generate products and unique locations.

First, we present a high-level steps to show how to use Spark's built-in `JavaPairRDD.leftOuterJoin()` method. Next, each step will be discussed in detail.

#### 4.6.1 High-Level Steps

##### **Listing 4.19:** High-Level Steps

```
1 // STEP-0: import required classes and interfaces
2 public class SparkLeftOuterJoin {
3     public static void main(String[] args) throws Exception {
```

```

4 // STEP-1: read input parameters
5 // STEP-2: create Spark's context object
6 // STEP-3: create RDD for user's data
7 // STEP-4: create (K=userID, V=location) pairs for users (the "right" table)
8 // STEP-5: create RDD for transaction's data
9 // STEP-6: create (K=userID, V=product) pairs for transactions (the "left" table)
10 // STEP-7: use Spark's built-in JavaPairRDD.leftOuterJoin() method
11 // STEP-8: create (product, location) pairs
12 // STEP-9: group (K=product, V=location) pairs by K
13 // STEP-10: create final output (K=product, V=Set<location>) pairs by K
14 System.exit(0);
15 }
16 }

```

---

## 4.6.2 STEP-0: import required classes and interfaces

An `Optional`<sup>4</sup> represents immutable object that may contain a non-null reference to another object (useful for left outer join, since the joined values may contain `null` if the key is in left table but not in the right table). `SparkConf` object is used to define Spark's configurations and then we use it to create an instance of `JavaSparkContext`, which will be used to create RDDs.

**Listing 4.20:** STEP-0: import required classes and interfaces

```

1 // STEP-0: import required classes and interfaces
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 com.google.common.base.Optional;
9 import org.apache.spark.SparkConf;
10 import java.util.Set;
11 import java.util.HashSet;

```

---

## 4.6.3 STEP-1: read input parameters

This step reads the location of HDFS input files: users and transactions. These input files will be used to create left table (`transactionsRDD`) and right table (`usersRDD`).

---

<sup>4</sup>`com.google.common.base.Optional`<T> is an abstract class. For details, see <https://code.google.com/p/guava-libraries/>

#### **Listing 4.21: STEP-1: read input parameters**

```
1 // STEP-1: read input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: LeftOuterJoin <users> <transactions>");
4     System.exit(1);
5 }
6
7 String usersInputFile = args[0];
8 String transactionsInputFile = args[1];
9 System.out.println("users="+ usersInputFile);
10 System.out.println("transactions="+ transactionsInputFile);
```

#### **4.6.4 STEP-2: create Spark's context object**

This step creates an instance of JavaSparkContext object, which will be used to create new RDDs. Note that I have hard-coded some host names (such as value for key `yarn.resourcemanager.hostname`), but for your production deployments, you should read these from a configuration object (such as an XML file).

#### **Listing 4.22: STEP-2: create Spark's context object**

```
1 // STEP-2: create Spark's context object
2 JavaSparkContext ctx = createJavaSparkContext();
3
4 static JavaSparkContext createJavaSparkContext() throws Exception {
5     SparkConf conf = new SparkConf();
6     conf.set("yarn.resourcemanager.hostname", "server100");
7     conf.set("yarn.resourcemanager.scheduler.address", "server100:8030");
8     conf.set("yarn.resourcemanager.resource-tracker.address", "server100:8031");
9     conf.set("yarn.resourcemanager.address", "server100:8032");
10    conf.set("mapreduce.framework.name", "yarn");
11    conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
12    conf.set("spark.executor.memory", "7g");
13    JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "SparkLeftOuterJoin", conf);
14    return ctx;
15 }
```

#### **4.6.5 STEP-3: create RDD for user's data**

This step creates `usersRDD`, which is a set of (`userID`, `location`) pairs. The `usersRDD` represents the "right" table for the left outer join operation.

#### **Listing 4.23: STEP-3: create RDD for user's data**

```
1 // STEP-3: create RDD for user's data
2 JavaRDD<String> users = ctx.textFile(usersInputFile, 1);
```

#### **4.6.6 STEP-4: Create usersRDD: The "right" Table**

This step creates the right table represented as usersRDD, which contains (K=userID, V=location) pairs from users input data.

#### **Listing 4.24: STEP-4: create (K=userID,V=location) pairs for users**

```
1 // STEP-4: create (K=userID,V=location) pairs for users
2 // <K2, V2> JavaPairRDD<K2, V2> mapToPair(PairFunction<T, K2, V2> f)
3 // Return a new RDD by applying a function to all elements of this 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   });
```

#### **4.6.7 STEP-5: create transactionRDD for transaction's data**

This step creates transactionRDD, which is a set of (userID, product) pairs. The transactionRDD represents the "left" table for the left outer join operation.

#### **Listing 4.25: STEP-5: create transactionRDD for transaction's data**

```
1 // STEP-5: create RDD for transaction's data
2 JavaRDD<String> transactions = ctx.textFile(transactionsInputFile, 1);
```

#### 4.6.8 STEP-6: Create transactionsRDD: The Left Table

This step creates the left table represented as transactionsRDD, which contains (K=userID,V=product) pairs from transactions input data.

**Listing 4.26:** STEP-6: create (K=userID,V=product) pairs for transactions

```
1 // STEP-6: create (K=userID, V=product) pairs for transactions
2 // PairFunction<T, K, V>
3 // T => Tuple2<K, V>
4 // sample transaction input: 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    });

```

#### 4.6.9 STEP-7: use Spark's built-in JavaPairRDD.leftOuterJoin() method

This is core step for performing the left outer join operation by using Spark's JavaPairRDD.leftOuterJoin() method.

**Listing 4.27:** STEP-7: use Spark's built-in JavaPairRDD.leftOuterJoin() method

```
1 // STEP-7: use Spark's built-in JavaPairRDD.leftOuterJoin() method
2 // JavaPairRDD<K, scala.Tuple2<V, Optional<W>>> leftOuterJoin(JavaPairRDD<K, W> other)
3 // Perform a left outer join of this and other. For each element (k, v) in this,
4 // the resulting RDD will either contain all pairs (k, (v, Some(w))) for w in
5 // other, or the pair (k, (v, None)) if no elements in other have key k.
6 //
7 // here we perform a transactionsRDD.leftOuterJoin(usersRDD)
8 JavaPairRDD<String, Tuple2<String, Optional<String>>> joined =
9     transactionsRDD.leftOuterJoin(usersRDD);
10    joined.saveAsTextFile("/output/1");

```

STEP-7 creates the following output (result of the left outer join operation):

```
#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)))
```

#### 4.6.10 STEP-8: create (product, location) pairs

This step builds another JavaPairRDD, which contains (K=product, V=location) pairs. Note that we completely ignored the userIDs, since we are only interested in products and their unique user's locations.

**Listing 4.28:** STEP-8: create (product, location) pairs

```
1 // STEP-8: create (product, location) pairs
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, Tuple2<String, Optional<String>>> t) {
9         Tuple2<String, Optional<String>> value = t._2;
10        return new Tuple2<String, String>(value._1, value._2.get());
11    }
12 };
13 products.saveAsTextFile("/output/2");
```

STEP-8 creates the following output

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

#### 4.6.11 STEP-9: group (K=product, V=location) pairs by K

This step groups (K=product, V=location) pairs by K. The result will be (K, V2) where V2 is a list of locations (will have duplicate locations).

**Listing 4.29:** STEP-9: group (K=product, V=location) pairs by K

```
1 // STEP-9: group (K=product, V=location) pairs by K
2 JavaPairRDD<String, Iterable<String>> productByLocations = products.groupByKey();
3 productByLocations.saveAsTextFile("/output/3");
```

STEP-9 creates the following output

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

#### 4.6.12 STEP-10: create final output (K=product, V=Set(location))

This final step removes duplicate locations and creates (K,V2), where V2 is a Tuple2<Set<location>, size>.

**Listing 4.30:** STEP-10: create final output (K=product, V=Set<location>) pairs by K

```
1 // STEP-10: create final output (K=product, V=Set<location>) pairs by K
2 JavaPairRDD<String, Tuple2<Set<String>, Integer>> productByUniqueLocations =
3     productByLocations.mapValues(new Function< Iterable<String>,
4                                     Tuple2<Set<String>, Integer> // output
5                                     >() {
6     public Tuple2<Set<String>, Integer> call(Iterable<String> s) {
7         Set<String> uniqueLocations = new HashSet<String>();
8         for (String location : s) {
9             uniqueLocations.add(location);
10        }
11        return new Tuple2<Set<String>, Integer>(uniqueLocations, uniqueLocations.size());
12    }
13 });
14 productByUniqueLocations.saveAsTextFile("/output/4");
```

STEP-10 creates the following final output

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

### 4.6.13 Sample Run by YARN

#### 4.6.13.1 Input

**Input: Right Table**

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

**Input: Left Table**

```
# 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
```

#### 4.6.13.2 Script

```
# cat ./run_left_outer_join_spark.sh
#!/bin/bash
export HADOOP_HOME=/usr/local/hadoop/hadoop-2.4.0
export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
export JAVA_HOME=/usr/java/jdk7
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
```

```

export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export SPARK_HOME=/home/hadoop/spark-1.0.0
export MY_JAR=/home/hadoop/spark_mahmoud_examples/mp.jar
export YARN_APPLICATION_CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
USERS=/data/leftouterjoins/users.txt
TRANSACTIONS=/data/leftouterjoins/transactions.txt
$SPARK_HOME/bin/spark-submit --class SparkLeftOuterJoin \
    --master yarn-cluster \
    --num-executors 3 \
    --driver-memory 1g \
    --executor-memory 1g \
    --executor-cores 10 \
    $MY_JAR    $USERS   $TRANSACTIONS

```

#### 4.6.13.3 Generated HDFS Output

```

# 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))
```

Chapter **5**

## Order Inversion Pattern

### 5.1 Introduction

The main focus of this chapter is to discuss the order inversion (OI) pattern, which can be used in solving some problems using MapReduce framework. The OI pattern enables us to control the order of values received at a reducer (some computations require ordered data). Typically, the OI pattern happens during data analysis phase. In Hadoop, the order of values arriving at a reducer is undefined (there is no order unless, we exploit the sorting phase of MapReduce to push data needed for calculations to the reducer). The OI pattern works for pairs patterns, which uses simpler data structures and requires less reducer memory (due to the fact that there is no additional sorting and ordering of reducer values in the reducer phase).

To understand the OI pattern, we start with a simple example. Consider a reducer with composite  $key = (K_1, K_2)$  and furthermore assume that  $K_1$  is the natural key component of the composite key. Let this reducer to receive following values (there is no ordering between these values):

$$V_1, V_2, \dots, V_n$$

With implementation of the OI pattern, it is possible to sort and classify the values arriving at reducer with  $key = (K_1, K_2)$ . The sole purpose

of using the OI pattern is to properly sequence data presented to the reducer). Further assume that,  $K_1$  is a fixed part of the composite key and  $K_2$  has only 3 (this can be any number) distinct values  $\{K_{2a}, K_{2b}, K_{2c}\}$ , which generated the following values (note that we have to send the keys  $\{(K_1, K_{2a}), (K_1, K_{2b}), (K_1, K_{2c})\}$  to the same reducer)

Keys and Values for Natural Key $K_1$	
Composite Key	Values...
$(K_1, K_{2a})$	$\{A_1, A_2, \dots, A_m\}$
$(K_1, K_{2b})$	$\{B_1, B_2, \dots, B_p\}$
$(K_1, K_{2c})$	$\{C_1, C_2, \dots, C_q\}$

where

- $m + p + q = n$
- Sort Order

$$K_{2a} < K_{2b} < K_{2c} \text{ (ascending order)}$$

OR

$$K_{2a} > K_{2b} > K_{2c} \text{ (descending order)}$$

- $A_i, B_j, C_k \in \{V_1, V_2, \dots, V_n\}$

With proper implementation of the OI pattern, we will make the reducer values to appear in order

$$A_1, A_2, \dots, A_m, B_1, B_2, \dots, B_p, C_1, C_2, \dots, C_q$$

This ordering and sequencing of reducer values will enable us to do some calculations and computations first on  $A_i$ , then on  $B_j$  and finally on  $C_k$ . Note that buffering values are not needed in memory. The main question is how to produce the desired behavior? The answer lies in defining a "custom partitioner," which only pays attention to the left component ( $K_1$  — called the natural reducer key) of the composite key  $(K_1, K_2)$ . That is, the custom partitioner must partition based on the hash of the left key ( $K_1$ ) only.

## 5.2 Example of Order Inversion Pattern

The simple example to demonstrate the OI pattern is to compute relative frequencies of words for a given set of documents. The goal is to build a  $N \times N$  matrix (call this matrix  $M$ ), where  $N = |V|$  (the vocabulary size all given documents) and each cell  $M_{ij}$  contains the number of times word  $W_i$  co-occurs with word  $W_j$  within a specific context. For simplicity, we define this context as the neighbors of  $W_i$ . For example, given the following specific words:

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

If we define the neighborhood of a word as two words before and two words after that word, then we can say that

Neighborhood Table	
Word	Neighbors $\pm 2$
$W_1$	$W_2, W_3$
$W_2$	$W_1, W_3, W_4$
$W_3$	$W_1, W_2, W_4, W_5$
$W_4$	$W_2, W_3, W_5, W_6$
$W_5$	$W_3, W_4, W_6$
$W_6$	$W_4, W_5$

For our example, computing relative frequencies requires marginal counts. But marginal counts cannot be computed until you see all counts. Therefore, we do need to make the marginal counts to arrive at the reducer before the joint counts. It is possible to buffer these values in memory, but it might not scale up, if these can not fit in memory. Note that for computing relative frequencies, we will not use absolute count of words. According to Jimmy Lin and Chris Dyer[15]: "the drawback of absolute counts is that it doesn't take into account the fact that some words appear more frequently than others. Word  $W_i$  may co-occur frequently with  $W_j$  simply because one of the words is very common. A simple remedy is to convert absolute counts into relative frequencies,  $f(W_j|W_i)$ . That is, what proportion of the time does  $W_j$  appear in the context of  $W_i$ ?" This can be computed using the following equation:

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

Here,  $N(a, b)$  indicates the number of times a particular co-occurring word pair is observed in the corpus (a given set of all documents as input). Therefore, we need the count of the joint event (word co-occurrence), divided by what is known as the marginal (the sum of the counts of the conditioning variable co-occurring with anything else).

### 5.3 MapReduce for Order Inversion Pattern

Following our example of computing relative frequencies for a given set of documents, we do need to generate two sequence of data: the first sequence will be total neighborhood counts (the total number of co-occurrences of a word) for the word (let's denote this by composite  $Key = (W, *)$  —  $W$  denotes the word) and the second sequence will be the counts of that word against other specific words (let's denote this by composite  $Key = (W, W_2)$ ): therefore, a reducer will receive the following set of (key,value) pairs in order:

A Reducer (key-value) Pairs in Order	
Key	Values as Integer Numbers...
$(W, *)$	$A_1, A_2, \dots, A_m$
$(W, W_1)$	$B_1, B_2, \dots, B_p$
$(W, W_2)$	$C_1, C_2, \dots, C_q$
...	...

A Reducer (key-value) Pairs in Order	
Key	Values...
$(dog, *)$	100, 129, 500, ...
$(dog, bite)$	2, 1, 1, 1
$(dog, best)$	1, 1, 1, 1, 1
...	...

A concrete example will be:

Now, we want to discuss the relative term co-occurrence and the OI design pattern. For this, we need to compute the co-occurrence matrix, by using the relative frequencies of each pair, instead of the absolute value. Therefore,

we need to count the number of times each pair  $(W_i, W_j)$  occurs divided by the number of total pairs with  $W_i$  (marginal). To complete our MapReduce solution, we will provide a map(), reduce(), and a custom partitioner (`OrderInversionPartitioner` class), which apply the partitioner only according to the first element in the `PairOfWords`<sup>1</sup>, sending all data regarding the same word to the same reducer. Also, we use `PairOfWords` to represent a composite key (composed of two words).

### 5.3.1 Custom Partitioner

Hadoop provides a plugin architecture to plug a custom partitioner:

**Listing 5.1:** Custom Partitioner: 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 provides a proper API to plug-in a custom partitioner:

---

*job.setPartitionerClass(OrderInversionPartitioner.class);*

---

#### 5.3.1.1 Custom Partitioner Implementation in Hadoop

The custom partitioner must ensure that all pairs with the same left word (called the natural key) are sent to the same reducer. For example, the composite keys `{(man, tall), (man, strong), (man, moon), ...}` are assigned to the same reducer. To send these keys to the same reducer, we

---

<sup>1</sup> We will use `PairOfWords` class for representing a pair of words  $(W_i, W_j)$ . The method `PairOfWords.getLeftElement((W_i, W_j))` returns  $W_i$  and the method `PairOfWords.getRightElement((W_i, W_j))` returns  $W_j$

must define a custom partitioner that only pays attention to the left word (in our example, the left word is `man`). That is, the partitioner should partition based on the hash of the left word (so called the natural key) only.

**Listing 5.2:** Custom Partitioner Implementation: OrderInversionPartitioner

```
1 import org.apache.hadoop.io.IntWritable;
2 import org.apache.hadoop.mapreduce.Partitioner;
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     // adapted from String.hashCode()
17     private static long hash(String str) {
18         long h = 112589906842597L; // prime
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 }
```

### 5.3.2 Relative Frequency Mapper

The mapper class emits relative frequencies of a word's neighbors (2 words before and 2 words after). For example, if a mapper receives the following:

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

Then it will emit the following (key, value) pairs:

Mapper Generated (key-value) Pairs	
Key	Value
(w1, w2)	1
(w1, w3)	1
(w1, *)	2
(w2, w1)	1
(w2, w3)	1
(w2, w4)	1
(w2, *)	3
(w3, w1)	1
(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

The mapper algorithm is presented below:

### Listing 5.3: Mapper Class: 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         // driver will set "neighbor.window"
8         neighborWindow = context.getConfiguration().getInt("neighbor.window", 2);
9     }
10
11    // key is system generated, ignored here

```

```

12 // value is a String (set of words)
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) ? tokens.length - 1 : i + neighborWindow;
24         for (int j = start; j <= end; j++) {
25             if (i == j) {
26                 continue;
27             }
28             pair.setNeighbor(tokens[j]);
29             emit(pair, 1);
30         }
31         pair.setNeighbor("*");
32         int totalCount = end - start;
33         emit(pair, totalCount);
34     }
35 }
36 }
```

---

### 5.3.3 Relative Frequency Reducer

Since we have a custom partitioner and implementation of the OI pattern, the values received by a reducer will be based on the natural key of the keys generated by all mappers. For the example presented for the mapper phase, we will have 6 reducers with the following (key, value) input pairs:

Input to Reducers	
Key	Value
(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

The reducer algorithm is presented below.

#### **Listing 5.4:** Reducer Class: 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 {
11                currentWord = key.getLeft();
12                totalCount = getTotalCount(values);
13            }
14        }
15        else {
16            int count = getTotalCount(values);
17            double relativeCount = count / totalCount;
18            emit(key, relativeCount);
19        }
20    }
21
22    private int getTotalCount(Iterable<Integer> values) {
23        int count = 0;
24        for (Integer value : values) {
25            count += value.get();
26        }
27        return count;
28    }
29 }
30 }
```

#### **5.3.4 Implementation Classes in Hadoop**

Implementation Classes in Hadoop	
Class Name	Class Description
RelativeFrequencyDriver	Dirver to submit job
RelativeFrequencyMapper	Defines map()
RelativeFrequencyReducer	Defines reduce()
RelativeFrequencyCombiner	Defines combine()
OrderInversionPartitioner	Custom partitioner: How to partition natural keys
PairOfWords	Represents pair of words (Word1, Word2)

## 5.4 Sample Run

### 5.4.1 Input

```
# 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
```

### 5.4.2 Running MapReduce Job

```
1 # ./run.sh
2 ...
3 added manifest
4 adding: OrderInversionPartitioner.class(in = 1005) (out= 590)(deflated 41%)
5 adding: PairOfWords$Comparator.class(in = 933) (out= 603)(deflated 35%)
6 adding: PairOfWords.class(in = 2932) (out= 1285)(deflated 56%)
7 adding: RelativeFrequencyCombiner.class(in = 1518) (out= 681)(deflated 55%)
8 adding: RelativeFrequencyDriver.class(in = 3121) (out= 1534)(deflated 50%)
9 adding: RelativeFrequencyMapper.class(in = 2900) (out= 1256)(deflated 56%)
10 adding: RelativeFrequencyReducer.class(in = 2208) (out= 1026)(deflated 53%)
11 Deleted hdfs://localhost:9000/lib/order_inversion.jar
12 Deleted hdfs://localhost:9000/order_inversion/output
13 ...
14 14/01/04 14:56:56 INFO input.FileInputFormat: Total input paths to process : 1
15 ...
16 14/01/04 14:56:57 INFO mapred.JobClient: Running job: job_201401041453_0002
17 14/01/04 14:56:58 INFO mapred.JobClient: map 0% reduce 0%
18 14/01/04 14:57:03 INFO mapred.JobClient: map 100% reduce 0%
19 14/01/04 14:57:11 INFO mapred.JobClient: map 100% reduce 22%
20 14/01/04 14:57:12 INFO mapred.JobClient: map 100% reduce 66%
21 14/01/04 14:57:19 INFO mapred.JobClient: map 100% reduce 77%
22 14/01/04 14:57:21 INFO mapred.JobClient: map 100% reduce 100%
23 14/01/04 14:57:21 INFO mapred.JobClient: Job complete: job_201401041453_0002
24 ...
25 14/01/04 14:57:21 INFO mapred.JobClient: Map-Reduce Framework
26 14/01/04 14:57:21 INFO mapred.JobClient: Map output materialized bytes=906
27 14/01/04 14:57:21 INFO mapred.JobClient: Map input records=5
28 14/01/04 14:57:21 INFO mapred.JobClient: Reduce shuffle bytes=906
29 14/01/04 14:57:21 INFO mapred.JobClient: Spilled Records=106
30 14/01/04 14:57:21 INFO mapred.JobClient: Map output bytes=1154
31 14/01/04 14:57:21 INFO mapred.JobClient: Total committed heap usage (bytes)=439619584
32 14/01/04 14:57:21 INFO mapred.JobClient: Combine input records=85
33 14/01/04 14:57:21 INFO mapred.JobClient: SPLIT_RAW_BYTES=125
34 14/01/04 14:57:21 INFO mapred.JobClient: Reduce input records=53
35 14/01/04 14:57:21 INFO mapred.JobClient: Reduce input groups=53
36 14/01/04 14:57:21 INFO mapred.JobClient: Combine output records=53
37 14/01/04 14:57:21 INFO mapred.JobClient: Reduce output records=44
38 14/01/04 14:57:21 INFO mapred.JobClient: Map output records=85
39 14/01/04 14:57:21 INFO RelativeFrequencyDriver: Job Finished in milliseconds: 24869
40
```

### 5.4.3 Generated Output

```
1 # hadoop fs -cat /order_inversion/output/part*
2 (great, a)      0.2
3 (great, is)     0.4
```

```

4 | (great, java)      0.2
5 | (great, language)  0.2
6 | (with, is)         0.3333333333333333
7 | (with, java)       0.3333333333333333
8 | (with, programming) 0.3333333333333333
9 | (fun, green)      0.2
10 | (fun, is)         0.4
11 | (fun, java)       0.2
12 | (fun, language)   0.2
13 | (programming, a)   0.2
14 | (programming, is)  0.2
15 | (programming, java) 0.2
16 | (programming, language) 0.2
17 | (programming, with) 0.2
18 | (a, great)        0.125
19 | (a, is)           0.25
20 | (a, java)         0.25
21 | (a, language)     0.25
22 | (a, programming)  0.125
23 | (green, fun)      0.25
24 | (green, is)        0.25
25 | (green, java)     0.25
26 | (green, language) 0.25
27 | (is, a)            0.14285714285714285
28 | (is, fun)          0.14285714285714285
29 | (is, great)        0.14285714285714285
30 | (is, green)        0.07142857142857142
31 | (is, java)         0.35714285714285715
32 | (is, programming)  0.07142857142857142
33 | (is, with)         0.07142857142857142
34 | (java, a)           0.1666666666666666
35 | (java, fun)         0.0833333333333333
36 | (java, great)      0.0833333333333333
37 | (java, green)      0.0833333333333333
38 | (java, is)          0.4166666666666667
39 | (java, programming) 0.0833333333333333
40 | (java, with)        0.0833333333333333
41 | (language, a)        0.3333333333333333
42 | (language, fun)      0.1666666666666666
43 | (language, great)    0.1666666666666666
44 | (language, green)    0.1666666666666666
45 | (language, programming) 0.1666666666666666

```

# Chapter 6

## Moving Average

### 6.1 Introduction

The purpose of this chapter is to present "moving average" solution in MapReduce/Hadoop. Before presenting a MapReduce solution, we will look at the basic concepts of "moving average". To understand "moving average" concept, we do need to understand time series data. Time series data are quantities that represent the values taken by a variable over a period of time such as a second, minute, hour, day, week, month, quarter, or year. Semiformaly, we can represent a time series data as a sequence of triplets:

$$(k, t, v)$$

Where **k** is a key (such as a stock symbol), **t** is a time (in hours, minutes, or seconds), and **v** is the associated value (such as value of a stock at a point **t**).

Typically, **time series** data occurs wherever the same measurements are recorded on a regular period of time basis. For example, closing price of a company stock is a time series data over minutes, hours, or days. Mean (or average) of time series data (observations equally spaced in time such as per hour, per day) from several consecutive periods is called "moving average." Why do we call it "moving"? It is called "moving" because it is continually recomputed as new time series data becomes available, and it progresses by dropping the earliest value and adding the latest value.

### 6.1.1 Example-1: Time Series Data

Consider the following data for closing stock price of a company called MY-STOCK (note that this is a fake stock symbol):

Time Series Data		
Time Series	Date	Closing Price
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
...	...	...

For moving averages of 3-days we'll have as output:

Moving Average Calculation			
Time Series	Date	Moving Average	How Calculated
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)

### 6.1.2 Example-2: Time Series Data

Another example will be the "Moving Average," which calculates the moving average of unique visitors for different URLs for each date in a specific timeframe. We may use the following input:

Time Series Data		
URL	Date	Unique Visitors
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
URL2	2013-10-02	20
URL2	2013-10-03	30
URL2	2013-10-04	70

For moving averages of 3 days we'll have as output:

Time Series Data		
URL	Date	Moving Average of 3 Days
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

## 6.2 Formal Definition

Let A be a sequence of an ordered set of objects:

$$A = (a_1, a_2, a_3, \dots, a_N)$$

We may express A as:

$$\{a_i\}_{i=1}^N$$

Then an n-moving average is a new sequence  $\{s_i\}_{i=1}^{N-n+1}$  defined from

the  $a_i$  by taking the arithmetic mean of subsequences of  $n$  terms,

$$s_i = \frac{1}{n} \sum_{j=i}^{i+n-1} a_j$$

So the sequences  $S_n$  giving n-moving averages are

$$S_2 = \frac{1}{2}((a_1 + a_2), (a_2 + a_3), \dots, (a_{n-1} + a_n))$$

$$S_3 = \frac{1}{3}((a_1 + a_2 + a_3), (a_2 + a_3 + a_4), \dots, (a_{n-2} + a_{n-1} + a_n))$$

$$S_4 = \frac{1}{4}((a_1 + a_2 + a_3 + a_4), (a_2 + a_3 + a_4 + a_5), \dots, (a_{n-3} + a_{n-2} + a_{n-1} + a_n))$$

...

## 6.3 Moving Average by POJO

For solving moving average, we provide two POJO solutions:

- First solution: using `java.util.Queue`
- Second Solution : using array and simulating queue

In both solutions, we take a window (as a queue data structure) and fill it in a First-In-First-Out (FIFO) manner with time series data points until we have  $N$  points in it (average/mean of these  $N$  points will be the moving average).

### 6.3.1 First solution: using Queue

This solution uses a queue data structure implemented by `java.util.Queue`. At any time, we make sure that we keep only the required items (the size of moving average window size) in the queue.

#### Listing 6.1: SimpleMovingAverage Class

```
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();
21        }
22    }
23
24    public double getMovingAverage() {
25        if (window.isEmpty()) {
26            throw new IllegalArgumentException("average is undefined");
27        }
28        return sum / window.size();
29    }
30 }
```

---

### 6.3.2 Second Solution : using Array

This solution uses a simple array (called `window`) and simulates enqueue and dequeue operations. This solution is more efficient but less intuitive.

**Listing 6.2:** SimpleMovingAverageUsingArray Class

```

1 public class SimpleMovingAverageUsingArray {
2
3     private double sum = 0.0;
4     private final int period;
5     private double[] window = null;
6     private int pointer = 0;
7     private int size = 0;
8
9     public SimpleMovingAverageUsingArray(int period) {
10         if (period < 1) {
```

```

11         throw new IllegalArgumentException("period must be > 0");
12     }
13     this.period = period;
14     window = new double[period];
15 }
16
17 public void addNewNumber(double number) {
18     sum += number;
19     if (size < period) {
20         window[pointer++] = number;
21         size++;
22     }
23     else {
24         // size = period (size cannot be > period)
25         pointer = pointer % period;
26         sum -= window[pointer];
27         window[pointer++] = number;
28     }
29 }
30
31 public double getMovingAverage() {
32     if (size == 0) {
33         throw new IllegalArgumentException("average is undefined");
34     }
35     return sum / size;
36 }
37 }
```

### 6.3.3 Testing of Moving Average

The following is a program for testing "moving average" using SimpleMovingAverage class. The program is tested by two window sizes {3, 4}.

**Listing 6.3:** Test 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         // time series      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 windowHeight : allWindowSizes) {
10         SimpleMovingAverage sma = new SimpleMovingAverage(windowHeight);
11         THE_LOGGER.info("windowSize = " + windowHeight);
12         for (double x : testData) {
13             sma.addNewNumber(x);
14             THE_LOGGER.info("Next number = " + x + ", SMA = " + sma.getMovingAverage());
15         }
16         THE_LOGGER.info("----");
17     }
18 }
19 }
```

### 6.3.4 Sample Run

```

1 # javac SimpleMovingAverage.java
2 # javac TestSimpleMovingAverage.java
3 # java TestSimpleMovingAverage
4
5 13/10/10 22:24:08 INFO TestSimpleMovingAverage: windowHeight = 3
6 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 10.0, SMA = 10.00
7 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 18.0, SMA = 14.00
8 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 20.0, SMA = 16.00
9 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 30.0, SMA = 22.66
10 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 24.0, SMA = 24.66
11 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 33.0, SMA = 29.00
12 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 27.0, SMA = 28.00
13 13/10/10 22:24:08 INFO TestSimpleMovingAverage: -----
14 13/10/10 22:24:08 INFO TestSimpleMovingAverage: windowHeight = 4
15 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 10.0, SMA = 10.00
16 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 18.0, SMA = 14.00
17 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 20.0, SMA = 16.00
18 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 30.0, SMA = 19.50
19 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 24.0, SMA = 23.00
20 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 33.0, SMA = 26.75
21 13/10/10 22:24:08 INFO TestSimpleMovingAverage: Next number = 27.0, SMA = 28.50
22 13/10/10 22:24:08 INFO TestSimpleMovingAverage: -----
```

## 6.4 MapReduce Solution

Now that we understand the concept of a moving average for time series data, we will focus on MapReduce solution for "moving average" problem. Implementation will be provided by using Apache Hadoop. We will see that MapReduce is a very powerful framework for solving moving average for a lot

of time series data (such as stocks for over long period of time). Calculation of "moving average" will be done in the `reduce()` function, where we have all required data for a specific time series.

### 6.4.1 Input

The input to our MapReduce will have the following format:

```
<name-as-string><,><date-as-timestamp><,><value-as-double>
```

A sample example is of stock prices is provided for three stock symbols (GOOG, AAPL, and IBM). Here we listed stock prices for few companies: the first column is the stock symbol, the next field is the timestamp, and the last field is the adjusted closed price of the stock on that day – provided by timestamp):

```
1 GOOG,2004-11-04,184.70
2 GOOG,2004-11-03,191.67
3 GOOG,2004-11-02,194.87
4 AAPL,2013-10-09,486.59
5 AAPL,2013-10-08,480.94
6 AAPL,2013-10-07,487.75
7 AAPL,2013-10-04,483.03
8 AAPL,2013-10-03,483.41
9 IBM,2013-09-30,185.18
10 IBM,2013-09-27,186.92
11 IBM,2013-09-26,190.22
12 IBM,2013-09-25,189.47
13 GOOG,2013-07-19,896.60
14 GOOG,2013-07-18,910.68
15 GOOG,2013-07-17,918.55
```

### 6.4.2 Output

The output to our MapReduce will have the following format:

```
<name-as-string><,><date-as-timestamp><,><moving-average-as-double>
```

A sample example of output is provided below:

```
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
```

By knowing the moving average algorithm, we just need to group data based on the stock symbol, and then sort the values based on the timestamp and then finally apply the moving average algorithm. There are at least two ways to sort our time series data:

- **Option-1**: Sort the time series data in memory (RAM) per reducer key. This option has one problem: if you do not have enough RAM for your reducer's sort operation, then this option will not work.
- **Option-2**: Let the MapReduce framework to do the sorting for the time series data (one of the main features of a MapReduce framework is sorting a grouping and Hadoop does a pretty good job on this). This option is much more scalable than Option-1 and sorting is performed by the `sort()` and `shuffle()` functions of MapReduce Framework. Using this option, we do need to alter key-value pairs and write some plugin classes to perform secondary sorting.

We will provide solutions to both options.

### 6.4.3 MapReduce Solution: Option-1: sort in RAM

The mapper accepts a row of input as

```
<name-as-string><,><timestamp><,><value-as-double>
```

and emits key-value pairs, where key is the `<name-as-string>` and value is `<timestamp><,><value-as-double>`. The reducer will receive key-values, where key is the `<name-as-string>` and values will be a un-ordered list of `<timestamp><,><value-as-double>`.

#### 6.4.3.1 Time Series Data

Time series data is represented as a `TimeSeriesData` object, which is illustrated below. This class implements `Writable` (since these objects will persist in Hadoop) and `Comparable<TimeSeriesData>` (since we will do in memory sorting on these objects).

**Listing 6.4:** `TimeSeriesData` Class

```

1 import org.apache.hadoop.io.Writable;
2 ...
3 /**
4 *
5 * TimeSeriesData represents a pair of
6 * (time-series-timestamp, time-series-value).
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}

```

---

#### 6.4.3.2 The Mapper Function

**Listing 6.5:** The Mapper Function

```

1 /**
2 * @param key is the <name-as-string>
3 * @param value is the <timestamp><,><value-as-double>
4 */
5 map(key, value) {
6     TimeSeriesData timeseries = new TimeSeriesData(value.timestamp, value.value-as-double);
7     emit(key, timeseries);
8 }

```

---

#### 6.4.3.3 The Reducer Function

### Listing 6.6: The Reducer Function

```
1 public class MovingAverageSortInRAM_Reducer {
2
3     private int windowSize = 4; //default
4
5     /**
6      * called once at the start of the reducer task
7      */
8     setup() {
9         // "moving.average.window.size" will be set at the driver
10        // configuration is MRF's configuration object
11        windowSize = configuration.get("moving.average.window.size");
12    }
13
14    /**
15     * @param key is the <name-as-string>
16     * @param value is a List<TimeSeriesData>
17     * where TimeSeriesData represents a pair of (timestamp, value)
18     */
19    reduce(key, values) {
20        List<TimeSeriesData> sortedTimeSeries = sort(values);
21        // call movingAverge(sortedTimeSeries, windowSize) and emit output(s)
22        // apply moving average algorithm to sortedTimeSeries
23        double sum = 0.0;
24        // calculate prefix sum
25        for (int i=0; i < windowSize-1; i++) {
26            sum += sortedTimeSeries.get(i).getValue();
27        }
28
29        // now we have enough timeseries data to calculate moving average
30        for (int i = windowSize-1; i < sortedTimeSeries.size(); i++) {
31            sum += sortedTimeSeries.get(i).getValue();
32            double movingAverage = sum / windowSize;
33            long timestamp = sortedTimeSeries.get(i).getTimestamp();
34            Text outputValue = timestamp+ "," + movingAverage;
35            // send output to distributed file system
36            emit(key, outputValue);
37
38            // prepare for next iteration
39            sum -= sortedTimeSeries.get(i-windowSize+1).getValue();
40        }
41
42    }
43 }
```

---

#### 6.4.4 Hadoop Implementation: sort in RAM

Our Hadoop implementation is comprised of the following classes:

<i>Class name</i>	<i>Description</i>
SortInMemory_MovingAverageDriver	A driver program to submit Hadoop jobs
SortInMemory_MovingAverageMapper	Defines map()
SortInMemory_MovingAverageReducer	Defines reduce()
TimeSeriesData	Represents a time series data point expressed as a pair of (timestamp, double)
DateUtil	Provides basic Date conversion utilities
HadoopUtil	Provides basic Hadoop utilities

#### 6.4.5 Sample Run

```
mahmoud@Mahmouds-MacBook:~/zmp/map_reduce_book/hadooptests/moving_average/sort_in_
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

#!/bin/bash
```

```

export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true

export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.::/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin

CLASSPATH=.:$HADOOP_HOME/conf
HADOOP_JAR_FILES='find $HADOOP_HOME -name 'hadoop*.jar''
for.hadoopjarfile in $HADOOP_JAR_FILES ; do
    CLASSPATH=$CLASSPATH:$hadoopjarfile
done

export TESTS=/Users/mahmoud/zmp/map_reduce_book/hadooptests
LIB_DIR=$TESTS/lib
JAR_FILES='find $LIB_DIR -name '*.jar''
for jarfile in $JAR_FILES ; do
    CLASSPATH=$CLASSPATH:$jarfile
done

export JAR=$TESTS/moving_average/sort_in_memory/sort_in_memory.jar
export CLASSPATH=$CLASSPATH:$JAR
export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class
$HADOOP_HOME/bin/hadoop fs -rm /lib/sort_in_memory.jar
$HADOOP_HOME/bin/hadoop fs -put $JAR /lib/
export INPUT=/moving_average/sort_in_memory/input
export OUTPUT=/moving_average/sort_in_memory/output
$HADOOP_HOME/bin/hadoop fs -rmdir $OUTPUT
export driver=SortInMemory_MovingAverageDriver
export WINDOW_SIZE=2
$HADOOP_HOME/bin/hadoop jar $JAR $driver $WINDOW_SIZE $INPUT $OUTPUT

```

```

mahmoud@Mahmouds-MacBook:~/zmp/map_reduce_book/hadooptests/moving_average/sort_in_
...
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 15:07:09 INFO mapred.JobClient: map 100% reduce 0%
14/05/12 15:07:17 INFO mapred.JobClient: map 100% reduce 3%
...
14/05/12 15:07:53 INFO mapred.JobClient: map 100% reduce 86%
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 output materialized bytes=401
14/05/12 15:07:55 INFO mapred.JobClient: Map input records=15
14/05/12 15:07:55 INFO mapred.JobClient: Reduce shuffle bytes=401
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: Total committed heap usage (bytes)=10
14/05/12 15:07:55 INFO mapred.JobClient: Combine input records=0
14/05/12 15:07:55 INFO mapred.JobClient: SPLIT_RAW_BYTES=133
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

# hadoop fs -cat /moving_average/sort_in_memory/output/part*
GOOG 2004-11-03,193.269999999999998
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

```

#### 6.4.6 MapReduce Solution: Option-2: Sort by MR Framework

In previous sections, we showed that how to solve moving average problem using MapReduce framework for lots of time series data by grouping by `name` and then sorting (by `timestamp`) time series values in memory. There is a problem with sorting data in memory: if you have a lot of data, then it might not fit in memory (unless you have a lot of RAM in all cluster nodes, which is not the case to date, since memory is not as cheap as hard disks). In this section, we will show how to use MRF (MapReduce Framework) to do sorting of data without sorting in memory. When MRF does sorting by shuffling, it is called as "secondary sorting." For example, in Hadoop, we can sort the data by the `shuffle` phase of MapReduce framework. The question is how this "secondary sorting" will be accomplished. We do need to write some additional plug-in custom Java classes, which will be plugged-in to the MapReuce's framework.

How do we plug-in the custom Java classes to accomplish sorting by MRF? We set these classes in the driver class:

**Listing 6.7:** The SortByMRF MovingAverageDriver Class

```

1 import ...
2 /**
3  * MapReduce job for moving averages of time series data by
4  * using MapReduce's "secondary sort" (sort by shuffle function).
5 */

```

```

6 public class SortByMRF_MovingAverageDriver {
7
8     public static void main(String[] args) throws Exception {
9         Configuration conf = new Configuration();
10        JobConf jobconf = new JobConf(conf, SortByMRF_MovingAverageDriver.class);
11        ...
12
13        // the following 3 setting are needed for "secondary sorting"
14        // Partitioner decides which mapper output goes to which reducer
15        // based on mapper output key. In general, different key is in
16        // different group (Iterator at the reducer side). But sometimes,
17        // we want different key in the same group. This is the time for
18        // Output Value Grouping Comparator, which is used to group mapper
19        // output (similar to group by condition in SQL). The Output Key
20        // Comparator is used during sort stage for the mapper output key.
21        jobconf.setPartitionerClass(NaturalKeyPartitioner.class);
22        jobconf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
23        jobconf.setOutputValueGroupingComparator(NaturalKeyGroupingComparator.class);
24
25        JobClient.runJob(jobconf);
26    }
27 }

```

---

To implement "secondary sort" for moving average, we need to make the mapper's output key as a composite of the natural key (name-as-string) and the natural value (timeseries-timestamp). The following diagram ( 6.1 ) illustrates the natural and composite keys needed.

The custom CompositeKey (comprised of "name" and "timestamp") is defined as:

#### **Listing 6.8:** CompositeKey Class Definition

```

1 import org.apache.hadoop.io.WritableComparable;
2 ...
3 public class CompositeKey
4     implements WritableComparable<CompositeKey> {
5     // composite key is a pair (name, timestamp)
6     private String name;
7     private long timestamp;
8     ...
9 }

```

---

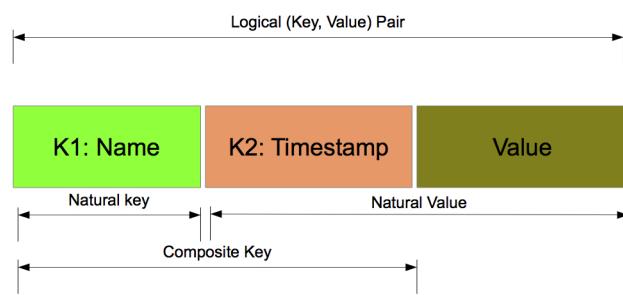


Figure 6.1: Composite and Natural Keys

The `CompositeKey` class has to implement `WritableComparable` interface, since these objects will persist in HDFS. Therefore, the custom `CompositeKey` class gives Hadoop the needed information during the shuffle to perform a sort on two fields ("name" and "timestamp") rather than the ("name") only. But how the `CompositeKey` objects will be sorted? We do need to provide a class that provides comparison of `CompositeKey` objects. This comparison is accomplished by the `CompositeKeyComparator` class (the main function of this class is to provide the `compare()` method):

**Listing 6.9:** `CompositeKeyComparator` Class

```
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             // names are equal here
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` is a `Comparator` for `WritableComparable(s)`. This

base implementation uses the natural ordering. To define alternate orderings, we had to override `compare(WritableComparable, WritableComparable)` method. So sorting of the `CompositeKey` (the combination of the natural key — name — and the natural value — timestamp —) objects is provided by the custom `CompositeKeyComparator` class.

Now that we know how the composite keys will be sorted, the next question is how will data arrive to reducers? This will be done by another custom class: `NaturalKeyPartitioner`, which implements the `Partitioner`<sup>1</sup> interface, which partitions the key space generated by all mappers. Even though all keys are sorted by the whole `CompositeKey` (comprised of `name` and `timestamp`), but in partitioning we will use only by the `name` and the reason is that we want all the sorted values with the same `name` to go to a single reducer.

Here is the code for our custom partitioner called `NaturalKeyPartitioner`:

#### Listing 6.10: CompositeKey Class Definition

```
1 import org.apache.hadoop.mapred.JobConf;
2 import org.apache.hadoop.mapred.Partitioner;
3 public class NaturalKeyPartitioner implements
4     Partitioner<CompositeKey, TimeSeriesData> {
5
6     @Override
7     public int getPartition(CompositeKey key,
8                           TimeSeriesData value,
9                           int numberofPartitions) {
10        return Math.abs((int) (hash(key.getName()) % numberofPartitions));
11    }
12
13    @Override
14    public void configure(JobConf jobconf) {
15    }
16
17    /**
```

---

<sup>1</sup> `org.apache.hadoop.mapred.Partitioner`. The `Partitioner` controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the  $m$  reduce tasks the intermediate key (and hence the record) is sent for reduction. (source: <http://hadoop.apache.org/docs/current/api/index.html?org/apache/hadoop/mapred//class-usePartitioner.html>)

```

18     * adapted from String.hashCode()
19     */
20     static long hash(String str) {
21         long h = 1125899906842597L; // prime
22         int length = str.length();
23         for (int i = 0; i < length; i++) {
24             h = 31*h + str.charAt(i);
25         }
26         return h;
27     }
28 }
```

---

The last piece of a plug-in class is `NaturalKeyGroupingComparator`, which is used during Hadoop's shuffle phase to group composite keys by the natural part of their key (in our example, the first components of composite key is the `name` component).

**Listing 6.11:** NaturalKeyGroupingComparator Class Definition

```

1 import org.apache.hadoop.io.WritableComparable;
2 import org.apache.hadoop.io.WritableComparator;
3 /**
4 *
5 * NaturalKeyGroupingComparator
6 *
7 * This class is used during Hadoop's shuffle phase to
8 * group composite key's by the natural of their key.
9 * The natural key for time series data is the "name".
10 *
11 */
12 public class NaturalKeyGroupingComparator extends WritableComparator {
13     protected NaturalKeyGroupingComparator() {
14         super(CompositeKey.class, true);
15     }
16
17     @Override
18     public int compare(WritableComparable w1, WritableComparable w2) {
19         CompositeKey key1 = (CompositeKey) w1;
20         CompositeKey key2 = (CompositeKey) w2;
21         // natural keys are: key1.getName() and key2.getName()
22         return key1.getName().compareTo(key2.getName());
23     }
24 }
```

---

The following classes are used to implement "moving average" by sorting reducer's values by using the MapReducer's framework.

<i>Class name</i>	<i>Description</i>
<code>MovingAverage</code>	A simple moving average algorithm
<code>SortByMRF_MovingAverageDriver</code>	A driver program to submit Hadoop jobs
<code>SortByMRF_MovingAverageMapper</code>	Defines map()
<code>SortByMRF_MovingAverageReducer</code>	Defines reduce()
<code>TimeSeriesData</code>	Represents a time series data point expressed as a pair of (timestamp, double)
<code>CompositeKey</code>	Defines a custom composite key of (string, timestamp)
<code>CompositeKeyComparator</code>	defines sorting order for CompositeKey
<code>NaturalKeyPartitioner</code>	partitions the data output from the map phase before it is sent through the shuffle phase.
<code>NaturalKeyGroupingComparator</code>	this class is used during Hadoop's shuffle phase to group composite Key's by the first part (natural) of their key.
<code>DateUtil</code>	Provides basic Date conversion utilities
<code>HadoopUtil</code>	Provides basic Hadoop utilities

## 6.5 Sample Run

### 6.5.0.1 HDFS Input

```

1 # hadoop fs -cat /moving_average/sort_by_mrf/input/*
2 GOOG,2004-11-04,184.70
3 GOOG,2004-11-03,191.67
4 GOOG,2004-11-02,194.87
5 AAPL,2013-10-09,486.59
6 AAPL,2013-10-08,480.94
7 AAPL,2013-10-07,487.75
8 AAPL,2013-10-04,483.03
9 AAPL,2013-10-03,483.41
10 IBM,2013-09-30,185.18
11 IBM,2013-09-27,186.92
12 IBM,2013-09-26,190.22
13 IBM,2013-09-25,189.47
14 GOOG,2013-07-19,896.60
15 GOOG,2013-07-18,910.68
16 GOOG,2013-07-17,918.55

```

### 6.5.0.2 Sample Script

```
1 # cat run.sh
2 #!/bin/bash
3 export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
4 export HADOOP_HOME_WARN_SUPPRESS=true
5
6 export JAVA_HOME='/usr/libexec/java_home'
7 echo "JAVA_HOME=$JAVA_HOME"
8
9 PATH=.:${PATH}:$HADOOP_HOME/bin:$JAVA_HOME/bin
10 export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin
11
12 #CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
13 CLASSPATH=.:$HADOOP_HOME/conf
14 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.2.1.jar
15 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.2.1.jar
16 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.2.1.jar
17 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.2.1.jar
18 CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.2.1.jar
19
20 export TESTS=/Users/mahmoud/zmp/map_reduce_book/hadooptests
21 LIB_DIR=$TESTS/lib
22 JAR_FILES='find $LIB_DIR -name "*.jar"'
23 for jarfile in $JAR_FILES ; do
24     CLASSPATH=$CLASSPATH:$jarfile
25 done
26
27 export JAR=$TESTS/moving_average/sort_by_mrf/sort_by_mrf.jar
28 export CLASSPATH=$CLASSPATH:$JAR
29 export HADOOP_CLASSPATH=$CLASSPATH
30
31 javac *.java
32 jar cvf $JAR *.class
33 $HADOOP_HOME/bin/hadoop fs -rm /lib/sort_by_mrf.jar
34 $HADOOP_HOME/bin/hadoop fs -put $JAR /lib/
35 export input=/moving_average/sort_by_mrf/input
36 export output=/moving_average/sort_by_mrf/output
37 $HADOOP_HOME/bin/hadoop fs -rmr $output
38 export driver=SortByMRF_MovingAverageDriver
39 export window_size=2
40 $HADOOP_HOME/bin/hadoop jar $JAR $driver $window_size $input $output
```

### 6.5.0.3 Script Run

```
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 14/05/12 16:15:36 INFO mapred.JobClient: map 25% reduce 0%
8 ...
9 14/05/12 16:16:31 INFO mapred.JobClient: map 100% reduce 86%
10 14/05/12 16:16:32 INFO mapred.JobClient: map 100% reduce 100%
11 14/05/12 16:16:33 INFO mapred.JobClient: Job complete: job_201405121612_0002
12 ...
13 14/05/12 16:16:33 INFO mapred.JobClient: Map-Reduce Framework
14 14/05/12 16:16:33 INFO mapred.JobClient: Map output materialized bytes=3176
15 14/05/12 16:16:33 INFO mapred.JobClient: Map input records=15
16 14/05/12 16:16:33 INFO mapred.JobClient: Reduce shuffle bytes=3176
17 14/05/12 16:16:33 INFO mapred.JobClient: Spilled Records=30
18 14/05/12 16:16:33 INFO mapred.JobClient: Map output bytes=446
19 14/05/12 16:16:33 INFO mapred.JobClient: Total committed heap usage (bytes)=4545941504
20 ...
21 14/05/12 16:16:33 INFO mapred.JobClient: Reduce input records=15
22 14/05/12 16:16:33 INFO mapred.JobClient: Reduce input groups=3
23 14/05/12 16:16:33 INFO mapred.JobClient: Combine output records=0
24 14/05/12 16:16:33 INFO mapred.JobClient: Reduce output records=15
25 14/05/12 16:16:33 INFO mapred.JobClient: Map output records=15
```

#### 6.5.0.4 Generated Output

```
1 # hadoop fs -cat /moving_average/sort_by_mrf/output/part*
2 GOOG      2004-11-02,194.87
3 GOOG      2004-11-03,193.2699999999998
4 GOOG      2004-11-04,188.185
5 GOOG      2013-07-17,551.625
6 GOOG      2013-07-18,914.614999999999
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       2013-09-30,186.0499999999995
```

Chapter **7**

# Market Basket Analysis

## 7.1 What is Market Basket Analysis?

Market Basket Analysis algorithm is a popular data mining algorithm, frequently used by marketing and e-commerce professionals to reveal affinities between individual products or product groupings. The general goal of data mining is to extract interesting correlated information from a large collection of data, for example millions of sales in supermarket or credit card transactions. Finding *frequent sets* in mining association rules for market basket analysis is a computationally intensive problem. This is an ideal case for MapReduce to solve this kind of a problem. Market basket analysis is a technique to identify items likely to be purchased together and association rule mining finds correlations between items in a set of transactions. Finally, marketers may use association rules to place the items next to each other so that users buy more items.

This chapter provides two solutions to market basket analysis:

- MapReduce/Hadoop solution to MBA for tuples of order of N (where  $N=1, 2, 3, \dots$ ). This solution just finds the frequent patterns.
- Spark/Hadoop solution, which not only finds frequent patterns, but also generates association rules for frequent patterns.

## 7.2 MapReduce/Hadoop Solution

The goal of this chapter is to present a MapReduce solution for data mining analysis to find the most frequently occurred pair of products (order of 1, 2, ...) in baskets at a given supermarket or e-commerce store. Our MapReduce solution is expandable to find the most frequently occurred Tuple-N ( $N=1, 2, 3, \dots$ ) of products (order of  $N$ ) in baskets. The "order of  $N$ " (as an integer) will be passed as an argument to the MapReduce driver, which then will set that in Hadoop's Configuration object. Finally, the `map()` will read that parameter from the Hadoop's Configuration object in the `setup()` method. Once we have The most frequent item sets  $F_i$  ( $i=1, 2, \dots$ ) can be used to produce association rule of the transaction. For example, if we have 5 items  $\{A, B, C, D, E\}$  with the following six transactions:

```
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
```

then the goal is to build frequent item sets  $F_1$  (size = 1) and  $F_2$  (size = 2) as

- $F_1 = \{[C, 3], [A, 3], [B, 3], [E, 4]\}$
- $F_2 = \{[<A,C>, 2], [<C,E>, 2], [<A,E>, 2], [<B,E>, 2]\}$

Note that in this example, we applied a minimum "support" of 2, which is a count of how many times a frequent pattern occurs in the entire transaction set. Thus,  $[D, 1]$  is eliminated in  $F_1$ . The item sets  $F_1$  and  $F_2$  can be used to produce association rule of the transaction. An association rule is something like:

```
LHS => RHS  
coke => chips  
if customers purchase coke (called left-hand-side),  
they also buy chips (called right-hand-side).
```

In data mining, an association rule has two metrics:

- **Support:** Frequency of occurrence of an itemset. For example  $\text{Support}(\{A, C\}) = 2$ . This means that items A and C appear together only in 2 transactions
- **Confidence:** How often does the left-hand-side of the rule appear with the right-hand-side.

Market Basket Analysis enables us to understand behavior of shoppers

- what items are bought together?
- what is in each shopping basket?
- what items should be put on sale?
- how should items be placed next to each other for maximum profit? For example if a supermarket has 10,000 or more unique items, then there are 50 million 2-item combinations and 100 billion 3-item combinations.
- how to determine te of the catalog of an e-commerce site

To understand the main ideas of MBA, consider the picture of the shopping cart at a checkout counter in a supermarket filled with various products purchased by a customer. This basket contains a set of products: tuna, milk, orange juice, bananas, eggs, tooth paste, window cleaner, and detergent. One basket tells us about what one customer purchased at one time. A complete list of purchases made by all customers (all transactions in a supermarket) provides much more information; it describes the most important part of a retailing businesswhat merchandise customers are buying and when. Therefore, the purpose of MBA is how to improve sales by properly marketing which products should be placed next to each other on the shelves, and by carefully designing the layout of products in the catalog of an e-commerce site. The final goal of market basket analysis is the automatic generation of association rules.

## 7.3 What are the Application areas for MBA?

Market Basket Analysis can be applied to supermarket shoppers and it is important to realize that there are many other areas in which it can be applied. These include:

- Analysis of credit card purchases
- Analysis of telephone calling patterns
- Identification of fraudulent medical insurance claims (Consider cases where common rules are broken)
- Analysis of telecom service purchases
- Analysis of daily/weekly purchases at Amazon.com

## 7.4 Market Basket Analysis using MapReduce

Since there will be a lot of big data, MapReduce is an ideal framework to do Market Basket Analysis. We follow Market Basket Analysis Algorithm [30] presented by Jongwook Woo and Yuhang Xu. What is the problem we are trying to solve with MapReduce? Given a set of transactions (where each transaction is a set of items), we want to answer this question: what is a pair of items that people frequently buy at store.

The high-level Mapreduce algorithm is presented below. Each `map()` function accepts a transaction, which is a set of items  $\{I_1, I_2, \dots, I_n\}$  purchased by a customer. A mapper first sort (it can be ascending or descending) items, which generates  $\{S_1, S_2, \dots, S_n\}$  and then emits  $(key, 1)$  pairs, where  $key = \text{Tuple2}(S_i, S_j)$  and  $S_i \leq S_j$ . The job of combiners and reducers is to aggregate and count the frequencies.

Before we discuss the MapReduce algorithm in detail, let's look at the input and expected output.

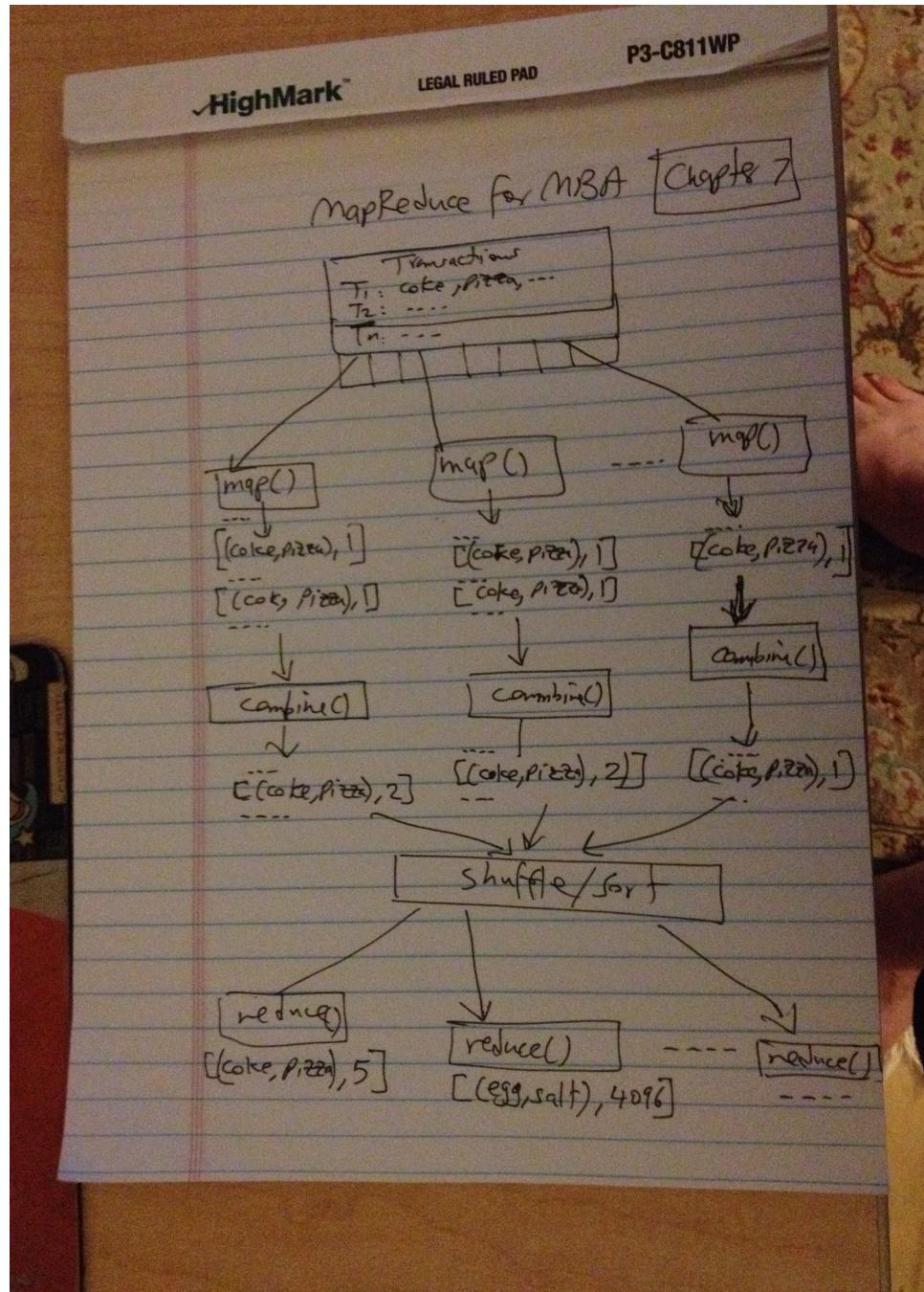


Figure 7.1: MapReduce Algorithm for Market Basket Analysis

#### 7.4.0.5 Input

We assume that input is given as a sequence of transactions (one transaction per-line). Transaction items are separated by spaces. Sample input will be (red colors are not part of actual data):

```
Transaction 1: cracker, ice cream, coke, apple  
Transaction 2: chicken, pizza, coke, bread  
Transaction 3: baguette, soda, shampoo, cracker, pepsi  
Transaction 4: baguette, cream cheese, diapers, milk  
...
```

#### 7.4.0.6 Expected Output for Tuple2 (Order of 2)

For Order of 2, we want to find frequency of items for all unique pairs of ( $Item_1$ ,  $Item_2$ ). Sample expected output is presented below:

Pair of Items	Frequency
...	...
(bear, cracker)	8709
(baguette, eggs)	7524
(cracker, coke)	4300
...	...

#### 7.4.0.7 Expected Output for Tuple3 (Order of 3)

For Order of 3, we want to find frequency of items for all unique sets of ( $Item_1$ ,  $Item_2$ ,  $Item_3$ ). Sample expected output is presented below:

Triplet of Items	Frequency
...	...
(bear, cracker, eggs)	1940
(baguette, diapers, eggs)	1900
(cracker, coke, meat)	1843
...	...

#### 7.4.0.8 Mapper Informal

The `map()` will take a single transaction and generate a set of (key, value) pairs to be consumed by `reduce()`. The mapper pairs the transaction items as a key and the number of key occurrences as its value in the basket (for all transactions and without the transaction numbers).

For example, `map()` for "Transaction 1" will emit the following (key, value) pairs:

```
[<cracker, icecream>, 1]  
[<cracker, coke>, 1]  
[<cracker, apple>, 1]  
[<icecream, coke>, 1]  
[<icecream, apple>, 1]  
[<coke, apple>, 1]
```

We should note that if we select the two items in a basket as a key, there should be incorrect counting for the occurrence of the items in the pairs. For example, if transactions  $T_1$  and  $T_2$  have the following items

```
T1: cracker, ice cream, coke  
T2: ice cream, coke, cracker
```

which have the same items but in different order.

For transaction  $T_1$ , `map()` will generate:

```
[(cracker, ice cream), 1]  
[(cracker, coke), 1]  
[(ice cream, coke), 1]
```

For transaction  $T_2$ , `map()` will generate:

```
[(ice cream, coke), 1]  
[(ice cream, cracker), 1]  
[(coke, cracker), 1]
```

As you observe from `map()` outputs for transactions  $T_1$  and  $T_2$ , we have a total of 6 different pairs of items that occur only once respectively, which should amount to 3 different pairs. That is, keys `(cracker, ice cream)` and `(ice cream, cracker)` are not the same even though they are, which is not correct. We can avoid this problem if we sort the transaction items in alphabetical order before generating (key, value) pairs. After sorting items in transactions we will have:

```
Sorted T1: coke, cracker, ice cream
Sorted T2: coke, cracker, ice cream
```

Now each transaction ( $T_1$  and  $T_2$ ) has the following 3 pair of (key, value) pairs:

```
[(coke, cracker), 1]
[(coke, ice cream), 1]
[(cracker, ice cream), 1]
```

That is TWO different pairs of items that occur twice respectively so that we can accumulate the value of the occurrence for these two transactions as follows: `[(coke, cracker), 2], [(coke, ice cream), 2], [(cracker, ice cream), 2]`, which leads to the correct count of the total number of occurrences.

#### 7.4.1 Mapper Formal

The Market Basket Analysis (MBA) Algorithm for Mapper is illustrated below. Mapper reads the input data and creates a list of items for each transaction. For each transaction, its time complexity is  $O(n)$  where  $n$  is the number of items for a transaction. Then, the items in the transaction list are sorted to avoid the duplicated keys; `(cracker, coke)` and `(coke, craker)` are the same keys. Time complexity of quicksort is  $O(n \log n)$ . Then, the sorted transaction items should be converted to pairs of items as keys, which is a cross operation in order to generate cross pairs of the items in the list.

##### Listing 7.1: MBA `map()` Function

```
1 // key is transaction ID and ignored here
2 // value = transaction items (I1, I2, ..., In).
3 map(key, value) {
4     (S1, S2, ..., Sn) = sort(I1, I2, ..., In);
```

```

5  // now, we have: S1 < S2 < ... < Sn
6  List<Tuple2<Si, Sj>> listOfPairs =
7      Combinations.generateCombinations(S1, S2, ..., Sn);
8  for ( Tuple2<Si, Sj> pair : listOfPairs) {
9      // reducer key is: Tuple2<Si, Sj>
10     // reducer value is: integer 1
11     emit([Tuple2<Si, Sj>, 1]);
12 }
13 }
```

---

`Combinations` is a simple Java utility class, which generate combinations of basket items for a given "list of items" and "number of pairs" (as 2, 3, 4, ...). Assume that  $(S_1, S_2, \dots, S_n)$  is sorted (i.e.,  $S_1 \leq S_2 \leq \dots \leq S_n$ ). Then the `Combinations.generateCombinations(S1, S2, ..., Sn)` method generate all combinations between any two items in a given transaction. For example, `generateCombinations(S1, S2, S3, S4)` will return the following pairs:

```

(S1, S2)
(S1, S3)
(S1, S4)
(S2, S3)
(S2, S4)
(S3, S4)
```

#### 7.4.2 Reducer

The Market Basket Analysis (MBA) Algorithm for Reducer is illustrated below. The reducer is to sum up the number of values per reducer key. Thus, its time complexity is  $O(v)$  where  $v$  is the number of values per key.

**Listing 7.2:** MBA reduce() Function

```

1 // key is in form of Tuple2(Si, Sj)
2 // value = List<integer>, where each element is an integer number
3 reduce(Tuple2(Si, Sj) key, List<integer> values) {
4     integer sum = 0;
5     for (integer i : values) {
6         sum += i;
7     }
8     emit(key, sum);
9 }
10 }
```

---

## 7.5 MapReduce/Hadoop Implementation Classes

The hadoop implementation is comprised of the following Java classes.

Implementation Classes in MapReduce/Hadoop	
Class Name	Class Description
Combination	A utility class to create combination of items
MBADriver	Submits the job to Hadoop
MBAMapper	Defines map()
MBAReducer	Defines reduce()

### 7.5.1 Find Sorted Combinations

The `Combination.findSortedCombinations()` is a recursive function, which will create a unique combinations for any order of  $N$  (where  $N = 2, 3, \dots$ ).

**Listing 7.3:** Find Sorted Combinations

```
1 /**
2  * If elements = { a, b, c, d },
3  * then findCollections(elements, 2) will return:
4  *      { [a, b], [a, c], [a, d], [b, c], [b, d], [c, d] }
5  *
6  * and findCollections(elements, 3) will return:
7  *      { [a, b, c], [a, b, d], [a, c, d], [b, c, d] }
8  *
9 */
10 public static <T extends Comparable<? super T>> List<List<T>>
11         findSortedCombinations(Collection<T> elements, int n) {
12     List<List<T>> result = new ArrayList<List<T>>();
13
14     // handle initial step for recursion
15     if (n == 0) {
16         result.add(new ArrayList<T>());
17         return result;
18     }
19
20     // handle recursion for n-1
21     List<List<T>> combinations = findSortedCombinations(elements, n - 1);
22     for (List<T> combination: combinations) {
23         for (T element: elements) {
24             if (combination.contains(element)) {
25                 continue;
26             }
27
28             List<T> list = new ArrayList<T>();
29             list.addAll(combination);
```

```

31         if (list.contains(element)) {
32             continue;
33         }
34
35         list.add(element);
36         //sort items not to duplicate the items
37         // example: (a, b, c) and (a, c, b) might become
38         // different items to be counted if not sorted
39         Collections.sort(list);
40
41         if (result.contains(list)) {
42             continue;
43         }
44         result.add(list);
45     }
46 }
47 return result;
48 }
```

---

### 7.5.2 Market Basket Analysis Driver: MBADriver

The driver class accepts 3 parameters and submits job to the MapReduce/Hadoop framework. The driver also sets "number.of.pairs" configuration parameter to be read by mappers.

**Listing 7.4:** MBADriver

```

1 public class MBADriver extends Configured implements Tool {
2 ...
3     public int run(String args[]) throws Exception {
4         String inputPath = args[0];
5         String outputPath = args[1];
6         int numberOfPairs = Integer.parseInt(args[2]);
7 ...
8         // job configuration
9         Job job = new Job(getConf());
10 ...
11         job.getConfiguration().setInt("number.of.pairs", numberOfPairs);
12
13         // set input/output path
14         FileInputFormat.setInputPaths(job, new Path(inputPath));
15         FileOutputFormat.setOutputPath(job, new Path(outputPath));
16
17         //Mapper K, V output
18         job.setMapOutputKeyClass(Text.class);
19         job.setMapOutputValueClass(IntWritable.class);
20         //output format
21         job.setOutputFormatClass(TextOutputFormat.class);
22
23         //Reducer K, V output
24         job.setOutputKeyClass(Text.class);
```

```

25     job.setOutputValueClass(IntWritable.class);
26
27     // set mapper/reducer/combiner
28     job.setMapperClass(MBAMapper.class);
29     job.setCombinerClass(MBAReducer.class);
30     job.setReducerClass(MBAReducer.class);
31
32     //delete the output path if exists to avoid "existing dir/file" error
33     Path outputDir = new Path(outputPath);
34     FileSystem.get(getConf()).delete(outputDir, true);
35
36     // submit job
37     boolean status = job.waitForCompletion(true);
38     ...
39 }
```

### 7.5.3 Market Basket Analysis Mapper: MBAMapper

The map() function reads one transaction at a time and generates a set of (key, value) pairs, where key is a Order-N combination of items in a given transaction and value will be an integer 1 (to indicate this combination has been seen once). The mapper's setup() method reads "number.of.pairs" from Hadoop's Configuration object.

**Listing 7.5:** MBAMapper

```

1 public class MBAMapper extends Mapper<LongWritable, Text, Text, IntWritable> {
2
3     public static final int DEFAULT_NUMBER_OF_PAIRS = 2;
4
5     //output key2: list of items paired; can be 2 or 3 ...
6     private static final Text reducerKey = new Text();
7
8     //output value2: number of the paired items in the item list
9     private static final IntWritable NUMBER_ONE = new IntWritable(1);
10
11    int numberOfPairs; // will be read by setup(), set by driver
12
13    protected void setup(Context context)
14        throws IOException, InterruptedException {
15        this.numberOfPairs = context.getConfiguration()
16            .getInt("number.of.pairs", DEFAULT_NUMBER_OF_PAIRS);
17    }
18
19    public void map(LongWritable key, Text value, Context context)
20        throws IOException, InterruptedException {
21        String line = value.toString().trim();
22        List<String> items = convertItemsToList(line);
23        if ((items == null) || (items.isEmpty())) {
24            // no mapper output will be generated
```

```

25         return;
26     }
27     generateMapperOutput(numberOfPairs, items, context);
28 }
29
30 private static List<String> convertItemsToList(String line) {
31     // see the next section
32 }
33
34 private void generateMapperOutput(...){
35     // see the next section
36 }
37 }

```

---

**Listing 7.6:** MBAMapper Helper Methods: convertItemsToList

```

1  private static List<String> convertItemsToList(String line) {
2      if ((line == null) || (line.length() == 0)) {
3          // no mapper output will be generated
4          return null;
5      }
6      String[] tokens = StringUtils.split(line, ",");
7      if ((tokens == null) || (tokens.length == 0)) {
8          return null;
9      }
10     List<String> items = new ArrayList<String>();
11     for (String token : tokens) {
12         if (token != null) {
13             items.add(token.trim());
14         }
15     }
16     return items;
17 }

```

---

**Listing 7.7:** MBAMapper Helper Methods: generateMapperOutput

```

1 /**
2 *
3 * This method builds set of <key, value> by sorting the input list.
4 * key is a combination of items in a transaction, and value = 1.
5 * Sorting is required to make sure that (a, b) and (b, a)
6 * represent the same key.
7 * @param numberOfPairs    number of pairs associated
8 * @param items    list of items (from input line)
9 * @param context   Hadoop Job context
10 * @throws IOException
11 * @throws InterruptedException
12 */
13 private void generateMapperOutput(int numberOfPairs,
14                                 List<String> items,
15                                 Context context)
16 throws IOException, InterruptedException {

```

```

17     List<List<String>> sortedCombinations =
18         Combination.findSortedCombinations(items, number_of_pairs);
19     for (List<String> itemList: sortedCombinations) {
20         reducerKey.set(itemList.toString());
21         context.write(reducerKey, NUMBER_ONE);
22     }
23 }
24 }
```

---

## 7.5.4 Sample Run

### 7.5.4.1 Input

```
# hadoop fs -cat /market_basket_analysis/input/input.txt
cracker,bread,banana
cracker,coke,butter,coffee
cracker,bread
cracker,bread
cracker,bread,coffee
butter,coke
butter,coke,bread,cracker
```

### 7.5.4.2 Actual Run

```
# 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: number_of_pairs: 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 13:36:43 INFO mapred.JobClient: map 100% reduce 0%
14/05/02 13:36:51 INFO mapred.JobClient: map 100% reduce 3%
...
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 output materialized bytes=328
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: SPLIT_RAW_BYTES=125
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

```

#### 7.5.4.3 Output

```

# hadoop fs -cat /market_basket_analysis/output/part*
[bread, coffee]    1
[butter, coffee]   1
[banana, cracker] 1
[butter, coke]     3
[coffee, cracker] 2
[bread, butter]    1
[banana, bread]   1
[bread, cracker]   5
[coke, cracker]   2
[bread, coke]      1
[coffee, coke]    1
[butter, cracker] 2

```

## 7.6 Spark/Hadoop Solution

This section presents a Spark solution for generating all "Association Rules" for a set of transactions. Association rules are statements of the if-then form  $X \rightarrow Y$ , which means that if we find all of  $X$  items in the market basket (denoted by a transaction), then we have a good chance of finding  $Y$ . Before delving into generation of "Association Rules", we need some basic definitions. Let  $I = \{I_1, I_2, \dots, I_n\}$ : a set of items. Transaction  $t$  is defined as  $I = \{S_1, S_2, \dots, S_m\}$  where  $\{S_i \in I\}$  and  $m \leq n$ . Then transaction Database  $T$  is defined to be a set of transactions  $T = \{t_1, t_2, \dots, t_n\}$ . Therefore, each transaction  $t$ , contains a set of items in  $I$ . Given these definitions, now, we can define an association rule as:

$$X \rightarrow Y, \text{ where } X, Y \in I, \text{ and } X \cap Y = \emptyset$$

for example, we might have:  $X=\{\text{milk, bread}\}$  and  $Y=\{\text{cereal, sugar, butter}\}$ .  $X$  (item set of 2 elements) and  $Y$  (item set of 3 elements) are called the left-hand-side and right-hand-side of an association rule respectively. Therefore, in a nutshell, we can say that an association rule is a pattern that states when  $X$  occurs, then  $Y$  occurs with certain probability. We need few more definitions before defining two important metrics ("support" and "confidence") for an association rule. An *itemset* is a collection of

one or more items. For example an itemset of  $Y=\{\text{cereal, sugar, butter}\}$  contains 3 elements. A *support count* (denoted by  $\theta$ ) is frequency of occurrence of an itemset in all transactions. for example

$$\theta(\{I_1, I_2\}) = 3$$

means that among all transactions,  $I_1$  and  $I_2$  appear together in only 3 transactions.

Now, we can define "support" and "confidence" metrics for an association rule. In data mining association rules for set of transactions, two important metrics are defined and used:

- Support: is the fraction of transactions that contain an itemset. For example if  $\theta(\{I_1, I_2\}) = 3$  and total number of transactions is 24, then support of  $\{I_1, I_2\}$  is defined as  $\frac{3}{24}$ . In frequent itemset generation, a support threshold is used. So for a given association rule:  $X \rightarrow Y$ , support is defined as fraction of transactions that contain both  $X$  and  $Y$ . So, we may write this as

$$\text{support} = \frac{\theta(\{I_1, I_2\})}{\#\text{of transactions}}$$

- Confidence: for a given association rule:  $X \rightarrow Y$ , confidence basically measures how often items in  $Y$  appear in transactions that contain  $X$ . In association rules generation, a confidence threshold is used. Confidence can be expressed as:

$$\text{confidence} = \frac{\theta(Y)}{\theta(X)}$$

Therefore the goal is to find all association rules that satisfy the user-specified minimum support (minimum-support) and minimum confidence (minimum-confidence). These two parameters may be passed to limit the number of frequent set generation and finally limit emitting the association rules. In summary, we can say that an association rule (denoted by  $X \rightarrow Y$ ) is about relationships between two disjoint itemsets  $X$  and  $Y$  and it presents the pattern when  $X$  occurs,  $Y$  also occurs in transactions.

In our solution, we will not limit frequent sets by minimum-support, but our algorithm will calculate the confidence of every association rule generated.

Our spark solution finds all frequent item sets, generates all proper association rules, and finally it calculates the confidence of generated association rules. Calculating "support" is pretty straightforward: divide frequency of an itemset by the total number of all transactions.

### 7.6.1 MapReduce Algorithm

MapReduce algorithm workflow is presented below. The algorithm is comprised of two MapReduce phases:

- Mapreduce Phase-1: mappers convert transactions to patterns and reducers find frequent patterns
- MapReduce Phase-2: mappers convert frequent patterns into sub patterns and finally reducers generate association rules and their associated confidence.

### 7.6.2 Input

Our input is a set of transactions (one transaction per line). For example, for testin, we will use the following input (comprised of 3 transactions):

```
a,b,c  
a,b,d  
b,c  
b,c
```

### 7.6.3 Spark Implementation

The spark implementation follows the two-phase MapReduce algorithm presented. First, we present all high-level steps, and then each step is presented in detail.

Chapter 7 : Market Basket Analysis  
MapReduce Algorithm: Generate Association Rules

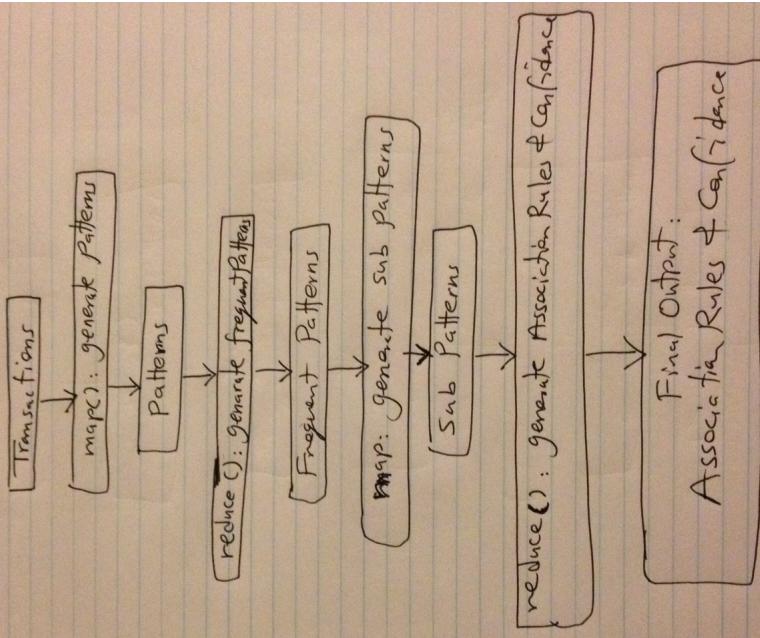


Figure 7.2: Generating Association Rules

### 7.6.3.1 High Level Steps

We present Spark solution in a single Java driver class. This is possible due to availability of high-level abstraction of Spark API. Overall, Spark's API has a higher abstraction layer than Mapreduce/hadoop and that is why we can put 2 phases of Mapreduce algorithms in a single Java class.

#### Listing 7.8: High Level Steps

```
1 // STEP-0: import required classes and interfaces
2 public class FindAssociationRules {
3
4     static JavaSparkContext createJavaSparkContext() {...}
5     static List<String> toList(String transaction) {...}
6     static List<String> removeOneItem(List<String> list, int i) {...}
7
8     public static void main(String[] args) throws Exception {
9         // STEP-1: handle input parameters
10        // STEP-2: create a Spark context object
11        // STEP-3: read all transactions from HDFS and create the first RDD
12        // STEP-4: generate frequent patterns (map() phase 1)
13        // STEP-5: combine/reduce frequent patterns (reduce phase 1)
14        // STEP-6: generate all sub-patterns (map() phase 2)
15        // STEP-7: combine sub-patterns
16        // STEP-8: generate association rules (reduce() phase 2)
17        System.exit(0);
18    }
19 }
```

STEP-4 and STEP-5 solve the first phase of MapReduce and is illustrated below ( [7.3](#)).

STEP-6, STEP-7, and STEP-8 solve the second phase of MapReduce and is illustrated below ( [7.4](#)).

### 7.6.3.2 STEP-0: Import required classes and interfaces

#### Listing 7.9: STEP-0: import required classes and interfaces

```
1 // STEP-0: import required classes and interfaces
2 import java.util.List;
3 import java.util.ArrayList;
4 import scala.Tuple2;
5 import scala.Tuple3;
6 import org.apache.spark.api.java.JavaRDD;
```

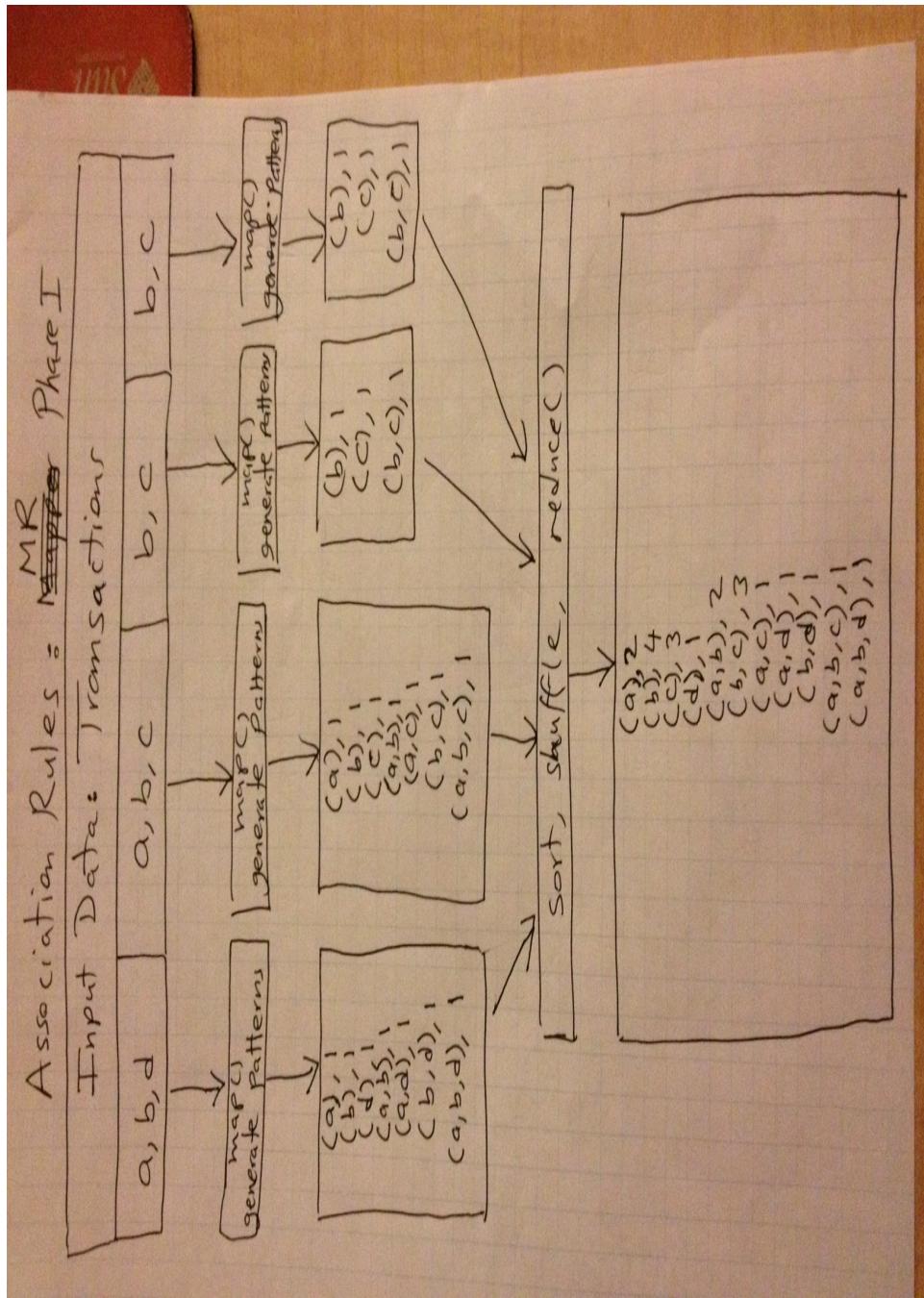


Figure 7.3: Generating Association Rules: MR Phase I

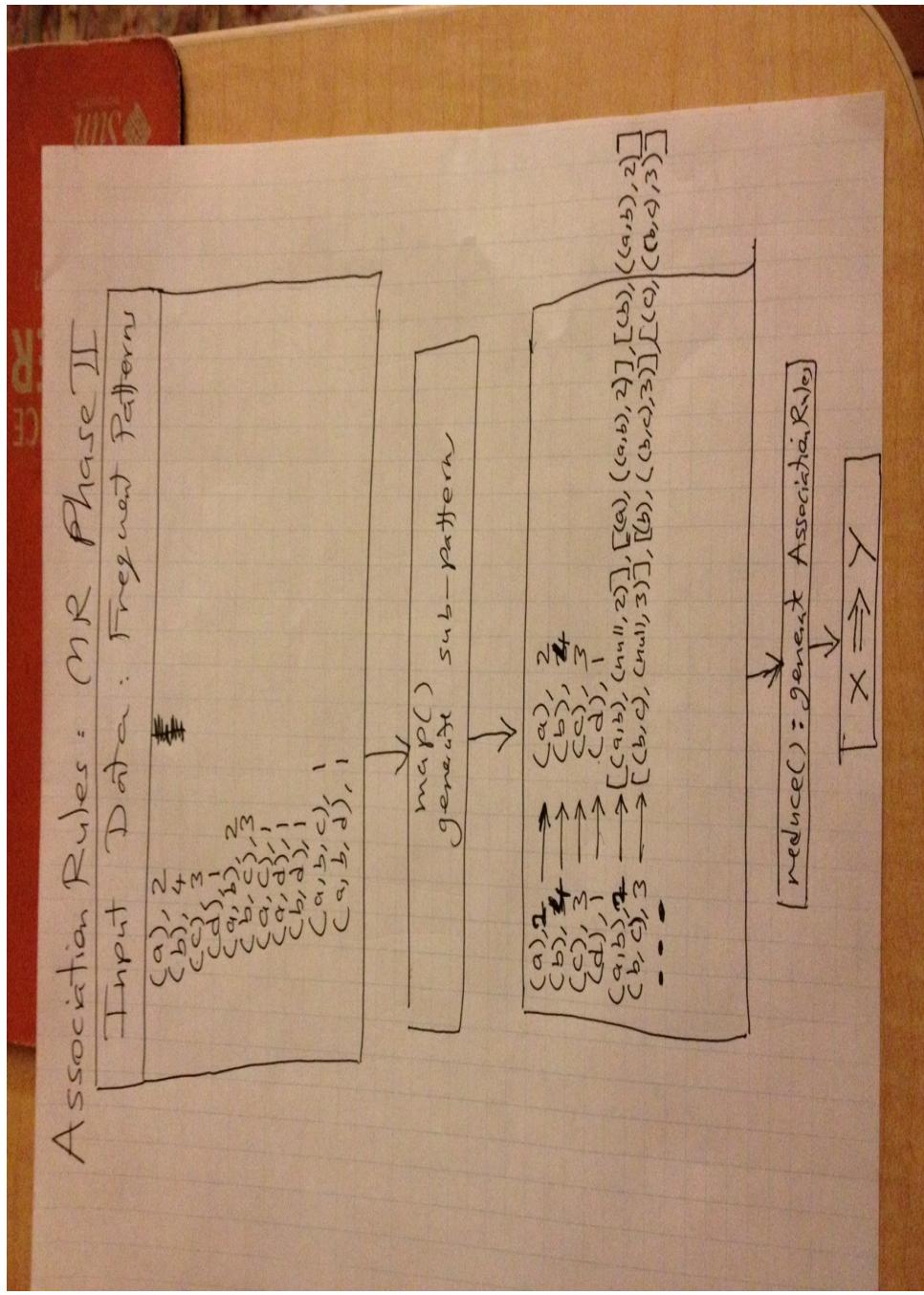


Figure 7.4: Generating Association Rules: MR Phase I

```

7 import org.apache.spark.api.java.JavaPairRDD;
8 import org.apache.spark.api.java.JavaSparkContext;
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.Function;
12 import org.apache.spark.api.java.function.Function2;
13 import org.apache.spark.SparkConf;

```

---

### 7.6.3.3 Create Spark Context Object

To create RDDs, you need a `JavaSparkContext` object. Here, I have hard-coded the YARN's resource manager as `myserver100`. For production deployments, you should read these values from a configuration file. This method of creation of `JavaSparkContext` object is more suitable when your run Spark applications on YARN. If you are going to use Spark cluster (without using YARN), then you should pass "`spark master URL`" (such as: `spark://<spark-master-node>:<port>`) to `JavaSparkContext`'s constructor.

**Listing 7.10:** Create Spark Context Object

```

1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     conf.set("yarn.resourcemanager.hostname", "myserver100");
4     conf.set("yarn.resourcemanager.scheduler.address", "myserver100:8030");
5     conf.set("yarn.resourcemanager.resource-tracker.address", "myserver100:8031");
6     conf.set("yarn.resourcemanager.address", "myserver100:8032");
7     conf.set("mapreduce.framework.name", "yarn");
8     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
9     conf.set("spark.executor.memory", "1g");
10    JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "FindAssociationRules", conf);
11    return ctx;
12 }

```

---

### 7.6.3.4 Utility List Functions

The `toList()` accepts a transaction (items separated by ",") and returns list of items. The `removeOneItem()` removes a single item from a given list and returns a new list with that item removed (used for generating the LHS of association rules).

### **Listing 7.11: Utility List Functions**

```
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 }
```

#### **7.6.3.5 STEP-1: handle input parameters**

This step reads the transactions file parameter supplied by command line.

### **Listing 7.12: STEP-1: handle input parameters**

```
1 // STEP-1: handle input parameters  
2 if (args.length < 1) {  
3     System.err.println("Usage: FindAssociationRules <transactions>");  
4     System.exit(1);  
5 }  
6 String transactionsFileName = args[0];
```

#### **7.6.3.6 STEP-2: create a Spark context object**

This step creates a `JavaSparkContext` object, which is suitable for running Spark applications on YARN. If you want to run your application on a Spark cluster (rather than YARN), then you may use the following Spark API:

```
public JavaSparkContext(String master,  
                        String appName)
```

Parameters:

master - Cluster URL to connect to (e.g. spark://host:port)  
appName - A name for your application, to display on the cluster web UI

#### **Listing 7.13: STEP-2: create a Spark context object**

```
1 // STEP-2: create a Spark context object
2 JavaSparkContext ctx = createJavaSparkContext();
```

#### **7.6.3.7 STEP-3: Read Transactions from HDFS**

This step reads all transactions from an HDFS file and creates the first RDD (`JavaRDD<String>`), where each item is a transaction.

#### **Listing 7.14: STEP-3: read all transactions from HDFS**

```
1 // STEP-3: read all transactions from HDFS and create the first RDD
2 JavaRDD<String> transactions = ctx.textFile(transactionsFileName, 1);
3 transactions.saveAsTextFile("/rules/output/1");
```

For debugging purposes, we write the intermediate data to HDFS. Here, we just emit all raw transactions.

```
# hadoop fs -cat /rules/output/1/part*
a,b,c
a,b,d
b,c
b,c
```

#### **7.6.3.8 STEP-4: generate frequent patterns**

This step generates all patterns for a given transaction. For example, given a transaction with 3 items: "a, b, c", the mappers emit the following (K,V) pairs:

hadoop fs -cat /rules/output/1/part*	
Key	Value
a	1
b	1
c	1
a,b	1
a,c	1
b,c	1
a,b,c	1

Here is the definition of the mapper, which generates all patterns. Typically, in market basket analysis, you might select patterns of length two or three. Selecting all patterns might be very time consuming process. Here to generate all patterns, we use a recursive function:

```
List<List<String>> Combination.findSortedCombinations(List<String>)

For example, findSortedCombinations(List<a, b, c, d>) generates:

[
  [],
  [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]
]
```

Note that combinations are always sorted, for example, for list of 2 items {a, b}, we always create [a, b] (we never create [b, a]). Having sorted item sets enable us to avoid duplicates. Definition of the **Combination** class is given at the end of this chapter.

**Listing 7.15:** STEP-4: generate frequent patterns

```

1   // STEP-4: generate frequent patterns
2   // PairFlatMapFunction<T, K, V>
3   // T => Iterable<Tuple2<K, V>>
4   JavaPairRDD<List<String>, Integer> patterns =
5       transactions.flatMapToPair(new PairFlatMapFunction<
6           String,          // T
7           List<String>,    // K
8           Integer          // V
9       >() {
10      public Iterable<Tuple2<List<String>, Integer>> call(String transaction) {
11          List<String> list = toList(transaction);
12          List<List<String>> combinations = Combination.findSortedCombinations(list);
13          List<Tuple2<List<String>, Integer>> result =
14              new ArrayList<Tuple2<List<String>, Integer>>();
15          for (List<String> combList : combinations) {
16              if (combList.size() > 0) {
17                  result.add(new Tuple2<List<String>, Integer>(combList, 1));
18              }
19          }
20      }
21  });
22  patterns.saveAsTextFile("/rules/output/2");

```

---

Here is the ouput of STEP-4:

```
# 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)
```

#### 7.6.3.9 STEP-5: combine/reduce frequent patterns

This step implements the `reduceByKey()` function for Phase-1 of MapReduce algorithm. It finds all unique frequent patterns and their associated frequency among all transactions.

**Listing 7.16:** STEP-5: combine/reduce frequent patterns

```

1   // STEP-5: combine/reduce frequent patterns
2   JavaPairRDD<List<String>, Integer> combined =
3       patterns.reduceByKey(new Function2<Integer, Integer, Integer>() {
4           public Integer call(Integer i1, Integer i2) {
5               return i1 + i2;
6           }
7       });
8       combined.saveAsTextFile("/rules/output/3");
9
10  // now, we have: patterns(K,V)
11  //      K = pattern as List<String>
12  //      V = frequency of pattern
13  // now given (K,V) as (List<a,b,c>, 2) we will
14  // generate the following (K2,V2) pairs:
15  //
16  //  (List<a,b,c>, T2(null, 2))
17  //  (List<a,b>, T2(List<a,b,c>, 2))
18  //  (List<a,c>, T2(List<a,b,c>, 2))
19  //  (List<b,c>, T2(List<a,b,c>, 2))
```

Here is the ouput of STEP-5:

```
# 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)
```

### 7.6.3.10 STEP-6: generate all sub-patterns

This step creates all sub patterns, which are needed for creation of association rules. Given a frequent pattern of

$(K = List < A_1, A_2, \dots, A_n >, V = Frequency),$

we create the following sub-patterns (K2, V2) as:

$(K2 = List < A_1, A_2, \dots, A_n >, V2 = Tuple2(null, Frequency))$   
 $(K2 = List < A_1, A_2, \dots, A_{n-1} >, V2 = Tuple2(K, Frequency))$   
 $(K2 = List < A_1, A_2, \dots, A_{n-2}, A_n >, V2 = Tuple2(K, Frequency))$   
 ...

For example given (K,V) as (List(a,b,c), 2) we will generate the following (K2,V2) pairs:

Sub Patterns (K2,V2)	
K2	V2
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)

**Listing 7.17:** STEP-6: generate all sub-patterns

```

1   // STEP-6: generate all sub-patterns
2   // PairFlatMapFunction<T, K, V>
3   // T => Iterable<Tuple2<K, V>>
4   JavaPairRDD<List<String>, Tuple2<List<String>, Integer>> subpatterns =
5       combined.flatMapToPair(new PairFlatMapFunction<
6           Tuple2<List<String>, Integer>, // T
7           List<String>, // K
8           Tuple2<List<String>, Integer> // V
9       >() {
10      public Iterable<Tuple2<List<String>, Tuple2<List<String>, Integer>>>
11          call(Tuple2<List<String>, Integer> pattern) {
12              List<Tuple2<List<String>, Tuple2<List<String>, Integer>>> result =
13                  new ArrayList<Tuple2<List<String>, Tuple2<List<String>, Integer>>>();
14              List<String> list = pattern._1;
15              Integer frequency = pattern._2;
16              result.add(new Tuple2(list, new Tuple2(null, frequency)));
17              if (list.size() == 1) {
18                  return result;
19              }

```

```

20
21      // pattern has more than one items
22      // result.add(new Tuple2(list, new Tuple2(null,size)));
23      for (int i=0; i < list.size(); i++) {
24          List<String> sublist = removeOneItem(list, i);
25          result.add(new Tuple2(sublist, new Tuple2(list, frequency)));
26      }
27      return result;
28  }
29 };
30 subpatterns.saveAsTextFile("/rules/output/4");

```

---

Here is the ouput of STEP-6:

```
# 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))
```

#### 7.6.3.11 STEP-7: combine sub-patterns

This step groups sub patterns by key using Spark's `groupByKey()` method.

##### **Listing 7.18:** STEP-7: combine sub-patterns

```
1 // STEP-7: combine sub-patterns
2 JavaPairRDD<List<String>, Iterable<Tuple2<List<String>, Integer>>>
3     rules = subpatterns.groupByKey();
4     rules.saveAsTextFile("/rules/output/5");
```

Here is the output of STEP-7:

```
# 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)])
```

#### 7.6.3.12 STEP-8: Generate Association Rules

Now, given a frequent pattern and all sub patterns, we will be able to generate all association rules and their associated "confidence" value. This step is implemented by `JavaPairRDD.map()` function.

##### **Listing 7.19:** STEP-8: generate association rules

```
1 // STEP-8: generate association rules
2 // Now, use (K=List<String>, V=Iterable<Tuple2<List<String>, Integer>>)
3 // to generate association rules
4 // JavaRDD<R> map(Function<T,R> f)
5 // Return a new RDD by applying a function to all elements of this RDD.
6 JavaRDD<List<Tuple3<List<String>, List<String>, Double>>> assocRules =
7     rules.map(new Function<
```

```

8     Tuple2<List<String>, Iterable<Tuple2<List<String>, Integer>>, // T: input
9     List<Tuple3<List<String>, List<String>, Double>> // R: output
10    // T: input
11    // R: ( ac => b, 1/3): T3(List(a,c), List(b), 0.33)
12    // ( ad => c, 1/3): T3(List(a,d), List(c), 0.33)
13    >() {
14        public List<Tuple3<List<String>, List<String>, Double>>
15            call(Tuple2<List<String>, Iterable<Tuple2<List<String>, Integer>> in) {
16                List<Tuple3<List<String>, List<String>, Double>> result =
17                    new ArrayList<Tuple3<List<String>, List<String>, Double>>();
18                List<String> fromList = in._1;
19                Iterable<Tuple2<List<String>, Integer>> to = in._2;
20                List<Tuple2<List<String>, Integer>> toList =
21                    new ArrayList<Tuple2<List<String>, Integer>>();
22                Tuple2<List<String>, Integer> fromCount = null;
23                for (Tuple2<List<String>, Integer> t2 : to) {
24                    // find the "count" object
25                    if (t2._1 == null) {
26                        fromCount = t2;
27                    }
28                    else {
29                        toList.add(t2);
30                    }
31                }
32                // Now, we have the required objects for generating association rules:
33                // "fromList", "fromCount", and "toList"
34                if (toList.isEmpty()) {
35                    // no output generated, but since Spark does not
36                    // like null objects, we will fake a null object
37                    return result; // an empty list
38                }
39            }
40            // now using 3 objects: "from", "fromCount", and "toList",
41            // create association rules:
42            for (Tuple2<List<String>, Integer> t2 : toList) {
43                double confidence = (double) t2._2 / (double) fromCount._2;
44                List<String> t2List = new ArrayList<String>(t2._1);
45                t2List.removeAll(fromList);
46                result.add(new Tuple3<fromList, t2List, confidence));
47            }
48        }
49        return result;
50    }
51 });
52 assocRules.saveAsTextFile("/rules/output/6");

```

Here is the output of STEP-8 (final output): each output entry depicts list of  $\text{Tuple3}(X, Y, \text{confidence})$  where  $X \rightarrow Y$  and "confidence" (as a double value).

```
# hadoop fs -cat /rules/output/6/part*
[[[a, b], [d], 0.5), ([a, b], [c], 0.5)]
[]
```

```

[[[c],[b],1.0), ([c],[a],0.333333333333333)]
[[[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],0.333333333333333)]
[]
[[[a, c],[b],1.0)]
[[[a, d],[b],1.0]]
[[[b],[a],0.5), ([b],[d],0.25), ([b],[c],0.75)]

```

### 7.6.3.13 YARN Script for Spark

**Listing 7.20:** High Level Steps

```

1 # cat run_assoc_rules.sh
2 #!/bin/bash
3 export JAVA_HOME=/usr/java/jdk7
4 export HADOOP_HOME=/usr/local/hadoop2.4.0
5 export SPARK_HOME=/usr/local/spark-1.0.0
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 MP=/mp/spark_mahmoud_examples
10 export MY_JAR=$MP/mp.jar
11 export SPARK_JAR=$SPARK_HOME/assembly/target/spark-assembly-1.0.0-hadoop2.4.0.jar
12 INPUT=/data/data_mining_transactions.txt
13 prog=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   $MY_JAR $INPUT

```

### 7.6.4 Creating Item Sets From Transactions

Given a transaction  $T = \{I_1, I_2, \dots, I_n\}$ , to create item sets for a given transaction  $T$  (comprised of a set of items), we have a simple POJO class (**Combination**) to generate all combinations of sorted item sets. The **Combination** class is defined below.

### **Listing 7.21:** Combination Class

```
1 import java.util.List;
2 import java.util.ArrayList;
3 import java.util.Arrays;
4 import java.util.Collection;
5 import java.util.Collections;
6
7 public class Combination {
8
9     public static <T extends Comparable<? super T>> List<List<T>>
10        findSortedCombinations(Collection<T> elements) {...}
11
12    public static <T extends Comparable<? super T>> List<List<T>>
13        findSortedCombinations(Collection<T> elements, int n) {...}
14
15    public static void main(String[] args) throws Exception {
16        test();
17    }
18
19    public static void test() throws Exception {
20        List<String> list = Arrays.asList("a", "b", "c", "d");
21        System.out.println("list=" + list);
22        List<List<String>> comb = findSortedCombinations(list);
23        System.out.println(comb.size());
24        System.out.println(comb);
25    }
26}
```

Method definitions are listed below.

### **Listing 7.22:** Combination Class: findSortedCombinations(List)

```
1 /**
2  * Will return combinations of all sizes...
3  * If elements = { a, b, c }, then findCollections(elements) will return:
4  * { [], [a], [b], [c], [a, b], [a, c], [b, c], [a, b, c] }
5  *
6  */
7 public static <T extends Comparable<? super T>> List<List<T>>
8    findSortedCombinations(Collection<T> elements) {
9    List<List<T>> result = new ArrayList<List<T>>();
10   for (int i = 0; i <= elements.size(); i++) {
11       result.addAll(findSortedCombinations(elements, i));
12   }
13   return result;
14 }
```

### **Listing 7.23:** Combination Class: findSortedCombinations(List, n)

```

1  /**
2  * If elements = { a, b, c }, then findCollections(elements, 2) will return:
3  * { [a, b], [a, c], [b, c] }
4  *
5  */
6  public static <T extends Comparable<? super T>> List<List<T>>
7      findSortedCombinations(Collection<T> elements, int n) {
8      List<List<T>> result = new ArrayList<List<T>>();
9      if (n == 0) {
10          result.add(new ArrayList<T>());
11          return result;
12      }
13
14      List<List<T>> combinations = findSortedCombinations(elements, n - 1);
15      for (List<T> combination: combinations) {
16          for (T element: elements) {
17              if (combination.contains(element)) {
18                  continue;
19              }
20
21              List<T> list = new ArrayList<T>();
22              list.addAll(combination);
23
24              if (list.contains(element)) {
25                  continue;
26              }
27
28              list.add(element);
29              // sort items not to duplicate the items, for example: (a, b, c) and
30              // (a, c, b) might become different items to be counted if not sorted
31              Collections.sort(list);
32              if (result.contains(list)) {
33                  continue;
34              }
35              result.add(list);
36          }
37      }
38      return result;
39  }

```

---

# Chapter 8

## Common Friends

### 8.1 Introduction

Given a social network with tens of millions of users, the purpose of this chapter is to implement a MapReduce program to identify "common friends" among all pairs of users. Let  $U$  be a set of all users:  $\{U_1, U_2, \dots, U_n\}$ . Then the goal is to find common friends for every pair of  $(U_i, U_j)$  where  $i \neq j$ .

For finding common friends, we provide three solutions:

- Classic MapReduce/Hadoop using Primitive data types
- Classic MapReduce/Hadoop using Custom data types
- Spark/Hadoop using RDDs

These days most social network sites (such as Facebook, hi5, LinkedIn) offer services to help you share messages, pictures, and videos among friends. Some sites even offer video chat services to help you connect with friends. By definition, friend is a person whom one knows, likes, and trusts. For example, Facebook has a list of friends, and friends are a bi-directional thing on Facebook. If I'm your friend, you're mine too. Typically social networks (such as Facebook) pre-compute calculations when they can to reduce the processing time of requests. One common processing request is the "You and Mary (as your friend) have 185 friends in common" feature. When you visit someone's profile, you see a list of friends that you have in common. This

list doesn't change frequently so it had to be wasteful to recalculate it every time you visited the profile. There are many way to find out the common friends between users of a social network. There are at least two possible solutions:

1. use a caching strategy and save the common friends in cache (such as Redis or Memcache)
2. use MapReduce to calculate everyone's common friends once a day and store those results

We will focus on a MapReduce solution for finding common friends.

## 8.2 Input

We prepare input as a set of records, where each record has the following format:

$$<person><,><friend_1><friend_2>\dots<friend_N> \quad (8.1)$$

where  $<friend_1><friend_2>\dots<friend_N>$  are friends of the  $<person>$ . Note that in real projects/applications, each person/friend will be identified as a unique user-id. A very simple and complete example of input will be as follows.

```
100, 200 300 400 500 600
200, 100 300 400
300, 100 200 400 500
400, 100 200 300
500, 100 300
600, 100
```

In this example, user 500 has two friends identified by user-ids 100 and 300 and user identified by 600 has only one friend: user 100.

### 8.3 Common Friends Algorithm

Let  $\{A_1, A_2, \dots, A_m\}$  be a set of friends for User<sub>1</sub> and  $\{B_1, B_2, \dots, B_n\}$  be a set of friends for User<sub>2</sub>, then common friends of User<sub>1</sub> and User<sub>2</sub> can be defined as the intersection (common elements) of these two sets. The POJO solution is given below:

**Listing 8.1:** Common Friends Algorithm

```
1 import java.util.Set;
2 import java.util.TreeSet;
3
4 public class CommonFriends {
5
6     // user1friends = {A1, A2, ..., Am}
7     // user2friends = {B1, B2, ..., Bn}
8     public static Set<Integer> intersection(Set<Integer> user1friends,
9                                              Set<Integer> user2friends) {
10        if ((user1friends == null) || (user1friends.isEmpty())) {
11            return null;
12        }
13
14        if ((user2friends == null) || (user2friends.isEmpty())) {
15            return null;
16        }
17
18        // both sets are non-null
19        if (user1friends.size() < user2friends.size()) {
20            return intersect(user1friends, user2friends);
21        }
22        else {
23            return intersect(user2friends, user1friends);
24        }
25    }
26
27    private static Set<Integer> intersect(Set<Integer> smallSet,
28                                           Set<Integer> largeSet) {
29        Set<Integer> result = new TreeSet<Integer>();
30        // iterate on small set to improve performance
31        for (Integer x : smallSet) {
32            if (largeSet.contains(x)) {
33                result.add(x);
34            }
35        }
36        return result;
37    }
38 }
```

## 8.4 MapReduce Algorithm

Our MapReduce solution to find "common friends" has a `map()` and `reduce()` functions. The mapper accepts a  $(key_1, value_1)$  pair, where  $key_1$  is a person and  $value_1$  is a list of associated friends of this person. The mapper emits a set of new  $(key_2, value_2)$  pairs where  $key_2$  is a `Tuple2(key1, friendi)` where  $friend_i \in value_1$  and  $value_2$  is the same as  $value_1$  (list of all friends for  $key_1$ ). The reducer's key is a pair of two users ( $User_j, User_k$ ) and value is a list of sets of friends. The `reduce()` function will intersect all sets of friends to find common and mutual friends for  $(User_j, User_k)$  pair.

Here are the `map()` and `reduce()` functions:

**Listing 8.2:** Finding Common Friends `map()` Function

```
1 // key is the person
2 // value is a list of friends for this key=person
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 }
```

Mapper's output keys are sorted and this property will prevent duplicate keys. Note that we assume that friendship is a bi-directional thing: if Alex is a friend of Bob, then Bob is a friend of Alex.

Mappers output sorted keys are generated as:

**Listing 8.3:** Finding Common Friends `buildSortedKey()` Function

```
1 Tuple2 buildSortedKey(person1, person2) {
2     if (person1 < person2) {
3         return Tuple2(person1, person2)
4     }
5     else {
6         return Tuple2(person2, person1)
7     }
8 }
```

The `reduce()` function finds the common friends for every pair of users by intersecting all associated friends in between.

#### **Listing 8.4:** Finding Common Friends reduce() Function

```
1 // key = Tuple2(person1, person2)
2 // value = List {List_1, List_2, ..., List_M}
3 //   where each List_i = { set of unique user id's }
4 reduce(key, value) {
5   outputKey = key;
6   outputValue = intersection (List_1, List_2, ..., List_M);
7   emit (outputKey, outputValue);
8 }
```

### **8.4.1 MapReduce Algorithm in Action**

To understand our MapReduce algorithm, I will show all (key, value) pairs generated by mappers and reducers. Lets apply map() to our sample input:

map(100, (200 300 400 500 600)) will generate:

```
([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)) will generate:

```
([100, 200], [100 300 400])
([200, 300], [100 300 400])
([200, 400], [100 300 400])
```

map(300, (100 200 400 500)) will generate:

```
([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)) will generate:

```
([100, 400], [100 200 300])  
([200, 400], [100 200 300])  
([300, 400], [100 200 300])
```

map(500, (100 300)) will generate:

```
([100, 500], [100 300])  
([300, 500], [100 300])
```

map(600, (100)) will generate:

```
([100, 600], [100])
```

So the mappers generate the following (key, value) pairs:

```
([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])
```

Before these key-value pairs are sent to the reducers, they are grouped by keys:

```

([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])

```

So, reducers will receive the following set of (key, value) pairs:

```

([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]))

```

Finally, reducer(s) will generate:

```

([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], [])

```

Following the generated output, we observe that users 100 and 600 have no common friends. The business case is now that when user-100 visits user-200's profile, we can quickly look up [100, 200] key and see that they have two friends in common, [300, 400]. Meanwhile, the users identified by 100 and 500 have one friend (identified by user-id 300) in common.

## 8.5 Solution 1: Hadoop Implementation using Text

Assuming that each user id is a long data type, in this implementation, we represent "list of friends" as a "string" object. For example, a list of three user ids: 100, 200, 300 can be represented as a string object: "100,200,300." To traverse this kind of list, we do need to tokenize the string object and then retrieve items from it.

<i>Class name</i>	<i>Description</i>
CommonFriendsDriver.java	A driver program to submit Hadoop jobs
CommonFriendsMapper.java	Defines map()
CommonFriendsReducer.java	Defines reduce()
HadoopUtil.java	Utility methods for Hadoop

### 8.5.1 Sample Run for Solution 1

```
$ cat run.sh
#!/bin/bash

export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true

export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.:/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin

CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.2.1.jar

LIB_DIR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/lib
JAR_FILES='find $LIB_DIR -name \'*.jar\''
for jarfile in $JAR_FILES ; do
CLASSPATH=$CLASSPATH:$jarfile
done

echo "CLASSPATH=$CLASSPATH"

export JAR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/common_friends/common_fr
```

```

export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class

$HADOOP_HOME/bin/hadoop fs -rmr /common_friends/output
$HADOOP_HOME/bin/hadoop jar $JAR CommonFriendsDriver /common_friends/input /commo

$ 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-- 1 mahmoud staff 74 2013-09-21 17:07 /common_friends/input/file1.txt
-rw-r--r-- 1 mahmoud staff 20 2013-09-21 17:07 /common_friends/input/file2.txt

$ ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
adding: CommonFriendsDriver.class(in = 2844) (out= 1369)(deflated 51%)
adding: CommonFriendsMapper.class(in = 2485) (out= 1084)(deflated 56%)
adding: CommonFriendsReducer.class(in = 2197) (out= 952)(deflated 56%)
adding: HadoopUtil.class(in = 1797) (out= 839)(deflated 53%)
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 17:17:34 INFO mapred.JobClient: map 50% reduce 0%
...
13/09/21 17:18:19 INFO mapred.JobClient: map 100% reduce 83%
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 output materialized bytes=510
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

$ 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]

```

## 8.6 Solution 2: Hadoop Implementation using ArrayListOfLongsWritable

This implementation represents a "list of longs" as an `ArrayListOfLongsWritable`<sup>1</sup> class (a customized class), which extends `ArrayListOfLongs` and implements `WritableComparable<ArrayListOfLongsWritable>` object. As long as your object (`ArrayListOfLongsWritable`) implements the `Writable`<sup>2</sup> interface, then you can use it as key or value by mappers and reducers. The rule of thumb is that, in Hadoop, you can have a customized objects as key or value as long as it implements the `Writable` interface. This is how hadoop persists objects on Hadoop Distributed File System.

<i>Class name</i>	<i>Description</i>
<code>CommonFriendsDriverUsingList.java</code>	A driver program to submit Hadoop jobs
<code>CommonFriendsMapperUsingList.java</code>	Defines <code>map()</code>
<code>CommonFriendsReducerUsingList.java</code>	Defines <code>reduce()</code>
<code>HadoopUtil.java</code>	Utility methods for Hadoop

### 8.6.1 Sample Run for Solution 2

```
$ cat run.sh
#!/bin/bash

export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true

export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.:~/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
PATH=$PATH:$HADOOP_HOME/bin
PATH=$PATH:$JAVA_HOME/bin
```

---

<sup>1</sup> `edu.umd.cloud9.io.array.ArrayListOfLongsWritable`

<sup>2</sup> `org.apache.hadoop.io.Writable` (a serializable object which implements a simple, efficient, serialization protocol, based on `DataInput` and `DataOutput`). Note that any key or value type in the Hadoop MapReduce framework implements this interface.

```

export PATH

CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.2.1.jar

LIB_DIR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/lib
JAR_FILES='find $LIB_DIR -name '*.jar'
for jarfile in $JAR_FILES ; do
CLASSPATH=$CLASSPATH:$jarfile
done

echo "CLASSPATH=$CLASSPATH"

export JAR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/common_friends_using_lists.jar
export CLASSPATH=$CLASSPATH:$JAR
export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class
$HADOOP_HOME/bin/hadoop fs -rm /lib/common_friends_using_lists.jar
$HADOOP_HOME/bin/hadoop fs -put $JAR /lib/
#
input=/common_friends_using_lists/input
output=/common_friends_using_lists/output
$HADOOP_HOME/bin/hadoop fs -rmr /common_friends_using_lists/output
$HADOOP_HOME/bin/hadoop jar $JAR CommonFriendsDriverUsingList $input $output

$ ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
adding: CommonFriendsDriverUsingList.class(in = 2934) (out= 1413)(deflated 51%)
adding: CommonFriendsMapperUsingList.class(in = 2645) (out= 1127)(deflated 57%)
adding: CommonFriendsReducerUsingList.class(in = 2915) (out= 1286)(deflated 55%)
adding: HadoopUtil.class(in = 1797) (out= 840)(deflated 53%)

```

```

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 CommonFriendsDriverUsingList: inputDir=/common_friends_usin
13/09/23 15:13:33 INFO CommonFriendsDriverUsingList: outputDir=/common_friends_usi
...
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 15:13:39 INFO mapred.JobClient: map 100% reduce 0%
...
13/09/23 15:14:23 INFO mapred.JobClient: map 100% reduce 83%
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

$ 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]

```

## 8.7 Spark Solution

We present a Spark solution by writing a `map()` and `reduce()` functions using Spark's RDDs. Our users data is represented as HDFS text files and each record line has the following format ( $P$  is the person and  $\{F_1, F_2, \dots, F_n\}$  are

direct friends of  $P$ ):

$$P, F_1, F_2, \dots, F_n$$

Our Spark solution will be based on the following map() and reduce() functions. The mapper function is listed below:

```
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);
    }
}
```

The reducer function is listed below:

```
// key = Tuple2<user1,user2>
// values = List<List<user>>
reduce(key, values) {
    commonFriends = intersection(values);
    emit(key, friends);
}
```

Since Spark provides a much higher level API than classic MapReduce/Hadoop, the entire solution is presented in a single Java driver class. First, we present all steps, and then we will go details on the steps. The high-level solution is given as:

**Listing 8.5:** High Level Steps

```

1 // STEP-0: Import required classes and interfaces
2 public class FindCommonFriends {
3     public static void main(String[] args) throws Exception {
4
5         // STEP-1: Check input parameters
6         // STEP-2: Create a JavaSparkContext object
7         // STEP-3: Read input text file from HDFS and create
8         //          the first JavaRDD to represent input file
9         // STEP-4: map JavaRDD<String> into (key, value) pairs,
10        //          where key=Tuple<user1,user2>, value=List of friends
11        // STEP-5: reduce (key=Tuple2<u1,u2>, value=List<friends>) pairs
12        //          into (key=Tuple2<u1,u2>, value=List<List<friends>>)
13        // STEP-6: Find common friends by intersection of all List<List<Long>>
14
15        System.exit(0);
16    }
17
18    // build a sorted Tuple to avoid duplicates
19    static Tuple2<Long,Long> buildSortedTuple(long a, long b) {...}
20 }
```

---

Next, we present details of each steps.

### 8.7.1 STEP-0: Import Required Classes

Spark's main Java classes and interfaces are defined in the `org.apache.spark.api.java` package.

**Listing 8.6:** STEP-0: Import required classes and interfaces

```

1 // STEP-0: Import required classes and interfaces
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.PairFlatMapFunction;
7 import org.apache.spark.api.java.function.FlatMapFunction;
8 import org.apache.spark.api.java.function.Function;
9
10 import java.util.Arrays;
11 import java.util.List;
12 import java.util.ArrayList;
13 import java.util.Map;
14 import java.util.HashMap;
```

---

### 8.7.2 STEP-1: Check Input Parameters

For our program, we need two input parameters: "Spark's master URL" (as `args[0]`) and "input text file stored in HDFS" (as `args[1]`).

**Listing 8.7:** STEP-1: Check input parameters

```
1 // STEP-1: Check input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: FindCommonFriends <master> <file>");
4     System.exit(1);
5 }
6 System.out.println("spark master URL="+args[0]);
7 System.out.println("HDFS input file =" + args[1]);
```

### 8.7.3 STEP-2: Create a JavaSparkContext Object

A `JavaSparkContext` object is created from a "spark master URL". This a factory object to create new RDDs.

**Listing 8.8:** STEP-2: Create a JavaSparkContext object

```
1 // STEP-2: Create a JavaSparkContext object
2 JavaSparkContext ctx = new JavaSparkContext(
3     args[0], // spark master URL
4     "FindCommonFriends",
5     System.getenv("SPARK_HOME"),
6     System.getenv("SPARK_EXAMPLES_JAR"));
```

### 8.7.4 STEP-3: Read Input

We create the first RDD from an input file by using a context object (`JavaSparkContext`) created in STEP-2. The new RDD (`JavaRDD<String>`) represent all records of the input file. Each input record is represented as a Java String object.

**Listing 8.9:** STEP-3: Read Input and Create RDD

```
1 // STEP-3: Read input text file from HDFS and create
2 //          the first JavaRDD to represent input file
3 JavaRDD<String> records = ctx.textFile(args[1], 1);
4
5 // debug
6 List<String> debug0 = records.collect();
7 for (String t : debug0) {
```

```

8     System.out.println("debug0 record=" + t);
9 }
```

### 8.7.5 STEP-4: Apply a Mapper

This step maps every record of  $[P, F_1, F_2, \dots, F_n]$  into set of (key, value) pairs where key=Tuple2( $P, F_i$ ) and value is a list  $[F_1, F_2, \dots, F_n]$ . If size of friends list is one, then no friends list will be created (in this case, there cannot be any friends in common). To implement mapper, we use `JavaRDD.flatMapToPair()` function. This is accomplished by using `PairFlatMapFunction` as (T is input and (K, V) is generated as an output):

```

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

Here is the complete implementation of STEP-4:

**Listing 8.10: STEP-4: Apply a Mapper**

```

1 // STEP-4: map JavaRDD<String> into (key, value) pairs,
2 //           where key=Tuple<user1,user2>, value=List of friends
3 //
4 // PairFlatMapFunction<T, K, V>
5 // T => Iterable<Tuple2<K, V>>
6 JavaPairRDD<Tuple2<Long, Long>, Iterable<Long>> pairs =
7     //          T      K            V
8     records.flatMapToPair(new PairFlatMapFunction<String, Tuple2<Long, Long>, Iterable<Long>>() {
9         public Iterable<Tuple2<Tuple2<Long, Long>, Iterable<Long>>> call(String s) {
10             String[] tokens = s.split(",");
11             long person = Long.parseLong(tokens[0]);
12             String friendsAsString = tokens[1];
13             String[] friendsTokenized = friendsAsString.split(" ");
14             if (friendsTokenized.length == 1) {
15                 Tuple2<Long, Long> key = buildSortedTuple(person, Long.parseLong(friendsTokenized[0]));
16                 return Arrays.asList(new Tuple2<Tuple2<Long, Long>, Iterable<Long>>(key, new ArrayList<Long>()));
17             }
18             List<Long> friends = new ArrayList<Long>();
19             for (String f : friendsTokenized) {
20                 friends.add(Long.parseLong(f));
21             }
22             List<Tuple2<Tuple2<Long, Long>, Iterable<Long>>> result =
23                 new ArrayList<Tuple2<Tuple2<Long, Long>, Iterable<Long>>>();
24             for (Long f : friends) {
25                 Tuple2<Long, Long> key = buildSortedTuple(person, f);
```

```

27         result.add(new Tuple2<Tuple2<Long,Long>, Iterable<Long>>(key, friends));
28     }
29     return result;
30   }
31 });

```

---

To debug STEP-4, we use the `JavaRDD.collect()` function.

```

// 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);
}

```

### 8.7.6 STEP-5: Apply a Reducer

The reducer is applied by calling `JavaPairRDD.groupByKey()` method.

#### **Listing 8.11:** STEP-5: Apply a Reducer

```

1 // STEP-5: reduce (key=Tuple2<u1,u2>, value=Iterable<friends>) pairs
2 //      into (key=Tuple2<u1,u2>, value=Iterable<Iterable<friends>>)
3 JavaPairRDD<Tuple2<Long, Long> , Iterable<Iterable<Long>>> grouped = pairs.groupByKey();
4
5 // debug2
6 List<Tuple2<Tuple2<Long, Long> ,Iterable<Iterable<Long>>>> debug2 = grouped.collect();
7 for (Tuple2<Tuple2<Long,Long>, Iterable<Iterable<Long>>> t2 : debug2) {
8     System.out.println("debug2 key="+t2._1+"\t value="+t2._2);
9 }

```

---

### 8.7.7 STEP-6: Find Common Friends

To find common friends, we intersect all friends for two users by just altering the values. This is accomplished by `JavaPairRDD.mapValues()` method without changing the keys.

#### **Listing 8.12:** STEP-6: Find intersection of all

```

1 // STEP-6: Find intersection of all List<List<Long>>
2 // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
3 // Pass each value in the key-value pair RDD through a map
4 // function without changing the keys;
5 // this also retains the original RDD's partitioning.
6 // Find intersection of all List<List<Long>>
7 JavaPairRDD<Tuple2<Long, Long>, Iterable<Long>> commonFriends =
8     grouped.mapValues(new Function<Iterable<Iterable<Long>>, // input
9                         Iterable<Long> // output
10                        >() {
11     public Iterable<Long> call(Iterable<Iterable<Long>> s) {
12         Map<Long, Integer> countCommon = new HashMap<Long, Integer>();
13         int size = 0;
14         for (Iterable<Long> iter : s) {
15             size++;
16             List<Long> list = iterableToList(iter);
17             if ((list == null) || (list.isEmpty())) {
18                 continue;
19             }
20             for (Long f : list) {
21                 Integer count = countCommon.get(f);
22                 if (count == null) {
23                     countCommon.put(f, 1);
24                 }
25                 else {
26                     countCommon.put(f, ++count);
27                 }
28             }
29         }
30         // if countCommon.Entry<f, count> == countCommon.Entry<f, s.size()
31         // then that is a common friend
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     }
39     return finalCommonFriends;
40 }
41 );

```

---

To debug STEP-6, we used the following code:

```

// 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);
}

```

Finally the following function is used to make sure that we will not have duplicate Tuple2<user1,user2> objects.

```
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);  
    }  
}
```

## 8.8 Sample Run of a Spark Program

### 8.8.1 HDFS Input

```
# 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
```

### 8.8.2 Script to Run Spark Program

```
# cat run_find_common_friends.sh  
source /home/hadoop/conf/env_2.3.0.sh  
export SPARK_HOME=/home/hadoop/spark-1.0.0  
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh  
source $SPARK_HOME/conf/spark-env.sh  
  
# system jars:
```

```

CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop

jars='find $SPARK_HOME -name \'*.jar\''
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
export SPARK_CLASSPATH=$CLASSPATH
export SPARK_MASTER=spark://hnode01319.nextbiosystem.net:7077
#-Dsun.lang.ClassLoader.allowArraySyntax=true
USERS=/data/users_and_friends.txt
OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS FindCommonFriends $SPARK_MASTER $USERS

```

### 8.8.3 Log of Sample Run

We ran this program in a cluster environment by using 3 servers identified by myserver100 (as a Spark master), myserver200, and myserver300. The actual log is trimmed to fit the page.

```

hadoop@hnode01319:~/spark_mahmoud_examples# ./run_find_common_friends.sh
spark master URL=spark://myserver100 :7077
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]

```

# Chapter 9

## Recommendation Engines using MapReduce

This chapter deals with recommendation engines using MapReduce algorithms.

If you are a frequent user of Amazon.com website, you're probably familiar with the lists of related products they feature to help customers find what they are looking for on the site. Amazon.com, presents several such lists on every page, including "Frequently Bought Together" and "Customers Who Bought This Item Also Bought." These features have roots and solutions in recommendation engines and systems. Examples of such applications include recommending books, CDs and other products at Amazon.com. Typically, recommendation engines and systems enhance user experience in the following ways:

- Assist users in finding information
- Reduce search and navigation time
- User satisfaction and return to the site frequently

The purpose of a recommender engine and systems are to predict or recommend:

- items that user have not rated, bought, or navigated to those items yet
- movies or books that a user had not yet considered

- restaurants or locations that a user have not been there

Recommender systems have become extremely common in recent years. A few examples of such systems are:

- Amazon.com and Mybuys.com, providers of recommender systems for buying similar items
- When viewing a product on Amazon.com, the store will recommend additional items based on a matrix of what other shoppers bought along with the currently selected item.
- Tripbase.com, a travel recommendation website, which recommends travel/vacation packages based on your input or preferences.
- Netflix offers predictions of movies that a user might like to watch based on the user's previous ratings and watching habits (as compared to the behavior of other users).

In this chapter, we will address the following problems, which have roots in recommendation engines and systems. For details on recommendation systems, refer to [1], [26], and [9].

- Customers Who Bought This Item Also Bought
- Frequently Bought Together
- Movie Recommendation
- Recommend People Connection

## 9.1 Customers Who Bought This Item Also Bought

Most of the e-commerce vendors, including [Amazon.com](#), use the feature "Customers Who Bought This Item Also Bought" (CWBTIAB) on their web site for recommending books, CDs, and other items. Here we will build a simple recommendation system to answer CWBTIAB question.

Suppose that the [Amazon.com](#) store log contains "user-id" and "bought-item" for each transaction sale. We are going to implement CWBTIAB functionality by MapReduce paradigm. Whenever the item is shown, the store will suggest five other items most often bought by the buyers of the item.

### 9.1.1 Input

We assume that the input is a set of large transactions (a transaction log has a lot of data including, transaction-id, date, price, ...), which have the following fields:

```
<user-id><,><bought-item>
```

### 9.1.2 Expected Output

The recommendation engine should emit the following (key, value) pairs:

- **key:** item
- **value:** list of five most common "Customers Who Bought This Item Also Bought" items

### 9.1.3 MapReduce Solution

We solve CWBTIAB problem with two iterations of MapReduce.

**PHASE-1:** The first MapReduce iteration generates lists of all items bought by the same user. Grouping is done by the Hadoop framework, both mapper and reducer perform an identity function.

**PHASE-2:** The second MapReduce iteration solves co-occurrences problem on list items. It uses the stripes approach and emits only five most common co-occurrences.

Before we discuss PHASE-1 and PHASE-2, I will explain the concept of stripes by a simple example.

### 9.1.3.1 Stripes Design Pattern

Stripes is a design pattern and the main idea behind it is to group together pairs into an associative array. Consider the following (K,V)'s emitted by a mapper (note that in this example, mappers output key is a composite key as Tuple2):

A Mappers Output: Classic	
Key	Value
(k, k <sub>1</sub> )	3
(k, k <sub>2</sub> )	2
(k, k <sub>3</sub> )	4
(k, k <sub>4</sub> )	6
(z, z <sub>1</sub> )	7
(z, z <sub>2</sub> )	8
(z, z <sub>3</sub> )	5

The idea behind stripes approach is that rather than emitting many (K,V)'s, just emit one per stripe: for our example, we will emit only one (key, value) pair per stripe as (k and z are called natural keys):

A Mappers Output: Stripes Approach	
Key	Value
k	{ (k <sub>1</sub> , 3), (k <sub>2</sub> , 2), (k <sub>3</sub> , 4), (k <sub>4</sub> , 6) }
z	{ (z <sub>1</sub> , 7), (z <sub>2</sub> , 8), (z <sub>3</sub> , 5) }

The stripes approach creates an associative array (or a hash table) per natural key and it reduces the number of (K,V)'s emitted per mapper. But the emitted value of mappers becomes a complex object (an associative array), but with Stripes approach there will be less sorting and shuffling of (K,V) pairs.

How does a reducer work in Stripes approach? Reducers perform element-wise sum of associative arrays: consider the following three (K,V)'s for a reducer (as input):

```

K -> { (a, 1), (b, 2), (c, 4), (d, 3) }
K -> { (a, 2), (c, 2) }
K -> { (a, 3), (b, 5), (d, 5) }

```

Then, the generated output will be:

```
K -> { (a, 1+2+3), (b, 2+5), (c, 4+2), (d, 3+5) }
```

or

```
K -> { (a, 6), (b, 7), (c, 6), (d, 8) }
```

The advantages and disadvantages of the Stripes approach is summarized below:

- Advantages

- Since mappers generate less (K,V) pairs than classic approach, then there will be less sorting and shuffling of key-value pairs generated by mappers
- Stripes enable us to utilize combiners and this way it can make better use of combiners (as a local per node optimization)
- Better performance[12]

- Disadvantages

- More difficult to implement (since value of the mappers is an associative array and we have to write a serializer and deserializer for that associative array)
- Underlying object (values generated by mappers as an associative array) more heavyweight
- Fundamental limitation in terms of size of event space (since we are creating an associate array per natural key, we need to make sure that mappers have enough RAM to hold these hash tables).

### 9.1.3.2 MapReduce PHASE-1

This iteration generates lists of all items bought by the same user. Grouping is done by the MapReduce framework on the userID (as a key). Both mapper and reducer perform an identity function. The goal of PHASE-1 is to find all items purchased by all users.

#### Mapper PHASE-1

The mapper is an identity function, which just emits (key,value) pairs as received.

##### **Listing 9.1:** Mapper PHASE-1

```
1 // key = userID
2 // value = item bought by userID
3 map(userID, item) {
4     emit(userID, item);
5 }
```

#### Reducer PHASE-1

The reducer is an identity function, which just groups all items for a single user.

##### **Listing 9.2:** Reducer PHASE-1

```
1 // key = userID
2 // value = list of items bought by userID
3 reduce(userID, items[I1, I2, ..., In]) {
4     emit(userID, items);
5 }
```

### 9.1.3.3 MapReduce PHASE-2

The second MapReduce iteration solves co-occurrences problem on list items. We use the stripes approach, which basically keeps mappers busy by doing more work similar to the combiners. So with the stripes approach, the mappers aggregate as much as data, and then pass them to combiners and reducers. Finally the reducers emit the expected outputs (an item followed by list of five most common co-occurrences).

Since we might be creating so many hash tables (associative arrays) per mappers/reducers, you need to make sure that you have enough memory to hold these data structures. If number of users or items were large enough, then it might not fit in memory. If your memory/RAM is limited, then you might consider creating these associative arrays (has tables) in disk (such a solution is available in MapDB<sup>1</sup>).

## Mapper PHASE-2

The mapper functionality includes combiner functionality as well (since we are doing as much as possible by stripes technique in mappers). The stripes approach minimizes the number of keys generated and therefore the MapReduce's execution framework has less shuffling and sorting to perform. Since we are using non-primitive type for our values (an associative array), more serialization and deserialization will be required.

### **Listing 9.3:** Mapper PHASE-2

```
1 // key = userID
2 // value = list of items bought by userID
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 }
```

## Reducer PHASE-2

The reducer generates "top 5" items for every item in all transactions. The reducer performs an item-wise sum of all stripes (represented as an associative arrays) for a given item.

### **Listing 9.4:** Reducer PHASE-2

---

<sup>1</sup>MapDB (developed by Jan Kotek) provides concurrent Maps, Sets and Queues backed by disk storage or off-heap-memory. It is a fast and easy to use embedded Java database engine. Source: <http://www.mapdb.org>

```
1 // key = item
2 // value = list of stripes[M1, M2, ..., Mn]
3 reduce(item, stripes[M1, M2, ..., Mn]) {
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) + value;
8         }
9     }
10    emit(key, top(5, final))
11 }
```

---

The `top(N, Map<Item, Integer>)` will return N items, which have the top/maximum frequencies for a given associate array.

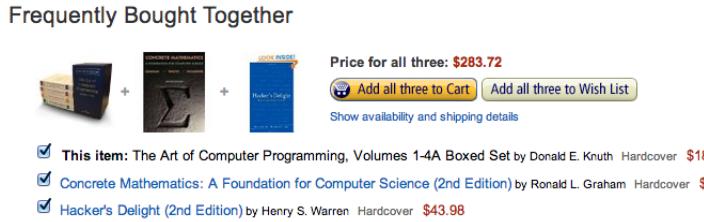


Figure 9.1: Frequently Bought Together

## 9.2 Frequently Bought Together

The purpose of this section is to implement "Frequently Bought Together" (FBT) feature by a MapReduce/Hadoop. FBT is a behavioral targeting technique, which is used for selecting advertisements and products that leverage a user's previous history in order to select and display other relevant products that the user may desire to purchase. For example, Amazon.com uses "Frequently Bought Together" links underneath an item that a buyer is considering.

Suppose you are searching for Donald Knuth's book ("The Art of Computer Programming") on Amazon.com. When you view this book on Amazon.com site, they present a section called "Frequently Bought Together," which lists books bought together. This list also includes the book you searched. Here is the example:

Example: [http://www.amazon.com/Computer-Programming-Volumes-1-4A-Boxed/dp/0321751043/ref=sr\\_1\\_1?ie=UTF8&qid=1400102273&sr=8-1&keywords=knuth](http://www.amazon.com/Computer-Programming-Volumes-1-4A-Boxed/dp/0321751043/ref=sr_1_1?ie=UTF8&qid=1400102273&sr=8-1&keywords=knuth)

How do they do come up with this list of "Frequently Bought Together" for most of their items? This is basically derived by a search for relationships between items (such as books and CDs). Typically, e-commerce sites (such as Amazon.com) gathers data on customer purchasing habits. Using association rule learning, the Amazon.com can determine which products are frequently bought together and use this information for marketing purposes. This is sometimes referred to as variation on market basket analysis, a well-known topic in data mining.

### 9.2.1 Input

Let us assume that we have an input of product sales transactions for all customers. Let's then assume that we have  $n$  Transactions (labeled as  $T_1, \dots, T_n$ ) and  $m$  products (labeled as  $P_1, \dots, P_m$ ), and furthermore let's assume that we have compiled the input as:

<i>Transaction</i>	<i>Purchased Items</i>
$T_1$	$\{P_{1,1}, P_{1,2}, \dots, P_{1,k_1}\}$
$T_2$	$\{P_{2,1}, P_{2,2}, \dots, P_{2,k_2}\}$
...	...
$T_n$	$\{P_{n,1}, P_{n,2}, \dots, P_{n,k_n}\}$

where  $P_{i,j} \in \{P_1, \dots, P_m\}$

and  $k_i$  is the number of items purchased in transaction  $T_i$

So each line of input is a transaction ID, followed by a list of products purchased.

Therefore our goal is to build a hash table such for which the key will be  $P_i$  for  $i = 1, 2, \dots, m$  and value will be list of products purchased together.

For example, if we have the following as an input:

<i>Transaction</i>	<i>Purchased Items</i>
$T_1$	$\{P_1, P_2, P_3\}$
$T_2$	$\{P_2, P_3\}$
$T_3$	$\{P_2, P_3, P_4\}$
$T_4$	$\{P_5, P_6\}$
$T_5$	$\{P_3, P_4\}$

Then our desired output will look like:

<i>Item</i>	<i>Frequently Bought Together</i>
$P_1$	$\{P_2, P_3\}$
$P_2$	$\{P_1, P_3, P_4\}$
$P_3$	$\{P_1, P_2, P_4\}$
$P_4$	$\{P_2, P_3\}$
$P_5$	$\{P_6\}$
$P_6$	$\{P_5\}$

Therefore if a customer is browsing product  $P_3$ , then we will say the *Frequently Bought Together* products are  $P_1$ ,  $P_2$ , and  $P_4$ .

### 9.2.2 MapReduce Solution

The map() will take a single transaction and generate a set of (key, value) pairs to be consumed by reduce(). The mapper pairs the transaction items (i.e., products) as a key and the number of key occurrences as its value in the transaction (for all transactions and without the transaction ID – transaction ID is ignored)

For example, map() for "Transaction 1 (T1)" will emit the following (key, value) pairs:

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1]
```

We should note that if we select the two products in a transaction as a key, there should be incorrect counting for the occurrence of the products in the pairs. NEEDS CLARIFICATION: For example, if transactions T1 and T2 have the same products T1: ( $P_1, P_2, P_3$ ) and T2: ( $P_1, P_3, P_2$ ), which have the same products but in different order.

For transaction T1, map() will generate:

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1]
```

For transaction T2, map() will generate:

```
[<P1, P3>, 1]
[<P1, P3>, 1]
[<P3, P2>, 1]
```

As you observe from `map()` outputs for transactions T1 and T2, therefore, we have a total of six different pairs of products that occur only once respectively, when there should be three different pairs. That is, keys (P2, P3) and (P3, P2) are not the same even though they are. We know that this is not correct. We can avoid this problem if we sort the transaction products in alphabetical order before generating (key, value) pairs. After sorting items in transactions we will have:

```
sorted T1: (P1, P2, P3)
sorted T2: (P1, P2, P3)
```

Now each transaction (T1 and T2) have the following 3 pair of (key, value) pairs:

```
[<P1, P2>, 1]
[<P1, P3>, 1]
[<P2, P3>, 1)
```

That is TWO different pairs of products that occurs twice respectively so that we accumulate the value of the occurrence for these two transactions as follows: [ $i$ P1,  $P2_i$ , 2], [ $i$ P1,  $P3_i$ , 2], ( $j$ P2,  $P3_j$ , 2), which is a correct count of the total number of occurrences.

### 9.2.2.1 Mapper

Mapper reads the input data and creates a list of items for each transaction. For each transaction, its time complexity is  $O(n)$  where  $n$  is the number of items for a transaction. Then, the items in the transaction list are sorted to avoid the duplicated keys; (P2, P3) and (P3, P2) are the same keys. Time complexity of quicksort is  $O(n \log n)$ . Then, the sorted transaction items should be converted to pairs of items as keys, which is a cross operation that allows us to generate cross pairs of the items in the list.

Listing 9.1: MBA `map()` Function

```

// key is transaction ID and ignored here
// value = transaction items (P1, P2, ..., Pm).
map(key, value) {
    (S1, S2, ..., Sm) = sort(P1, P2, ..., Pm);
    // now, we have: S1 < S2 < ... < Sm
    ListOfPair<Si, Sj> = generateCombinations(S1, S2, ..., Sm)
    for ( (Si, Sj) pair : ListOfPair<Si, Sj>) {
        // reducer key is: (Si, Sj)
        // reducer value is: integer 1
        emit([(Si, Sj), 1]);
    }
}

```

---

The `generateCombinations(S1, S2, ..., Sm)` generates all combinations between any two items in a given transaction. For example, `generateCombinations(S1, S2, S3, S4)` will return the following pairs:

```

(S1, S2)
(S1, S3)
(S1, S4)
(S2, S3)
(S2, S4)
(S3, S4)

```

Finally, `map()` will output the following (*key, value*) pairs:

```

<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

```

### 9.2.2.2 Reducer

The "Frequently Bought Together" algorithm for Reducer is illustrated below. The reducer is to sum up the number of values per reducer key. Thus, its time complexity is  $O(v)$  where  $v$  is the number of values per key.

Listing 9.2: MBA reduce() Function

```
// key is in form of (Si, Sj)
// value = List<integer>, where each element is an integer number
reduce(key, value) {
    int sum = 0;
    for (int i : List<integer>) {
        sum += i;
    }
    emit(key, sum);
}
```

### 9.2.2.3 Reducer Output

The reducer will create the following output format:

$\langle P_i, P_j \rangle, N$

where  $N$  is number of transaction, where products  $P_i$  and  $P_j$  have been purchased together. The higher  $N$  indicates that there is close relationship between these two products. Now, with this output, we can create the desired hash table, where keys are in  $\{P_1, P_2, \dots, P_m\}$ .

For our input, the reducer will generate the following output:

```
<P1, P2> 1
<P1, P3> 1
<P2, P3> 3
<P2, P4> 1
<P3, P4> 2
<P5, P6> 1
```

## 9.3 Recommend People Connection

In this section, we provide a complete Spark-based MapReduce algorithm to recommend people connection. These days, there are lots of social network sites (Facebook, Instagram, LinkedIn, Pinterest...). One common feature in social network site is to recommend that people connect. For example, the "People you may know" feature from LinkedIn, allows members to consider who they might want to link with. The basic idea is this: if Alex is a friend of Bob and Alex is a friend of Barbara (Alex is a common friend of Bob and Barbara, but Bob and Barbara do not know each other) then the social network system should recommend Bob to connect with Barbara and vice versa. Therefore, the key point is that if two people have a set of mutual friends, but they are not friends, then the MapReduce solution should recommend them to be connected to each other.

Friendship among all users can be expressed as a graph. For our simple example, we can show graph as:

In mathematics, a graph is an ordered pair  $G = (V, E)$  comprising a set  $V$  of vertices or nodes together with a set  $E$  of edges or lines, which are 2-element subsets of  $V$  (i.e., an edge is related with two vertices, and the relation is represented as an unordered pair of the vertices with respect to the particular edge). This type of graph may be described precisely as undirected and simple. Our assumption is that friendship between all people can be represented in an undirected graph (if person  $A$  is a friend with person  $B$ , then  $B$  is a friend with  $A$ ). For our MapReduce solutions, we assume that the friendships are undirected: if  $A$  is a friend of  $B$  then  $B$  is also a friend of  $A$ . Most social networks (such as Facebook and LinkedIn) use bidirectional friendships, while twitter's friendship is directional. Therefore, we assume that our graph is undirected.

In our case, the graph is an ordered pair  $G = (V, E)$  where

- $V$  - finite set of people (users of social network)
- $E$  - is a binary relation on  $V$  called the edge set, whose elements are called friendship

From a graph theory perspective, for each person or member of a specific social network, who is  $2 - \text{degree}$  reachable from person  $A$ , we count how

## Friendship Connection

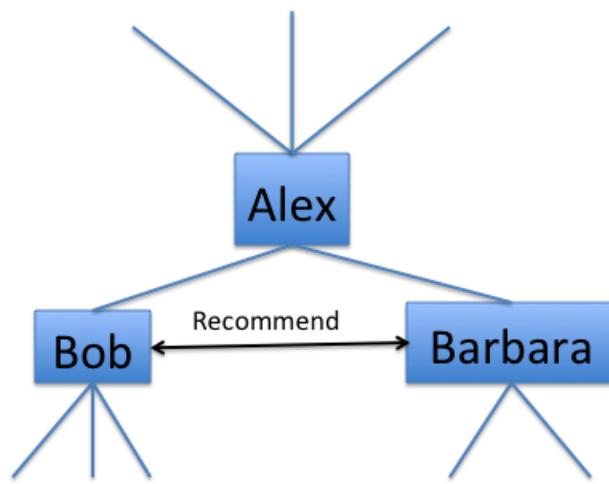


Figure 9.2: Friendship Graph

many distinct paths (with 2 connecting edges) exist between this person and person  $A$ . Rank this list in terms of the number of paths and show the top 10 persons that person  $A$  should connect with. We will show that how we can use MapReduce solution to compute this  $top - 10$  connection list for every person. Therefore, the goal is to pre-compute (as a batch job) the top 10 recommended persons for every member of a social network.

The friendship-recommendation problem can be stated as: For every person  $X$ , we determine a list of person  $X_1, X_2, \dots, X_{10}$  which is the top 10 persons that person  $X$  has common friends with.

### 9.3.1 Input

The social network graph is generally very sparse. Here we assume the input record is an adjacency list sorted by name. Therefore, every line of input will be a member ID (such as a user ID) followed by a list of immediate friends identified by  $F_1, F_2, \dots, F_n$

X    F1    F2    ...    Fn

where  $F_1 < F_2 < \dots < F_n$

For example, for three members, we have an adjacency list sorted by name (here we use names instead of IDs to better comprehend the input):

```
alex => bob, jane, dave, terry
bob => alex, dan, jeff, phil
terry => alex, jeb, russel
```

We will read our unput from HDFS as:

```
# 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
```

User 1 is friends with all, so we should not have any recommendation for this user. On the other hand, User 3 is friends with Users 1 and 2, therefore we can recommend Users 4,5,6,7, and 8 to User 3 since  $\{4,5,6,7,8\}$  are mutual friends of Users 1 and 2.

### 9.3.2 Output

The output of our Spark program will be provided in the following format:

```
<USER><:><F(M: [I1, I2, I3, ...]), ...>
```

where

F is "Recommended friend to USER"

M is the number of mutual friends

I1, I2, I3, ... are the ids of mutual friends

For our sample input, the expected output will be:

```
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]),
```

Note that User 1 does not have any friend recommendation, since he is friends with everyone. The output can be verified from the graph.

### 9.3.3 MapReduce Solution

This section provides a general MapReduce solution without using any specific framework implementations. To solve "Recommend People Connection" problem, we just need a mapper and a reducer outlined below. The mapper identifies direct friends and possible future friends (which will be recommended).

**Listing 9.5:** Recommend People Connection: Mapper

```
1 // key = person (as Long for userID)
2 // value = friends = List<userID> = direct friends of person as {F1, F2, F3, ...}
3 map(key, friends) {
4
5     // emit all direct friendships
6     for (friend : friends) {
7         // -1 denotes direct friendship
8         directFriend = Tuple2(friend, -1);
9         emit(key, directFriend);
10    }
11
12    // emit all possible friendships, where
13    // they have mutual friend, person (as a key)
14    for (int i = 0; i < friends.size(); i++) {
15        for (int j = i + 1; j < friends.size(); j++) {
16            // possible friend 1
17            possibleFriend1 = Tuple2(friends.get(j), person);
18            emit(friends.get(i), possibleFriend1);
19            // possible friend 2
20            possibleFriend2 = Tuple2(friends.get(i), person);
21            emit(friends.get(j), possibleFriend2);
22        }
23    }
24 }
```

Reducer is presented below. The reducer finds mutual friends between the key (as a person), and the value (which is a list of `Tuple2<Long, Long>`). If any of `values` has a mutual friend (denoted by special long value as -1), then we do not make the recommendation since they are already friends. Finally, we prepare a formatted output, which might be read by other programs for further processing.

**Listing 9.6:** Recommend People Connection: Reducer

```
1 // key = person
2 // values = possible recommendations as List<Tuple2<userID1, userID2>>
3 reduce(key, values) {
4 }
```

```

5   // mutualFriends.key is the recommended friend
6   // mutualFriends.value is the list of mutual friends
7   Map<Long, List> mutualFriends = new HashMap<Long, List>();
8
9   for (Tuple2<toUser, mutualFriend> t2 : values) {
10     Long toUser = t2.toUser; // t2._1
11     Long mutualFriend = t2.mutualFriend; // t2._2
12     boolean alreadyFriend = (mutualFriend == -1);
13
14     if (mutualFriends.containsKey(toUser)) {
15       if (isAlreadyFriend) {
16         mutualFriends.put(toUser, null);
17       }
18       else if (mutualFriends.get(toUser) != null) {
19         mutualFriends.get(toUser).add(mutualFriend);
20       }
21     }
22     else {
23       if (alreadyFriend) {
24         mutualFriends.put(toUser, null);
25       }
26       else {
27         mutualFriends.put(toUser, List<mutualFriend>)
28       }
29     }
30   }
31
32   String reducerOutput = buildOutput(mutualFriends);
33   emit(key, reducerOutput);
34 }
```

---

The buildOutput() method definition is given below:

**Listing 9.7: buildOutput() Method**

```

1 String buildOutput(Map<Long, List> map) {
2   String output = "";
3   for (Map.Entry<Long, List> entry : map.entrySet()) {
4     String K = entry.getKey();
5     List V = entry.getValue();
6     output += K + " (" + V.size() + ":" + V + ")";
7   }
8   return output;
9 }
```

---

## 9.4 Spark Implementation

Our implementation of MapReduce solution uses Spark-1.0.0 Java API (as a Spark/Hadoop/YARN job). Since Spark provides a higher-level Java API

than MapReduce/Hadoop API, we will present the entire solution in a single Java class (called `SparkFriendRecommendation`), which will include a series of `flatMapToPair()` and `groupBy()` functions. Spark's MapReduce abstraction is based on RDDs<sup>2</sup>. We use `JavaRDD<T>` (to represent a set of objects of type T) and `JavaPairRDD<K,V>` (to represent a set of (K,V) pairs).

First, I will provide a high-level solution in 10 basic steps, and then we will dissect into each step with a proper working Spark code. Note that some of these steps can be merged (to create somewhat complex steps), but for simplicity reasons I did not merge them.

### **Listing 9.8:** SparkFriendRecommendation: High Level Steps

```

1 // STEP-0: import required classes and interfaces
2 public class SparkFriendRecommendation {
3
4     public static void main(String[] args) throws Exception {
5         // STEP-1: handle input parameters
6         // STEP-2: create a Spark's context object
7         // STEP-3: read hdfs input text and create the first RDD
8         // STEP-4: implement map() function
9         // STEP-5: implement reduce() function
10        // STEP-6: generate desired final output
11        System.exit(0);
12    }

```

The following helper methods are used in our Spark solution:

### **Listing 9.9:** SparkFriendRecommendation: Helper Methods

```

1     static String buildRecommendations(Map<Long, List<Long>> mutualFriends) {
2         // body of buildRecommendations()
3     }
4
5     // convenient methods to create Tuple2<long, long>
6     static Tuple2<Long,Long> T2(long a, long b) {
7         return new Tuple2<Long,Long>(a, b);
8     }
9
10    // convenient methods to create Tuple2<long, Tuple2<long, long>>
11    static Tuple2<Long,Tuple2<Long,Long>> T2(long a, Tuple2<Long,Long> b) {
12        return new Tuple2<Long,Tuple2<Long,Long>>(a, b);
13    }

```

---

<sup>2</sup>In Spark, an RDD (Resilient Distributed Dataset) is the basic abstraction for data representation. Represents an immutable, partitioned collection of elements that can be operated on in parallel.

### 9.4.1 STEP-0: Import Required Classes

This step imports required classes and interfaces. We just utilize two important packages from Spark:

- `org.apache.spark.api.java`  
This package defines RDDs (JavaRDD, JavaPairRDD, and JavaSparkContext)
- `org.apache.spark.api.java.function`  
This package defines transformation functions (Function and PairFlatMapFunction)

**Listing 9.10:** STEP-0: Import Required Classes

```
1 // STEP-0: import required classes and interfaces
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;
```

### 9.4.2 STEP-1: Handle Input Parameters

We read 2 input parameters: "spark master URL" and users data (as HDFS text file), which contains users and its associated friends.

**Listing 9.11:** STEP-1: Handle Input Parameters

```
1 // STEP-1: handle input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: SparkFriendRecommendation <master> <file>");
4     System.exit(1);
5 }
6 String sparkMasterURL = args[0];
7 String hdfssInputFile = args[1];
```

### 9.4.3 STEP-2: Create Spark's Context Object

A JavaSparkContext<sup>3</sup> object is created by using spark master URL. This object is used to create first RDD. JavaSparkContext returns JavaRDDs and works with Java collections such as List<T> and Iterable<T>. There are several ways to create JavaSparkContext object, for details refer to Spark's Java API.

**Listing 9.12:** STEP-2: create a Spark's context object

```
1 // STEP-2: create a Spark's context object
2 JavaSparkContext ctx = new JavaSparkContext(
3     sparkMasterURL,
4     "SparkFriendRecommendation",
5     System.getenv("SPARK_HOME"),
6     System.getenv("SPARK_EXAMPLES_JAR"));
```

Note that, we created an instance of a JavaSparkContext object by connecting to a Spark cluster. If you are running Spark on a Hadoop/-YARN cluster (without starting an Spark cluster), then you should create JavaSparkContext object as (assuming that your YARN resource manager server is "server100"):

```
SparkConf conf = new SparkConf();
conf.set("yarn.resourcemanager.hostname", "server100");
conf.set("yarn.resourcemanager.scheduler.address", "server100:8030");
conf.set("yarn.resourcemanager.resource-tracker.address", "server100:8031");
conf.set("yarn.resourcemanager.address", "server100:8032");
conf.set("mapreduce.framework.name", "yarn");
conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
conf.set("spark.executor.memory", "7g");
JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "myprogram", conf);
```

### 9.4.4 STEP-3: Read HDFS Input File

JavaSparkContext returns the first JavaRDD<String> object, which represents all records from the HDFS Input File. I use JavaRDD.collect() for debugging purposes to make sure we have the correct input data.

---

<sup>3</sup>org.apache.spark.api.java.JavaSparkContext

### **Listing 9.13: STEP-3: Read HDFS Input File**

```
1 // STEP-3: read hdfs text and create the first RDD
2 JavaRDD<String> records = ctx.textFile(hdfsInputFile, 1);
3
4 // debug0
5 List<String> debug1 = records.collect();
6 for (String t : debug1) {
7     System.out.println("debug1 record=" + t);
8 }
```

#### **9.4.5 STEP-4: Implement map() Function**

This step implements the map() algorithm we presented earlier as part of the MapReduce algorithm. The mapper converts each record of

```
<person><TAB><friend1><,><friend2><,><friend3><,>...
```

into (key, value) pairs where key is the <person> and value a Tuple2<user1, user2> signifying direct friendship or possible future friendship. The mapper is implemented by JavaRDD.flatMapToPair() function. This implementation is accomplished by providing call() method as:

```
JavaPairRDD<K2, V2> flatMapToPair(PairFlatMapFunction<T, K2, V2> f)
// Return a new RDD by first applying a function to all
// elements of this RDD, and then flattening the results.
```

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

where

T is an input parameter/record  
(K,V) pairs are generated from T

Here is the detail of mapper implementation:

### Listing 9.14: STEP-4: implement map() function

```
1  // STEP-4: implement map() function
2  // flatMapToPair
3  //   <K2, V2> JavaPairRDD<K2, V2> flatMapToPair(PairFlatMapFunction<T, K2, V2> f)
4  //   Return a new RDD by first applying a function to all elements of this RDD,
5  //   and then flattening the results.
6  //
7  // PairFlatMapFunction<T, K, V>
8  // T => Iterable<Tuple2<K, V>>
9  JavaPairRDD<Long, Tuple2<Long, Long>> pairs =
10    //          T      K      V
11    records.flatMap(new PairFlatMapFunction<String, Long, Tuple2<Long, Long>>() {
12    public Iterable<Tuple2<Long, Tuple2<Long, Long>>> call(String record) {
13      // record=<person><TAB><friend1><,><friend2><,><friend3><,>...
14      String[] tokens = record.split("\t");
15      long person = Long.parseLong(tokens[0]);
16      String friendsAsString = tokens[1];
17      String[] friendsTokenized = friendsAsString.split(",");
18
19      List<Long> friends = new ArrayList<Long>();
20      List<Tuple2<Long, Tuple2<Long, Long>>> mapperOutput =
21        new ArrayList<Tuple2<Long, Tuple2<Long, Long>>>();
22      for (String friendAsString : friendsTokenized) {
23        long toUser = Long.parseLong(friendAsString);
24        friends.add(toUser);
25        Tuple2<Long, Long> directFriend = T2(toUser, -1L);
26        mapperOutput.add(T2(person, directFriend));
27      }
28
29      for (int i = 0; i < friends.size(); i++) {
30        for (int j = i + 1; j < friends.size(); j++) {
31          // possible friend 1
32          Tuple2<Long, Long> possibleFriend1 = T2(friends.get(j), person);
33          mapperOutput.add(T2(friends.get(i), possibleFriend1));
34          // possible friend 2
35          Tuple2<Long, Long> possibleFriend2 = T2(friends.get(i), person);
36          mapperOutput.add(T2(friends.get(j), possibleFriend2));
37        }
38      }
39      return mapperOutput;
40    }
41  });
```

To debug STEP 4, we used the following code:

```
// 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);
}
```

#### 9.4.6 STEP-5: Implement reduce() Function

**Listing 9.15:** STEP-5: implement reduce() function

```
1 // STEP-5: implement reduce() function
2 JavaPairRDD<Long, Iterable<Tuple2<Long, Long>>> grouped = pairs.groupByKey();
3
4 // debug3
5 List<Tuple2<Long, Iterable<Tuple2<Long, Long>>> debug3 = grouped.collect();
6 for (Tuple2<Long, Iterable<Tuple2<Long, Long>>> t2 : debug3) {
7     System.out.println("debug3 key="+t2._1+"\t value="+t2._2);
8 }
```

#### 9.4.7 STEP-6: Generate Final Output

The final output is created from aggregating all values grouped by in STEP-5. This step generates final recommendations for possible future friendships. This step uses `JavaPairRDD.mapValues()` method:

```
public <U> JavaPairRDD<K,U> mapValues(Function<V,U> f)
```

Description: Pass each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.

This step converts `Iterable<Tuple2<Long, Long>>` (as input) into a `String` object (as output). Details of implementation for this step is given below.

**Listing 9.16:** STEP-6: generate desired final output

```
1 // STEP-6: generate desired final output
2 // Find intersection of all List<List<Long>>
3 // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
4 // Pass each value in the key-value pair RDD through a map
5 // function without changing the keys;
6 // this also retains the original RDD's partitioning.
7 JavaPairRDD<Long, String> recommendations =
8     grouped.mapValues(new Function< Iterable<Tuple2<Long, Long>>, // input
```

```

9                               String           // final output
10                          >() {
11      public String call(Iterable<Tuple2<Long, Long>> values) {
12
13          // mutualFriends.key = the recommended friend
14          // mutualFriends.value = the list of mutual friends
15          final Map<Long, List<Long>> mutualFriends = new HashMap<Long, List<Long>>();
16          for (Tuple2<Long, Long> t2 : values) {
17              final Long toUser = t2._1;
18              final Long mutualFriend = t2._2;
19              final boolean alreadyFriend = (mutualFriend == -1);
20
21              if (mutualFriends.containsKey(toUser)) {
22                  if (alreadyFriend) {
23                      mutualFriends.put(toUser, null);
24                  } else if (mutualFriends.get(toUser) != null) {
25                      mutualFriends.get(toUser).add(mutualFriend);
26                  }
27              } else {
28                  if (alreadyFriend) {
29                      mutualFriends.put(toUser, null);
30                  } else {
31                      List<Long> list1 = new ArrayList<Long>(Arrays.asList(mutualFriend));
32                      mutualFriends.put(toUser, list1);
33                  }
34              }
35          }
36      }
37      return buildRecommendations(mutualFriends);
38  }
39 });
40 });
41 );

```

---

To debug STEP 6, we used the following code:

```

// 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);
}

```

#### 9.4.8 Convenient Methods

### Listing 9.17: Convenient Methods

```
1 static String buildRecommendations(Map<Long, List<Long>> mutualFriends) {
2     StringBuilder recommendations = new StringBuilder();
3     for (Map.Entry<Long, List<Long>> entry : mutualFriends.entrySet()) {
4         if (entry.getValue() == null) {
5             // already a friend, no need to recommend again!
6             continue;
7         }
8         recommendations.append(entry.getKey());
9         recommendations.append(" (");
10        recommendations.append(entry.getValue().size());
11        recommendations.append(": ");
12        recommendations.append(entry.getValue());
13        recommendations.append("),");
14    }
15    return recommendations.toString();
16 }
17
18 static Tuple2<Long,Long> T2(long a, long b) {
19     return new Tuple2<Long,Long>(a, b);
20 }
21
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 }
```

#### 9.4.9 HDFS Input

```
# 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
```

#### 9.4.10 Script to Run Spark Program

```
# cat run_friends_recommendations2.sh
```

```

#!/bin/bash
source /home/hadoop/conf/env_2.3.0.sh
export SPARK_HOME=/home/hadoop/spark-1.0.0
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
source $SPARK_HOME/conf/spark-env.sh

# system jars:
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop

jars='find $SPARK_HOME -name \'*.jar\''
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
export SPARK_CLASSPATH=$CLASSPATH
export SPARK_MASTER=spark://myserver100:7077
#-Dsun.lang.ClassLoader.allowArraySyntax=true
USERS=/data/friends2.txt
OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS SparkFriendRecommendation $SPARK_MASTER

```

#### 9.4.11 Program Run Log

The log is trimmed and edited to fit the page.

```

# ./run_friends_recommendations2.sh
...
14/06/02 12:46:50 WARN spark.SparkConf: null jar passed to SparkContext constructo
14/06/02 12:46:51 INFO slf4j.Slf4jLogger: Slf4jLogger started
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])
```

## Content-Based Recommendation: Movies

Have you ever asked how does Netflix (or similar web sites) create movie or book recommendation to its users? There must be some kind of magic algorithm to do this kind of recommendation, right? Netflix even offered one million dollars for finding the optimal solution for movie recommendations[18]. Content-based recommendation systems, such as Netflix for movies, examine properties of items (such as movies) recommended. For example, if a user has watched a lot of **action** movies, then the recommendation system will suggest movies, which are categorized as **tt "action"** movies.

Here we present a basic MapReduce solution, which is based on Edwin Chen's blog[6]. Suppose you run an online movie business, and you want to generate movie recommendations. You have a rating system (people can rate movies with 1 to 5 stars), and well assume for simplicity that all of the ratings are stored in a TSV (Tab Separated Value) file in HDFS. After presenting a generic MapReduce solution, we provide a concrete Spark implementation for movie recommendation.

We should note that in content-based recommendation techniques, the more information (such as a domain knowledge and metadata) we have about contents, the algorithms become more complex (more variables are involved), but recommendations become more accurate and reasonable. For example, for movie recommendations the recommendation system needs to know the

additional metadata such as actors, directors, and producers. For our examples here, we will limit our algorithms only to "ratings of movies". Typically, in a recommendation-system application there are two classes of entities users and items. In a movie recommendation, users are people, who watched and rated movies and items are movies. The input data in the following section will show the relationship (rating a movie from 1 to 5) between users and movies.

The purpose of this section is to present a three-phase MapReduce solution for movie recommendations:

- PHASE-1: Find total number of raters for each movie
- PHASE-2: For every pair of movies A and B, find all the people who rated both A and B.
- PHASE-3: Find correlation of every two related movies

## 10.1 Input

We assume that the raw input data in the following format:

```
user  movie  rating
```

Where *user* is a user id, who rated the *movie* and *rating* is a an integer number from 1 to 5. An example is given below:

Users and Movies		
User	Movie	Rating
User1	Movie1	1
User1	Movie2	2
User1	Movie3	3
User2	Movie1	1
User2	Movie2	2
User2	Movie3	3
User2	Movie5	5
...	...	...

From this input we generate another input, which has the following format (we basically group by movie).

## 10.2 MapReduce PHASE-1

The goal of this phase is to generate the following output.

```
user movie rating numberOfRaters
```

where `numberOfRaters` is the number of people who rated that specific movie. For finding `numberOfRaters`, we write a simple MapReduce job comprised of a mapper and a reducer. Given an input record of `<user><movie><rating>`, the mapper emits  $(K_2, V_2)$  pair, where  $K_2$  is `movie` and  $V_2$  is a `Tuple2(user, rating)`. The reducer finds out the number of times a movie has been rated and then generates our desired output (as defined above).

### Mapper PHASE-1

**Listing 10.1:** Mapper PHASE-1

```
1 map( <user> <movie> <rating>) {
2     K2 = <movie>;
3     V2 = Tuple2(user, rating);
4     emit(K2, V2);
5 }
```

### Reducer PHASE-1

**Listing 10.2:** Reducer PHASE-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 }
```

The output of PHASE-1 will be used as in input to PHASE-2.

## 10.3 MapReduce PHASE-2 and PHASE-3

MapReduce solution will calculate how similar pairs of movies are, so that if someone watches *The Lion King*, you can recommend films like *Toy Story*. So how should you define the similarity between two movies? One way is to use their correlation[29]:

- For every pair of movies A and B, find all the people who rated both A and B.
- Use these ratings to form a Movie A vector and a Movie B vector.
- Calculate the correlation (mutual relation of two or more movies – here, the correlation is a way to measure how associated or related two movies are) between these two vectors.
- Whenever someone watches a movie, you can then recommend the movies most correlated with it.

The first two steps will be done by MapReduce-Phase-2 (will generate a Movie A vector and a Movie B vector). The last two steps will be done by MapReduce-Phase-3 (to calculate the correlation between every two-related movies).

To get all pairs of co-rated movies, we'll join our input (all movie ratings) against itself (similar to SQL's joining a table against itself). To join ratings input against itself, the mapper will collect all movies per user and then reducer will generate unique combinations of all movies rated by a user.

To understand this better, lets create a sample input for two users (note that this input is generated by PHASE-1 – each record contains: `user`, `movie`, `rating`, and `numberOfRaters`):

Users and Movies			
User	Movie	Rating	Number of Raters
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
...	...	...	...

## 10.4 MapReduce-Phase-2 Mapper

The map() will accept one line of input as:

```
<user> <movie> <rating> <numberOfRaters>
```

and emit a (key, value) pair as:

```
key: <user>
value: <movie> <rating> <numberOfRaters>
```

Here is the definition of map() function:

### Listing 10.3: Mapper PHASE-2

```

1 // key is <user>
2 // value = <movie> <rating> <NumOfRaters>
3 map(key, value) {
4     String[] tokens = value.split("\t");
5     movie = tokens[0];
6     rating = tokens[1];
7     numberofRaters = tokens[2];
8     Tuple3<String, Integer, Integer> t3 = Tuple3(movie, rating, numberofRaters);
9     emit(key, t3);
10 }
```

The mapper will generate the following output for our sample input (to be consumed by reducers):

Mappers Output	
KEY	VALUE
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>
...	...

## 10.5 MapReduce-Phase-2 Reducer

The reduce() will accept a user (as a key) and list of Tuple3(movie, rating, numOfRaters) and will emit unique combinations of all movies rated by user.

Therefore, input for reducer will look like:

Output of Sort and Shuffle	
KEY	VALUE(s)
User1	[<Movie1, 1, 10>, <Movie2, 2, 20>, <Movie3, 3, 30>]
User2	[<Movie1, 1, 10>, <Movie2, 2, 20>, <Movie3, 3, 30>, <Movie5, 5, 50>]
...	...

Here is the definition of reduce() function, which will join output of "Sort and Shuffle" phase against itself (join will be on the "user" key).

**Listing 10.4:** Reducer PHASE-2

```

1 // key is user (generated by map())
2 // value = List{ Tuple3(<movie> <rating> <NumOfRaters>) }
3 reduce(key, value) {
4     // Generate unique combinations of all movies against each other
5     // make sure not create duplicate entries like (movie1, movie2) and
6     // (movie2, movie1).

```

```

7  // To avoid this, we will create (movie1, movie2)
8  // where movie1 < movie2
9  List[ Tuple2( Tuple3(<movie1> <rating1> <numOfRaters1>),
10           Tuple3(<movie2> <rating2> <numOfRaters2>) ) ]
11           list = generateUniqueCombinations(value);
12
13  for (Tuple2( Tuple3(<movie1> <rating1> <numOfRaters1>),
14           Tuple3(<movie2> <rating2> <numOfRaters2>) ) pair : list) {
15      m1 = pair._1; // = Tuple3(<movie1> <rating1> <numOfRaters1>)
16      m2 = pair._2; // = Tuple3(<movie2> <rating2> <numOfRaters2>)
17
18      //define the reducer key
19      reducerKey = Tuple2(m1.movie, m2.movie);
20
21      // calculate additional information, which will be needed by correlation
22      int ratingProduct = m1.rating * m2.rating;
23      int rating1Squared = m1.rating * m1.rating;
24      int rating2Squared = m2.rating * m2.rating;
25
26      // define the reducer value
27      reducerValue = Tuple7(m1.rating,
28                            m1.NumOfRaters,
29                            m2.rating,
30                            m2.NumOfRaters,
31                            ratingProduct,
32                            rating1Squared,
33                            rating2Squared);
34      emit(reducerKey, reducerValue);
35  } // end-for-loop
36 }

```

---

Therefore, reducer will generate the following (key, value) pairs:

- Generated from key=User1:

Reducer Output							
KEY	VALUE						
<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>

- Generated from key=User2:

Reducer Output								
KEY	VALUE							
<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>	

## 10.6 MapReduce-Phase-3 Mapper

The mapper is an identity mapper. The map() will accept (key, value) input pairs as:

```

key: Tuple2(<movie1>, <movie2>)
value: Tuple7(<rating1>,
              <numOfRaters1>,
              <rating2>,
              <numOfRaters2>,
              <ratingProduct>,
              <rating1Squared>,
              <rating2Squared>
            )
  
```

The mapper is an identity mapper. It just emits (key, value) as received. Here is the definition of map() function:

### Listing 10.5: Mapper PHASE-3

```

1 // key is Tuple2(<movie1>, <movie2>) and generated by reducer of phase 2
2 // value = Tuple7(<rating1> <numOfRaters1> <rating2> <numOfRaters2>
3 //                  <ratingProduct> <rating1Squared> <rating2Squared>)
4 map(key, value) {
5   emit(key, value);
6 }
  
```

The mapper will generate the following for our sample input (used as an input to reducer of phase 3):

Mapper Output							
KEY	VALUE						
<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>

## 10.7 MapReduce-Phase-3 Reducer

The reduce() will accept (`<movie1>, <movie2>`) as a key and list of Tuple7 (`< rating1 >< numOfRaters1 >, < rating2 >< numOfRaters2 >< ratingProduct >< rating1Squared >< rating2Squared >`) and will emit correlation of every two related movies.

So the input for reducer will be:

Reducer Input							
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>						

Following is the definition of `reduce()`. Recall that we can write the Pearson correlation in the following form:

$$\text{correlation}(X, Y) = \frac{n \sum xy - \sum x \sum y}{\sqrt{n \sum x^2 - (\sum x)^2} \sqrt{n \sum y^2 - (\sum y)^2}}$$

This correlation is "Pearson product-moment correlation coefficient" [?].

### Listing 10.6: Reducer PHASE-3

```

1 // key is Tuple2(<movie1, movie2>)
2 // values = List{ Tuple7(<rating1
3 //                  <numOfRaters1>
4 //                  <rating2>
5 //                  <numOfRaters2>
6 //                  <ratingProduct>
7 //                  <rating1Squared>
8 //                  <rating2Squared>)
9 // }
10 reduce(key, value) {
11     // calculate additional information, which will be needed by correlation
12     int groupSize = value.size(); // length of each vector
13     int dotProduct = 0; // sum of ratingProd
14     int rating1Sum = 0; // sum of rating1
15     int rating2Sum = 0; // sum of rating2
16     int rating1NormSq = 0; // sum of rating1Squared
17     int rating2NormSq = 0; // sum of rating2Squared
18     int maxNumOfumRaters1 = 0; // max of numOfRaters1
19     int maxNumOfumRaters2 = 0; // max of numOfRaters2
20     for (Tuple7(<rating1
21             <numOfRaters1>
22             <rating2>
23             <numOfRaters2>
24             <ratingProduct>
25             <rating1Squared>
26             <rating2Squared>)) : values) {
27         dotProduct += ratingProd;
28         rating1Sum += rating1;
29         rating2Sum += rating2;
30         rating1NormSq += rating1Squared;
31         rating2NormSq += rating2Squared;
32         if (numOfRaters1 > maxNumOfumRaters1) {
33             maxNumOfumRaters1 = numOfRaters1;
34         }
35         if (numOfRaters2 > maxNumOfumRaters2) {
36             maxNumOfumRaters2 = numOfRaters2;
37         }
38     }
39
40     double pearson = calculatePearsonCorrelation(
41         groupSize,
42         dotProduct,
43         rating1Sum,
44         rating2Sum,
```

```

45             rating1NormSq,
46             rating2NormSq);
47
48     double cosine = calculateCosineCorrelation(dotProduct, Math.sqrt(rating1NormSq), Math.sqrt(rating2NormSq));
49     double jaccard = calculateJaccardCorrelation(groupSize, maxNumOfumRaters1, maxNumOfumRaters2);
50     return Tuple3(pearson, cosine, jaccard);
51 }

```

---

The definition of `correlation()` function is given below:

**Listing 10.7: Pearson Correlation Function**

```

1 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 }

```

---

Therefore, reducer will generate similarities of every two movies:

```

<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

```

## 10.8 More Similarity Measures

Note that for the correlation calculation, we used "Pearson product-moment correlation coefficient" [?]. We may use other formulas to calculate correlation. Of course, there are lots of other similarity measures we could use

besides correlation. For example, we may use Cosine Similarity, which is another common vector-based similarity measure and defined as:

Listing 10.1: Cosine Similarity

```
double calculateCosineCorrelation
    (double dotProduct,
     double rating1Norm,
     double rating2Norm) {
    return dotProduct / (rating1Norm * rating2Norm)
}
```

There are other Correlations or Similarity Measures:

- Regularized Correlation
- Jaccard Similarity
- Cosine Similarity
- Euclidean Distance
- Manhattan Distance

Selecting Correlations or Similarity Measures algorithm is domain/problem specific. You do need to investigate to see which algorithm gives you better (pragmatic optimal) answers.

## 10.9 Movie Recommendation in Spark

This section presents Spark implementation for the 3 phases of MapReduce algorithms provided in earlier sections of this chapter. The entire Spark solution is written in a single Java driver class called `MovieRecommendationsWithJoin`. The following are presented for Spark implementation:

- First, we present all high-level steps
- Second, each step is presented in detail with complete Spark code in Java.
- Finally, a sample run is provided using Spark-1.0.0

### 10.9.1 High-Level Solution in Spark

The solution in Spark is presented in 12 steps, which use RDDs. Spark's primary data abstraction and programming model is based on RDDs. In a nutshell, Resilient distributed datasets (RDDs) are immutable data structures, which have the following properties:

- Created from HDFS files or "parallelized" arrays
- Can be transformed with `map()` and `filter`
- Can be cached across parallel operations such as `reduce`, `collect`, and `foreach`.

### 10.9.2 High-Level Solution: All Steps

The driver class, `MovieRecommendationsWithJoin`, is presented below.

**Listing 10.8:** High-Level Solution: All Steps

```
1 //STEP-0: import required classes and interfaces
2 public class MovieRecommendationsWithJoin {
3
4     public static void main(String[] args) throws Exception {
5         //STEP-1: handle input parameters
6         //STEP-2: create a Spark's context object
7         //STEP-3: read HDFS file and create the first RDD
8         //STEP-4: find who has seen and rated this movie
9         //STEP-5: group moviesRDD by movie
10        //STEP-6: find number of raters per movie and then create (K,V) pairs as
11        //        usersRDD = <K=user, V=<movie,rating,numberOfRaters>>
12        //STEP-7: join usersRDD against itself to find all (movie1, movie2) pairs
13        //        joinedRDD = usersRDD.join(usersRDD);
14        //        joinedRDD = (user, T2((m1,r1,n1), (m2,r2,n2)) )
15        //STEP-8: remove duplicate (movie1, movie2) pairs.
16        //        note that (movie1, movie2) and (movie2, movie1) are the same
17        //STEP-9: generate all (movie1, movie2) combinations
18        //        The goal of this step is to create (K,V) pairs of
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        //STEP-10: group moviePairs by key(movie1,movie2)
29        //STEP-11: calculate correlation/similarities between every (movie1,movie2)
30        //STEP-12: print final results
```

```

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 }
```

### 10.9.3 STEP-0: Import Required Classes

Most of the classes and interfaces we need are included in the following two packages: `org.apache.spark.api.java` and `org.apache.spark.api.java.function`.

**Listing 10.9:** STEP-0: Import Required Classes

```

1 //STEP-0: import required classes and interfaces
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;
```

### 10.9.4 STEP-1: Handle Input Parameters

Two parameters are needed to run the Spark program: "spark master URL" and "HDFS input file". The "spark master URL" will be used to create a `JavaSparkContext` object.

**Listing 10.10:** STEP-1: Handle Input Parameters

```

1 //STEP-1: handle input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: MovieRecommendationsWithJoin <master> <users-ratings>");
4     System.exit(1);
5 }
6
7 String sparkMaster = args[0];
8 String usersRatingsInputFile = args[1];
9 System.out.println("sparkMaster=" + sparkMaster);
10 System.out.println("usersRatingsInputFile=" + usersRatingsInputFile);
```

---

### 10.9.5 STEP-2: Create a Spark's Context Object

This step creates a JavaSparkContext object that returns JavaRDDs and works with Java collections. JavaSparkContext is a factory class, which creates JavaRDD<T> and JavaPairRDD<K,V> objects.

**Listing 10.11:** STEP-2: Create a Spark's Context Object

```
1 //STEP-2: create a Spark's context object
2 JavaSparkContext ctx = new JavaSparkContext(
3     sparkMaster,
4     "MovieRecommendationsWithJoin",
5     System.getenv("SPARK_HOME"),
6     System.getenv("SPARK_EXAMPLES_JAR"));
```

---

### 10.9.6 STEP-3: Read Input File and Create RDD

Using a JavaSparkContext object we read an HDFS input text file and create the first JavaRDD<String> object (set of String type records) , which represents the input file.

```
//STEP-3: read HDFS file and create the first RDD
JavaRDD<String> usersRatings = ctx.textFile(usersRatingsInputFile, 1);
```

After executing STEP-3, the usersRating RDD is a list of:

```
{
    "user1 movie1 1",
    "user1 movie2 2",
    "user1 movie3 3",
    "user2 movie1 1",
    "user2 movie2 2"
    "user2 movie3 3"
    "user2 movie5 5"
    ...
}
```

### 10.9.7 STEP-4: Find Who Has Rated Movies

The `usersRating` RDD is an input to STEP-4. The three steps (STEP-4, STEP-5, and STEP-6) find the "number of raters for every movie". STEP-4 implements a mapper: it accepts a single record (comprised of `user`, `movie`, and `rating`) and create a (key,value) pair of (`movie`, (`user,rating`)). `JavaRDD.mapToPair()` accepts a single input record and creates a `JavaPairRDD<K2,V2>` where `K2=movie` and `V2=Tuple2(user,rating)`.

**Listing 10.12:** STEP-4: Find Who Has Rated Movies

```
1 //STEP-4: find who has seen this movie
2 // <K2, V2> JavaPairRDD<K2, V2> mapToPair(PairFunction<T, K2, V2> f)
3 // Return a new RDD by applying a function to all elements of this RDD.
4 // PairFunction<T, K2, V2>
5 // T => Tuple2<K2, V2>
6 // T = <user> <movie> <rating>
7 // K2 = <movie>
8 // V2 = Tuple2<user, rating>
9 JavaPairRDD<String, Tuple2<String, Integer>> moviesRDD =
10    //          T      K      V
11    usersRatings.mapToPair(new PairFunction<String, String, Tuple2<String, Integer>>() {
12    //          T
13    public Tuple2<String, Tuple2<String, Integer>> call(String s) {
14      String[] record = s.split("\t");
15      String user = record[0];
16      String movie = record[1];
17      Integer rating = new Integer(record[2]);
18      Tuple2<String, Integer> userAndRating = new Tuple2<String, Integer>(user, rating);
19      return new Tuple2<String, Tuple2<String, Integer>>(movie, userAndRating);
20    }
21  });
```

To debug STEP-4, we use `JavaPairRDD.collect()` method.

```
System.out.println("== debug1: moviesRDD: K = <movie>, V = 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);
}
```

After executing STEP-4, the `moviesRDD` is a list of (K,V) pairs:

```
{
```

```

((movie1, (user1, 1)),
((movie2, (user1, 2)),
((movie3, (user1, 3)),
((movie1, (user2, 1)),
((movie2, (user2, 2)),
((movie3, (user2, 3)),
((movie5, (user2, 5))

...
}

}

```

### 10.9.8 STEP-5: Group moviesRDD by Movie

STEP-4 created a JavaPairRDD<K2, V2> where K2=movie and V2=Tuple2(user, rating). This step groups moviesRDD by its key. The result of this step will be:

```

JavaPairRDD<K, V>
where   K=movie
        V=List(Tuple2(user, rating))

```

This step is implemented by JavaPairRDD.groupByKey() method.

#### Listing 10.13: STEP-5: Group moviesRDD by Movie

```

1 //STEP-5: group moviesRDD by movie
2 JavaPairRDD<String, Iterable<Tuple2<String, Integer>>> moviesGrouped = moviesRDD.groupByKey();
3
4 System.out.println("== debug2: moviesGrouped: K = <movie>, V = Iterable<Tuple2<user, rating>> ==");
5 List<
6     Tuple2<String, Iterable<Tuple2<String, Integer>>>
7     > debug2 = 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 }

```

After executing STEP-5, the moviesGrouped is a list of the following (K,V) pairs. Clearly, we can count the number of raters (which is the size of list value per movie) for each movie.

```

{
    (movie1, [(user1, 1), (user2, 1), ...]),
    (movie2, [(user1, 2), (user2, 2), ...]),

```

```

        (movie3, [(user1, 3), (user2, 3), ...]),
        (movie5, [(user2, 5), ...])
    ...
}

```

### 10.9.9 STEP-6: Find Number of Raters per Movie

This step tallies the raters for each movie and generates another JavaPairRDD<K,V> where K=user and V=Tuple3(movie, rating, numberofRaters). This step is implemented by JavaPairRDD.flatMapToPair() method.

**Listing 10.14:** STEP-6: Find Number of Raters per Movie

```

1  //STEP-6: find number of raters per movie and then create (K,V) pairs as
2  //          K = user
3  //          V = Tuple3<movie, rating, numberofRaters>
4  //
5  // PairFlatMapFunction<T, K, V>
6  // T => Iterable<Tuple2<K, V>>
7  JavaPairRDD<String,Tuple3<String, Integer, Integer>> usersRDD =
8      moviesGrouped.flatMapToPair(new PairFlatMapFunction<
9          Tuple2<String, Iterable<Tuple2<String, Integer>>>, // T
10         String, // K
11         Tuple3<String, Integer, Integer>>() { // V
12     public Iterable<Tuple2<String, Tuple3<String, Integer, Integer>>>
13         call(Tuple2<String, Iterable<Tuple2<String, Integer>>> s) {
14             List<Tuple2<String, Integer>> listOfUsersAndRatings =
15                 new ArrayList<Tuple2<String, Integer>>();
16             // now read inputs and generate desired (K,V) pairs
17             String movie = s._1;
18             Iterable<Tuple2<String, Integer>> pairsOfUserAndRating = s._2;
19             int numberofRaters = 0;
20             for (Tuple2<String, Integer> t2 : pairsOfUserAndRating) {
21                 numberofRaters++;
22                 listOfUsersAndRatings.add(t2);
23             }
24
25             // now emit (K, V) pairs
26             List<Tuple2<String, Tuple3<String, Integer, Integer>>> results =
27                 new ArrayList<Tuple2<String, Tuple3<String, Integer, Integer>>>();
28             for (Tuple2<String, Integer> t2 : listOfUsersAndRatings) {
29                 String user = t2._1;
30                 Integer rating = t2._2;
31                 Tuple3<String, Integer, Integer> t3 =
32                     new Tuple3<String, Integer, Integer>(movie, rating, numberofRaters);
33                 results.add(new Tuple2<String, Tuple3<String, Integer, Integer>>(user, t3));
34             }
35             return results;
36         }
37     });

```

---

To debug STEP-6, we use `JavaPairRDD.collect()` method.

```
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);  
}
```

After executing STEP-6, the `usersRDD` is a list of the following (K,V) pairs, which include the number of raters per movie. Here K=user and V=(movie, rating, numberofRatters).

```
{  
    (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)),  
    ...  
}
```

### 10.9.10 STEP-7: Perform Self-Join

This step uses a very powerful feature of Spark API: the `join` operation. Self-join is performed on `usersRDD` as (the join will be done on the "user" key):

```
joinedRDD = usersRDD.join(usersRDD);
```

where both `usersRDD` and `joinedRDD` are `JavaRDD` types. The signature of `join()` is given as:

```

public <W> JavaPairRDD<K,scala.Tuple2<V,W>> join(JavaPairRDD<K,W> other)
Description: Return an RDD containing all pairs of elements with matching
keys in this and other. Each pair of elements will be returned
as a (k, (v1, v2)) tuple, where (k, v1) is in this and (k, v2)
is in other. Performs a hash join across the cluster.

```

Here is the self-join (self-join is needed to find combinations of (movie1, movie2)) implementation by JavaPairRDD.join(JavaPairRDD):

#### Listing 10.15: STEP-7: Perform Self-Join

```

1  //STEP-7: join usersRDD against itself to find all (movie1, movie2) pairs
2  // usersRDD = <K=user, V=<movie,rating,numberOfRaters>>
3  // the result will be joinedRDD = (user, T2((m1,r1,n1), (m2,r2,n2)) )
4  // the join will be done on the "user" key
5  JavaPairRDD<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>>
6      joinedRDD = usersRDD.join(usersRDD);
7  List<
8      Tuple2<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>>
9      > debug5 = joinedRDD.collect();
10     for (Tuple2<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>> t2 : debug5) {
11         System.out.println("debug5 key="+t2._1 + "\t value="+t2._2);
12     }

```

After executing STEP-7, the joinedRDD is a list of the following (K,V) pairs:

```

{
    (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)]),
    ...
}

```

As you can see for each K=user, we have duplicate movie pairs (note that, (movie1, movie2) is the same as (movie2, movie1) pairs). The next step will remove duplicates by filter API.

### 10.9.11 STEP-8: Remove Duplicate (movie1, movie2) Pairs

In previous step, STEP-7, we generated (movie1, movie2) combinations pairs, which have duplicates. This step removes duplicates by using `JavaPairRDD.filter()` method. The `filter()` method is defined as:

```
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.
```

Filter implementation is provided below. The `filter()` function only keeps (movie1, movie2) pairs if `movie1 < movie2` (this is the predicate the we implemented below):

**Listing 10.16:** STEP-8: Remove Duplicate (movie1, movie2) Pairs

```
1 //STEP-8: remove duplicate (movie1, movie2) pairs.
2 // note that (movie1, movie2) and (movie2, movie1) are the same
3
4 // now we have: filteredRDD = (user, T2((m1,r1,n1), (m2,r2,n2)) ) where m1 < m2
5 // to guarantee that we will not have duplicate movie pairs per user
6 JavaPairRDD<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>>
7 filteredRDD = joinedRDD.filter(new Function<
8     Tuple2<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>>,
9     Boolean
10    >() {
11    public Boolean call(Tuple2<String,
12        Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>> s) {
13        // to remove duplicates, make sure that movie1 < movie2 for all (movie1, movie2) pairs
14        Tuple3<String, Integer, Integer> movie1 = s._2._1;
15        Tuple3<String, Integer, Integer> movie2 = s._2._2;
16        String movieName1 = movie1.first();
17        String movieName2 = movie2.first();
18        if (movieName1.compareTo(movieName2) < 0) {
19            return true;
20        }
21        else {
22            return false;
23        }
24    }
25});
```

To debug STEP-8, we use `JavaPairRDD.collect()` method.

```
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>>
    System.out.println("debug55 key="+t2._1 + "\t value="+t2._2);
}

```

After executing STEP-8, the `filteredRDD` is a list of the following (K,V) pairs (without duplicates):

```

{
    (user1, [(movie1, 1, 2), (movie2, 2, 2)]),
    (user1, [(movie1, 1, 2), (movie3, 3, 2)]),
    (user1, [(movie2, 2, 2), (movie3, 3, 2)]),
    ...
}

```

### 10.9.12 STEP-9: Generate All (movie1, movie2) Combinations

For each (movie1, movie2) pair, this step creates some derived data, which will be used for calculating three correlation algorithms (Pearson, Cosine, Jaccard). This is achieved by another transformation by using `JavaPairRDD.mapToPair()`.

**Listing 10.17:** STEP-9: Generate All (movie1, movie2) Combinations

```

1 //STEP-9: generate all (movie1, movie2) combinations
2 // The goal of this step is to create (K,V) pairs of
3 //   K: Tuple2(movie1, movie2)
4 //   V: Tuple7(movie1.rating,
5 //             movie1.numOfRaters,
6 //             movie2.rating,
7 //             movie2.numOfRaters,
8 //             ratingProduct,
9 //             rating1Squared, = movie1.rating * movie1.rating
10 //            rating2Squared = movie2.rating * movie2.rating
11 //            )
12 // PairFunction<T, K, V>
13 // T => Tuple2<K, V>
14 // user attribute will be dropped at this phase.
15 //   K
16 JavaPairRDD<Tuple2<String, String>, Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>>>
17     moviePairs = filteredRDD.mapToPair(new PairFunction<
18         <Tuple2<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>>,           // T
19         Tuple2<String, String>,                                         // K
20         Tuple7<Integer, Integer, Integer, Integer, Integer, Integer> // V

```

```

21     >() {
22     public Tuple2<Tuple2<String, String>, Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>>
23         call(Tuple2<String, Tuple2<Tuple3<String, Integer, Integer>, Tuple3<String, Integer, Integer>>> s) {
24             // String user = s._1; // will be dropped
25             Tuple3<String, Integer, Integer> movie1 = s._2._1;
26             Tuple3<String, Integer, Integer> movie2 = s._2._2;
27             Tuple2<String, String> m1m2Key = new Tuple2<String, String>(movie1.first(), movie2.first());
28             int ratingProduct = movie1.second() * movie2.second(); // movie1.rating * movie2.rating;
29             int rating1Squared = movie1.second() * movie1.second(); // movie1.rating * movie1.rating;
30             int rating2Squared = movie2.second() * movie2.second(); // movie2.rating * movie2.rating;
31             Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer> t7 =
32                 new Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>(
33                     movie1.second(), // movie1.rating,
34                     movie1.third(), // movie1.numberOfRaters,
35                     movie2.second(), // movie2.rating,
36                     movie2.third(), // movie2.numberOfRaters,
37                     ratingProduct,
38                     rating1Squared,
39                     rating2Squared);
40             return new Tuple2<Tuple2<String, String>, Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>>(
41                 );
42             });
}

```

---

### 10.9.13 STEP-10: Group Movie Pairs

This step groups data by movie pairs: (movie1, movie2), which gather required data needed for computing correlations.

```

//STEP-10: group moviePairs by key(movie1,movie2)
JavaPairRDD< Tuple2<String, String>,
              Iterable<Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>>
            > corrRDD = moviePairs.groupByKey();

```

### 10.9.14 STEP-11: Calculate Correlations

This step uses 3 different correlation algorithms to find similarities between every pair of movies. There is no golden rule for selecting a correlation algorithm for a specific application. You should select a correlation algorithm, which fits into your applicatin environment.

#### **Listing 10.18:** STEP-11: Calculate Correlations

```

1  //STEP-11: calculate correlation/similarities between every (movie1,movie2)
2  // coor.key = (movie1,movie2)
3  // coor.value= (pearson, cosine, jaccard) correlations

```

```

4   JavaPairRDD<Tuple2<String, String>, Tuple3<Double, Double, Double>> corr =
5       corrRDD.mapValues(new Function<
6           Iterable<Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>>, // input
7           Tuple3<Double, Double, Double>
8       >() {
9           public Tuple3<Double, Double, Double> call(
10               Iterable<Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>> s) {
11                   return calculateCorrelations(s);
12               }
13           });

```

---

### 10.9.15 STEP-12: Print Final Results

```

//STEP-12: print final results
System.out.println("== Movie Correlations ==");
List<
    Tuple2< Tuple2<String, String>, Tuple3<Double, Double, Double>>
> debug6 = corr.collect();
for (Tuple2<Tuple2<String, String>, Tuple3<Double, Double, Double>> t2 : debug6)
    System.out.println("debug5 key="+t2._1 + "\t value="+t2._2);
}

corr.saveAsTextFile("/movies/output");

```

### 10.9.16 Helper Method: calculateCorrelations()

**Listing 10.19:** Helper Method: calculateCorrelations()

```

1 static Tuple3<Double, Double, Double> calculateCorrelations(
2     Iterable<Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer>> values) {
3     int groupSize = 0; // length of each vector
4     int dotProduct = 0; // sum of ratingProd
5     int rating1Sum = 0; // sum of rating1
6     int rating2Sum = 0; // sum of rating2
7     int rating1NormSq = 0; // sum of rating1Squared
8     int rating2NormSq = 0; // sum of rating2Squared
9     int maxNumOfRaters1 = 0; // max of numOfRaters1
10    int maxNumOfRaters2 = 0; // max of numOfRaters2
11    for (Tuple7<Integer, Integer, Integer, Integer, Integer, Integer, Integer> t7 : values) {
12        // Tuple7(<rating1>: t7._1
13        //         <numOfRaters1>: t7._2
14        //         <rating2>: t7._3
15        //         <numOfRaters2> : t7._4
16        //         <ratingProduct> : t7._5

```

```

17         //      <rating1Squared> : t7._6
18         //      <rating2Squared>): t7._7
19     groupSize++;
20     dotProduct += t7._5; // ratingProduct;
21     rating1Sum += t7._1; // rating1;
22     rating2Sum += t7._3; // rating2;
23     rating1NormSq += t7._6; // rating1Squared;
24     rating2NormSq += t7._7; // rating2Squared;
25     int numOfRaters1 = t7._2;
26     if (numOfRaters1 > maxNumOfumRaters1) {
27         maxNumOfumRaters1 = numOfRaters1;
28     }
29     int numOfRaters2 = t7._4;
30     if (numOfRaters2 > maxNumOfumRaters2) {
31         maxNumOfumRaters2 = numOfRaters2;
32     }
33 }
34
35     double pearson = calculatePearsonCorrelation(
36             groupSize,
37             dotProduct,
38             rating1Sum,
39             rating2Sum,
40             rating1NormSq,
41             rating2NormSq);
42
43     double cosine = calculateCosineCorrelation(dotProduct, Math.sqrt(rating1NormSq), Math.sqrt(rating2NormSq));
44
45     double jaccard = calculateJaccardCorrelation(groupSize, maxNumOfumRaters1, maxNumOfumRaters2);
46
47     return new Tuple3<Double,Double,Double>(pearson, cosine, jaccard);
48 }
```

---

### 10.9.17 Helper Method: calculatePearsonCorrelation()

**Listing 10.20:** Helper Method: calculateCorrelations()

```

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 }
```

---

### 10.9.18 Helper Method: calculateCosineCorrelation()

**Listing 10.21:** Helper Method: calculateCosineCorrelation()

```
1  /**
2   * The cosine similarity between two vectors A, B is
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  }
```

---

### 10.9.19 Helper Method: calculateJaccardCorrelation()

**Listing 10.22:** Helper Method: calculateJaccardCorrelation()

```
1  /**
2   * The Jaccard Similarity between two sets A, B is
3   * |Intersection(A, B)| / |Union(A, B)|
4   */
5  static double calculateJaccardCorrelation(double inCommon,
6                                         double totalA,
7                                         double totalB) {
8      double union = totalA + totalB - inCommon;
9      return inCommon / union;
10 }
```

---

## 10.10 Sample Run of Spark Program

### 10.10.1 HDFS Input

To refresh, we assume that our input data is stored in HDFS as a text file

```
# 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

```

### 10.10.2 Script

```

# cat run_movies.sh
#!/bin/bash
source /home/hadoop/conf/env_2.3.0.sh
export SPARK_HOME=/home/hadoop/spark-1.0.0
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
source $SPARK_HOME/conf/spark-env.sh

# system jars:
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop

jars='find $SPARK_HOME -name \'*.jar\''
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH
export SPARK_CLASSPATH=$CLASSPATH
export SPARK_MASTER=spark://hnode01319.nextbiosystem.net:7077
MOVIES=/movies.txt
OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS MovieRecommendationsWithJoin $SPARK_MA

```

## 10.11 Log of Sample Run

The actual output is edited to fit the page.

```
# ./run_movies.sh
sparkMaster=spark://hnode01319.nextbiosystem.net:7077
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=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])

```

```

debug5 key=User4      value=(Tuple3[Movie3,3,4],Tuple3[Movie4,2,1])
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=User1      value=(Tuple3[Movie2,4,4],Tuple3[Movie3,3,4])
debug5 key=User1      value=(Tuple3[Movie1,3,4],Tuple3[Movie2,4,4])
debug5 key=User1      value=(Tuple3[Movie1,3,4],Tuple3[Movie3,3,4])
==== 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[0.5222329678670935,0.982069984444069,1]
debug5 key=(Movie1,Movie3)      value=Tuple3[0.47140452079103173,0.9422657923052785,1]
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]
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

```

### 10.11.1 Inspecting HDFS Output

```

# hadoop fs -ls /movies/output/
Found 2 items
-rw-r--r--  3 hadoop root,hadoop          0 2014-06-07 23:18 /movies/output/_SUCCESS
-rw-r--r--  3 hadoop root,hadoop      504 2014-06-07 23:18 /movies/output/part-r0000

```

```
# 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])
```

# Chapter 11

## Smarter Email Marketing with Markov Model

### 11.1 Introduction

The goal of this chapter is to show that how Markov Model (in the simplest form known as Markov Chain) can be used in predicting the "next smart email marketing date" to customers based on their transaction history. Given a set of random variables (such as last purchase date by customers), Markov model indicates that the distribution for this variable (last purchase date) depends only on the distribution of the previous state (another last purchase date). We implement a MapReduce solution for smarter email marketing with using Markov Model.

For writing this chapter, I was inspired by the blog "*Smarter Email Marketing with Markov Model*"<sup>1</sup> posted by Pranab Ghosh <sup>1</sup>, and this chapter is based on his blog. For implementation of MapReduce phases of this chapter, I have developed brand new modular Java classes to demonstrate the core values (such as sorting of reduced values by MapReduce's "secondary sort" technique, defining custom partitioner class, and defining "grouping comparator" class). Therefore, given customer's transaction history (`purchase-date`, `amount-purchased`) the goal is to use MapReduce and Markov Chain Model to predict the next date (predictive modeling technique) for sending an email

---

<sup>1</sup><http://pkghosh.wordpress.com/2013/04/15/smarter-email-marketing-with-markov-model/>

marketing to that customer. This is kind of a machine learning algorithm. Typically, a machine learning based solutions consist of two distinct phases:

- Phase-1: build a model by using historical training data.
- Phase-2: make a prediction for next new data using the model built in Phase-1.

## 11.2 Markov Chain in a Nutshell

Let  $S = \{S_1, S_2, S_3, \dots\}$  be a set of finite states and we want to collect the following probabilities:

$$P(S_n | S_{n-1}, S_{n-2}, \dots, S_1)$$

Markov's first-order assumption is the following:

$$P(S_n | S_{n-1}, S_{n-2}, \dots, S_1) \approx P(S_n | S_{n-1})$$

This approximation states the Markov Property: the state of the system at time  $t+1$  depends only on the state of the system at time  $t$ , and Markov second-order assumption is the following:

$$P(S_n | S_{n-1}, S_{n-2}, \dots, S_1) \approx P(S_n | S_{n-1}, S_{n-2})$$

Now, we can express the joint probability using the Markov assumption:

$$P(S_1, S_2, \dots, S_n) = \prod_{i=1}^n P(S_i, S_{i-1})$$

Markov random processes can be summarized as<sup>2</sup>:

- A random sequence has the Markov property if its distribution is determined solely by its current state. Any random process having this property is called a Markov random process.

---

<sup>2</sup>Based on slides by Mehmet Yunus Dnmez: <http://classes.soe.ucsc.edu/cmp264/Fall106/LecHMM.pdf>

- For observable state sequences (state is known from data), this leads to a Markov chain model. This is what we will use in creating a Markov model to predict the next email marketing date.
- For non-observable states, this leads to a Hidden Markov Model (HMM).

Now, we formalize Markov Chain (which we will use in this chapter):  
Markov Chain has 3 components:

1. State Space: a finite set of states  $S = \{S_1, S_2, S_3, \dots\}$
2. Transition Probabilities: a function  $f : S \times S \rightarrow R$  such that  $0 \leq f(a, b) \leq 1$  for all  $a, b \in S$  and  $\sum_{b \in S} f(a, b) = 1$  for every  $a \in S$ .
3. Initial Distribution: a function  $g : S \times R$  such that  $0 \leq g(a) \leq 1$  for every  $a \in S$  and  $\sum_{a \in S} g(a) = 1$ .

Then a Markov Chain is a random process in  $S$  such that

1. At time 0 the state of the chain is distributed according to function  $g$
2. If at time  $t$  the state of chain is  $a$ , then at time  $t + 1$  it will end up at state  $b$  with probability of  $f(a, b)$  for every  $b \in S$ .

Let's illustrate this by an example: let a weather pattern for a city be in 4 states: **sunny**, **cloudy**, **rainy**, **foggy** and further assume that the state does not change for a day. The sum of each row is 1.00.

		Tomorrow's Weather				
		sunny	rainy	cloudy	foggy	
		Today's Weather				
	sunny	0.6	0.1	0.2	0.1	
	rainy	0.5	0.2	0.2	0.1	
	cloudy	0.1	0.7	0.1	0.1	
	foggy	0.0	0.3	0.4	0.3	

Now, we can answer the following questions:

- Given that today is **sunny**, what is the probability that tomorrow is **cloudy** and the day after is **foggy**?

$$\begin{aligned}
& P(S_2 = \text{cloudy}, S_3 = \text{foggy} | S_1 = \text{sunny}) \\
&= P(S_3 = \text{foggy} | S_2 = \text{cloudy}, S_1 = \text{sunny}) \times \\
&\quad P(S_2 = \text{cloudy} | S_1 = \text{sunny}) \\
&= P(S_3 = \text{foggy} | S_2 = \text{cloudy}) \times \\
&\quad P(S_2 = \text{cloudy} | S_1 = \text{sunny}) \\
&= 0.1 \times 0.2 \\
&= 0.02
\end{aligned}$$

- Given that today is `foggy`, what is the probability that it will `rainy` two days from now? (this means that the second day can be any of `sunny`, `cloudy`, `rainy`, `foggy`):

$$\begin{aligned}
& P(S_3 = \text{foggy} | S_1 = \text{foggy}) = \\
& P(S_3 = \text{foggy}, S_2 = \text{sunny} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy}, S_2 = \text{cloudy} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy}, S_2 = \text{rainy} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy}, S_2 = \text{foggy} | S_1 = \text{foggy}) + \\
& = P(S_3 = \text{foggy} | S_2 = \text{sunny}) \times P(S_2 = \text{sunny} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy} | S_2 = \text{cloudy}) \times P(S_2 = \text{cloudy} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy} | S_2 = \text{rainy}) \times P(S_2 = \text{rainy} | S_1 = \text{foggy}) + \\
& P(S_3 = \text{foggy} | S_2 = \text{foggy}) \times P(S_2 = \text{foggy} | S_1 = \text{foggy}) \\
& = 0.1 \times 0.0 + \\
& 0.1 \times 0.4 + \\
& 0.1 \times 0.3 + \\
& 0.3 \times 0.3 \\
& = 0.00 + 0.04 + 0.03 + 0.09 \\
& = 0.16
\end{aligned}$$

So one of the main goal of this chapter is to build the model (the probability transition table), which will define  $f(a, b)$  for all  $a \in S$ . Once we have this model the rest is easy.

### 11.3 Markov Model using MapReduce

We assume that we have historical customer transaction data, which includes `transaction-id`, `customer-id`, `purchase-date`, `amount`. Therefore, each input record will have:

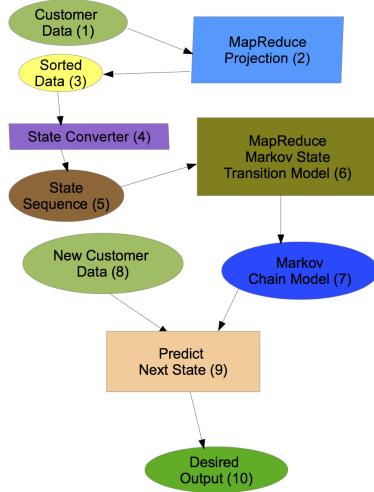


Figure 11.1: Markov Work Flow

`transaction-id, customer-id, purchase-date, amount`

The entire solution involves two MapReduce jobs and a set of Ruby scripts (the Ruby scripts are developed by Pranab Ghosh – will provide pointers for these Ruby scripts as we use them).

The entire workflow is depicted below ([11.1](#)):

Steps are outlined below:

1. We use a script to generate fake Customer Data (1).
2. The MapReduce Prjection (2) accepts Customer Data (1) as an input and generates Sorted Data (3). The Sorted Data (3) is a date-ordered-in-ascending order.
3. The State Converter (4) script accepts the Sorted Data (3) and generates State Sequence (5).

4. The MapReduce Markov State Transition Model (6) accepts State Sequence (5) as an input and generates Markov Chain Model (7). This model enable us to predict the next state.
5. Finally, we use a script, Predict Next State (9), which accepts new Customer Data (8) and Markov Chain Model (7) and gives/predicts the next email date for marketing.

### 11.3.1 MapReduce to Generate Time-ordered Transactions

The goal of this MapReduce phase will be to accept historical customer transaction data and generate the following output for every `customer-id`:

$customerid, (Date_1, Amount_1), (Date_2, Amount_2), \dots (Date_N, Amount_N)$

such that

$$Date_1 \leq Date_2 \leq \dots \leq Date_N$$

Output of MapReduce is sorted on the "purchased-date" by ascending order. After output is created, we will convert `(Date, Amount)` into a two-letter-symbol (this step is done by a script), which stands for a Markov Chain state. Generating sorted output can be accomplished in two ways: each reducer can sort its output by the "purchased-date" by ascending order (here you need enough RAM to hold all your data for sorting) or you can use MapReduce's "secondary sort" (by using the sort capability of MapReduce framework — this way you do not need much RAM at all) to sort data by date. We will present solutions for both cases. The final output generated from this phase will have the following format:

$customerid, State_1, State_2, \dots, State_n$

#### 11.3.1.1 Mapper

Listing 11.1: Time-ordered Transactions map() Function

```

/**
 * @param key is ignored
 * @param value is transaction-id, customer-id, purchase-date, amount
 */
map(key, value) {
    pair(Date, Integer) pair = (value.purchase-date, value.amount);
    emit(value.customer-id, pair);
}

```

---

### 11.3.1.2 Reducer

Listing 11.2: Time-ordered Transactions reduce() Function

```

/**
 * @param key is a customer-id
 * @param values is a list of pairs of Pair(Date, Integer)
 */
reduce(String key, List<Pair<Date, Integer>> values) {
    sortedValues = sortByDateInAscendingOrder(values);
    emit(key, sortedValues);
}

```

---

### 11.3.1.3 Hadoop Solution 1: Time-ordered Transactions

Mappers emit (key, value), where key is a "customer-id" and value is pair of ("purchase-date", "amount"). Data arrive to reducers unsorted. This solution performs sorting of transactions in reducer. In the number of transactions arriving at reducers are too big, then this might cause "out-of-memory" exception in reducers. Our "Hadoop Solution 2" will not have this restriction: rather than sorting data in memory per reducer, we use "secondary sort", which is a feature of MapReduce paradigm for sorting the values of reducers.

Hadoop Solution 1: Implementation Classes	
Class Name	Description
SortInMemoryProjectionDriver	Driver class to submit jobs
SortInMemoryProjectionMapper	Mapper class
SortInMemoryProjectionReducer	Reducer class
DateUtil	Basic Date Utility class
HadoopUtil	Basic Hadoop Utility class

### 11.3.1.4 Hadoop Solution 1: Sample Run

#### 11.3.1.4.1 Hadoop Solution 1: Partial Input

Pranab Ghosh has provided a Ruby script (`buy_xaction.rb`<sup>3</sup>), which generates fake customer transaction data:

```
./buy_xaction.rb 80000 210 .05 > training.txt

# hadoop fs -copyFromLocal training.txt
    /markov_email_marketing/projection_by_sorting_in_ram/input/

# hadoop fs -cat
    /markov_email_marketing/projection_by_sorting_in_ram/input/training.txt
...
EY2I3D12PZ,1382705171,2013-07-29,28
VC38QFM2IF,1382705172,2013-07-29,84
1022R2QPWG,1382705173,2013-07-29,27
4G02MW73CK,1382705174,2013-07-29,31
VKV2K1S0D2,1382705175,2013-07-29,28
LDFK8WZQFH,1382705176,2013-07-29,25
8874144Q11,1382705177,2013-07-29,180
...
...
```

#### 11.3.1.4.2 Hadoop Solution 1: Run

```
# ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
adding: DateUtil.class(in = 1126) (out= 614)(deflated 45%)
adding: HadoopUtil.class(in = 1797) (out= 840)(deflated 53%)
adding: SortInMemoryProjectionDriver.class(in = 2780) (out= 1361)(deflated 51%)
adding: SortInMemoryProjectionMapper.class(in = 2112) (out= 899)(deflated 57%)
adding: SortInMemoryProjectionReducer.class(in = 2477) (out= 1143)(deflated 53%)
Deleted hdfs://localhost:9000/lib/projection_by_sorting_in_ram.jar
Deleted hdfs://localhost:9000/markov_email_marketing/projection_by_sorting_in_ram/
...

```

---

<sup>3</sup>[https://github.com/pranab/avenir/blob/master/resource/buy\\_xaction.rb](https://github.com/pranab/avenir/blob/master/resource/buy_xaction.rb)

```

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 12:03:26 INFO mapred.JobClient: map 100% reduce 0%
13/11/27 12:03:34 INFO mapred.JobClient: map 100% reduce 6%
...
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 output materialized bytes=2080706
13/11/27 12:04:17 INFO mapred.JobClient: Map input records=832280
13/11/27 12:04:17 INFO mapred.JobClient: Reduce shuffle bytes=20807060
13/11/27 12:04:17 INFO mapred.JobClient: Spilled Records=2496840
13/11/27 12:04:17 INFO mapred.JobClient: Map output bytes=19142440
13/11/27 12:04:17 INFO mapred.JobClient: Total committed heap usage (bytes)=10
13/11/27 12:04:17 INFO mapred.JobClient: Combine input records=0
13/11/27 12:04:17 INFO mapred.JobClient: SPLIT_RAW_BYT=157
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: Combine output records=0
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)

```

#### 11.3.1.4.3 Hadoop Solution 1: partial Output

```

...
ZTOBR28AH2 2013-01-06,190;2013-04-02,109;2013-04-09,26;2013-05-04,40;2013-05-12,34;
ZV2A56WNI6 2013-01-22,51;2013-01-24,34;2013-02-09,52;2013-04-11,200;2013-05-25,58;
ZVPH1EE3CI 2013-01-20,45;2013-03-02,107;2013-03-31,31;2013-06-12,198;2013-07-21,64;
ZXN7727FBA 2013-02-07,164;2013-02-23,30;2013-03-28,107;2013-04-02,26;2013-04-04,49;
ZY44ATNBK7 2013-03-27,191;2013-04-27,51;2013-05-06,31;2013-06-30,81
...

```

The next task is to convert (using a Ruby script developed by Pranab Ghosh) sorted consecutive sequence of (purchase-date, purchase-amount) pairs into a set of "Markov Chain states". The state is defined by a "two-letter-symbol" and is defined below:

<i>Time elapsed since last transaction</i>	<i>Amount spent compared to previous transaction</i>
S : small	L : significantly less than
M : medium	E : more or less same
L : large	G : significantly greater than

Therefore, we will have 9 ( $3 \times 3$ ) states:

<i>State Name</i>	<i>Time elapsed since last transaction : Amount spent compared to previous transaction</i>
SL	small : significantly less than
SE	small : more or less same
SG	small : significantly greater than
ML	medium : significantly less than
ME	medium : more or less same
MG	medium : significantly greater than
LL	large : significantly less than
LE	large : more or less same
LG	large : significantly greater than

Therefore our Markov Chain model has 9 states (transition matrix of  $9 \times 9$ ). The next step is to build our model's Transition Probabilities: that is to define

$$0.0 \leq P(state_1, state_2) \leq 1.0$$

where  $state_1$  and  $state_2 \in \{SL, SE, SG, ML, ME, MG, LL, LE, LG\}$ .

#### 11.3.1.5 Hadoop Solution 2: Time-ordered Transactions

This implementation provides a solution, which sorts reducers values by using "secondary sort" technique, which is a feature of MapReduce paradigm for sorting the values of reducers. To accomplish this, we need some custom classes, which will be plugged in to the MapReduce's framework implementation. Mappers emit (key, value), where key is a pair ("customer-id", "purchase-date") and value is pair of ("purchase-date", "amount"). Data arrive to reducers sorted. As you can see, to generate sorted values for each reducer key, we put the "purchase-date" (which is the first part of the emitted mapper value) as part of the mapper key. So, the **CompositeKey** is comprised of a pair ("customer-id", "purchase-date"). We represent value as pair of

("purchase-date", "amount") as a class `edu.umd.cloud9.io.pair.PairOfLongInt`<sup>4</sup>, where the "Long" part will represent Date and "Int" part represents purchase "amount".

MapReduce framework states that once the data values reaches a reducer, all data is grouped by key. Since we have a composite key (represented as `CompositeKey` — comprised of a pair ("customer-id", "purchase-date")), we need to make sure that records are grouped solely by the natural key (the natural key is the "customer-id"). This is accomplished by writing a custom partitioner class: `NaturalKeyPartitioner`.

We do need to provide more plug-in classes:

```
Configuration conf = new Configuration();
JobConf jobconf = new JobConf(conf, SecondarySortProjectionDriver.class);
...
jobconf.setPartitionerClass(NaturalKeyPartitioner.class);
jobconf.setOutputKeyComparatorClass(CompositeKeyComparator.class);
jobconf.setOutputValueGroupingComparator(NaturalKeyGroupingComparator.class);
```

Let's summarize "secondary sort": mappers generate (key, value) pairs. The order of the values arriving to a reducer is unspecified and can vary between job runs. For example, let all mappers generate (K, V1), (K, V2), and (K, V3). So, for the key K, we have 3 values: V1, V2, V3: so a reducer, which is processing key K might get one (out of 6) of the following orders:

```
V1, V2, V3
OR  V1, V3, V2
OR  V2, V1, V3
OR  V2, V3, V1
OR  V3, V1, V2
OR  V3, V2, V1
```

In most of the situations (this depends on your requirements and how you will process reducer values), it really does not matter in what order we

---

<sup>4</sup> <http://lintool.github.io/Cloud9/docs/api/edu/umd/cloud9/io/pair/PairOfLongInt.html>

get these values (i.e., V1, V2, V3). But if you want to put some ordering in receiving and processing values (such as sorted by ascending or sorted by descending), then the first option is to get all values V1, V2, V3 and then apply a sort function on these values, which you will get your desired order of values. This sort technique might not be feasible if you do not have enough RAM in your servers. There is another preferable method (which scales out very well and you do not need to worry about big RAM requirements) to have sorted values arriving at a reducer by utilizing the sort and shuffle feature of the MapReduce's framework. This technique is called a "**secondary sort**". **Secondary sort** is a technique that allows the MapReduce programmer to control the order that the values show up within a reduce function call. To achieve this, you need to use a composite key that contains both the information needed to sort by key and the information needed by value, and then decoupling the grouping of the intermediate data from the sorting of the intermediate data. The "secondary sorting" enables to define sorting order of the values generated by mappers. Sorting is done on both by the keys and the values. Further, grouping enable us to decide which sets of (key, value) are put together into a single call of the reduce function. For our example, the composite key is illustrated below ([11.2](#)):

In Hadoop, the grouping is controlled in two places: (1) the partitioner, which sends map outputs to reduce tasks, and the (2) grouping comparator, which groups data within a reduce task. Both of these (partitioner and group comparator) can be accomplished by pluggable classes per MapReduce job. To sort values for reducers, we have to define a pluggable class, which sets the output key comparator. Therefore, using raw MapReduce, getting secondary sort working can take a little bit of work - we need to define a composite key and specify three functions (by defining pluggable classes to the MapReduce job — details are given below) that each use it in different ways. Note that, for our example the natural key is a customerID (as a String object) and there is no need to wrap it in another class, which we might call it as a NaturalKey. The natural key is what you would normally use as the key or "group by" operator.

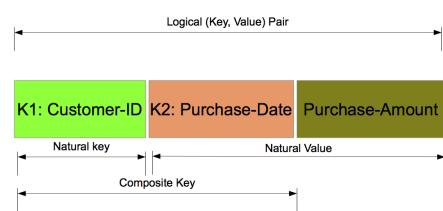


Figure 11.2: Composite Key for Secondary Sorting

Hadoop Solution 2: Implementation Classes	
Class Name	Description
SecondarySortProjectionDriver	Driver class to submit jobs
SecondarySortProjectionMapper	Mapper class
SecondarySortProjectionReducer	Reducer class
CompositeKey	Custom key to hold a pair of (customer-id, purchase-date) which is a combination of the natural key and the natural value we want to sort by
CompositeKeyComparator	How to sort CompositeKey objects, compares two composite keys for sorting.
NaturalKeyGroupingComparator	considers the natural key; makes sure that a single reducer sees a custom view of the groups (how to group CustomerID)
NaturalKeyPartitioner	How to partition by the natural key (customerID) to reducers; blocks all data into a logical group, inside which we want the secondary sort to occur on the natural value
DateUtil	Basic Date Utility class
HadoopUtil	Basic Hadoop Utility class

### 11.3.1.6 Hadoop Solution 2: Sample Run

#### 11.3.1.6.1 Hadoop Solution 2: Partial Input

```
# hadoop fs -cat /markov_email_marketing/projection_by_secondary_sort/input/trainin
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

#### 11.3.1.6.2 Hadoop Solution 1: Run

```
# ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
adding: CompositeKey$CompositeKeyComparator.class(in = 448) (out= 305)(deflated 31%)
adding: CompositeKey.class(in = 1767) (out= 883)(deflated 50%)
adding: CompositeKeyComparator.class(in = 715) (out= 448)(deflated 37%)
adding: DateUtil.class(in = 1126) (out= 614)(deflated 45%)
adding: HadoopUtil.class(in = 2541) (out= 1139)(deflated 55%)
adding: NaturalKeyGroupingComparator.class(in = 588) (out= 354)(deflated 39%)
adding: NaturalKeyPartitioner.class(in = 1204) (out= 672)(deflated 44%)
adding: SecondarySortProjectionDriver.class(in = 3290) (out= 1576)(deflated 52%)
adding: SecondarySortProjectionMapper.class(in = 2136) (out= 916)(deflated 57%)
adding: SecondarySortProjectionReducer.class(in = 1999) (out= 897)(deflated 55%)
Deleted hdfs://localhost:9000/lib/projection_by_secondary_sort.jar
Deleted hdfs://localhost:9000/markov_email_marketing/projection_by_secondary_sort/
...
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 15:14:45 INFO mapred.JobClient: map 25% reduce 0%
...
13/11/27 15:16:01 INFO mapred.JobClient: map 100% reduce 93%
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 output materialized bytes=7560600
13/11/27 15:16:03 INFO mapred.JobClient: Map input records=832280
13/11/27 15:16:03 INFO mapred.JobClient: Reduce shuffle bytes=7560600
13/11/27 15:16:03 INFO mapred.JobClient: Spilled Records=1664560
13/11/27 15:16:03 INFO mapred.JobClient: Map output bytes=26632960
13/11/27 15:16:03 INFO mapred.JobClient: Total committed heap usage (bytes)=4732200
13/11/27 15:16:03 INFO mapred.JobClient: Map input bytes=30089506
```

```

13/11/27 15:16:03 INFO mapred.JobClient: Combine input records=0
13/11/27 15:16:03 INFO mapred.JobClient: SPLIT_RAW_BYTES=2900
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)

```

### 11.3.1.6.3 Hadoop Solution 2: partial Output

```

...
ZSY40NVPS6 2013-01-01,110;2013-01-11,32;2013-03-04,111;2013-04-09,65;2013-05-15,10
ZTLNF004LN 2013-01-16,55;2013-03-21,164;2013-05-14,66;2013-06-29,81;
ZV20AIXG8L 2013-01-13,210;2013-02-03,32;2013-02-10,48;2013-02-23,27;2013-03-08,55;
ZVA9U4EQ46 2013-01-21,199;2013-01-26,34;2013-01-28,49;2013-02-18,25;2013-04-29,198
ZW8612MI1V 2013-02-27,107;2013-03-14,33;2013-03-23,41;2013-04-13,30;2013-05-12,42;
...

```

### 11.3.2 MapReduce to Generate Markov State Transition

The goal of this MapReduce phase is to generate Markov State Transition matrix. The input for this phase will be the "state sequence" and will have the following format:

$$customerid, State_1, State_2, \dots, State_n$$

and the output will be a matrix of  $N \times N$  where  $N$  is the number of states (in our model  $N = 9$ ) for Marokov Chain model. The matrix entries will indicate the probability of going from one state to another one. The goal of this MapReduce is to count the number of instances of state transitions. Since the number of states for our model is 9, therefore, we will have  $9 \times 9 = 81$  poossible state transitions.

#### 11.3.2.1 Mapper

Listing 11.3: Markov State Transition map() Function

```
/**  
 * @param key is the Customer-ID, ignored  
 * @param value is sequence of states = {S1, S2, ..., Sn}  
 * we assume value is an array of n states (indexed from 0 to n-1)  
 */  
map(key, value) {  
    for (i=0, i < n-1, i++) {  
        // value[i] denotes "from state"  
        // value[i+1] denotes "to state"  
        reducerKey = pair(value[i], value[i+1]);  
        emit(reducerKey, 1);  
    }  
}
```

### 11.3.2.2 Combiner

Listing 11.4: Markov State Transition combine() Function

```
/**  
 * @param key is a Pair(state1, state2)  
 * @param value is a list of integers (partial count of "state1" to "  
 * state2")  
 */  
combine(Pair(state1, state2) key, List<integer> values) {  
    int partialSum = 0;  
    for (int count : values) {  
        partialSum += count;  
    }  
    emit(key, partialSum);  
}
```

### 11.3.2.3 Reducer

Listing 11.5: Markov State Transition reduce() Function

```
/**  
 * @param key is a Pair(state1, state2)
```

```

* @param value is a list of integers (partial count of "state1" to "
    state2")
*/
reduce(Pair(state1, state2) key, List<integer> value) {
    int sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}

```

---

#### 11.3.2.4 Hadoop Implementation of State Transition Model

Hadoop Solution 2: Implementation Classes	
Class Name	Description
MarkovStateTransitionModelDriver	Driver class to submit jobs
MarkovStateTransitionModelMapper	Mapper class
MarkovStateTransitionModelCombiner	Combiner class
MarkovStateTransitionModelReducer	Reducer class
ReadDataFromHDFS	Reads data from HDFS and creates List<TableItem>
StateTransitionTableBuilder	builds transition table and defines P(state1, state2)
TableItem	A class to represent a triplet of (fromState, toState, count)
HadoopUtil	Basic Hadoop Utility class

#### 11.3.2.5 Sample Run of State Transition Model

##### 11.3.2.5.1 Sample Run: Input

```
# hadoop fs -cat /markov_email_marketing/state_transition_model/input/state_seq.txt
000IA1PHVZ,SG,SL,SG,SL,ML,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
```

0030B44HDO,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

### 11.3.2.5.2 Sample Job Run

```

# ./run.sh
...
adding: HadoopUtil.class(in = 1797) (out= 840)(deflated 53%)
adding: MarkovStateTransitionModelCombiner.class(in = 1711) (out= 704)(deflated 58%)
adding: MarkovStateTransitionModelDriver.class(in = 1976) (out= 988)(deflated 50%)
adding: MarkovStateTransitionModelMapper.class(in = 2018) (out= 826)(deflated 59%)
adding: MarkovStateTransitionModelReducer.class(in = 2018) (out= 869)(deflated 56%)
adding: ReadDataFromHDFS.class(in = 3691) (out= 1848)(deflated 49%)
adding: StateTransitionTableBuilder.class(in = 1287) (out= 708)(deflated 44%)
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 21:16:25 INFO mapred.JobClient: map 100% reduce 0%
13/11/29 21:16:32 INFO mapred.JobClient: map 100% reduce 3%
...
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 output materialized bytes=844
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: Spilled Records=268
13/11/29 21:17:11 INFO mapred.JobClient: Map output bytes=8067660
13/11/29 21:17:11 INFO mapred.JobClient: Total committed heap usage (bytes)=1000000000
13/11/29 21:17:11 INFO mapred.JobClient: Combine input records=672461
13/11/29 21:17:11 INFO mapred.JobClient: SPLIT_RAW_BYTES=152
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

```

#### 11.3.2.5.3 Sample Run: Output

```

# hadoop fs -ls /markov_email_marketing/state_transition_model/output/part*
-rw-r--r-- 1 mahmoud staff 58 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 85 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 60 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 43 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 53 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 60 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 76 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 54 2013-11-29 21:16 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 30 2013-11-29 21:17 /markov_email_marketing/
-rw-r--r-- 1 mahmoud staff 46 2013-11-29 21:17 /markov_email_marketing/

# hadoop fs -cat /markov_email_marketing/state_transition_model/output/part*
LL, MG 2990
ME, SG 172
MG, LL 803
ML, LG 59
SE, MG 5
SG, ME 606
LG, SG 1516
LL, SL 69
MG, ML 3477
ML, MG 12391
SE, SL 3014
SG, LG 69
SL, LL 7885
SL, SE 2099
LE, LG 2
LG, LE 1
MG, SG 19485

```

ML,SL 268  
SG,MG 337  
SL,ML 35265  
LE,MG 10  
ME,LG 1  
SG,SL 242614  
SL,SG 177254  
LE,SL 140  
LG,LG 1  
ME,MG 31  
SE,LL 62  
SE,SE 2  
SL,LE 24  
LG,MG 327  
ME,SL 606  
MG,LG 18  
ML,LL 2465  
SE,ML 305  
SL,ME 151  
LG,SL 510  
LL,SG 17062  
MG,MG 172  
SE,SG 1478  
SG,LL 10975  
SG,SE 3013  
SL,LG 240  
LL,LE 1  
MG,SL 2096  
ML,SG 63843  
SG,ML 48428  
SL,MG 1325  
ME,LL 5  
SG,SG 5090  
SL,SL 2772  
LE,SG 58  
LG,LL 64  
LL,LG 507  
SE,LG 2

SG,LE 140

```
# javac ReadDataFromHDFS.java TableItem.java
# java ReadDataFromHDFS /markov_email_marketing/state_transition_model/output
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - path=hdfs://hnode00815.ne
Nov 30 2013 12:24:18 [main] [INFO ] [ReadDataFromHDFS] - line=LL,MG 2990
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}, {ME,S}
```

## 11.4 Using Markov Model to Predict Next Email Marketing Date

Now that we have built the Markov Chain model from a given training data, we can use it to predict the next email marketing date for customers.

To persist the model (`model.txt`), we use the following piece of code:

```
# cat StateTransitionTableBuilder.java
...
public class StateTransitionTableBuilder {
    ...
    public static void main(String[] args) {
        String hdfsDirectory = args[0];
        generateStateTransitionTable(hdfsDirectory);
    }
}

# export hdfsDir="/markov_email_marketing/state_transition_model/output"
# java StateTransitionTableBuilder $hdfsDir > model.txt
```

Now that we have the model (`model.txt`) — represented as a two-dimensional

table), we can use a Ruby scripts (`buy_xaction.rb` and `mark_plan.rb`) developed by Pranab Ghosh to predict the next email marketing date:

```
# 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
F7LS37R08X,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
F7LS37R08X, 2013-02-14
...
```

Therefore, we are able to predict the next date for sending an email marking based on the transation hisory of users.

# Chapter 12

## K-Means Clustering

### 12.1 Introduction

The main goal of this chapter is to provide a MapReduce solution for K-Means clustering algorithm. K-Means MapReduce algorithm is interesting and different from other MapReduce algorithms. K-Means MapReduce algorithm is an iterative MapReduce algorithm, you execute it many times (with different centroids) until it converges (meaning that all optimal K clusters are found after many iterations of the same MapReduce job).

What is clustering? What is K-Means? In a nutshell, we can say that

- clustering is the process of grouping a set of d-dimensional (2-dimensional, 3-dimensional, ...) objects into clusters of similar objects, and
- objects should be similar to one another within the same cluster and dissimilar with those in other clusters.

K-Means is a distance-based clustering algorithm. K-Means clustering has many useful applications. For example, it can be used to find a group of consumers with common behaviors, or to cluster documents based on similarity of their contents. The first question is how do we determine the value of  $K$  as the number of groups or clusters we want to generate from our input data? Selection of  $K$  is specific to the application or problem domain you are trying to solve. There is no magic formula to find  $K$ .

An example of a K-Means for  $K = 3$  is given below. The first image shows the raw data. The second image shows an application of K-Means. In

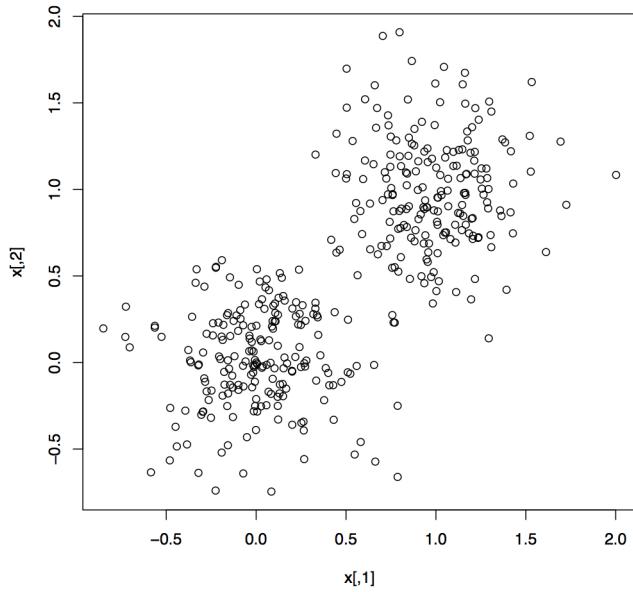


Figure 12.1: Raw Data

the second image, the red points represents the first cluster and black points represent the second cluster, and finally the green points represent the third cluster. The goal of K-Means is how to convert the raw data (first image) into clusters (second image).

How does K-Means work? This is how it works: the algorithm is given as inputs a set of  $N$  d-dimensional points and a number of desired clusters  $K$ . For the simplicity, we will consider points in a Euclidean space. However, the K-Means clustering algorithm will work in any space provided a distance metric is given as input as well. Therefore, our input of  $N$  (we use  $n$  and  $N$  interchangeably here) d-dimensional points will be:

$$\begin{aligned} p_1 &= (a_{11}, a_{12}, \dots, a_{1d}) \\ p_2 &= (a_{21}, a_{22}, \dots, a_{2d}) \end{aligned}$$

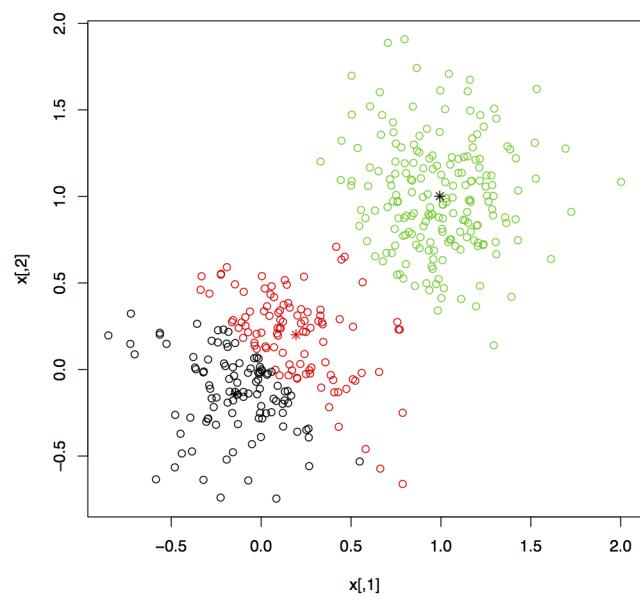


Figure 12.2: Clustered Data

...  
 $p_n = (a_{n1}, a_{n2}, \dots, a_{nd})$

For example, in a 2-dimentional environment, our input data can be as follows, where each row represents a point  $(x, y)$  in a 2-dimentional space:

```
p1 = (1,1)
p2 = (2,1)
p3 = (1,2)
p4 = (5,5)
p5 = (6,5)
p6 = (5,6)
p7 = (7,7)
p8 = (9,6)
```

Initially,  $K$  points are chosen as cluster centers; these are called cluster centroids. There are many ways to initialize the cluster centroids: one way is to choose the  $K$  points randomly from the sample of  $n$  points. Once the  $K$  initial cluster centroids are chosen, the distance is calculated from every point in the input set to each of the  $K$  centers and then each point is assigned to the specific cluster center whose distance is closest. When all objects have been assigned, then again we recalculate the positions of the  $K$  centroids. The above two steps are repeated until the cluster centroids no longer change (or change very little).

Before we present our MapReduce solution, we will look at the basic definition of K-Means clustering and then we will briefly focus on informal and formal algorithms of K-Means clustering. The better we understand K-Means clustering, then it will be easier to comprehend the MapReduce solution.

## 12.2 What is K-Means Clustering

What is K-Means Clustering? In a nutshell, K-Means Clustering (unsupervised learning) is a data mining algorithm to cluster, classify or to group your  $N$  objects based on their attributes or features into  $K$  number of groups (so called clusters).  $K$  is a positive integer number ( $K = 2, 3, 4, \dots$ ). If we apply

K-Means to a set of  $N$  objects, then the result will be  $K$  disjoint groups (adding all objects in  $K$  groups will result in  $N$  objects). The next question is how do we group these  $N$  objects into  $K$  disjoint groups? The grouping is done by minimizing the sum of squares of distances between data and the corresponding cluster centroid. The next two questions are: why do we use clustering? and how to find cluster centroids? The simple answer is: we use K-Mean clustering to classify the data (for example to classify student's grades into 5 categories ( $K = 5$ ): A (excellent), B (good), C (average), D (poor), and F (failure). Another example will be: use K-Means to predict student academic performance[22]. I will return to the question of "how to find cluster centroids" in the next sections.

Next, we formalize the K-Means clustering. Given  $n$  d-dimentional points:

$$\begin{aligned} X_1 &= (x_{11}, x_{12}, \dots, x_{1d}) \\ X_2 &= (x_{21}, x_{22}, \dots, x_{2d}) \\ &\dots \\ X_n &= (x_{n1}, x_{n2}, \dots, x_{nd}) \end{aligned}$$

The goal is to partition these  $\{X_1, X_2, \dots, X_n\}$  into  $k$  clusters:  $\{C_1, C_2, \dots, C_k\}$ . K-means aims to find the positions  $\mu_i$  ( $i=1,\dots,k$ ) of the clusters that minimize the distance from the data points to the cluster. K-means clustering solves the following cost minimization algorithm:

$$\arg \min_C \sum_{i=1}^k \sum_{\mathbf{X}_j \in C_i} \|\mathbf{X}_j - \boldsymbol{\mu}_i\|^2$$

where  $\boldsymbol{\mu}_i$  is the mean of points in  $C_i$ .

### 12.3 What are the Applications of Clustering?

Clustering algorithms have many possible applications. The following are partial list of clustering applications:

- Marketing: Given a large set of customer transactions, finding groups of customers with similar purchase behaviors
- Document classification: clustering web log data to discover groups of similar access patterns.
- Insurance: identifying groups of vehicle insurance policy holders with a high average claim cost; identifying frauds;

## 12.4 K-Means Clustering Method: Partitioning Approach

K-Means clustering algorithm is an iterative algorithm. In subsequent sections, I will provide a MapReduce algorithm, which we will continue running until the K-Means clustering algorithm converges by finding a proper optimal solution. Informally, we can state K-Means Clustering algorithm as:

Given a set of  $N$  objects, which has to be grouped into  $K$  clusters, the k-means algorithm performs the following steps:

Step-1: Partition  $N$  objects into  $K$  non-empty subsets

Step-2: Compute the centroids of the clusters in the current partition (the centroid is the center, i.e., mean point, of the cluster). For all  $i = 1, 2, \dots, k$  compute  $\mu_i$  as:

$$\mu_i = \frac{1}{|c_i|} \sum_{j \in c_i} \mathbf{x}_j, \forall i$$

Step-3: Assign each object to the cluster with the nearest centroid. This is simply attributing the closest cluster to each data point:

$$c_i = \{j : d(x_j, \mu_i) \leq d(x_j, \mu_l), l \neq i, j = 1, \dots, n\}$$

where  $d(a, b)$  is the distance function for two points:  $a$  and  $b$ .

Step-4: Stop when no more new assignments. Otherwise go back to Step-2. Basically, we repeat steps Steps 2-3 until convergence.

Basically, the algorithm iterates until there are no changes in the centroids; and when there are no changes in the centroids, then we have found our  $K$  clusters. K-Means algorithm is an iterative process, which means that at each step the membership of each object in a cluster is re-evaluated based on the current centers of each existing cluster. This process is repeated until the desired number of clusters (or the number of objects) is reached.

## 12.5 K-Means Distance Function

The first step of the K-Means clustering algorithm involves exclusive assignment of data points to the closest cluster centroids. For a given data point (of  $d$ -dimensions), how do we determine the closest cluster centroid? For this purpose, we use a distance function, which calculates how probable (or likely) that the centroid generated the data point. There are many distance functions (few are mentioned here):

- Euclidean Distance
- Manhattan Distance
- Inner Product Space
- Maximum Norm
- Your Own Custom Function (any metric you define over the  $d$ -dimensional space)

For example, Euclidean distance have been used in many K-Means algorithms. The **Euclidean distance** between two data point instances  $X$  and  $Y$  which are represented by  $d$  continuous attributes ( $d$ -dimensional) is:

Let

$$\begin{aligned} X &= (x_1, x_2, \dots, x_d) \\ Y &= (y_1, y_2, \dots, y_d) \end{aligned}$$

Then

$$\text{distance}(X, Y) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + \dots + (x_d - y_d)^2}$$

The Euclidean distance function has some interesting properties:

- $\text{distance}(i, j) \geq 0$
- $\text{distance}(i, i) = 0$
- $\text{distance}(i, j) = \text{distance}(j, i)$
- $\text{distance}(i, j) \leq \text{distance}(i, k) + \text{distance}(k, j)$

For example, Euclidean distance function can be expressed in Java-like as:

```
public class EuclideanDistance {  
    // Let Vector[i] be the i'th position of the Vector  
    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);  
    }  
}
```

When implementing K-Means, we have to make sure that our algorithm converges and we can properly calculate "mean" and "distance" functions properly over the space we have.

## 12.6 K-Means Clustering Step-by-Step Example

## 12.7 K-Means Clustering Formalized

K-Means is a simple learning algorithm for clustering analysis. The goal of K-Means clustering algorithm is to find the best division of  $n$  entities – so called objects or points–in  $k$  groups, so that the total distance between

the group's members and its corresponding centroid, representative of the group, is minimized. Formally, the goal is to partition the  $n$  entities into  $k$  sets  $\{S_i, i = 1, 2, \dots, k\}$  in order to minimize the within-cluster sum of squares (WCSS), defined as:

$$\min \sum_{j=1}^k \sum_{i=1}^n \|x_i^j - c_j\|$$

where term  $\|x_i^j - c_j\|$  provides the distance between an entity point and the cluster's centroid.

## 12.8 MapReduce Solution for K-Means Clustering

MapReduce solution for K-Means clustering is an iterative solution, where each iteration is implemented as a MapReduce job. K-Means needs an iterative version of MapReduce, which is not a standard formulation of MapReduce paradigm. To implement an iterative MapReduce solution, we do need a driver or control program on the client side to initialize the  $K$  centroid positions, call the iteration of MapReduce jobs and determine whether the iteration should continue or end. The mapper needs to fetch the data point and all cluster centroids; the cluster centroids have to be shared among all mappers. This can be done in many different ways by using HDFS or a global data structure server like Redis or memcached. The reducer recomputes new means.

Note that each iteration of the algorithm is structured as a single MapReduce job, which iteratively improves partitioning of data into  $k$  clusters. The overall MapReduce pseudo code solution is illustrated below. The change() function is used to terminate the MapReduce iteration when there is not much change in centroids.

The pseudocode of the K-Means algorithm is shown below.

**Listing 12.1:** K-Means Algorithm

```

1 // k = number of desired clusters
2 // delta = acceptable error for convergence
3 // data = input data
4 kmeans(k, delta, data) {
5     // initialize the cluster centroids
6     initial_centroids = pick(k, data);
7
8     // this is how we broadcast centers to mappers
9     writeToHDFS(initial_centroids);
10
11    // iterate as long as necessary
12    current_centroids = initial_centroids;
13    while (true) {
14        // map_reduce_job() does 2 tasks:
15        //   1. uses current_centroids in map()
16        //   2. reduce() creates new_centroids and write it to HDFS
17        map_reduce_job();
18        new_centroids = readFromHDFS();
19        if change(new_centroids, current_centroids) <= delta {
20            // we are done, terminate loop-iteration
21            break;
22        }
23        else {
24            current_centroids = new_centroids;
25        }
26    }
27
28    result = readFromHDFS();
29    return result;
30}

```

---

The `change()` method is presented below:

**Listing 12.2: K-Means Algorithm: `change()` Method**

```

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 }

```

---

The next three subsections provide the details of verb—`map_reduce_job()`—(as a MapReduce job). The MapReduce job will have 3 functions: `map()`, `combine()`, and `reduce()`.

### 12.8.1 MapReduce Solution: map()

The `map()` function classifies data. It uses the cluster centroids and assigns each point  $p \in 1, 2, \dots, n$  to the nearest center. Note that in the first iteration, the centroids will be the ones selected at random or created manually. The `map()` accepts in-points in the data set and outputs one (Cluster-ID, Vector) pair for each point, where the Cluster-ID is the integer ID of the cluster which is closest to the point and Vector is a d-dimensional vector (array of double data type).

The `map()` function is summarized below:

- Read the cluster centroids into memory from a SequenceFile<sup>1</sup> (note that we may also use Redis or memcached for persisting cluster centroids). In Hadoop's implementation, this will be done in the `setup()` method of the mapper class.
- Iterate over each cluster centroid for each input key-value pair. In Hadoop's implementation, for the `map()` function, key is generated by Hadoop and ignored (not used)
- Compute the Euclidean distances and save the nearest center which has the lowest distance to the input point (as a d-dimensional vector)
- Write the *key – value* pair to be consumed by reducers, where *key* is the nearest cluster center with its input point (a d-dimensional vector as a *value*). Both, key and value are Vector (as a d-dimensional vector) type.

**Listing 12.3:** K-Means MapReduce: `map()`

```
1 public class KmeansMapper ... {
2
3     private List<Vector> centers = null;
4
5     private List<Vector> readCentersFromSequenceFile() {
6         // read cluster centroids from a SequenceFile,
7         // which is a set of key-value pairs
8         ...
9     }
10
11    // called once at the beginning of the map task
12    public void setup(Context context) {
```

---

<sup>1</sup>org.apache.hadoop.io.SequenceFile

```

13     this.centers = readCentersFromSequenceFile();
14 }
15
16 /**
17 * @param key is MapReduce generated, ignored here
18 * @param value is the d-dimentional Vector (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     // prepare key-value for reducers
38     emit(nearest, value);
39 }
40
41 } // end of class KmeansMapper

```

---

## 12.8.2 MapReduce Solution: combine()

After each map task, a combiner is applied to mix the intermediate data of the same map task. The combiner sums up the values—the sum is needed for computing the mean values—for each dimension of vector objects. In the `combine()` function, we partially sum the values of the points (as vectors) assigned to the same cluster. The combiner function improves the efficiency of our algorithm by avoiding network traffic by transferring less data between slave nodes.

**Listing 12.4:** K-Means MapReduce: `combine()`

```

1 /**
2 * @param key is the Centroid
3 * @param values is a list of Vector
4 */
5 combine(Vector key, Iterable<Vector> values) {
6     // all dimensions in sum Vector are initialized to 0.0
7     Vector sum = new Vector();

```

```

8   for (Vector value : values) {
9     // note that value.length = d,
10    // where d is the number of dimensions for input objects
11    for (int i = 0; i < value.length; i++) {
12      sum[i] += value[i];
13    }
14  }
15
16  emit(key, sum);
17 }

```

---

### 12.8.3 MapReduce Solution: reduce()

The reducer does re-centering: it re-creates the centroids for all clusters by recalculating the means for all clusters. In reduce phase, the outputs of the map() functions are grouped by Cluster-ID, and for each Cluster-ID the centroid of the points associated with that Cluster-ID is calculated. Each reduce function generates (Cluster-ID, Centroid) pairs, which represent the newly calculated cluster centers.

The reduce() function is summarized below:

- The main task of the reduce() function is to re-center.
- Each reducer, iterates over each value vector and calculate the mean vector. Once you have found the mean, then this is the new center; finally save it into a SequenceFile.

**Listing 12.5:** K-Means MapReduce: reduce()

```

1 /**
2 * @param key is the Centroid
3 * @param values is a list of Vector
4 */
5 reduce(Vector key, Iterable<Vector> values) {
6   // all dimensions in newCenter are initialized to 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     // set new mean for each dimension
18     newCenter[i] = newCenter[i] / count;

```

```
19     }
20
21     emit(key.ID, newCenter);
22 }
```

---

## 12.9 MapReduce K-Means Clustering Step-by-Step Example

## 12.10 K-Means Implementation by Spark

Spark framework provides some common machine learning (ML) functionality, called MLlib. Spark's MLlib<sup>2</sup> supports the following types of machine learning algorithms:

- binary classification
- regression
- clustering (includes K-Means)
- collaborative filtering
- gradient descent optimization primitive.

Spark's MLlib supports M-Means clustering, that clusters the data points into predefined number of clusters. The MLlib implementation includes a parallelized variant of the **k-means++** (new algorithm for find the centroids) method called **kmeans||** (parallel implementation of K-Means algorithm). Spark's implementation of K-Means has the following parameters:

- **k** is the number of desired clusters.
- **maxIterations** is the maximum number of iterations to run.
- **initializationMode** specifies either random initialization or initialization via **kmeans||**.

---

<sup>2</sup><http://spark.apache.org/docs/0.9.1/mllib-guide.html>

- `runs` is the number of times to run the k-means algorithm (note that k-means is not guaranteed to find a globally optimal solution, and when run multiple times on a given dataset, the algorithm returns the best clustering result).
- `initializationSteps` determines the number of steps in the `kmeans||` algorithm.
- `epsilon` determines the distance threshold within which we consider k-means to have converged.

Sample implementation of K-Means is given by Spark as a `JavaKMeans`<sup>3</sup> class. Below, we present `JavaKMeans` class, and then we will show a sample run.

**Listing 12.6:** Example using MLLib KMeans from Java

```

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;
11 import org.apache.spark.mllib.linalg.Vectors;
12
13 /**
14 * Example using MLLib KMeans from Java.
15 */
16 public final class JavaKMeans {
17
18     private static class ParsePoint implements Function<String, Vector> {
19         private static final Pattern SPACE = Pattern.compile(" ");
20         @Override
21         public Vector call(String line) {
22             String[] tok = SPACE.split(line);
23             double[] point = new double[tok.length];
24             for (int i = 0; i < tok.length; ++i) {
25                 point[i] = Double.parseDouble(tok[i]);
26             }
27             return Vectors.dense(point);
28         }
29     }
30 }
```

---

<sup>3</sup>`org.apache.spark.examples.mllib.JavaKMeans`

```

31  public static void main(String[] args) {
32      if (args.length < 3) {
33          System.err.println(
34              "Usage: JavaKMeans <input_file> <k> <max_iterations> [<runs>]");
35          System.exit(1);
36      }
37      String inputFile = args[0];
38      int k = Integer.parseInt(args[1]);
39      int iterations = Integer.parseInt(args[2]);
40      int runs = 1;
41
42      if (args.length >= 4) {
43          runs = Integer.parseInt(args[3]);
44      }
45
46      SparkConf sparkConf = new SparkConf().setAppName("JavaKMeans");
47      JavaSparkContext sc = new JavaSparkContext(sparkConf);
48      JavaRDD<String> lines = sc.textFile(inputFile);
49      JavaRDD<Vector> points = lines.map(new ParsePoint());
50      KMeansModel model = KMeans.train(points.rdd(), k, iterations, runs, KMeans.K_MEANS_PARALLEL());
51
52      System.out.println("Cluster centers:");
53      for (Vector center : model.clusterCenters()) {
54          System.out.println(" " + center);
55      }
56      double cost = model.computeCost(points.rdd());
57      System.out.println("Cost: " + cost);
58
59      sc.stop();
60  }
61 }
```

---

Some of the key points of the `JavaKMeans` class are discussed below:

- Line 47: create `JavaSparkContext` object, which is a connection object to a Spark master and an entry point to run jobs in Spark cluster
- Line 48: read input file (as record of strings) and create a new `JavaRDD<String>` (a set of `String` objects)
- Line 49: Convert and tokenize each string record in `JavaRDD<String>` to a `Vector` of doubles. This is accomplished by using `ParsePoint` class, which takes a `String` object and converts it to a `Vector` of doubles.
- Line 50: The K-Means algorithm is called by the `KMeans`<sup>4</sup> class, which generates a `KMeansModel`<sup>5</sup> (a clustering model for K-means, where each

---

<sup>4</sup>`org.apache.spark.mllib.clustering.KMeans`

<sup>5</sup>`org.apache.spark.mllib.clustering.KMeansModel`

point belongs to the cluster with the closest center).

- Lines 52-57: prints the results

### 12.10.1 Sample Run of K-Means by Spark

MP TBDL

#### 1. Dependencies

Mlib uses the jblas linear algebra library, which itself depends on native Fortran routines. You may need to install the gfortran runtime library if it is not already present on your nodes. Mlib will throw a linking error if it cannot detect these libraries automatically.

2. yum install gcc-gfortran

nctions

# kNN: k-Nearest-Neighbors

## 13.1 Introduction

This chapter focuses on one of important machine learning algorithms, called k-nearest-neighbors (kNN), where  $k$  is an integer number greater than 0. The kNN classification problem is to find the  $k$ -nearest data points in a dataset (so called training dataset) to a given query data point. K-Nearest-Neighbors is the same as "kNN join". The kNN join can be defined as: given two data sets  $R$  and  $S$ , the goal is to find  $k$  nearest neighbors from a dataset  $S$  for every object in dataset  $R$ . In data mining,  $R$  and  $S$  are called query and training datasets respectively. Training dataset ( $S$ ) refers to data, which has already been classified and query dataset ( $R$ ) refers to data, which is going to be classified using  $S$ 's classifications.

The objective of this chapter is to present MapReduce solution (using Spark/Hadoop) for kNN (especially for kNN join algorithm). The kNN algorithm has been discussed extensively in many machine learning books such as Machine Learning for Hackers[?] and Machine Learning in Action[11]. kNN is an important clustering algorithm, which has many applications in a wide range of data mining (such as pattern recognition) and bioinformatics (such as Breast Cancer Diagnosis Problem[23], Weather Generating Model[?], and product recommender systems. The kNN algorithm can be fairly expensive especially when we have a large training set and this is why MapReduce solution will be suitable for handling large training data set.

What is kNN? Nearest Neighbor analysis is an algorithm for classifying

n-dimensional objects<sup>1</sup> based on their similarity to other n-dimensional objects. In machine learning, Nearest Neighbor analysis has been developed as a way to recognize patterns of data without requiring an exact match to any stored patterns or objects. Similar n-dimensional are near each other and dissimilar n-dimensional are distant from each other. Thus, the distance between two cases is a measure of their dissimilarity. According to wikipedia: "In pattern recognition, the k-nearest neighbor algorithm (kNN) is a non-parametric method for classifying objects based on closest training examples in the feature space. kNN is a type of instance-based learning, or lazy learning where the function is only approximated locally and all computation is deferred until classification. The k-nearest neighbor algorithm is amongst the simplest of all machine learning algorithms: an object is classified by a majority vote of its neighbors, with the object being assigned to the class most common amongst its  $k$  nearest neighbors ( $k$  is a positive integer, typically small). If  $k = 1$ , then the object is simply assigned to the class of its nearest neighbor."

## 13.2 kNN Classification

What is the main idea behind the kNN? The central idea of kNN is to build a classification method by using no assumptions about the form of the "smooth" function  $f$ :

$$x = (x_1, x_2, \dots, x_n)$$

$$y = f(x)$$

that relates  $y$  (the response variable) to the  $x$  (predictor variables). Function  $f$  is a non-parametric method because it does not involve estimation of parameters in any form or shape. In kNN, given a new point  $p = (p_1, p_2, \dots, p_n)$ , we dynamically identify  $k$  observations in the training dataset that are similar to  $p$  (k nearest neighbor). In kNN, neighbors are defined by a distance or dissimilarity measure that we can compute between observations based on the independent variables. For simplicity, we will use the Euclidean distance. The Euclidean distance between the points  $(x_1, x_2, \dots, x_n)$  and  $(p_1, p_2, \dots, p_n)$  is defined as:

---

<sup>1</sup>An n-dimensional object  $x$ , which has  $n$  attributes, will be represented as  $(x_1, x_2, \dots, x_n)$

$$\sqrt{(x_1 - p_1)^2 + (x_2 - p_2)^2 + \dots + (x_n - p_n)^2}$$

The question is **how do you find k nearest neighbors?**. For each queried n-dimensional object (an object which has  $n$  attributes), the Euclidean distances between the queried object and all the training data objects are calculated and the queried object is assigned the class label that most of the  $k$  closest training data have. Note that the kNN algorithm assumes that all data correspond to n-dimensional metric (denoted by  $R^n$ ) space, which means that the data type of all attributes are "double." This is required since we need to calculate the Euclidean distance between n-dimensional objects.

### 13.3 Distance Functions

The Euclidean distance function have been used in many data mining clustering algorithms. The Euclidean distance between two n-dimentional objects X and Y are given below. Let  $X$  and  $Y$  defined as:

$$X = (X_1, X_2, \dots, X_n)$$

$$Y = (Y_1, Y_2, \dots, Y_n)$$

Then `distance(X, Y)` is defined as:

$$distance(X, Y) = \sqrt{\sum_{i=1}^n (X_i - Y_i)^2}$$

$$distance(X, Y) = \sqrt{(X_1 - Y_1)^2 + (X_2 - Y_2)^2 + \dots + (X_n - Y_n)^2}$$

Note that the Euclidean distance function will work only for metric data, where all attributes have "double" primitive data type. What if our attributes are non-numeric. Take for example, "big spender," "medium spender," "small spender," and "non spender." Then we need to come up with custom distance functions so that we can accommodate all different data types. In this case we need to answer `distance ("big spender", "small spender")` and so on. Most of the time in real applications we do need custom distance functions.

There are other distance functions available for kNN:

- Manhattan:

$$\sum_{i=1}^n |X_i - Y_i|$$

- Minkowski:

$$\left( \sqrt{\sum_{i=1}^n (|X_i - Y_i|)^q} \right)^{1/q}$$

## 13.4 kNN Example

The kNN algorithm is an intuitive method which classifies unclassified data (called an input query) based on its similarity or distance with the data in the training dataset. For example, let our training dataset to have 4 classes identified as  $\{C_1, C_2, C_3, C_4\}$  as illustrated below:

Given  $k=6$  and  $X = (X_1, X_2, \dots, X_n)$ , the goal is to find a class label from  $\{C_1, C_2, C_3, C_4\}$  for the unknown data  $X$ . As you can observe from the above figure example, of the 6 ( $k=6$ ) closest neighbors, 4 belongs to  $C_1$ , 1 belongs to  $C_2$ , 1 belongs to  $C_3$ , and 0 belongs to  $C_4$ . Therefore, by a majority vote,  $X$  is assigned to  $C_1$ , which is a predominant class.

## 13.5 An Informal kNN Algorithm

kNN algorithm can be summarized in the following simple steps:

- Step-1: determine  $k$  (selection of  $k$  depends on your data and project requirements – there is no magic formula for  $k$ 's selection)
- Step-2: calculate the distances between the new input and all the training data (selection of a distance function also depends on type of data)
- Step-3: sort the distance and determine  $k$  nearest neighbors based on the  $k$ -th minimum distance.
- Step-4: gather the categories of those neighbors.
- Step-5: determine the category based on majority vote

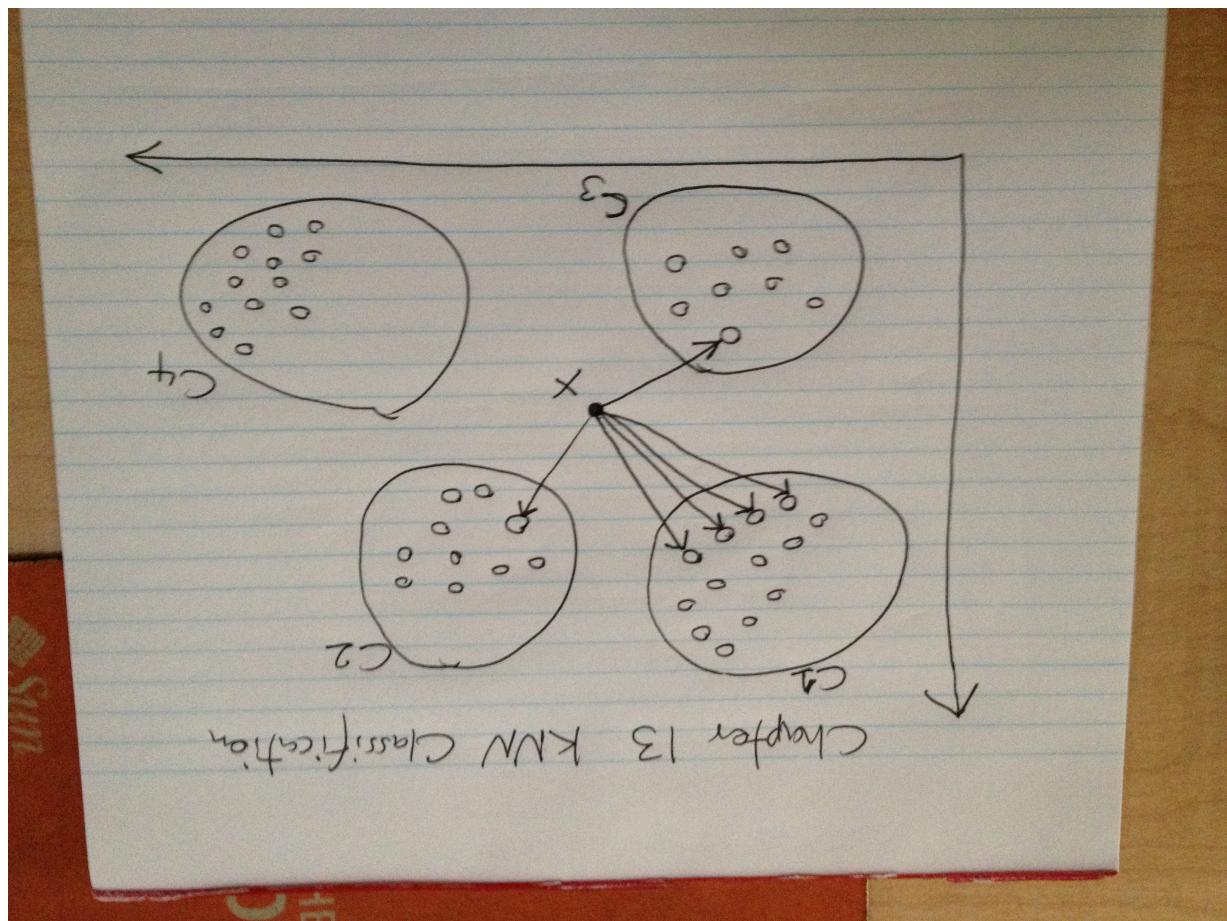


Figure 13.1: kNN Classification with 4 Classes {C<sub>1</sub>, C<sub>2</sub>, C<sub>3</sub>, C<sub>4</sub>}

## 13.6 Formal kNN Algorithm

To define kNN algorithm in formal terms, we need some definitions.

**Distance Function** : The distance function depends on application domain, but most of the time the Euclidean distance function works for most of metric data. The distance between two points  $(x_1, x_2, \dots, x_n)$  and  $(y_1, y_2, \dots, y_n)$  is defined as:  $|x, y|$ .

**k Nearest Neighbors** Given an object  $q$  (called a query object), a training dataset  $S$  and an integer  $k$ , the  $k$  nearest neighbors of  $q$  from  $S$ , denoted as  $\text{kNN}(q, S)$ , is a set of  $k$  objects from  $S$  such that

$$\forall o \in \text{kNN}(q, S), \forall s \in \{S - \text{kNN}(q, S)\}, |o, q| \leq |s, q|$$

**kNN Join** Given two data sets  $R$  and  $S$  (where  $S$  is a training dataset), and an integer  $k$ , then kNN join of  $R$  and  $S$  is defined as:

$$\text{kNNjoin}(R, S) = \{(r, s) | \forall r \in R, \forall s \in \text{kNN}(r, S)\}$$

Basically, this combines each object  $r \in R$  with its  $k$  Nearest Neighbors from  $S$ .

### 13.6.1 Java-like Non-MapReduce Solution for kNN

Given  $K > 0$ , query set (this is the data that we want to classify), and training set (this is the training data that we know classification of its data objects), we provide a Java-like non-MapReduce solution for kNN clustering algorithm. Below, *Point* class is used to represent an n-dimensional data.

**Listing 13.1:** Classify Method

```
1 /**
2 * Classify queried data using kNN Analysis
3 * @param k an integer > 0
4 * @param querySet a set of data objects that we want to classify
5 * @param trainingSet a set of training data (already classified)
6 */
7 public static void classify(int k,
8     Point[] querySet,
9     Point[] trainingSet) {
10    foreach (Point query : querySet) {
11        knn(k, query, trainingSet);
12    }
13 }
```

The kNN algorithm (non-MapReduce version) is illustrated below:

### Listing 13.2: kNN Algorithm

```
1 /**
2  * kNN Analysis
3  * @param k an integer > 0
4  * @param query a data object that we want to classify
5  * @param trainingSet a set of training data (already classified)
6 */
7 public static void knn(int k,
8                      Point query,
9                      Point[] trainingSet) {
10    foreach (Point training : trainingSet) {
11        // Create a fixed sized sorted map of length k,
12        // where we map the distance to a training point.
13        SortedMap<Double, Point> map = new TreeMap<Double, Point>(k);
14
15        // Calculate distance of test point to training point.
16        double d = calculateEuclidianDistance(query, training);
17
18        // Insert training point into sorted list, discarding
19        // if training point not within k nearest neighbours
20        // to query point.
21        map.put(d, training);
22    }
23
24    // Do majority vote on k nearest neighbours
25    // and assign the corresponding label to the
26    // query point.
27    query.label = majorityVote(map, k);
28 }
```

Below is the definition of the Euclidean distance function:

### Listing 13.3: Euclidean distance function

```
1 /**
2  * @param query an n-dimensional query data object
3  * @param reference an n-dimensional reference data object
4  * @return the quadratic Euclidean distance.
5 */
6 public static double calculateEuclidianDistance(Point query,
7                                                 Point reference) {
8     double sum = 0.0;
9
10    // n is the number of dimensions in the vector
11    // space. here n is equal to query.length, which
12    // is equal to reference.length
13    int n = query.length; // which is equal to
14
15    for (int i = 0; i < n; i++) {
16        double difference = reference.vector[i] - query.vector[i];
```

```

17         sum += difference * difference;
18     }
19
20     return Math.sqrt(sum);
21 }
```

---

## 13.7 kNN Implementation in Spark

Given two data sets R and S, the goal is to find  $k$  nearest neighbors from a dataset S for every object in dataset R. In data mining, S is called a training dataset. The complexity of this kNN join algorithm is  $N^2$ , because for every object r in R, you want to calculate  $\text{distance}(r, s)$  for every s in S. Finding  $\text{distance}(r, s)$  requires the Cartesian product of R and S. Spark provides higher level API to perform a Cartesian product between two data sets. For example, `JavaPairRDD.cartesian(JavaPairRDD)` finds the cartesian product of two data sets. The signature of `cartesian()` is defined as:

```
<U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,> other)
Description: Return the Cartesian product of this RDD
and another one, that is, the RDD of all pairs of elements
(a, b) where a is in this and b is in other.
```

For example, the following example shows how to perform cartesian product of two data sets in Spark:

```
// data set R
List<Tuple2<String, String>> listR = new ArrayList<Tuple2<String, String>>();
listR.add(new Tuple2<String, String>("r1", "R1"));
listR.add(new Tuple2<String, String>("r2", "R2"));
listR.add(new Tuple2<String, String>("r3", "R3"));

// data set S
List<Tuple2<String, String>> listS = new ArrayList<Tuple2<String, String>>();
listS.add(new Tuple2<String, String>("s1", "S1"));
listS.add(new Tuple2<String, String>("s2", "S2"));
listS.add(new Tuple2<String, String>("s3", "S3"));
```

```

listS.add(new Tuple2<String, String>("s4", "S4"));

JavaPairRDD<String, String> R = ctx.parallelizePairs(listR);
JavaPairRDD<String, String> S = ctx.parallelizePairs(listS);

// perform the Cartesian product of this R and S
JavaPairRDD<Tuple2<String, String>, Tuple2<String, String>> cart = R.cartesian(S);

// save output
cart.saveAsTextFile("/output/z");

```

The output of cartesian product of R and S are shown below.

```

# 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))

```

### 13.7.1 Formalizing kNN for Spark Implementation

Let R (called query data set) and S (called training data set) be d-dimensional data sets, which we want to find the  $k\text{NN}(R, S)$ . Further assume that all training data (S) has been classified into  $C = \{C_1, C_2, \dots\}$ , where  $C$  denotes all available classifications. We define R, S, and C as:

- $R = \{R_1, R_2, \dots, R_m\}$
- $S = \{S_1, S_2, \dots, S_n\}$
- $C = \{C_1, C_2, \dots, C_p\}$

where

- $R_i = (r_i, a_1, a_2, \dots, a_d)$
- $S_j = (s_j, b_1, b_2, \dots, b_d, C_j)$
- $r_i$  is a unique record ID for  $R_i$
- $a_1, a_2, \dots, a_d$  are attributes of  $R_i$
- $s_j$  is a unique record ID for  $S_j$
- $b_1, b_2, \dots, b_d$  are attributes of  $S_j$
- $C_j$  is a classification identifier for  $S_j$

The goal is to find  $kNN(R, S)$ . To find this, we will perform a cartesian product of  $R$  and  $S$ , and then group the data by  $r_i$  ( a unique record ID for  $R_i$ ). After grouping, we will find the  **$k$ -nearest-data-points** from  $S$  (by sorting distances ascendingly and then picking the first  $k$  elements). Once the  **$k$ -nearest-data-points** from  $S$  are found, then we will select the classification by a majority rule.

### 13.7.2 Input Data Set Formats

We are assuming that  $R$  and  $S$  both represent  $d$ -dimensional data points. The input record for query data set ( $R$ ) will have the following format:

```
<unique-record-id><;><a-1><,><a-2><,>...<,><a-d>
```

The input record for training data set ( $S$ ) will have the following format:

```
<unique-record-id><;><classification-id><;><b-1><,><b-2><,>...<,><b-d>
```

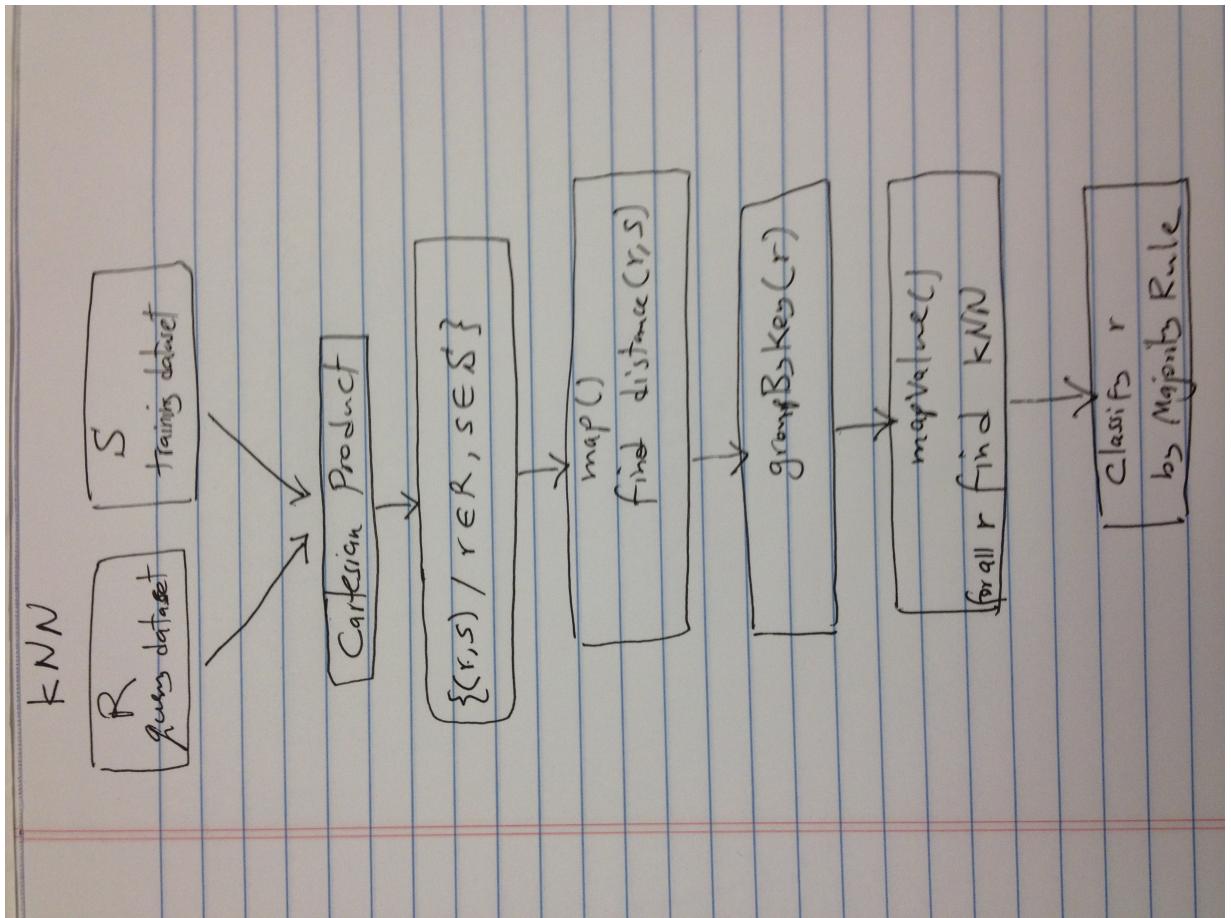


Figure 13.2: kNN Implementation in Spark

### 13.7.3 kNN Implementation in Spark

#### 13.7.3.1 High Level Steps

##### Listing 13.4: High Level Steps

```
1 // STEP-0: Import required classes and interfaces
2
3 public class kNN {
4
5     static JavaSparkContext createJavaSparkContext() throws Exception {...}
6     static List<Double> splitOnToListOfDouble(String str, String delimiter) {...}
7     static double calculateDistance(String rAsString, String sAsString, int d) {...}
8     static SortedMap<Double, String> findNearestK(Iterable<Tuple2<Double, String>> neighbors, int k) {...}
9     static Map<String, Integer> buildClassificationCount(Map<Double, String> nearestK) {...}
10    static String classifyByMajority(Map<String, Integer> majority) {...}
11
12    public static void main(String[] args) throws Exception {
13        // STEP-1: Handle input parameters
14        // STEP-2: Create a Spark context object
15        // STEP-3: Broadcast shared objects
16        // STEP-4: Create RDDs for query and training datasets
17        // STEP-5: Perform cartesian product of (R, S)
18        // STEP-6: Find distance(r, s) for r in R and s in S
19        // STEP-7: Group distances by r in R
20        // STEP-8: find the k-nearest-neighbors and classify r
21        System.exit(0);
22    }
23 }
```

#### 13.7.3.2 STEP-0: Import required classes and interfaces

Most of the Spark classes and interfaces are imported from two packages: `org.apache.spark.api.java` and `org.apache.spark.api.java.function`. The `Broadcast` class is used to broadcast shared objects and data structures to all cluster nodes (this is similar to Hadoop's `Configuration` object which you can `set()` and `get()` objects in mappers and reducers).

##### Listing 13.5: STEP-0: Import required classes and interfaces

```
1 // STEP-0: Import required classes and interfaces
2 import scala.Tuple2;
3 import org.apache.spark.SparkConf;
4 import org.apache.spark.broadcast.Broadcast;
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.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;

```

---

### 13.7.3.3 High Level Steps

The `createJavaSparkContext()` method creates an instance of a `JavaSparkContext` objects, which is a factory class for creating RDDs. Notes that I have hard coded the YARN's resource manager (as myserver100), but for production deployment, you should read and load all your configurations from a configuration file (such as XML).

#### **Listing 13.6:** Create a Spark context Object

```

1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     conf.set("yarn.resourcemanager.hostname", "myserver100");
4     conf.set("yarn.resourcemanager.scheduler.address", "myserver100:8030");
5     conf.set("yarn.resourcemanager.resource-tracker.address", "myserver100:8031");
6     conf.set("yarn.resourcemanager.address", "myserver100:8032");
7     conf.set("mapreduce.framework.name", "yarn");
8     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
9     conf.set("spark.executor.memory", "1g");
10    JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "kNN", conf);
11    return ctx;
12 }

```

---

### 13.7.3.4 Method `splitOnToListOfDouble()`

The `splitOnToListOfDouble()` method accepts all attributes as a String object and returns a `List<Double>`.

#### **Listing 13.7:** Method `splitOnToListOfDouble()`

```

1 /**
2  * @param str a comma (or semicolon) separated list of double values
3  * str is like "1.1,2.2,3.3" or "1.1;2.2;3.3"
4  *
5  * @param delimiter a delimiter such as ",", ";", ...

```

```

6   * @return a List<Double> (all attributes for a dataset record)
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  }

```

---

### 13.7.3.5 Method calculateDistance()

The `calculateDistance()` accepts two vectors of R and S and computes the **Euclidean distance** between them. The **Euclidean distance** between two points  $r = (r_1, r_2, \dots, r_d)$  and  $s = (s_1, s_2, \dots, s_d)$  is defined as

$$\text{distance}(r, s) = \sqrt{(r_1 - s_1)^2 + (r_2 - s_2)^2 + \dots + (r_d - s_d)^2}$$

Based on your project and data requirements, you may select and use other distance functions<sup>2</sup>.

**Listing 13.8:** Method calculateDistance()

```

1 /**
2  * @param rAsString = "r.1,r.2,...,r.d"
3  * @param sAsString = "s.1,s.2,...,s.d"
4  * @param d dimension of R and S
5 */
6 static double calculateDistance(String rAsString, String sAsString, int d) {
7   List<Double> r = splitOnToListOfDouble(rAsString, ",");
8   List<Double> s = splitOnToListOfDouble(sAsString, ",");
9
10  // d is the number of dimensions in the vector
11  if (r.size() != d) {
12    return Double.NaN;
13  }
14  if (s.size() != d) {
15    return Double.NaN;
16  }
17

```

---

<sup>2</sup>For details on distance functions, refer to [http://www.saedsayad.com/k\\_nearest\\_neighbors.htm](http://www.saedsayad.com/k_nearest_neighbors.htm)

```

18     // here we have (r.size() == s.size() == d)
19     double sum = 0.0;
20     for (int i = 0; i < d; i++) {
21         double difference = r.get(i) - s.get(i);
22         sum += difference * difference;
23     }
24     return Math.sqrt(sum);
25 }
```

### 13.7.3.6 Method findNearestK()

Given  $\{(distance, classification)\}$ , it finds the k-nearest-neighbors based on the distance.

**Listing 13.9:** Method buildClassificationCount()

```

1 static SortedMap<Double, String> findNearestK(Iterable<Tuple2<Double, String>> neighbors, int k) {
2     // keep only k-nearest-neighbors
3     SortedMap<Double, String> nearestK = new TreeMap<Double, String>();
4     for (Tuple2<Double, String> neighbor : neighbors) {
5         Double distance = neighbor._1;
6         String classificationID = neighbor._2;
7         nearestK.put(distance, classificationID);
8         // keep only k-nearest-neighbors
9         if (nearestK.size() > k) {
10             // remove the last highest distance neighbor from nearestK
11             nearestK.remove(nearestK.lastKey());
12         }
13     }
14     return nearestK;
15 }
16
17 \subsubsection{Method buildClassificationCount()}
18 The \tt buildClassificationCount() is a simple method, which counts the
19 classifications (will be used to select the classification based on
20 a mority count).
21
22 \begin{pyglist}[numbers=left, numbersep=5pt, language=java,
23 caption={Method buildClassificationCount()},%
24 listingnamefont=\sffamily\bfseries\color{white}, fontsize=\scriptsize,%
25 captionfont=\sffamily\color{white}, captionbgcolor=gray,%
26 fvsset={frame=bottomline, framerule=4pt, rulecolor=\color{gray}}]
27     static Map<String, Integer> buildClassificationCount(Map<Double, String> nearestK) {
28         Map<String, Integer> majority = new HashMap<String, Integer>();
29         for (Map.Entry<Double, String> entry : nearestK.entrySet()) {
30             String classificationID = entry.getValue();
31             Integer count = majority.get(classificationID);
32             if (count == null){
33                 majority.put(classificationID, 1);
34             }
35             else {
36                 majority.put(classificationID, count+1);
37             }
38         }
39         return majority;
40     }
41 }
```

```
37         }
38     }
39     return majority;
40 }
```

---

### 13.7.3.7 Method classifyByMajority()

The `classifyByMajority()` method select a classification based on a majority rule. For example, for a given query point  $r$ , if  $k=6$  and classifications are  $\{C_1, C_2, C_3, C_3, C_3, C_4\}$  (total of 6), then  $C_3$  is selected based on a majority rule.

**Listing 13.10:** Method classifyByMajority()

```
1  static String classifyByMajority(Map<String, Integer> majority) {
2     int votes = 0;
3     String selectedClassification = null;
4     for (Map.Entry<String, Integer> entry : majority.entrySet()) {
5         if (selectedClassification == null) {
6             selectedClassification = entry.getKey();
7             votes = entry.getValue();
8         }
9         else {
10             int count = entry.getValue();
11             if (count > votes) {
12                 selectedClassification = entry.getKey();
13                 votes = count;
14             }
15         }
16     }
17     return selectedClassification;
18 }
```

---

### 13.7.3.8 STEP-1: Handle input parameters

This step reads 4 input parameters:

1. k as an integer for kNN
2. d as an integer for dimension of R and S vectors
3. Query data set R (as an HDFS file)
4. Training data set S (as an HDFS file)

#### **Listing 13.11: STEP-1: Handle input parameters**

```
1 // STEP-1: Handle input parameters
2 if (args.length < 4) {
3     System.err.println("Usage: kNN <k-knn> <d-dimension> <R> <S>");
4     System.exit(1);
5 }
6 Integer k = Integer.valueOf(args[0]); // k for kNN
7 Integer d = Integer.valueOf(args[1]); // d-dimension
8 String datasetR = args[2];
9 String datasetS = args[3];
```

#### **13.7.3.9 STEP-2: Create a Spark context object**

This step creates a JavaSparkContext object, which is a factory class for creating new RDDs.

#### **Listing 13.12: STEP-2: Create a Spark context object**

```
1 // STEP-2: Create a Spark context object
2 JavaSparkContext ctx = createJavaSparkContext();
```

#### **13.7.3.10 STEP-3: Broadcast shared objects**

To access shared objects and data structures from all cluster nodes, Spark offers a `Broadcast` class, which you can register objects and read them from all cluster nodes. Below, we broadcast two integer values (k and d), which are accessed by all cluster nodes.

#### **Listing 13.13: STEP-3: Broadcast shared objects**

```
1 // STEP-3: Broadcast shared objects
2 // broadcast k and d as global shared objects,
3 // which can be accessed from all cluster nodes
4 final Broadcast<Integer> broadcastK = ctx.broadcast(k);
5 final Broadcast<Integer> broadcastD = ctx.broadcast(d);
```

#### **13.7.3.11 STEP-4: Create RDDs for query and training datasets**

This step create two RDDs (one for R and another one for S). These RDDs represent raw data as String objects.

### **Listing 13.14: STEP-4: Create RDDs for query and training datasets**

```
1 // STEP-4: Create RDDs for query and training datasets
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");
```

For debugging/understanding purposes, this step creates the following outputs.

#### **Query Data Set (R)**

```
# 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
```

#### **Training Data Set (S)**

```
# 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

```

### 13.7.3.12 STEP-5: Perform cartesian product of (R, S)

This step creates the cartesian product of query data set (R) with training data set (S). This step creates the following RDD:

$$\{(r, s) / r \in R, s \in S\}$$

**Listing 13.15:** STEP-5: Perform cartesian product of (R, S)

```

1 // STEP-5: Perform cartesian product of (R, S)
2 //<U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,> other)
3 // Return the Cartesian product of this RDD and another
4 // one, that is, the RDD of all pairs of elements (a, b)
5 // where a is in this and b is in other.
6 JavaPairRDD<String,String> cart = R.cartesian(S);
7 cart.saveAsTextFile("/output/cart");

```

For debugging/understanding purposes, this step creates the following outputs:

```

# 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;3.0,3.0,102;c1;1.2,1.0)
(1000;3.0,3.0,103;c1;1.6,1.5)
(1000;3.0,3.0,104;c1;1.3,1.7)
(1000;3.0,3.0,105;c1;2.0,2.1)
(1000;3.0,3.0,106;c1;2.0,2.2)
(1000;3.0,3.0,107;c1;2.3,2.3)
(1000;3.0,3.0,208;c2;9.0,9.0)
(1000;3.0,3.0,209;c2;9.1,9.2)
(1000;3.0,3.0,210;c2;9.2,9.0)
(1000;3.0,3.0,211;c2;10.6,10.5)
(1000;3.0,3.0,212;c2;10.3,10.7)
(1000;3.0,3.0,213;c2;9.6,9.1)
(1000;3.0,3.0,214;c2;9.4,10.4)
(1000;3.0,3.0,215;c2;10.3,10.3)
(1000;3.0,3.0,300;c3;10.0,1.0)
(1000;3.0,3.0,301;c3;10.1,1.2)
(1000;3.0,3.0,302;c3;10.2,1.0)
(1000;3.0,3.0,303;c3;10.6,1.5)
(1000;3.0,3.0,304;c3;10.3,1.7)
(1000;3.0,3.0,305;c3;10.0,2.1)
(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;10.1,3.2,102;c1;1.2,1.0)
(1001;10.1,3.2,103;c1;1.6,1.5)
(1001;10.1,3.2,104;c1;1.3,1.7)
(1001;10.1,3.2,105;c1;2.0,2.1)
(1001;10.1,3.2,106;c1;2.0,2.2)
(1001;10.1,3.2,107;c1;2.3,2.3)
(1001;10.1,3.2,208;c2;9.0,9.0)
(1001;10.1,3.2,209;c2;9.1,9.2)
(1001;10.1,3.2,210;c2;9.2,9.0)
(1001;10.1,3.2,211;c2;10.6,10.5)
(1001;10.1,3.2,212;c2;10.3,10.7)
(1001;10.1,3.2,213;c2;9.6,9.1)
(1001;10.1,3.2,214;c2;9.4,10.4)
(1001;10.1,3.2,215;c2;10.3,10.3)
(1001;10.1,3.2,300;c3;10.0,1.0)
(1001;10.1,3.2,301;c3;10.1,1.2)
(1001;10.1,3.2,302;c3;10.2,1.0)
(1001;10.1,3.2,303;c3;10.6,1.5)
(1001;10.1,3.2,304;c3;10.3,1.7)
(1001;10.1,3.2,305;c3;10.0,2.1)
(1001;10.1,3.2,306;c3;10.0,2.2)
(1001;10.1,3.2,307;c3;10.3,2.3)
(1003;2.7,2.7,100;c1;1.0,1.0)
(1003;2.7,2.7,101;c1;1.1,1.2)
(1003;2.7,2.7,102;c1;1.2,1.0)
(1003;2.7,2.7,103;c1;1.6,1.5)
(1003;2.7,2.7,104;c1;1.3,1.7)
(1003;2.7,2.7,105;c1;2.0,2.1)
(1003;2.7,2.7,106;c1;2.0,2.2)
(1003;2.7,2.7,107;c1;2.3,2.3)
(1003;2.7,2.7,208;c2;9.0,9.0)
(1003;2.7,2.7,209;c2;9.1,9.2)
(1003;2.7,2.7,210;c2;9.2,9.0)
(1003;2.7,2.7,211;c2;10.6,10.5)
(1003;2.7,2.7,212;c2;10.3,10.7)
(1003;2.7,2.7,213;c2;9.6,9.1)
(1003;2.7,2.7,214;c2;9.4,10.4)
(1003;2.7,2.7,215;c2;10.3,10.3)
(1003;2.7,2.7,300;c3;10.0,1.0)
(1003;2.7,2.7,301;c3;10.1,1.2)
(1003;2.7,2.7,302;c3;10.2,1.0)
(1003;2.7,2.7,303;c3;10.6,1.5)
(1003;2.7,2.7,304;c3;10.3,1.7)
(1003;2.7,2.7,305;c3;10.0,2.1)
(1003;2.7,2.7,306;c3;10.0,2.2)
(1003;2.7,2.7,307;c3;10.3,2.3)
(1004;5.0,5.0,100;c1;1.0,1.0)
(1004;5.0,5.0,101;c1;1.1,1.2)
(1004;5.0,5.0,102;c1;1.2,1.0)
(1004;5.0,5.0,103;c1;1.6,1.5)
(1004;5.0,5.0,104;c1;1.3,1.7)
(1004;5.0,5.0,105;c1;2.0,2.1)
(1004;5.0,5.0,106;c1;2.0,2.2)
(1004;5.0,5.0,107;c1;2.3,2.3)
(1004;5.0,5.0,208;c2;9.0,9.0)
(1004;5.0,5.0,209;c2;9.1,9.2)
(1004;5.0,5.0,210;c2;9.2,9.0)
(1004;5.0,5.0,211;c2;10.6,10.5)
(1004;5.0,5.0,212;c2;10.3,10.7)
(1004;5.0,5.0,213;c2;9.6,9.1)
(1004;5.0,5.0,214;c2;9.4,10.4)
(1004;5.0,5.0,215;c2;10.3,10.3)
(1004;5.0,5.0,300;c3;10.0,1.0)
(1004;5.0,5.0,301;c3;10.1,1.2)
(1004;5.0,5.0,302;c3;10.2,1.0)
(1004;5.0,5.0,303;c3;10.6,1.5)
(1004;5.0,5.0,304;c3;10.3,1.7)
(1004;5.0,5.0,305;c3;10.0,2.1)
(1004;5.0,5.0,306;c3;10.0,2.2)
(1004;5.0,5.0,307;c3;10.3,2.3)
(1005;13.1,2.2,100;c1;1.0,1.0)
(1005;13.1,2.2,101;c1;1.1,1.2)
(1005;13.1,2.2,102;c1;1.2,1.0)
(1005;13.1,2.2,103;c1;1.6,1.5)
(1005;13.1,2.2,104;c1;1.3,1.7)

```

```

(1005;13.1,2.2,105;c1;2.0,2.1)
(1005;13.1,2.2,106;c1;2.0,2.2)
(1005;13.1,2.2,107;c1;2.3,2.3)
(1005;13.1,2.2,208;c2;9.0,9.0)
(1005;13.1,2.2,209;c2;9.1,9.2)
(1005;13.1,2.2,210;c2;9.2,9.0)
(1005;13.1,2.2,211;c2;10.6,10.5)
(1005;13.1,2.2,212;c2;10.3,10.7)
(1005;13.1,2.2,213;c2;9.6,9.1)
(1005;13.1,2.2,214;c2;9.4,10.4)
(1005;13.1,2.2,215;c2;10.3,10.3)
(1005;13.1,2.2,300;c3;10.0,1.0)
(1005;13.1,2.2,301;c3;10.1,1.2)
(1005;13.1,2.2,302;c3;10.2,1.0)
(1005;13.1,2.2,303;c3;10.6,1.5)
(1005;13.1,2.2,304;c3;10.3,1.7)
(1005;13.1,2.2,305;c3;10.0,2.1)
(1005;13.1,2.2,306;c3;10.0,2.2)
(1006;12.7,12.7,100;c1;1.0,1.0)
(1006;12.7,12.7,101;c1;1.1,1.2)
(1006;12.7,12.7,102;c1;1.2,1.0)
(1006;12.7,12.7,103;c1;1.6,1.5)
(1006;12.7,12.7,104;c1;1.3,1.7)
(1006;12.7,12.7,105;c1;2.0,2.1)
(1006;12.7,12.7,106;c1;2.0,2.2)
(1006;12.7,12.7,107;c1;2.3,2.3)
(1006;12.7,12.7,208;c2;9.0,9.0)
(1006;12.7,12.7,209;c2;9.1,9.2)
(1006;12.7,12.7,210;c2;9.2,9.0)
(1006;12.7,12.7,211;c2;10.6,10.5)
(1006;12.7,12.7,212;c2;10.3,10.7)
(1006;12.7,12.7,213;c2;9.6,9.1)
(1006;12.7,12.7,214;c2;9.4,10.4)
(1006;12.7,12.7,215;c2;10.3,10.3)
(1006;12.7,12.7,300;c3;10.0,1.0)
(1006;12.7,12.7,301;c3;10.1,1.2)
(1006;12.7,12.7,302;c3;10.2,1.0)
(1006;12.7,12.7,303;c3;10.6,1.5)
(1006;12.7,12.7,304;c3;10.3,1.7)
(1006;12.7,12.7,305;c3;10.0,2.1)
(1006;12.7,12.7,306;c3;10.0,2.2)
(1006;12.7,12.7,307;c3;10.3,2.3)

```

### 13.7.3.13 STEP-6: Find distance(r, s) for r in R and s in S

This step finds an Euclidian distance between every pair of (R, S). Based on your data and project requirements, you may select and use different distance algorithms (such as Minkowski). Note that the selection of the distance algorithm will influence the bias of kNN classification. This step creates the following RDD:

$$\{(r, (\text{distance}, \text{classification}))\}$$

**Listing 13.16:** STEP-6: Find distance(r, s) for r in R and s in S

```

1 // STEP-6: Find distance(r, s) for r in R and s in S
2 // (K, V), where K = unique-record-id-of-R, V=Tuple2(distance, classification)
3 // distance = distance(r, s) where r in R and s in S
4 // classification is extracted from s
5 JavaPairRDD<String, Tuple2<Double, String>> knnMapped =

```

```

6          //           input      K      V
7          cart.mapToPair(new PairFunction<Tuple2<String, String>, String, Tuple2<Double, String>>() {
8      public Tuple2<String, Tuple2<Double, String>> call(Tuple2<String, String> cartRecord) {
9          String rRecord = cartRecord._1;
10         String sRecord = cartRecord._2;
11         String[] rTokens = rRecord.split(",");
12         String rRecordID = rTokens[0];
13         String r = rTokens[1]; // r.1, r.2, ..., r.d
14         String[] sTokens = sRecord.split(",");
15         // sTokens[0] = s.recordID
16         String sClassificationID = sTokens[1];
17         String s = sTokens[2]; // s.1, s.2, ..., s.d
18         Integer d = broadcastD.value();
19         double distance = calculateDistance(r, s, d);
20         String K = rRecordID; // r.recordID
21         Tuple2<Double, String> V = new Tuple2<Double, String>(distance, sClassificationID);
22         return new Tuple2<String, Tuple2<Double, String>>(K, V);
23     }
24 });
25 knnMapped.saveAsTextFile("/output/knnMapped");

```

---

For debugging/understanding purposes, this step creates the following outputs:

```
# hadoop fs -cat /output/knnMapped/part*
(1000, (2.8284271247461903, c1))
(1000, (2.6172504656604803, c1))
(1000, (2.6907248094147422, c1))
(1000, (2.0518284528683193, c1))
(1000, (2.1400934559032696, c1))
(1000, (1.345362404707371, c1))
(1000, (1.2806248474865696, c1))
(1000, (0.9899494936611668, c1))
(1000, (8.48528137423857, c2))
(1000, (8.697700845625812, c2))
(1000, (8.627861844049196, c2))
(1000, (10.67754653466797, c2))
(1000, (10.61037228376083, c2))
(1000, (8.987213138676527, c2))
(1000, (9.783659846908007, c2))
(1000, (10.323759005323595, c2))
(1000, (7.280109889280518, c3))
(1000, (7.3246160308919945, c3))
(1000, (7.472616676907761, c3))
(1000, (7.746612162745725, c3))
(1000, (7.414849964766652, c3))
(1000, (7.057619995437555, c3))
(1000, (7.045565981523415, c3))
(1000, (7.333484846919642, c3))
(1001, (9.362157870918434, c1))
(1001, (9.219544457292887, c1))
(1001, (9.167878707749137, c1))
(1001, (8.668333173107735, c1))
(1001, (8.926925562588723, c1))
(1001, (8.174350127074323, c1))
(1001, (8.161494961096281, c1))
(1001, (7.85175139698144, c1))
(1001, (5.903388857258177, c2))
(1001, (6.082762530298219, c2))
(1001, (5.869412236331676, c2))
(1001, (7.3171032519706865, c2))
(1001, (7.502666192761076, c2))
(1001, (5.921148537234984, c2))
(1001, (7.23394774656273, c2))
(1001, (7.102816342831906, c2))
(1001, (2.202271554554524, c3))
```

(1001,(2.0,c3))
(1001,(2.202271554554524,c3))
(1001,(1.7720045146669352,c3))
(1001,(1.513274895042156,c3))
(1001,(1.104536101718726,c3))
(1001,(1.00498756212089,c3))
(1001,(0.9219544457292893,c3))
(1003,(2.4041630560342617,c1))
(1003,(2.193171219946131,c1))
(1003,(2.267156809750927,c1))
(1003,(1.6278820596099708,c1))
(1003,(1.7204650534085255,c1))
(1003,(0.9219544457292889,c1))
(1003,(0.8602325267042628,c1))
(1003,(0.5656854249492386,c1))
(1003,(8.909545442950499,c2))
(1003,(9.121951545584968,c2))
(1003,(9.052071586106685,c2))
(1003,(11.101801655587257,c2))
(1003,(11.034491379306976,c2))
(1003,(9.411163583744573,c2))
(1003,(10.206860437960342,c2))
(1003,(10.748023074035522,c2))
(1003,(7.495331880577404,c3))
(1003,(7.5504966724050675,c3))
(1003,(7.690253571892151,c3))
(1003,(7.99061950038919,c3))
(1003,(7.66550715869472,c3))
(1003,(7.3246160308919945,c3))
(1003,(7.3171032519706865,c3))
(1003,(7.610519036176179,c3))
(1004,(5.656854249492381,c1))
(1004,(5.445181356024793,c1))
(1004,(5.517245689653488,c1))
(1004,(4.879549159502341,c1))
(1004,(4.957822102496216,c1))
(1004,(4.172529209005013,c1))
(1004,(4.1036569057366385,c1))
(1004,(3.818376618407357,c1))
(1004,(5.656854249492381,c2))
(1004,(5.869412236331675,c2))
(1004,(5.8,c2))
(1004,(7.849203781276162,c2))
(1004,(7.783315488916019,c2))
(1004,(6.161980201201558,c2))
(1004,(6.9656299069072,c2))
(1004,(7.495331880577405,c2))
(1004,(6.4031242374328485,c3))
(1004,(6.360031446463138,c3))
(1004,(6.56048778674269,c3))
(1004,(6.603786792439623,c3))
(1004,(6.243396511515186,c3))
(1004,(5.78013840664737,c3))
(1004,(5.730619512757761,c3))
(1004,(5.948108943185221,c3))
(1005,(12.159358535712318,c1))
(1005,(12.041594578792296,c1))
(1005,(11.960351165413163,c1))
(1005,(11.52128465059344,c1))
(1005,(11.81058846967415,c1))
(1005,(11.10045044131093,c1))
(1005,(11.1,c1))
(1005,(10.800462953040487,c1))
(1005,(7.940403012442126,c2))
(1005,(8.06225774829855,c2))
(1005,(7.839005038904045,c2))
(1005,(8.66833173107735,c2))
(1005,(8.94930164873215,c2))
(1005,(7.736924453553879,c2))
(1005,(8.996110270555825,c2))
(1005,(8.570297544426332,c2))
(1005,(3.3241540277189316,c3))
(1005,(3.1622776601683795,c3))
(1005,(3.1384709652950433,c3))
(1005,(2.596150997149434,c3))
(1005,(2.8442925306655775,c3))
(1005,(3.101612483854164,c3))

```

(1005, (3.099999999999999,c3))
(1005, (2.801785145224379,c3))
(1006, (16.54629867976521,c1))
(1006, (16.33431969810803,c1))
(1006, (16.405486887014355,c1))
(1006, (15.76863976378432,c1))
(1006, (15.84171707865028,c1))
(1006, (15.06154042586614,c1))
(1006, (14.99130828181999,c1))
(1006, (14.707821048680186,c1))
(1006, (5.23259018078045,c2))
(1006, (.020956084253276,c2))
(1006, (5.093132631298737,c2))
(1006, (3.0413812651491092,c2))
(1006, (3.12409870362661,c2))
(1006, (4.75078940808788,c2))
(1006, (4.0224370722237515,c2))
(1006, (3.3941125496954263,c2))
(1006, (12.0074976577137,c3))
(1006, (11.79025020938911,c3))
(1006, (11.964113005150026,c3))
(1006, (11.395174417269795,c3))
(1006, (11.258774356030056,c3))
(1006, (10.938464243210744,c3))
(1006, (10.84158659975559,c3))
(1006, (10.673331251301065,c3))

```

### 13.7.3.14 STEP-7: Group distances by r in R

After finding the distances for  $\{(r, s)\}$ , to find the k-nearest-neighbors, we group data by  $r$ . Once data is grouped by  $r$ , then we scan (STEP-8) the group values and find the **k-smallest** (or **k-nearest**) distances.

#### **Listing 13.17:** STEP-7: Group distances by r in R

```

1 // STEP-7: Group distances by r in R
2 // now group the results by r.recordID and then find the k-nearest-neighbors.
3 JavaPairRDD<String, Iterable<Tuple2<Double, String>>> knnGrouped = knnMapped.groupByKey();

```

This step creates the following RDD:

$$\{(r, \{(distance, classification)\})\}$$

### 13.7.3.15 STEP-8: find the k-nearest-neighbors and classify R

This step scans the group values and find the **k-smallest** (or **k-nearest**) distances. To find k elements, which have the smallest distances, we use **SortedMap**, which keeps only k nearest neighbors. At every iteration, we make sure that we keep only k nearest elements. Once the k-nearest-neighbors found, then we classify the query data by the majority rule.

**Listing 13.18:** STEP-8: find the k-nearest-neighbors and classify R

```
1 // STEP-8: find the k-nearest-neighbors and classify r
2 // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
3 // Pass each value in the key-value pair RDD through a
4 // map function without changing the keys;
5 // this also retains the original RDD's partitioning.
6 // Generate (K,V) pairs where K=r.recordID, V = classificationID
7 JavaPairRDD<String, String> knnOutput =
8     knnGrouped.mapValues(new Function<Iterable<Tuple2<Double, String>>, // input
9                         String // output (classification)
10                        >() {
11     public String call(Iterable<Tuple2<Double, String>> neighbors) {
12         Integer k = broadcastK.value();
13         SortedMap<Double, String> nearestK = findNearestK(neighbors, k);
14         // now we have the k-nearest-neighbors in nearestK
15         // we need to find out the classification by majority
16         // by counting classifications
17         Map<String, Integer> majority = buildClassificationCount(nearestK);
18
19         // find a classificationID with majority of vote
20         String selectedClassification = classifyByMajority(majority);
21         return selectedClassification;
22     }
23 });
24 knnOutput.saveAsTextFile("/output/knnOutput");
25
26 System.exit(0);
27 }
28 }
```

This step creates the final output, which includes classification of all query dataset:

```
# hadoop fs -cat /output/knnOutput/part*
(1005,c3)
(1001,c3)
(1000,c1)
(1004,c1)
(1006,c2)
(1003,c1)
```

**13.7.3.16 YARN Shell Script****Listing 13.19:** YARN Shell Script

```

1 hadoop@hnode01319:~/spark_mahmoud_examples# cat run_knn.sh
2 #!/bin/bash
3 source /home/hadoop/conf/env_2.4.0.sh
4 export SPARK_HOME=/home/hadoop/spark-1.0.0
5 source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
6 source $SPARK_HOME/conf/spark-env.sh
7
8 # system jars:
9 CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
10 #
11 jars='find $SPARK_HOME -name \'*.jar\''
12 for j in $jars ; do
13     CLASSPATH=$CLASSPATH:$j
14 done
15
16 # app jar:
17 export MP=/home/hadoop/spark_mahmoud_examples
18 export CLASSPATH=$MP/mp.jar:$CLASSPATH
19 export CLASSPATH=$MP/commons-math3-3.0.0.jar:$CLASSPATH
20 export CLASSPATH=$MP/commons-math-2.2.jar:$CLASSPATH
21 export SPARK_CLASSPATH=$CLASSPATH
22 export HADOOP_HOME=/usr/local/hadoop/hadoop-2.4.0
23 export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
24 export JAVA_HOME=/usr/java/jdk7
25 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
26 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
27 export MY_JAR=$MP/mp.jar
28 export SPARK_JAR=$MP/spark-assembly-1.0.0-hadoop2.4.0.jar
29 export YARN_APPLICATION_CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
30 k=4
31 d=2
32 R=/knn/R.txt
33 S=/knn/S.txt
34 $SPARK_HOME/bin/spark-submit --class kNN \
35   --master yarn-cluster \
36   --num-executors 12 \
37   --driver-memory 3g \
38   --executor-memory 7g \
39   --executor-cores 12 \
40   $MY_JAR $k $d $R $S

```

---

# Chapter 14

## Naive Bayes

### 14.1 Introduction

In data mining and machine learning, there are many classification algorithms. One of the simplest, but effective algorithms is Naive Bayesian classifier that uses the Bayes<sup>1</sup>. The main focus of this chapter is to present a distributed MapReduce solution (using Spark/Hadoop) to Naive Bayesian classifier, which is a combination of a supervised learning method and probabilistic classification. The Naive Bayes classifier is a linear classifier. To understand Naive Bayes, we need to understand some basic probabilities and conditional probabilities. When we are dealing with numeric data, it is better to use clustering (such as K-means<sup>2</sup> and nearest-neighbour methods and algorithms), but for classification of names, symbols, emails, and texts, it may be better to use a probabilistic method, such as the Naive Bayes Classifier (NBC). In some cases, NBC is used to classify numeric data as well. In the following sections, we will have an example of symbolic and numeric data.

A naive Bayes classifier is a probabilistic classifier based on applying Bayes' theorem with strong (naive) independence assumptions. In a nutshell, a naive Bayes classifier assigns inputs into one of the  $k$  classes  $\{C_1, C_2, \dots, C_k\}$  based on some properties (features) of inputs. Naive Bayes Classifiers have applications such as e-mail spam filtering and classification of documents.

---

<sup>1</sup>For details, see [http://en.wikipedia.org/wiki/Thomas\\_Bayes](http://en.wikipedia.org/wiki/Thomas_Bayes)

<sup>2</sup>For details, see [http://en.wikipedia.org/wiki/K-means\\_clustering](http://en.wikipedia.org/wiki/K-means_clustering)

## Building Naïve Bayes Classifier



Figure 14.1: Naive Bayes: Training Phase

For example, a spam filter (using Naive Bayes classifier) will assign each e-mail to one of the two clusters: "spam mail" or "not a spam mail". Since Naive Bayes is a supervised learning method, it has two distinct stages:

- **STAGE-1: Training** (see Figure 14.1):  
This stage trains data from a set of finite instances of data samples - this is the so called supervised learning method - learn from a set of samples and then use this learning information for the new data classification). The main goal at this stage is to use training data and to build a classifier (to be used in STAGE-2).
- **STAGE-2: Classification** (see Figure 14.2):  
Given a new instance of data, we want to classify this data; using Training data and Naive Bayes theorem we classify new data to one of the

## Classification Process

New Data =  $(X) = (X_1, X_2, \dots, X_m)$

Class C is a member of  $\{C_1, C_2, \dots, C_k\}$

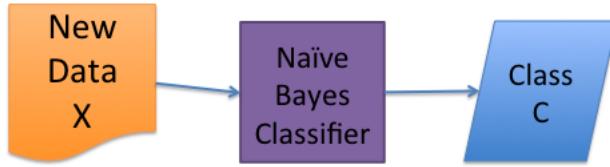


Figure 14.2: Naive Bayes: Classification

categories identified in Training.

## 14.2 Training and Learning Stage

Let each data set have  $m$  attributes ( $X = (x_1, x_2, \dots, x_m)$ ), the size of training data be  $n$ , and our classification has  $k$  distinct categories  $\{C_1, C_2, \dots, C_k\}$  where  $k \leq n$ , then we will have:

$$\begin{aligned} & (X_{11}, X_{12}, \dots, X_{1m}), \quad (c_1) \\ & (X_{21}, X_{22}, \dots, X_{2m}), \quad (c_2) \\ & \dots \\ & (X_{n1}, X_{n2}, \dots, X_{nm}), \quad (c_n) \end{aligned}$$

where  $c_i$  is a member of  $\{C_1, C_2, \dots, C_k\}$ .

### 14.2.1 Example: Training Data (Numeric Data)

The following is an example of training numeric data<sup>3</sup> (the classification column (Sex) is shown in blue and orange colors). Note that the first column is a meta data (not part of an actual data).

Person	Height (feet)	Weight (lbs)	Foot Siz (inches)	Classification (sex)
1	6.00	180	12	male
2	5.92	190	11	male
3	5.58	170	12	male
4	5.92	165	10	male
5	5.00	100	6	female
6	5.50	150	8	female
7	5.42	130	7	female
8	5.75	150	9	female

Here are the facts about this numeric training data:

- The Sex column is the classification column (in blue and orange colors). There are 2 ( $k = 2$ ) classification classes: { male, female }, where  $C_1 = \text{male}$  and  $C_2 = \text{female}$ .
- There are 3 attributes ( $m = 3$ ) per data set: (Height, Weight, Foot Size). We consider each data instance to be an m-dimensional vector of attribute values:  $X = (X_1, X_2, \dots, X_m)$ .
- Training data size is 8 ( $n = 8$ ) identified by first column (numbers 1, 2, ..., 8).
- The goal is to use this training data and build a classifier system (using Naive Bayes theorem) which will enable us to decide whether or not a person is male or female. This classification will be based on the values of the three attributes "height", "weight", and "foot size".

---

<sup>3</sup><http://www.ic.unicamp.br/~rocha/teaching/2011s2/mc906/aulas/naive-bayes-classifier.pdf>

### 14.2.2 Example: Training Data (Symbolic Data)

The following is an example of training data from Tom Mitchell's book[17]. Note that the first column is a meta data (not part of an actual data).

Day	Outlook	Temperature	Humidity	Wind	<i>PlayTennis Classification</i>
$D_1$	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
$D_7$	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

Here are the facts about this symbolic training data:

- The PlayTennis column is the classification column (in blue and red colors). There are 2 ( $k = 2$ ) classification classes: { Yes, No }, where  $C_1 = \text{Yes}$  and  $C_2 = \text{No}$ .
- There are 4 attributes ( $m = 4$ ) per data set: (Outlook, Temperature, Humidity, Wind). We consider each data instance to be an m-dimensional vector of attribute values:  $X = (X_1, X_2, \dots, X_m)$ .
- Training data size is 14 ( $n = 14$ ) identified by  $\{D_1, D_2, \dots, D_{14}\}$ .
- The goal is to use this training data and build a classifier system (using Naive Bayes theorem) which will enable us to decide whether or not to play the tennis game on the basis of the weather conditions, i.e. we wish to classify the data into two classes, one where the attribute PlayTennis has the value "Yes", and the other where it has the value "No." This

classification will be based on the values of the four attributes `outlook`, `temperature`, `humidity`, and `windy`.

Now the real question is: if input data is the following

```
X = (X1 = u1, X2=u2, X3=u3, X4=u4)
= (Outlook = Overcast,
  Temperature = Hot,
  Humidity = High,
  Wind = Strong)
```

then what will be our classification to PlayTennis? Will it be "[Yes](#)" or "[No](#)"? Naive Bayes classifier (using Bayes Theorem) can answer this question based on the data given in the training/learning phase. Here, Naive refers to the fact that we assume that all probabilities for our data attributes are independent. This means that for a given data  $X = (X_1, X_2, \dots, X_m)$ , we will assume that these attributes are independent of each other and therefore probabilities for each attribute will be independent. This is a very strong ("naive") assumption. Some statisticians are somewhat disturbed by use of the Naive Bayes Classifier because the "naive" assumption of independence is almost always invalid in the real world. But in real-world examples, the Naive Bayes method has been shown to perform surprisingly well in a wide variety of contexts.

### 14.3 Conditional Probability

Since Naive Bayes is based on Probabilistic Classification (especially on conditional probability), we will have a very short introduction to probability and conditional probability.

The conditional probability (denoted by  $P$ ) of event  $A$  given event  $B$  is dened as follows:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

Furthermore we say that if the events  $A$  and  $B$  are independent if and only if

$$P(A \cap B) = P(A)P(B)$$

## 14.4 The Naive Bayes Classifier

The Naive Bayes Classifier is a simple and stable method for classification and predictor selection. A Bayes classifier is a probabilistic classifier based on applying Bayes' theorem (from Bayesian statistics) with strong (naive) independence assumptions. The general form of Bayes' theorem is stated next. Let  $A$  be a sequence of mutually exclusive events  $\{A_1, A_2, \dots, A_n\}$  whose union is the sample space, and let  $E$  be any event. It is assumed that all of the events have non-zero probability ( $P(E) > 0$ ) and ( $P(A_i) > 0$  for all  $i$ ). The Bayes' Theorem states:

$$P(A_j|E) = \frac{P(A_j)P(E|A_j)}{\sum_{i=1}^n P(A_i)P(E|A_i)}$$

for any  $j \in \{1, 2, \dots, n\}$ .

A simpler formulation of Bayes's theorem can be stated as: Let  $A$  and  $B$  be two events (with some attribute values in a given statistical space). Then, Bayes' theorem gives the relationship between the probabilities of  $A$  and  $B$ ,  $P(A)$  and  $P(B)$ , and the conditional probabilities of  $A$  given  $B$  (denoted by  $P(A|B)$ ) and  $B$  given  $A$  (denoted by  $P(B|A)$ ). Bayes' theorem can be stated as (for both events,  $A$  and  $B$ , we assume with non-zero probability):

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

For more details on Bayes' formula/theorem, see [3].

Next, we formalize Bayes' therorm for classification: let  $X = (X_1 = u_1, \dots, X_m = u_m)$  be an instance of data, which needs to be classified, let  $C = \{C_1, C_2, \dots, C_k\}$  be a set of finite distinct classes (generated/deducted by training data).

Then, using Bayes's theorem, we can predict a class  $C^{predict} \in \{C_1, C_2, \dots, C_k\}$  for a given  $X$  as:

$$\begin{aligned}
C^{predict} &= \operatorname{argmax}_c P(C = c | X_1 = u_1, \dots, X_m = u_m) \\
&= \operatorname{argmax}_c \frac{P(C=c, X_1=u_1, \dots, X_m=u_m)}{P(X_1=u_1, \dots, X_m=u_m)} \\
&= \operatorname{argmax}_c \frac{P(X_1=u_1, \dots, X_m=u_m | C=c) P(C=c)}{P(X_1=u_1, \dots, X_m=u_m)} \\
&= \operatorname{argmax}_c P(X_1 = u_1, \dots, X_m = u_m | C = c) P(C = c) \\
&= \operatorname{argmax}_c P(C = c) \prod_{j=1}^m P(X_j = u_j | C = c)
\end{aligned}$$

Note that we dropped the denominator  $P(X_1 = u_1, \dots, X_m = u_m)$  from our classification algorithm since it is the same constant for all calculations, and it does not change the outcome of classification algorithm. Therefore, constructing a classifier from the probability model can be expressed as:

$$classify(X_1 = u_1, \dots, X_m = u_m) = C^{predict}$$

$$C^{predict} = \operatorname{argmax}_c P(C = c) \prod_{j=1}^m P(X_j = u_j | C = c)$$

#### 14.4.1 The Naive Bayes Classifier Example

Now the real question is if input data is the following

```

X = (Outlook = Overcast,
      Temperature = Hot,
      Humidity = High,
      Wind = Strong)
X = (Overcast, Hot, High, Strong)

```

$$X = (X_1, X_2, X_3, X_4)$$

Then how do we classify this data? Will the answer be "Yes" (play tennis) or "No" (do not play tennis). For our example, we have two classes:

- $C = (C_1, C_2) = ("Yes", "No")$
- $P(C_1) = P("Yes") = 9/14$
- $P(C_2) = P("No") = 5/14$

Now, following Bayes's classifier, we will have:

$$\begin{aligned} C^{predict} &= \underset{c}{\operatorname{argmax}} P(C = c) \prod_{j=1}^m P(X_j = u_j | C = c) \\ &= \max\{v_1, v_2\} \end{aligned}$$

where

$$\begin{aligned} v_1 &= \{P(C = C_1)P(X_1 | C = C_1)P(X_2 | C = C_1)P(X_3 | C = C_1)P(X_4 | C = C_1)\} \\ v_2 &= \{P(C = C_2)P(X_1 | C = C_2)P(X_2 | C = C_2)P(X_3 | C = C_2)P(X_4 | C = C_2)\} \end{aligned}$$

If  $v_1 > v_2$ , then our classification of  $X$  will be  $C_1 = "Yes"$ , otherwise it will be  $C_2 = "No"$ . The following are calculations of conditional probabilities for  $C_1 = "Yes"$ :

- $P(X_1 | C = C_1) = P("Overcast" | C = "Yes") = ?$   
Of the 9 cases where  $PlayTennis = "Yes"$ , there are 4 where  $Outlook = "Overcast"$ , thus  $P(Outlook = Overcast | play = Yes) = 4/9$ . In the notation of equation, we may write  $P(X_1 = Overcast | C_1) = 4/9$ .
- $P(X_2 | C = C_1) = P("Hot" | C = "Yes") = ?$   
Of the 9 cases where  $PlayTennis = "Yes"$ , there are 2 where  $Temperature = "Hot"$ , thus  $P(Temperature = Hot | PlayTennis = Yes) = 2/9$ . In the notation of equation, we may write  $P(X_2 = Hot | C_1) = 2/9$ .
- $P(X_3 | C = C_1) = P("High" | C = "Yes") = ?$   
Of the 9 cases where  $PlayTennis = "Yes"$ , there are 3 where  $Humidity = "High"$ , thus  $P(Humidity = High | PlayTennis = Yes) = 3/9$ . In the notation of equation, we may write  $P(X_3 = High | C_1) = 3/9$ .

- $P(X_4|C = C_1) = P(\text{"Strong"}|C = \text{"Yes"}) = ?$   
Of the 9 cases where  $PlayTennis = \text{"Yes"}$ , there are 3 where  $Wind = \text{"Strong"}$ , thus  $P(Wind = \text{"Strong"}|PlayTennis = Yes) = 3/9$ . In the notation of equation, we may write  $P(X_4 = Strong|C_1) = 3/9$ .

The following are calculations of conditional probabilities for  $C_2 = \text{"No"}$ :

- $P(X_1|C = C_2) = P(\text{"Overcast"}|C = \text{"No"}) = ?$   
Of the 5 cases where  $PlayTennis = \text{"No"}$ , there are 0 where  $Outlook = \text{"Overcast"}$ , thus  $P(Outlook = Overcast|play = No) = 0/5$ . In the notation of equation, we may write  $P(X_1 = Overcast|C_2) = 0/5$ .
- $P(X_2|C = C_2) = P(\text{"Hot"}|C = \text{"No"}) = ?$   
Of the 5 cases where  $PlayTennis = \text{"No"}$ , there are 2 where  $Temperature = \text{"Hot"}$ , thus  $P(Temperature = \text{"Hot"}|PlayTennis = No) = 2/5$ . In the notation of equation, we may write  $P(X_2 = Hot|C_2) = 2/5$ .
- $P(X_3|C = C_2) = P(\text{"High"}|C = \text{"No"}) = ?$   
Of the 5 cases where  $PlayTennis = \text{"No"}$ , there are 4 where  $Humidity = \text{"High"}$ , thus  $P(Humidity = \text{"High"}|PlayTennis = No) = 4/5$ . In the notation of equation, we may write  $P(X_3 = High|C_2) = 4/5$ .
- $P(X_4|C = C_2) = P(\text{"Strong"}|C = \text{"No"}) = ?$   
Of the 5 cases where  $PlayTennis = \text{"No"}$ , there are 3 where  $Wind = \text{"Strong"}$ , thus  $P(Wind = \text{"Strong"}|PlayTennis = No) = 3/5$ . In the notation of equation, we may write  $P(X_4 = Strong|C_2) = 3/5$ .

Plugging the values, we will have:

$$V_1 = \left(\frac{9}{14}\right)\left(\frac{4}{9}\right)\left(\frac{2}{9}\right)\left(\frac{3}{9}\right)\left(\frac{3}{9}\right) = \frac{648}{91854}$$

$$V_2 = \left(\frac{5}{14}\right)\left(\frac{0}{5}\right)\left(\frac{2}{5}\right)\left(\frac{4}{5}\right)\left(\frac{3}{5}\right) = 0$$

Since  $V_1 > V_2$ , therefore, we classify  $X$  as  $C_1 = \text{"Yes"}$ .

## 14.5 The Naive Bayes Classifier: MapReduce Solution for Symbolic Data

This section will present a MapReduce solution for classifying millions or billions of data for symbolic (non-numeric) data. For building a MapReduce solution, we will assume that we have a proper training data (this training data will enable us to use the Bayes' theorem for computing probabilities and conditional probabilities). The goal of MapReduce solution is to classify data into a set of  $k$  well-defined classes (defined by training data) and identified by  $\{C_1, C_2, \dots, C_k\}$

### 14.5.1 STAGE-1: Building Classifier Using Symbolic Training Data

The goal of building a classifier is to build a function (using training data), which will accept an instance of data  $X = (X_1 = u_1, \dots, X_m = u_m)$  and outputs a class  $c \in \{C_1, C_2, \dots, C_k\}$  (assuming that all of our training data maps into one of these classes). So we need to compute  $P(X_i = u_i | C = C_j)$  for all distinct  $X_i$  and all distinct  $C_j (j = 1, 2, \dots, k)$  in the training data. When the size of training data is small, we can write a non-MapReduce program to build the classifier, but for big training data, we should write a MapReduce job to compute  $P(X_i = u_i | C = C_j)$ .

#### 14.5.1.1 Mapper for Building Classifier Using Symbolic Data

**Listing 14.1:** map() for Building Classifier

```
1 /**
2 * @param key is generated by MapReduce framework, ignored here
3 * @param value as a String has the following format:
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     String reducerKey = "CLASS," + theClass;
15     emit(reducerKey, 1);
16 }

```

---

To understand the mapper, next we apply the map() function to all of our training data, which generates a set of input to be consumed by the reduce() function. The map() function just wants to count the attributes and their associations to the classification classes.

map(Sunny, Hot, High, Weak, **No**) will generate:

```
<Sunny,No>, <1>
<Hot,No>, <1>
<High,No>, <1>
<Weak,No>, <1>
<CLASS,No>, <1>
```

map(Sunny, Hot, High, Strong, **No**)

```
<Sunny,No>, <1>
<Hot,No>, <1>
<High,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

map(Overcast, Hot, High, Weak, **Yes**)

```
<Overcast,Yes>, <1>
<Hot,Yes>, <1>
<High,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Mild, High, Weak, **Yes**)

```
<Rain,Yes>, <1>
<Mild,Yes>, <1>
<High,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Cool, Normal, Weak, Yes)

```
<Rain,Yes>, <1>
<Cool,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Cool, Normal, Strong, No)

```
<Rain,No>, <1>
<Cool,No>, <1>
<Normal,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

map(Overcast, Cool, Normal, Strong, Yes)

```
<Overcast,Yes>, <1>
<Cool,Yes>, <1>
<Normal,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Sunny, Mild, High, Weak, No)

```
<Sunny,No>, <1>
<Mild,No>, <1>
<High,No>, <1>
<Weak,No>, <1>
<CLASS,No>, <1>
```

map(Sunny, Cool, Normal, Weak, Yes)

```
<Sunny,Yes>, <1>
<Cool,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Mild, Normal, Weak, Yes)

```
<Rain,Yes>, <1>
<Mild,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Sunny, Mild, Normal, Strong, Yes)

```
<Sunny,Yes>, <1>
<Mild,Yes>, <1>
<Normal,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Overcast, Mild, High, Strong, Yes)

```
<Overcast,Yes>, <1>
<Mild,Yes>, <1>
<High,Yes>, <1>
<Strong,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Overcast, Hot, Normal, Weak, Yes)

```
<Overcast,Yes>, <1>
<Hot,Yes>, <1>
<Normal,Yes>, <1>
<Weak,Yes>, <1>
<CLASS,Yes>, <1>
```

map(Rain, Mild, High, Strong, No)

```
<Rain,No>, <1>
<Mild,No>, <1>
<High,No>, <1>
<Strong,No>, <1>
<CLASS,No>, <1>
```

#### 14.5.1.2 Reducer for Building Classifier Using Symbolic Data

Below, we present the reducer for building the classifier. Since, we are only aggregating values (the counts), the reduce() can be used as a combiner too. For our example, the reducers will receive the following (key, value) pairs:

Reducer Input	
Key	Value
<CLASS, No>	[<1>, <1>, <1>, <1>, <1>]
<CLASS, Yes>	[<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>]
<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>]

Listing 14.1: reduce() for Building Classifier

```
/**
 * @param key is <Data, Class> or <CLASS, Class>
 * @param values as list of Integer
 */
reduce(key, values) {
    int total = 0;
    for(int value : values) {
```

```

        total += value;
    }

    emit(key, total);
}

```

---

The reduce() function just adds up frequencies (basically tallys the counters). The reducers will generate the following output:

Reducer Output	
<i>Key</i>	<i>Value</i>
<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
<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

We will customize our reducers in such a way to generate two type of outputs, one for emitting data for classes, and one for emitting conditional probabilities. Therefore, our customized reducers will create the following two outputs:

- Class Output

- Conditional Probabilities Output

Class Output is presented below:

CLASS OUTPUT	
<i>Key</i>	<i>Value</i>
<CLASS, No>	5
<CLASS, Yes>	9

Conditional Probabilities Output is presented below:

CONDITIONAL PROBABILITIES OUTPUT	
<i>Key</i>	<i>Value</i>
<Cool, No>	1
<Cool, Yes>	3
<High, No>	4
<High, Yes>	3
<Hot, No>	2
<Hot, Yes>	2
<Mild, No>	2
<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

From the output of reducers (CLASS OUTPUT and CONDITIONAL PROBABILITIES OUTPUT), we will generate a final probability table, which will be used to classify new data:

Probability Table	
<i>Key</i>	<i>Probability</i>
<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
<Strong, Yes>	3/9
<Sunny, No>	3/5
<Sunny, Yes>	2/9
<Weak, No>	2/5
<Weak, Yes>	6/9

If any key is not in our Probability Table, then its probability is zero. Typically, training data should cover all attributes in such a way that probability of zero will not happen, this requirement truly depends on the project and data mining requirements.

#### 14.5.2 STAGE-2: Using Classifier To Classify New Symbolic Data

Now that we have built our classifier (generated Probability Table), we can use it to classify new data. The goal is to classify  $X = \{x_1 = u_1, x_2 = u_2, \dots, x_m = u_m\}$  into one of the classes in  $\{C_1, C_2, \dots, C_k\}$ . We will assume that each record of MapReduce input is an instance of data, which needs to be classified.

For classification, the map() is an identity mapper, which helps us not to

classify the same (duplicate) data more than once. Because classification is an expensive operation, we eliminate duplicate calculations. Now that we have built the "Probability Table" from the given training data, all classifications will be done by reducers.

#### 14.5.2.1 Mapper to Classify New Symbolic Data

##### Listing 14.2: map() for Naive Bayes Classifier

```
1 public class NaiveBayesClassifierMapper ... {  
2  
3     /**  
4      * @param key is generated by MapReduce framework, ignored here  
5      * @param value as a String has the the following format:  
6      *       $X = (X_1, X_2, \dots, X_m) = <Data_1><,><Data_2><,>\dots<,><Data_m>$   
7      */  
8     map(key, value) {  
9         // if desired, we can find out how many  
10        // times each input data is duplicated  
11        emit(value, 1);  
12    }  
13}
```

#### 14.5.2.2 Reducer to Classify New Symbolic Data

##### Listing 14.3: reduce() for Naive Bayes Classifier

```
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 is  $X = (X_1, X_2, \dots, X_m)$   
13     *          = <Data_1><,><Data_2><,>\dots<,><Data_m>  
14     *  
15     * @param values as a list of integers (shows duplicate records)
```

```

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(attributes[i], aClass);
27          }
28          if (selectedClass == null) {
29              // computing values for the first classification
30              selectedClass = aClass;
31              maxPosterior = posterior;
32          }
33          else {
34              if (posterior > maxPosterior) {
35                  selectedClass = aClass;
36                  maxPosterior = posterior;
37              }
38          }
39      }
40      reducerOutputValue = selectedClass + "," + maxPosterior;
41      emit(key, reducerOutputValue);
42  }
43 }

```

---

## 14.6 The Naive Bayes Classifier: MapReduce Solution for Numeric Data

This section will present a MapReduce solution for classifying millions or billions of data for numeric data or so called continuous data. For building a MapReduce solution, we will assume that we have a proper training data (this training data will enable us to use the Bayes' theorem for computing probabilities and conditional probabilities). The goal of MapReduce solution is to classify data into a set of  $k$  well-defined classes (defined by training data) and identified by  $\{C_1, C_2, \dots, C_k\}$

To work with numeric data, we do need to calculate the mean and variance of the training data. Then we will use these values (mean and variance) in the classifier to classify new numeric data. In this numeric example, we have two classes:  $\{C_1, C_2\} = \{\text{male}, \text{female}\}$ . The classifier created from the training set using a Gaussian distribution assumption would be:

Sex Class	mean (height)	variance (height)	mean (weight)	variance (weight)	mean (foot size)	variance (foot size)
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

Since we have 4 males and 4 females in our training data, then we have equiprobable classes so  $P(\text{male}) = P(\text{female}) = 0.5$ .

For numeric data (continuous attribute such as height, weight, and foot size) it is recommended to use a Gaussian Normal Distribution as outlined below. Let  $x$  be continuous attribute (i.e., numeric value). Then first, we segment the data by the class, and then compute the mean ( $\mu$ ) and variance ( $\sigma^2$ ) of  $x$  in each class. Gaussian Normal Distribution for conditional probability can be expressed as:

$$P(x = v|c) = \frac{1}{\sigma_c \sqrt{2\pi}} e^{-\frac{(v-\mu_c)^2}{(2\sigma_c^2)}}$$

where

- $\mu_c$  is the mean of values in  $x$  associated with class  $c$
- $\sigma_c^2$  is the variance of values in  $x$  associated with class  $c$  ( $\sigma_c$  is the standard deviation of values in  $x$  associated with class  $c$ ).

Following [7], let's classify a new data:

Height (feet)	Weight (lbs)	Foot Size (inches)	Sex
6.00	130	8	?

The goal is to classify this data as male/female (in plain English: we want to determine which posterior is greater, male or female). Following Bayes's theorem, we may write:

```
posterior(male) = evidenceMale / evidence
posterior(female) = evidenceFemale / evidence
```

The `evidenceMale`, `evidenceFemale`, and `evidence` (also termed normalizing constant) may be calculated since the sum of the posteriors equals one.

```

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

```

The `evidence` may be ignored since it is a positive constant. We now determine the sex (classification) of the sample.

```

P(male) = 0.5
P(height|male) = 1.5789 (A probability density greater
                        than 1 is OK. It is the area
                        under the bell curve that is
                        equal to 1.)
P(weight|male) = 5.9881e-06
P(footsize|male) = 1.3112e-3
posterior numerator (male) = their product = 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) = their product = 5.3778e-04

```

Since `posterior numerator (female) > posterior numerator (male)`, then we conclude that the sample is female.

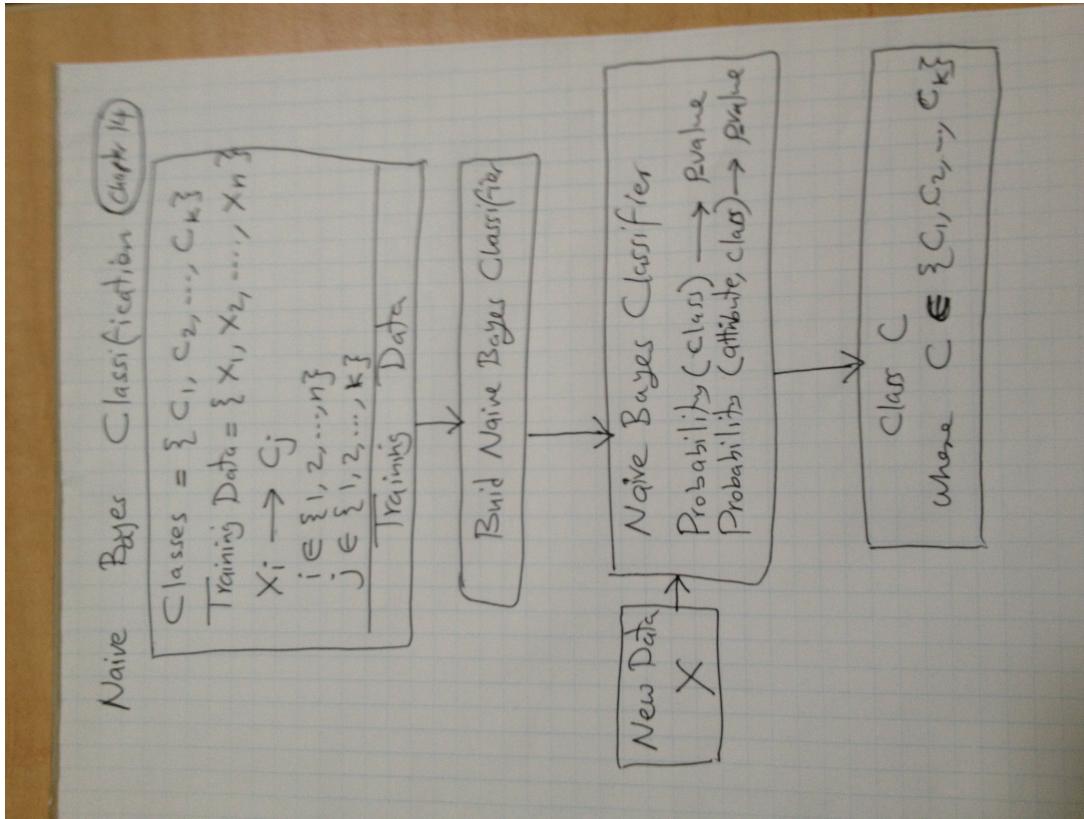


Figure 14.3: Naive Bayes: Training Phase

## 14.7 Naive Bayes Classifier Implementation in Spark

Spark implementation has two parts, which are illustrated in the following figure.

- Build Naive Bayes Classifier using Training Data. This is implemented by the `BuildNaiveBayesClassifier` class. This class reads training data and builds the Naive Bayes classifier. Let  $C = \{C_1, C_2, \dots, C_k\}$  be a set of classifications and let our training data to be
  - Let  $X = \{X_1, X_2, \dots, X_n\}$  be a raw data

- Each  $X_i$  has m attributes and classified as:

$$X_1 = \{X_{11}, X_{12}, \dots, X_{1m}\} \rightarrow c_1 \in C$$

$$X_2 = \{X_{21}, X_{22}, \dots, X_{2m}\} \rightarrow c_2 \in C$$

...

$$X_n = \{X_{n1}, X_{n2}, \dots, X_{nm}\} \rightarrow c_n \in C$$

- The goal is to create the following probabilistic table (pt) functions:

$$pt(C_i) = \text{p-value}$$

$$pt(A_j, C_i) = \text{p-value}$$

where  $A_j$  is an attribute of  $X$  and  $0.00 \leq \text{p-value} \leq 1.00$ .

- Use the Built Classifier to Classify New Data

Once the Naive Bayes classifier is built, we will use the `pt()` functions to classify new data. This is implemented by the `NaiveBayesClassifier` class. This class reads the classifier (expressed as `pt()` functions) and new data and classifies new data using the classifier.

#### 14.7.1 STAGE-1: Building Classifier Using Training Data

This stage is implemented by the `BuildNaiveBayesClassifier` class, which accepts training data (which is already classified), and builds a Naive Bayes Classifier. Naive Bayes classifier is a set of probability tables (pt), discussed in the preceding section. First, we present the `BuildNaiveBayesClassifier` class as a set of high-level steps, then we explain each step in a detailed manner.

### 14.7.1.1 Building Classifier: High-Level Steps

#### Listing 14.4: Building Classifier: High-Level Steps

```
1 // STEP-0: import required classes and interfaces
2 /**
3  * Build Naive Bayes Classifier. The goal is to build the following
4  * data structures (probabilities and conditional probabilities) to
5  * be used in classifying new data:
6  * Let C = {C1, C2, ..., Ck} be set of classifications,
7  * and let each training data element to have m attributes: A = {A1, A2, ..., Am}
8  * then we will build
9  *     ProbabilityTable(c) = p-value where c in C
10 *     ProbabilityTable(c, a) = p-value where c in C and a in A
11 * where      1 >= p-value >=0
12 *
13 * Record example in training data :
14 *     <attribute_1><,,><attribute_2><,,>...<,,><attribute_m><,,><classification>
15 *
16 *
17 * @author Mahmoud Parsian
18 */
19 public class BuildNaiveBayesClassifier implements java.io.Serializable {
20
21     static List<Tuple2<PairOfStrings, DoubleWritable>>
22         toWritableList(Map<Tuple2<String, String>, Double> PT) {...}
23
24     public static void main(String[] args) throws Exception {
25         // STEP-1: handle input parameters
26         // STEP-2: create a Spark context object
27         // STEP-3: read training data
28         // STEP-4: implement map() function to all elements of training data
29         // STEP-5: implement reduce() function to all elements of training data
30         // STEP-6: collect reduced data as Map
31         // STEP-7: build the classifier
32         // STEP-8: save the classifier
33         //     8.1: the PT (probability table) for classification of new entries
34         //     8.2: the Classification List (CLASSIFICATIONS)
35         System.exit(0);
36     }
37 }
```

### 14.7.1.2 STEP-0: import required classes and interfaces

#### Listing 14.5: STEP-0: import required classes and interfaces

```
1 // STEP-0: import required classes and interfaces
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;

```

---

#### 14.7.1.3 Method: toWritableList()

The `toWritableList()` prepares the classifier to be saved in HDFS. To persist data types, they have to implement the Hadoop's `Writable` interface.

##### Listing 14.6: STEP-0: import required classes and interfaces

```

1 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 }

```

---

#### 14.7.1.4 STEP-1: handle input parameters

This section reads 2 input parameters:

- `<training-data-filename>`: this is an HDFS file to represent the training data, to be used for building the BNaive Bayes Classifier
- `<resource-manager-host>`: this is the host name for YARN's resource manager (will be used to create a `SparkContextObject`)

#### **Listing 14.7: STEP-1: handle input parameters**

```
1 // STEP-1: handle input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: BuildNaiveBayesClassifier <training-data-filename> <resource-manager-host>");
4     System.exit(1);
5 }
6 final String trainingDataFilename = args[0];
7 final String resourceManagerHost = args[1];
```

#### **14.7.1.5 STEP-2: create a Spark context object**

This steps creates a `SparkContextObject`, which is a factory class for creating new RDDs. `SparkUtil` class has static methods for creating instances of `SparkContextObject` by using YARN's resource manager or by using Spark master URL.

#### **Listing 14.8: STEP-2: create a Spark context object**

```
1 // STEP-2: create a Spark context object
2 JavaSparkContext ctx = SparkUtil.createJavaSparkContext(resourceManagerHost);
```

#### **14.7.1.6 STEP-3: read training data**

This steps uses an instance of `SparkContextObject`, to read our training data and create `JavaRDD<String>`, where each element is a record of training data set. Each record has the following format:

<attribute\_1><,><attribute\_2><,>...<,><attribute\_m><,><classification>

#### **Listing 14.9: STEP-3: read training data**

```
1 // STEP-3: read training data
2 JavaRDD<String> training = ctx.textFile(trainingDataFilename, 1);
3 training.saveAsTextFile("/output/1");
4 // get the training data size, which will be
5 // used in calculating the conditional probabilities
6 long trainingDataSize = training.count();
```

To debug, we can view the created 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
```

#### 14.7.1.7 STEP-4: implement map() function to all elements of training data

This step maps all elements of training data, so that we can create count of attributes with respect to classifications. Then these counts are used to calculate conditional probabilities.

**Listing 14.10:** STEP-4: implement map() function

```
1 // STEP-4: implement map() function to all elements of training data
2 // PairFlatMapFunction<T, K, V>
3 // T => Iterable<Tuple2<K, V>>
4 // K = <CLASS, classification> or <attribute, classification>
5 JavaPairRDD<Tuple2<String, String>, Integer> pairs =
6     training.flatMapToPair(new PairFlatMapFunction<
7         String, // A1, A2, ..., An, classification
8         Tuple2<String, String>, // K = Tuple2(CLASS, classification) or
9             // Tuple2(attribute, classification)
10        Integer // V = 1
11    >());
12    public Iterable<Tuple2<Tuple2<String, String>, Integer>> call(String rec) {
13        List<Tuple2<String, Integer>> result =
14            new ArrayList<Tuple2<String, Integer>>();
15        String[] tokens = rec.split(",");
16        // tokens[0] = A1
17        // tokens[1] = A2
18        // ...
19        // tokens[m-1] = Am
20        // token[m] = classification
```

```

21     int classificationIndex = tokens.length -1;
22     String theClassification = tokens[classificationIndex];
23     for(int i=0; i < (classificationIndex-1); i++) {
24         Tuple2<String, String> K = new Tuple2<String, String>(tokens[i], theClassification);
25         result.add(new Tuple2<Tuple2<String, String>, Integer>(K, 1));
26     }
27
28     Tuple2<String, String> K = new Tuple2<String, String>("CLASS", theClassification);
29     result.add(new Tuple2<Tuple2<String, String>, Integer>(K, 1));
30     return result;
31 }
32 });
33 pairs.saveAsTextFile("/output/2");

```

---

To debug, we can view the created 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)
```

#### 14.7.1.8 STEP-5: implement reduce() function

This steps reduces the counts for preparation to calculate conditional probabilities, which will be used by the classifier.

##### **Listing 14.11:** STEP-5: implement reduce() function

```
1 // STEP-5: implement reduce() function to all elements of training data
2 JavaPairRDD<Tuple2<String, String>, Integer> counts =
3     pairs.reduceByKey(new Function2<Integer, Integer, Integer>() {
4         public Integer call(Integer i1, Integer i2) {
5             return i1 + i2;
6         }
7 });
8 counts.saveAsTextFile("/output/3");
```

To debug, we can view the created 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)
```

#### 14.7.1.9 STEP-6: collect reduced data as Map

This steps uses a powerful feature of Spark API to collect JavaPairRDD<K, V> as a Map<K, V>. Next we use the generated Map<K, V> to build the classifier.

##### Listing 14.12: STEP-6: collect reduced data as Map

```
1 // STEP-6: collect reduced data as Map
2 // java.util.Map<K,V> collectAsMap()
3 // Return the key-value pairs in this RDD to the master as a Map.
4 Map<Tuple2<String, String>, Integer> countsAsMap = counts.collectAsMap();
```

#### 14.7.1.10 STEP-7: build the classifier data structures

This steps builds the classifier, which consists of

- The Probability Table (PT)
- The Classification List (CLASSIFICATIONS)

##### Listing 14.13: STEP-7: build the classifier data structures

```
1 // STEP-7: build the classifier data structures, which will be used
2 // to classify new data; need to build the following
3 //   1. the Probability Table (PT)
4 //   2. the Classification List (CLASSIFICATIONS)
5 Map<Tuple2<String, String>, Double> PT = new HashMap<Tuple2<String, String>, Double>();
6 List<String> CLASSIFICATIONS = new ArrayList<String>();
7 for (Map.Entry<Tuple2<String, String>, Integer> entry : countsAsMap.entrySet()) {
8     Tuple2<String, String> k = entry.getKey();
9     String classification = k._2;
10    if (k._1.equals("CLASS")) {
11        PT.put(k, (double) entry.getValue() / (double) trainingDataSize);
12        CLASSIFICATIONS.add(k._2);
13    }
14    else {
15        Tuple2<String, String> k2 = new Tuple2<String, String>("CLASS", classification);
16        Integer count = countsAsMap.get(k2);
17        if (count == null) {
18            PT.put(k, 0.0);
19        }
20        else {
21            PT.put(k, (double) entry.getValue() / (double) count.intValue());
22        }
23    }
24 }
25 System.out.println("PT=" + PT);
```

To debug, we can view the created 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
}

```

#### 14.7.1.11 STEP-8: Save the classifier data structures

This step saves the classifier, which consists of

- The Probability Table (PT)
- The Classification List (CLASSIFICATIONS)

To save any data in Hadoop, your persisting class must implement the Hadoop's `org.apache.hadoop.io.Writable`<sup>4</sup> interface. In the code, I used `PairOfStrings` and `DoubleWritable` classes, which both implement the Hadoop's `Writable` interface.

**Listing 14.14:** STEP-8: Save the classifier data structures

```

1 // STEP-8: save the following, which will be used to classify new data
2 //   1. the PT (probability table) for classification of new entries
3 //   2. the Classification List (CLASSIFICATIONS)

```

---

<sup>4</sup>A serializable object which implements a simple, efficient, serialization protocol, based on `DataInput` and `DataOutput`.

```

4
5 // STEP 8.1: save the PT
6 // public <K,V> JavaPairRDD<K,V> parallelizePairs(java.util.List<scala.Tuple2<K,V>> list)
7 // Distribute a local Scala collection to form an RDD.
8 List<Tuple2<PairOfStrings, DoubleWritable>> list = toWritableList(PT);
9 JavaPairRDD<PairOfStrings, DoubleWritable> ptRDD = ctx.parallelizePairs(list);
10 ptRDD.saveAsHadoopFile("/naivebayes/part",
11                         PairOfStrings.class,           // name of path
12                         DoubleWritable.class,         // key class
13                         SequenceFileOutputFormat.class // value class
14                         );                           // output format class
15
16 // STEP 8.2: save the Classification List (CLASSIFICATIONS)
17 // List<Text> writableClassifications = toWritableList(CLASSIFICATIONS);
18 JavaRDD<String> classificationsRDD = ctx.parallelize(CLASSIFICATIONS);
19 classificationsRDD.saveAsTextFile("/naivebayes/classes"); // name of path

```

---

To debug, we can view the content of the saved classifier:

```

# 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

# hadoop fs -cat /naivebayes/classes/part*
Yes
No

```

#### 14.7.1.12 YARN Script to Build Naive Bayes Classifier

The following shell script builds Naive Bayes Classifier using YARN.

**Listing 14.15:** YARN Script to Build Naive Bayes Classifier

```
1 # cat run_build_naive_bayes_classifier.sh
2 #!/bin/bash
3
4 #... set the CLASSPATH accordingly...
5 # mp.jar contains all classes need to run in YARN/Spark environment
6 export MY_JAR=$MP/mp.jar
7 export THE_JARS=$MP/cloud9-1.3.2.jar
8 INPUT=/naivebayes/training_data.txt
9 RESOURCE_MANAGER_HOST=masternode100
10 $SPARK_HOME/bin/spark-submit --class $prog \
11     --master yarn-cluster \
12     --num-executors 12 \
13     --driver-memory 3g \
14     --executor-memory 7g \
15     --executor-cores 12 \
16     --jars $THE_JARS \
17     $MY_JAR $INPUT $RESOURCE_MANAGER_HOST
```

#### 14.7.2 STAGE-2: Using Classifier To Classify New Data

In STAGE-1, we built a Naive Bayes Classifier using the provided training data. The goal of STAGE-2 is to classify new data using the classifier built by STAGE-1. In STAGE-2, we read the classifier from HDFS and the classify according to the built classifier. I will use the following for the Naive Bayes classification of new data:

$$C^{predict} = \operatorname{argmax}_c P(C = c) \prod_{j=1}^m P(X_j = u_j | C = c)$$

The classification is implemented by a single driver class (`NaiveBayesClassifier`) using Spark API. First, a high-level steps are presented, then each step is discussed in detail.

##### 14.7.2.1 STAGE-2: High-Level Solution

**Listing 14.16:** STEP-8: Save the classifier data structures

```

1 // STEP-0: import required classes and interfaces
2 /**
3  * Naive Bayes Classifier, which classifies (using the
4  * classifier built by the BuildNaiveBayesClassifier class)
5  * new data.
6  *
7  * Now for a given  $X = (X_1, X_2, \dots, X_m)$ , we will classify it
8  * by using the following data structures (built by the
9  * BuildNaiveBayesClassifier class):
10 *
11 *     ProbabilityTable( $c$ ) =  $p$ -value where  $c$  in  $C$ 
12 *     ProbabilityTable( $c, a$ ) =  $p$ -value where  $c$  in  $C$  and  $a$  in  $A$ 
13 *
14 * Therefore, given  $X$ , we will classify it as  $C$  where  $C$  in  $\{C_1, C_2, \dots, C_k\}$ 
15 *
16 * @author Mahmoud Parsian
17 */
18 public class NaiveBayesClassifier implements java.io.Serializable {
19     public static void main(String[] args) throws Exception {
20         // STEP-1: handle input parameters
21         // STEP-2: create a Spark context object
22         // STEP-3: read new data to be classified
23         // STEP-4: read the classifier from hadoop
24         // STEP-5: cache the classifier components,
25         // which can be used from any node in the cluster.
26         // STEP-6: classify new data
27         System.exit(0);
28     }
29 }
```

---

#### 14.7.2.2 STEP-0: import required classes and interfaces

**Listing 14.17:** STEP-8: Save the classifier data structures

```

1 // STEP-0: import required classes and interfaces
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.SequenceFileInputFormat;
18 import org.apache.hadoop.io.DoubleWritable;
```

---

#### 14.7.2.3 STEP-1: handle input parameters

**Listing 14.18:** STEP-1: handle input parameters

```
1 // STEP-1: handle input parameters
2 if (args.length != 4) {
3     System.err.println("Usage: NaiveBayesClassifier <input-data-filename> "+
4                         "<NB-PT-path> <NB-CLASS-path> <resource-manager-host>");
5     System.exit(1);
6 }
7 final String inputDataFilename = args[0];           // data to be classified
8 final String nbProbabilityTablePath = args[1];    // part of classifier
9 final String nbClassesPath = args[2];              // part of classifier
10 final String resourceManagerHost = args[3];       // YARN's resource manager host
```

---

#### 14.7.2.4 STEP-2: create a Spark context object

Using YARN's resource manager host, `SparkUtil` class creates an instance of `JavaSparkContext`, which will be used to create new RDDs.

**Listing 14.19:** STEP-2: create a Spark context object

```
1 // STEP-2: create a Spark context object
2 JavaSparkContext ctx = SparkUtil.createJavaSparkContext(resourceManagerHost);
```

---

#### 14.7.2.5 STEP-3: read new data to be classified

The raw data to be classified has the following record format:

<attribute\_1><,><attribute\_2><,>...<,><attribute\_m>

**Listing 14.20:** STEP-3: read new data to be classified

```
1 // STEP-3: read new data to be classified
2 JavaRDD<String> newdata = ctx.textFile(inputDataFilename, 1);
```

---

#### 14.7.2.6 STEP-4: read the classifier from hadoop

This step reads the classifier components and data structures built by the `BuildNaiveBayesClassifier` class (STAGE-1). Once the classifier components are read, then we are ready to classify new data.

**Listing 14.21:** STEP-4: read the classifier from hadoop

```
1 // STEP-4: read the classifier from hadoop
2 // JavaPairRDD<K, V> hadoopFile(String path,
3 //                                Class<F> inputFormatClass,
4 //                                Class<K> keyClass,
5 //                                Class<V> valueClass)
6 // Get an RDD for a Hadoop file with an arbitrary InputFormat
7 // Note: Because Hadoop's RecordReader class re-uses the
8 // same Writable object for each record, directly caching the
9 // returned RDD will create many references to the same object.
10 // If you plan to directly cache Hadoop writable objects, you
11 // should first copy them using a map function.
12 JavaPairRDD<PairOfStrings, DoubleWritable> ptrDD = ctx.hadoopFile(
13     nbProbabilityTablePath, // "naivebayes/pt"
14     SequenceFileInputFormat.class, // input format class
15     PairOfStrings.class, // key class
16     DoubleWritable.class // value class
17 );
18
19 // <K2, V2> JavaPairRDD<K2, V2> mapToPair(PairFunction<T, K2, V2> f)
20 // Return a new RDD by applying a function to all elements of this 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(), pair.getRightElement());
32         Double V2 = new Double(rec._2.get());
33         return new Tuple2<Tuple2<String, String>, Double>(K2, V2);
34     }
35 });
```

#### 14.7.2.7 STEP-5: cache the classifier components

This step caches the classifier components (using Spark's `Broadcast` class) so that they can be accessed and used from any cluster nodes.

### **Listing 14.22: STEP-5: cache the classifier components**

```
1 // STEP-5: cache the classifier components,
2 // which can be used from any node in the cluster.
3 Map<Tuple2<String, String>, Double> classifier = classifierRDD.collectAsMap();
4 final Broadcast<Map<Tuple2<String, String>, Double>> broadcastClassifier =
5     ctx.broadcast(classifier);
6
7 // we need all classifications classes, which was created by the
8 // BuildNaiveBayesClassifier class.
9 JavaRDD<String> classesRDD = ctx.textFile("/naivebayes/classes", 1);
10 List<String> CLASSES = classesRDD.collect();
11 final Broadcast<List<String>> broadcastClasses =
12     ctx.broadcast(CLASSES);
```

#### **14.7.2.8 STEP-6: classify new data**

This step uses the classifier components to classify new data.

### **Listing 14.23: STEP-6: classify new data**

```
1 // STEP-6: classify new data
2 // Now, we have Naive Bayes classifier and new data
3 // Use the classifier to classify new data
4 // PairFlatMapFunction<T, K, V>
5 // T => Iterable<Tuple2<K, V>>
6 // K = <CLASS, classification> or <attribute,classification>
7 JavaPairRDD<String, String> classified = newdata.mapToPair(new PairFunction<
8     String,                                         // T = A1, A2, ..., Am (data to be classified)
9     String,                                         // K = A1, A2, ..., Am (data to be classified)
10    String                                         // V = classification for T
11    >() {
12        public Tuple2<String, String> call(String rec) {
13            // get the classifier from the Spark cache
14            Map<Tuple2<String, String>, Double> CLASSIFIER = broadcastClassifier.value();
15            // get the classes from the Spark cache
16            List<String> CLASSES = broadcastClasses.value();
17
18            // rec = (A1, A2, ..., Am)
19            String[] attributes = rec.split(",");
20            String selectedClass = null;
21            double maxPosterior = 0.0;
22            for (String aClass : CLASSES) {
23                double posterior = CLASSIFIER.get(new Tuple2<String, String>("CLASS", aClass));
24                for (int i=0; i < attributes.length; i++) {
25                    Double probability = CLASSIFIER.get(new Tuple2<String, String>(attributes[i], aClass));
26                    if (probability == null) {
27                        posterior = 0.0;
28                        break;
29                    }
30                    else {
31                        posterior *= probability.doubleValue();
```

```

32         }
33     }
34     if (selectedClass == null) {
35         // computing values for the first classification
36         selectedClass = aClass;
37         maxPosterior = posterior;
38     }
39     else {
40         if (posterior > maxPosterior) {
41             selectedClass = aClass;
42             maxPosterior = posterior;
43         }
44     }
45 }
46
47     return new Tuple2<String, String>(rec, selectedClass);
48 }
49 });
50 classified.saveAsTextFile("/output/classified");

```

---

Input and output for classified data are given below:

- Input: data to be classified:

```

# hadoop fs -cat /naivebayes/new_data_to_be_classified.txt
Rain,Hot,High,Strong
Overcast,Mild,Normal,Weak
Sunny,Mild,Normal,Weak

```

- Output: classified data:

```

# hadoop fs -cat /output/classified/part*
(Rain,Hot,High,Strong,Yes)
(Overcast,Mild,Normal,Weak,Yes)
(Sunny,Mild,Normal,Weak,Yes)

```

#### 14.7.2.9 YARN Script for Spark

##### **Listing 14.24:** STEP-6: classify new data

```

1 # cat run_classify_new_data.sh
2 #... set up required CLASSPATH
3 export MY_JAR=$MP/mp.jar
4 export THE_JARS=$MP/cloud9-1.3.2.jar
5 NEW_DATA=/naivebayes/new_data_to_be_classified.txt

```

```
6 RESOURCE_MANAGER_HOST=masternode100
7 CLASSIFIER_PT=/naivebayes/pt
8 CLASSIFIER_CLASSES=/naivebayes/classes
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   $MY_JAR  $NEW_DATA  $CLASSIFIER_PT  $CLASSIFIER_CLASSES  $RESOURCE_MANAGER_HOST
```

## 14.8 Using Apache Mahout

If you do not want to roll out your own machine learning algorithms, then you may use Apache Mahout<sup>5</sup>. The Apache Mahout project's goal is to build a scalable machine learning library. There are some examples on classification of tweets by Mahout:

- Using the Mahout Naive Bayes Classifier to automatically classify Twitter messages  
<http://chimpler.wordpress.com/2013/03/13/using-the-mahout-naive-bayes-classifier-to-automatically-classify-twitter-messages/>
- Using the Mahout Naive Bayes Classifier to automatically classify Twitter messages (part 2: distribute classification with hadoop)  
<http://chimpler.wordpress.com/2013/06/24/using-the-mahout-naive-bayes-classifier-to-automatically-classify-twitter-messages-part-2-distribute-classification-with-hadoop/>

---

<sup>5</sup><http://mahout.apache.org/>

# Chapter 15

## Sentiment Analysis

### 15.1 Introduction

Sentiment means "a general thought, feeling, emotion, opinion, or sense," and "sentiment analysis" (also known as opinion mining) refers to the use of natural language processing, text analysis and computational linguistics to identify and extract subjective information in source materials (source [Wikipedia](#)).<sup>19</sup> Bo Pang and Lillian Lee<sup>[19]</sup> define sentiment analysis as "sentiment analysis seeks to identify the viewpoint(s) underlying a text span; an example application is classifying a movie review as thumbs up or thumbs down." To perform a sentiment analysis about some event, we do need to teach computers what a sentiment is (how to define a "positive" or "negative" how to define "good" or "bad") since computers don't know about a sentiment unless they learn it from examples that a human has labeled as positive or negative. The is where "machine learning" comes: we do teach computers the meaning of "positive" and "negative" and so on. The first step is to build a model from a training data. After model is built, then we may use the model to analyze new data.

What is a sentiment data? Typically, sentiment data is an unstructured data that represents opinions and emotions contained in sources such as special news bulletins, customer support emails, social media posts (such as Tweets and Facebook blogs) and online product reviews.

To perform a good sentiment analysis, the sentiment analysis engine has to perform some level of speech analysis and word-sense-disambiguation to improve performance. Therefore, a sentiment analysis of a text/document

is more than tokenizing words and checking the words against "positive" and "negative" words. Sometimes, we do need to understand the intensity of words, and factor things like negation and diminishers. For example, consider the following sentence:

```
... movie was not good ...
```

If you just look at individual words without their relationships, you might indicate that the sentiment is "neutral" (since "not" is negative and "good" is positive. But if you look at the whole semantics of the sentence, clearly it is a negative sentiment.

### 15.1.1 Sentiment Examples

The following are some sentiment analysis examples:

- What bloggers are saying about a brand like Toyota when there was a problem with breaking system in some Toyota cars.
- What is the sentiment of people seeing the Iron Man III movie before and after view (this can be done by analyzing tweets before and after)? Did people really like the movie?
- What is the sentiment of customers before and after announcement of a new IPhone?
- What is the sentiment of people before and after a presidential debate? Is sentiment toward a Democratic or Republican party?
- What is the sentiment of people (happy or sad) about the customer service? Here, for sentiment analysis, we do need customer service log data and a dictionary of happy and sad words.

### 15.1.2 Sentiment Scores: Positive or Negative

Given a short sentence (such as a tweet, which is 140 characters long), how do you determine whether it is positive or negative sentiment. To score syn-

tactically (by ignoring contextual semantics), you can tokenize the sentence into words and then check these words against positive and negative words<sup>1</sup>

For example, given the following sentences (**green** color words are positive and **red** color words are negative):

- Sentence-1: "movie was **great** and I **loved** it"
- Sentence-2: "hamburger had a **bad** taste and **terrible**, but I **loved** fries"

What is sentiment score of Sentence-1 and Sentence-2:?

```
setimentScore(Sentence-1) = 1 positive +
                           1 positive
                           = 2 positives (Positive sentiment)
```

```
setimentScore(Sentence-2) = 1 negative +
                           1 negative +
                           1 positive
                           = 1 negative (Negative sentiment)
```

In most of the sentiment analysis situations, checking/examining the individual words against "positive" and "negative" words is not enough and might not give proper expected results. In real sentiment analysis, it is better just not to rely on pure syntax, but to understand the semantics: intensity of words, and to factor things like negation and diminishers. For example, consider the following sentence:

"movie was **not great** and I did **not love** it"

In a real world sentiment analysis, checking pure syntax will not yield great results. If you just look at individual words without their relationships, your pure syntactical algorithm might indicate that the sentiment is

---

<sup>1</sup>Hu and Liu have published an "opinion lexicon" which categorizes approximately 6,800 words as positive or negative and which can be downloaded from <http://www.cs.uic.edu/~liub/FBS/opinion-lexicon-English.rar>. For example positive words are: love, best, cool, great, good, ..., and negative words are: terrible, bad, hate, worst, sucks, ...

”neutral” (since ”not” is negative and {”good”, ”love”} are positive). But if you look at the whole semantics of the sentence, clearly it is a negative sentiment.

Now, after seeing the sentiment scores, we should ask what is **sentiment analysis**? In a nutshell, we can say that **sentiment analysis** is the task of finding whether the statement/opinion expressed in a text sentence is positive, negative, or neutral about a particular topic or trend.

## 15.2 Steps for Sentiment Analysis

What are the steps for performing Sentiment Analysis for a given set of tweets?

## 15.3 A Simple MapReduce for Sentiment Analysis

Let our keyword of interest be ”Obama” and ”Romney” and further assume that it is an election year and you want to know the sentiment of people tweeting about the two presidential candidates Obama and Romney. Let’s assume that you want to find the sentiment trend for each candidate per day. The mapper accepts a tweet, normalizing the text, looking for a keyword of interest (”Obama” or ”Romney”), counting the positive and negative keywords, then subtracting the ratio of negative words from the ratio of positive words. To perform map(), we do need two distinct set of words:

- set of positive words (such as ”good”, ”like”, ”enjoy”, ...)
- set of negative words (such as ”hate”, ”bad”, ”terrible”, ...)

These two sets are passed to mappers by the driver program and in hadoop we can accomplish this by using a distributed cache (`DistributedCache`<sup>2</sup> class).

---

<sup>2</sup>`DistributedCache` (`org.apache.hadoop.filecache.DistributedCache`) is a facility provided by the Hadoop’s MapReduce framework to cache text and archive files needed by applications.

### 15.3.1 map() for Sentiment Analysis

We assume that there exist a nomalizer and tokenizer function for a given tweet:

**Listing 15.1:** The Mapper Function

```
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 }
```

Using the normalizeAndTokenize() method, we can complete the map() method:

**Listing 15.2:** The Mapper Function

```
1 private Set<String> positiveWords = null;  
2 private Set<String> negativeWords = null;  
3 // default candidates values  
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 is the date of a tweet: YYYY-MM-DD:hh:mm:ss  
14 * @param value is a single tweet  
15 */  
16 map(key, value) {  
17     date = key; // date of a tweet as 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 }
```

---

### 15.3.2 reduce() for Sentiment Analysis

**Listing 15.3:** The Reducer Function

```
1 /**
2  * @param key is the Pair(Date, String)
3  *   where
4  *     Date=YYYY-MM-DD
5  *     String= a candidate ("obama" or "romney")
6  * @param values is a List<Double> where Double represents a ratio
7 */
8 reduce(key, values) {
9     double sumOfRatio = 0.0;
10    int n = 0; // number ratios
11    for (Double ratio : values) {
12        n++;
13        sumOfRatio += ratio;
14    }
15    emit (key, sumOfRatio / n);
16 }
```

---

# Chapter 16

## Finding, Counting and Listing all Triangles in Large Graphs

### 16.1 Introduction

Graphs and matrices are used by social network analysts to represent and analyze information about patterns of ties among social actors (users, friends, and "friends of friends"). In network analysis, data are usually modeled as a graph or set of graphs. A graph is a data structure, which has a finite set of nodes, called vertices, together with a finite set of lines, called edges, that join some or all of these nodes. Before we define some metrics using the count of triangles, we do need to define a triplet and a triangle: Let  $T=(a,b,c)$  be a set of three distinct nodes in a graph (identified by  $G$ ), Then  $T$  is a triplet if two of nodes are connected ( $\{(a,b), (a,c)\}$ ) and it is a triangle if all three nodes are connected ( $\{(a,b), (a,c), (b,c)\}$ ).

In analysis of graphs, there are three important metrics

- *Global Clustering Coefficient*
- *Transitivity Ratio* defined as  
$$T(G) = \frac{3 \times (\text{number of triangles on the graph})}{(\text{number of connected triples of vertices})}$$
- *Local Clustering Coefficient*

In order to calculate these three metrics in a large graph, we need to know the number of triangles in a graph. For details on these metrics, see [5], [25], and [32].

Typically a graph might have hundreds of millions of nodes (a node can be a user in a social network) and edges (the relationship between these users) and counting the number of triangles is very time consuming. This chapter provides two MapReduce solutions, which finds, counts, and lists all triangles for a given graph or set of graphs. The solutions are given in Hadoop and Spark:

- Hadoop solution: The MapReduce/Hadoop solution is comprised of 3 steps and each step is a separate MapReduce job.
- Spark solution: The MapReduce/Spark solution is comprised of a single Java driver class, which manipulates several JavaRDD objects. Spark's API is higher level than MapReduce/Hadoop API and this why we can fit all map() and reduce() functions in a single Java class.

## 16.2 Basic Graph Concepts

Let  $V$  be a finite set of nodes (a node can represent a user, a computer, or a tangible/intangible object) and  $E$  be finite set of edges (an edge represents a relationship and connectivity between 2 nodes), then an undirected simple graph can be defined as:  $G = (V, E)$ .

We use the symbol  $n$  for the number of nodes and the symbol  $m$  for the number of edges. The degree  $d(v) = |u \in V : \exists \{v, u\} \in E|$  of node  $v$  is defined to be the number of nodes in  $V$  that are adjacent to  $v$ . A triangle  $\Delta = (V_\Delta, E_\Delta)$  of a graph  $G = (V, E)$  is a three node subgraph with  $V_\Delta = \{u, v, w\} \subset V$  and  $E_\Delta = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subset E$ . We use the symbol  $T(G)$  to denote the number of triangles in graph  $G$ .

For example, the following graph has 2 triangles:

- $\{\{2, 3\}, \{3, 4\}, \{4, 2\}\}$
- $\{\{2, 4\}, \{4, 5\}, \{5, 2\}\}$

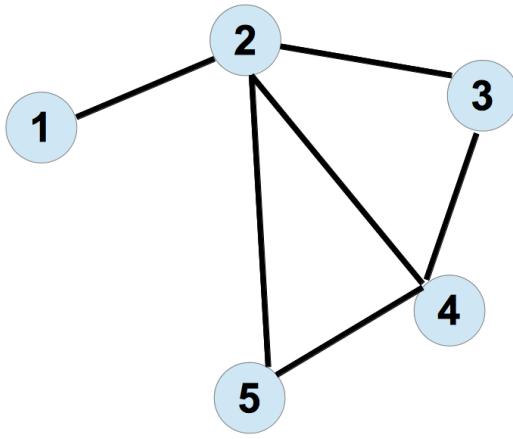


Figure 16.1: Graph Triangles

Let's formalize some of these graph metrics. Given an undirected graph  $G = (V, E)$ , social metrics for graph can be defined. But, first we need some basic definitions:

- Degree of a node  $v$  denoted by  $d(v)$  is the number of nodes in  $V$  that are adjacent to  $v$
- Number of triangles of node  $v$

$$\theta(v) = |(u, w) \in E : (v, u) \in E \text{ and } (v, w) \in E|$$

- Number of triangles of graph  $G$

$$\theta(G) = \frac{1}{3} \sum_{v \in V} \theta(v)$$

- The number of triples of node  $v$  defined as  $t(v)$

$$t(v) = \binom{d(v)}{2}$$

- The number of triples of graph  $G$  defined as  $t(G)$

$$t(G) = \sum_{v \in V} t(v)$$

Using these basic definition, now we can define clustering coefficient :

- Clustering coefficient for a node  $v$  defined as  $c(v)$  (where  $d(v) \geq 2$ ):

$$c(v) = \theta(v)/t(v)$$

- The clustering coefficient of a graph  $G$  (denoted as  $C(G)$ ) is the average over the clustering coefficients of its nodes:

$$C(G) = \frac{1}{|W|} \sum_{w \in W} c(w)$$

where  $W$  is the set of nodes  $w$  with  $d(w) \geq 2$ .

## 16.3 Importance of Counting Triangles

Why is counting the number of triangles in a graph important? Identifying and counting triangles help us to measure two important metrics (clustering coefficient and transitive ratio) about large graphs. Also, counting triangles help us in finding patterns (such as intrusion detection, spamming, and community detection) in social graphs. Also, triangle detections in biological networks help us to find protein-protein interaction networks.

It is shown that triangles and clustering coefficients play important roles in the analysis of complex networks using graph data structures. The existence of triangles and the resulting high clustering coefficient reveals important characteristics of social, biological, web and different other networks.

Given that counting the number of triangles in a graph is very time consuming (for details see [24]), distributing graph data and its computation can be done by MapReduce solutions. The goal of this chapter is to provide a MapReduce solution for finding, listing, and counting Triangles in Large Graphs. A sequential (non-MapReduce) algorithm [24] is provided for finding and counting all triangles in large graphs.

## 16.4 MapReduce Solution

We present a MapReduce solution in three steps (each step is a separate MapReduce job):

- STEP-1: Generate paths through  $u$  of length 2 and copy of all edges from  $u$  as keys, no associated data. From these paths we will generate possible triangles. This step can be solved by a map() and reduce() functions as:

$$\begin{aligned} \text{mapper}_1 : (u, v) &\rightarrow (u, v), (v, u) \\ \text{reducer}_1 : \{(u, v_1), (u, v_2), \dots, (u, v_d)\} &\rightarrow \{[(v_1, v_2), (u)], \\ &\quad [(v_1, v_3), (u)], \\ &\quad \dots, \\ &\quad [(v_{d-1}, v_d), (u)]\}, \\ &\quad \{[(u, v_1), (-)], \dots, [(u, v_d), (-)]\} \end{aligned}$$

- STEP-2: Identify triangles as  $\{\{u, v\}, \{v, w\}, \{w, u\}\}$ . This step will generate duplicate triangles and can be implemented by the following

map() and reduce() functions:

$$\begin{array}{ll}
 \text{mapper}_2 : [(u, v), (w)] & \rightarrow [(u, v), (w)] \\
 \text{reducer}_2 : \{[(u, v)(-)], [(u, v)(w_1)], \dots, [(u, v)(w_r)]\} & \rightarrow \{[(u, v, w_1)], \dots, [(u, v, w_r)]\} \\
 \text{reducer}_2 : \{[(u, v)(w_1)], \dots, [(u, v)(w_r)]\} & \rightarrow NoOutput
 \end{array}$$

- STEP-3: This step removes duplicate triangles:  $(a, b, c)$  is the same as  $(a, c, b)$ . This step can be solved by a map() and reduce() functions as:

$$\begin{array}{ll}
 \text{mapper}_3 : (k_3, v_3) & \rightarrow (\text{sort}(k_3), v_3) \\
 \text{reducer}_3 : \{(k_3, v_{31}), (k_3, v_{32}), (k_3, v_{33}), \dots\} & \rightarrow \{(k_3, \text{null})\}
 \end{array}$$

## 16.5 MapReduce in Action

The input to the first step (i.e.,  $\text{mapper}_1$  of STEP-1) will be the edges in a given undirected graph. Each line of input will be in the  $(u, v)$  format such that  $u < v$  ( $\{u, v\}$  is an edge of a graph). For our graph illustrated above, we will have the following records:

Input to $\text{mapper}_1$ of STEP-1	
key (begin-node)	value (end-node)
1	2
2	3
2	4
2	5
3	4
4	5

The  $\text{mapper}_1$  of STEP-1 generates the following output (which will be an input to  $\text{reducer}_1$  of STEP-1). Note that edges must be reciprocal, that is every  $\{\text{startNode}, \text{endNode}\}$  edge must have a corresponding  $\{\text{endNode}, \text{startNode}\}$ .

Output of $mapper_1$ of STEP-1	
key (start node)	value (end node)
1	2
2	1
2	3
3	2
2	4
4	2
2	5
5	2
3	4
4	3
4	5
5	4

After shuffle and sort phase, the following data will be ready as an input to  $reducer_1$  of STEP-1:

Input of $reducer_1$ of STEP-1	
key	value (as list of nodes)
1	[2]
2	[1, 3, 4, 5]
3	[2, 4]
4	[2, 3, 5]
5	[2, 4]

The  $reducer_1$  of STEP-1 is defined as:

$$\begin{aligned} \text{reducer}_1 : \{(u), (v_1, v_2, \dots, v_d)\} &\rightarrow \{[(v_1, v_2), (u)], \\ &[(v_1, v_3), (u)], \\ &\dots, \\ &[(v_{d-1}, v_d), (u)]\}, \\ &\{[(u, v_1), (-)], \dots, [(u, v_d), (-)]\} \end{aligned}$$

The  $reducer_1$  of STEP-1 will behave as:

<i>reducer</i> <sub>1</sub> Input/Output	
Input	Output
[1] , [2]	[(1, 2), (-)] [(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)]
[2] , [1, 3, 4, 5]	[(3, 2), (-)] [(3, 4), (-)] [(2, 4), (3)]
[3] , [2, 4]	[(4, 2), (-)] [(4, 3), (-)] [(4, 5), (-)] [(2, 3), (4)] [(2, 5), (4)] [(3, 5), (4)]
[4] , [2, 3, 5]	[(5, 2), (-)] [(5, 4), (-)] [(2, 4), (5)]
[5] , [2, 4]	

The *mapper*<sub>2</sub> of STEP-2 is an identity mapper; it just passes keys and values without any further processing. This enables us to generate proper keys and values for *reducer*<sub>2</sub> of STEP-2. The *reducer*<sub>2</sub> of STEP-2 will have the following input/output:

reducer <sub>2</sub> Input/Output	
Input	Output (as a triangle(s))
(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) [-]	

As you can observe, we have listed each triangle 3 times; this is to highlight the definition of a triangle. For example, a single triangle with 3 nodes of  $u, v, w$  can be equivalently expressed as:

$$\begin{aligned} &\{\{u, v\}, \{v, w\}, \{w, u\}\} \\ &\{\{v, w\}, \{w, u\}, \{u, v\}\} \\ &\{\{w, u\}, \{u, v\}, \{v, w\}\} \end{aligned}$$

## 16.6 STEP-3: Remove Duplicate Triangles

This step removes duplicate triangles and the result will be unique triangles. The input for this step will be in the form of  $(a, b, c)$ , where  $a, b$ , and  $c$  represent a vertex of a triangle.

### 16.6.1 STEP-3: Mapper

The mapper will accept a triangle in the form of  $(a, b, c)$  and will emit sorted vertexes. For example,

```
(1, 2, 3) will emit (1, 2, 3)
(1, 3, 2) will emit (1, 2, 3)
(2, 1, 3) will emit (1, 2, 3)
(2, 3, 1) will emit (1, 2, 3)
(3, 1, 2) will emit (1, 2, 3)
(3, 2, 1) will emit (1, 2, 3)
```

Therefore, the mapper is defined as:

```
// key is not used
// value = (a, b, c)
map(key, value) {
    triangle = sort(value);
    // triangle = (x, y, z)
    // x < y < z
    // x, y, z in {a, b, c}
    emit(triangle, null)
}
```

### 16.6.2 STEP-3: Reduer

Since output of mappers will be grouped by key of mappers. The reducer will just emit the received key. This will generate unique triangles. We really do not care about the values of this reducer at all (since a key represents a unique triangle).

```
// key is a triangle (a, b, c)
//   where a < b < c
// values = list of nulls
reduce(key, values) {
    emit(key, null)
}
```

## 16.7 Hadoop Implementation

Hadoop implementation classes are given below.

Implementation Classes in Hadoop		
Phase	Class Name	Class Description
Job Submitter	TriangleCounterDriver	Dirver to submit job for 3 phases
Phase-1	GeneratePathsMapper	Generate possible triangle paths
	GeneratePathsReducer	Identifies possible triangles
Phase-2	TriadsMapper	Identity mapper
	TriadsReducer	Identifies triangles with duplicates
Phase-3	UniqueTriadsMapper	Identity mapper
	UniqueTriadsReducer	Generates unique triangles

### 16.7.1 Sample Run

#### 16.7.1.1 The Script

```
# cat run.sh
#!/bin/bash
export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true
export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.: /usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin

CLASSPATH=.: $HADOOP_HOME/conf
HADOOP_JAR_FILES='find $HADOOP_HOME -name 'hadoop*.jar''
for hadoopjarfile in $HADOOP_JAR_FILES ; do
CLASSPATH=$CLASSPATH:$hadoopjarfile
done

export TESTS=/Users/mahmoud/zmp/map_reduce_book/hadooptests
LIB_DIR=$TESTS/lib
JAR_FILES='find $LIB_DIR -name '*.jar''
for jarfile in $JAR_FILES ; do
```

```

CLASSPATH=$CLASSPATH:$jarfile
done
echo "CLASSPATH=$CLASSPATH"

export JAR=$TESTS/find_count_list_triangles/triangles.jar
export CLASSPATH=$CLASSPATH:$JAR
export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class
$HADOOP_HOME/bin/hadoop fs -rm /lib/triangles.jar
$HADOOP_HOME/bin/hadoop fs -put $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 $JAR TriangleCounterDriver $INPUT $OUTPUT

```

### 16.7.1.2 HDFS Input

```

# hadoop fs -ls /triangles/input/
Found 1 items
-rw-r--r-- 1 mahmoud staff 24 2014-05-25 11:06 /triangles/input/sample_g
# hadoop fs -cat /triangles/input/sample_graph.txt
1 2
2 3
2 4
2 5
3 4
4 5

```

### 16.7.1.3 Log of Sample Run

```

# ./run.sh
...

```

```

Deleted hdfs://localhost:9000/lib/triangles.jar
Deleted hdfs://localhost:9000/triangles/output
Deleted hdfs://localhost:9000/triangles/tmp1
Deleted hdfs://localhost:9000/triangles/tmp2
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 13:40:23 INFO mapred.JobClient: map 100% reduce 0%
14/05/25 13:40:32 INFO mapred.JobClient: map 100% reduce 20%
...
14/05/25 13:40:59 INFO mapred.JobClient: map 100% reduce 80%
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 output materialized bytes=276
14/05/25 13:41:09 INFO mapred.JobClient: Map input records=6
14/05/25 13:41:09 INFO mapred.JobClient: Reduce shuffle bytes=276
14/05/25 13:41:09 INFO mapred.JobClient: Spilled Records=24
14/05/25 13:41:09 INFO mapred.JobClient: Map output bytes=192
14/05/25 13:41:09 INFO mapred.JobClient: Total committed heap usage (bytes)=10
14/05/25 13:41:09 INFO mapred.JobClient: Combine input records=0
14/05/25 13:41:09 INFO mapred.JobClient: SPLIT_RAW_BYTES=119
14/05/25 13:41:09 INFO mapred.JobClient: Reduce input records=12
14/05/25 13:41:09 INFO mapred.JobClient: Reduce input groups=5
14/05/25 13:41:09 INFO mapred.JobClient: Combine output records=0
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 13:41:17 INFO mapred.JobClient: map 50% reduce 0%
...
14/05/25 13:42:05 INFO mapred.JobClient: map 100% reduce 93%
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 output materialized bytes=922
14/05/25 13:42:06 INFO mapred.JobClient: Map input records=23
14/05/25 13:42:06 INFO mapred.JobClient: Reduce shuffle bytes=922
14/05/25 13:42:06 INFO mapred.JobClient: Spilled Records=46
14/05/25 13:42:06 INFO mapred.JobClient: Map output bytes=276
14/05/25 13:42:06 INFO mapred.JobClient: Total committed heap usage (bytes)=26
14/05/25 13:42:06 INFO mapred.JobClient: Combine input records=0
14/05/25 13:42:06 INFO mapred.JobClient: SPLIT_RAW_BYT
ES=1140
14/05/25 13:42:06 INFO mapred.JobClient: Reduce input records=23
14/05/25 13:42:06 INFO mapred.JobClient: Reduce input groups=16
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 13:42:13 INFO mapred.JobClient: map 50% reduce 0%
...
14/05/25 13:42:59 INFO mapred.JobClient: map 100% reduce 86%
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: Counters: 26
...
14/05/25 13:43:01 INFO mapred.JobClient: Map-Reduce Framework
14/05/25 13:43:01 INFO mapred.JobClient: Map output materialized bytes=654
14/05/25 13:43:01 INFO mapred.JobClient: Map input records=6
14/05/25 13:43:01 INFO mapred.JobClient: Reduce shuffle bytes=654
14/05/25 13:43:01 INFO mapred.JobClient: Spilled Records=12
14/05/25 13:43:01 INFO mapred.JobClient: Map output bytes=42
14/05/25 13:43:01 INFO mapred.JobClient: Total committed heap usage (bytes)=26
14/05/25 13:43:01 INFO mapred.JobClient: Combine input records=0
14/05/25 13:43:01 INFO mapred.JobClient: SPLIT_RAW_BYT
ES=1140
14/05/25 13:43:01 INFO mapred.JobClient: Reduce input records=6
14/05/25 13:43:01 INFO mapred.JobClient: Reduce input groups=2
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

```

#### 16.7.1.4 Inspecting Outputs

```
# 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
```

## 16.8 Spark Implementation

Hadoop implementation for finding all unique triangles comprised of 3 distinct MapReduce jobs (each job had a map() and reduce() functions). Since Spark offers a high-level API for mappers and reducers, we present the Spark solution in a single Java class, which has 9 steps. First, we present all these 9 steps and then we will dissect each step.

For Spark implementation, we closely follow the 3-phase MapReduce algorithms presented for the Hadoop implementation.

**Listing 16.1:** Spark's High Level Solution

```
1 // STEP-0: import required classes and interfaces
2 public class CountTriangles {
3
4     public static void main(String[] args) throws Exception {
5         // STEP-1: Read and validate input parameters
6         // STEP-2: create a JavaSparkContext object
7         // STEP-3: read an HDFS input text file representing a graph,
8         //          records are represented as JavaRDD<String>
9         // STEP-4: create a new JavaPairRDD for all edges, which includes
10        //          (source, destination) and (destination, source)
11        // STEP-5: create a new JavaPairRDD, which will generate triads
12        // STEP-6: create a new JavaPairRDD, representing all possible triads
13        // STEP-7: create a new JavaPairRDD, which will generate triangles
14        // STEP-8: create a new JavaPairRDD, representing all triangles
15        // Step-9: eliminate duplicate triangles and create unique triangles
16        System.exit(0);
17    }
18 }
```

Next, we present each step in detail. and finally we will have a sample run of the Spark solution.

### 16.8.1 STEP-0: Import Required Classes and Interfaces

Spark's binary distribution provides required JAR files. We import required classes and interfaces from these JAR files. Most of the classes and interfaces are defined in the `org.apache.spark.api.java` package.

#### **Listing 16.2: STEP-0: import required classes and interfaces**

```
1 // STEP-0: import required classes and interfaces
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;
```

#### **16.8.2 STEP-1: Read Input Parameters**

This step reads the "spark master node" (to connect to Spark cluster) and "HDFS input file", which represents our input graph as a set of edges in the form of (source, destination) and (destination, source).

#### **Listing 16.3: STEP-1: Read and validate input parameters**

```
1 // STEP-1: Read and validate input parameters
2 if (args.length < 2) {
3     System.err.println("Usage: CountTriangles <spark-master> <hdfs-file>");
4     System.exit(1);
5 }
6 String sparkMaster = args[0];
7 String hdfsFile = args[1];
```

#### **16.8.3 STEP-2: Create a Spark Context Object**

Using the spark master node name, we create a JavaSparkContext object, that returns JavaRDDs and works with Java collections. For this step, you need to define two environment variables: SPARK\_HOME and SPARK\_EXAMPLES\_JAR. Once we create a JavaSparkContext object, we next will be able to create and manipulate our data with RDDs. For example, if your spark cluster is comprised of 4 servers { myserver100, myserver200, myserver300, myserver400 } and spark master is represented by myserver100, then your args[0] might have the value: `spark://myserver100:7077`. Note that port number 7077 is configurable parameter in the spark cluster enviroment.

#### **Listing 16.4:** STEP-2: Create a Spark Context Object

```
1 // STEP-2: create a JavaSparkContext object
2 // This is the object for creating the first RDD
3 JavaSparkContext ctx = new JavaSparkContext(
4     sparkMaster, // spark master
5     "MyJavaWordCount",
6     System.getenv("SPARK_HOME"),
7     System.getenv("SPARK_EXAMPLES_JAR"));
```

#### **16.8.4 STEP-3: Read Graph via HDFS Input**

In this step, our input graph (represented by `args[1]`) is read by `JavaSparkContext` object and the first `JavaRDD<String>` is created. For example, we used HDFS file `/triangles/sample_graph.txt` as an input file to represent our input graph. Spark's main abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs. Spark's RDDs have actions (such as `reduce()` and `collect()`), which return values, and transformations (such as `map()`, `flatMap()`, `flatMapToPair()`, `union()`, and `filter()`), which return pointers to new RDDs.

#### **Listing 16.5:** STEP-3: read an hdfs input text file representing a graph

```
1 // STEP-3: read an hdfs input text file representing a graph,
2 //           records are represented as JavaRDD<String>
3 // args[1] = HDFS text file: /triangles/sample_graph.txt
4 JavaRDD<String> lines = ctx.textFile(hdfsFile, 1);
```

#### **16.8.5 STEP-4: Create All Graph Edges**

This step creates a new `JavaPairRDD<Long,Long>` from a set of edges represented as `lines` (as `JavaRDD<String>`). So we convert each edge (as "nodeA nodeB") into two pairs as

```
(nodeA, nodeB)
(nodeB, nodeA)
```

Note that edges must be reciprocal, that is every source, destination edge must have a corresponding destination, source. To accomplish our mapping we use `PairFlatMapFunction<T, K, V>`, where T is input and K and V are outputs (creating a `(key=K, value=V)` pair). Using Spark's `JavaRDD.flatMapToPair()` method, we generate all required (key,value) pairs (represented as `JavaPairRDD<Long, Long>`).

#### Listing 16.6: STEP-4: create a new JavaPairRDD for all edges

```

1  // STEP-4: create a new JavaPairRDD for all edges, which
2  // includes (source, destination) and (destination, source)
3  // PairFlatMapFunction<T, K, V>
4  // T => Iterable<Tuple2<K, V>>
5  JavaPairRDD<Long,Long> edges = //           T      K      V
6      lines.flatMap(new PairFlatMapFunction<String, Long, Long>() {
7          public Iterable<Tuple2<Long,Long>> call(String s) {
8              String[] nodes = s.split(" ");
9              long start = Long.parseLong(nodes[0]);
10             long end = Long.parseLong(nodes[1]);
11             // Note that edges must be reciprocal, that is every
12             // {source, destination} edge must have a corresponding {destination, source}.
13             return Arrays.asList(new Tuple2<Long, Long>(start, end),
14                                 new Tuple2<Long, Long>(end, start));
15         }
16     });

```

#### 16.8.6 STEP-5: Create RDD To Generate Triads

Now that we have all edges, we try to form data structures, which eventually will create triangles. For every node, we find all corresponding destinations, which might form possible triads. To accomplish this task, we use `JavaPairRDD.groupByKey()`. The new RDD will be denoted by `JavaPairRDD<Long, Iterable<Long>>`. Note that values are returned as `Iterable<Long>` instead of `List<Long>`. This is another high-level abstraction by Spark to keep implementation hidden from users (for possible optimizations by Spark platform – Spark can implement `Iterable<Long>` by `ArrayList`, `LinkedList`, or other proper data structures).

```
// STEP-5: create a new JavaPairRDD, which will generate triads
JavaPairRDD<Long, Iterable<Long>> triads = edges.groupByKey();
```

For debugging STEP-5, we collect all `triads` objects and display them:

```

// STEP-5.1: 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);
}

```

### 16.8.7 STEP-6: Create All Possible Triads

This step creates all possible triads (candidates for further checking that they form a triangle). To implement this step, we use `JavaPairRDD.flatMapToPair()` function to create proper (key,value) pairs, where key is an edge (represented by a `Tuple2<node1,node2>` and value is a node (as a Long data type). One thing you should note that all RDDs are immutable, this means that they are READ-ONLY and can not be modified, altered, sorted in any way. If you want to sort your RDD's then clone them and then do your transformation or action.

**Listing 16.7:** STEP-6: create a new JavaPairRDD, which will generate possible triads

```

1 // STEP-6: create a new JavaPairRDD, which will generate possible triads
2 JavaPairRDD<Tuple2<Long,Long>, Long> possibleTriads =
3     triads.flatMapToPair(
4         new PairFlatMapFunction<Tuple2<Long, Iterable<Long>>, // input
5             Tuple2<Long,Long>, // key (output)
6             Long // value (output)
7         ) {
8     public Iterable<Tuple2<Tuple2<Long,Long>, Long>>
9         call(Tuple2<Long, Iterable<Long>> s) {
10
11     // s._1 = Long (as a key)
12     // s._2 = Iterable<Long> (as a values)
13     Iterable<Long> values = s._2;
14     // we assume that no node has an ID of zero
15     List<Tuple2<Tuple2<Long,Long>, Long>> result =
16         new ArrayList<Tuple2<Tuple2<Long,Long>, Long>>();
17
18     // Generate possible triads.
19     for (Long value : values) {
20         Tuple2<Long,Long> k2 = new Tuple2<Long,Long>(s._1, value);
21         Tuple2<Tuple2<Long,Long>, Long> k2v2 =
22             new Tuple2<Tuple2<Long,Long>, Long>(k2, 01);
23         result.add(k2v2);

```

```

24     }
25
26     // RDD's values are immutable, so we have to copy the values
27     // copy values to valuesCopy
28     List<Long> valuesCopy = new ArrayList<Long>();
29     for (Long item : values) {
30         valuesCopy.add(item);
31     }
32     Collections.sort(valuesCopy);
33
34     // Generate possible triads.
35     for (int i=0; i< valuesCopy.size() -1; ++i) {
36         for (int j=i+1; j< valuesCopy.size(); ++j) {
37             Tuple2<Long,Long> k2 =
38                 new Tuple2<Long,Long>(valuesCopy.get(i), valuesCopy.get(j));
39             Tuple2<Tuple2<Long,Long>, Long> k2v2 =
40                 new Tuple2<Tuple2<Long,Long>, Long>(k2, s._1);
41             result.add(k2v2);
42         }
43     }
44
45     return result;
46 }
47 );

```

---

For debugging STEP-6, we collect all `possibleTriads` objects and display them:

```

// STEP-6.1: 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);
}

```

### 16.8.8 STEP-7: Create RDD To Generate Triangles

This step create an RDD, which will be used to generate the actual triangles. The resulting JavaPairRDD will have an edge (represented as `Tuple2<Long,Long>`) and a set of nodes (represented as `Iterable<Long>`) for possible triangle formation.

**Listing 16.8:** STEP-7: create a new JavaPairRDD, which will generate triangles

```
1 // STEP-7: create a new JavaPairRDD, which will generate triangles
2 JavaPairRDD<Tuple2<Long,Long>, Iterable<Long>> triadsGrouped =
3     possibleTriads.groupByKey();
4
5 // STEP-7.1: debug3
6 List<Tuple2<Tuple2<Long,Long>, Iterable<Long>>> debug3 = triadsGrouped.collect();
7 for (Tuple2<Tuple2<Long,Long>, Iterable<Long>> t2 : debug3) {
8     System.out.println("debug3 t2._1="+t2._1);
9     System.out.println("debug3 t2._2="+t2._2);
10 }
```

### 16.8.9 STEP-8: Create All Triangles

This step generates all possible triangles, but it will include duplicates. We use a FlatMapFunction, which works as:

```
FlatMapFunction<T, R>
T => Iterable<R>
```

where T is an input and Iterable<R> is an output. In our case:

```
T = Tuple2<Tuple2<Long,Long>, Iterable<Long>>
R = Tuple3<Long,Long,Long>

T = Tuple2 (Tuple2(nodeA,nodeB), <node1,node2, ...>)
R = Tuple3(nodeA,nodeB,nodeC) as a triangle
    where nodeC in {node1, node2, ...}
```

This step is accomplished by calling JavaPairRDD.flatMap() function.

**Listing 16.9:** STEP-8: create a new JavaPairRDD, which will generate all triangles

```
1 // STEP-8: create a new JavaPairRDD, which will generate all triangles
2 JavaRDD<Tuple3<Long,Long,Long>> trianglesWithDuplicates =
3     triadsGrouped.flatMap(new FlatMapFunction<
```

```

4         Tuple2<Tuple2<Long,Long>, Iterable<Long>>, // input
5         Tuple3<Long,Long,Long> // output
6         >() {
7     public Iterable<Tuple3<Long,Long,Long>>
8         call(Tuple2<Tuple2<Long,Long>, Iterable<Long>> s) {
9
10        // s._1 = Tuple2<Long,Long> (as a key) = "<nodeA><,><nodeB>"
11        // s._2 = Iterable<Long> (as a values) = {0, n1, n2, n3, ...} or {n1, n2, n3, ...}
12        // note that 0 is a fake node, which does not exist
13        Tuple2<Long,Long> key = s._1;
14        Iterable<Long> values = s._2;
15        // we assume that no node has an ID of zero
16
17        List<Long> list = new ArrayList<Long>();
18        boolean haveSeenSpecialNodeZero = false;
19        for (Long node : values) {
20            if (node == 0) {
21                haveSeenSpecialNodeZero = true;
22            }
23            else {
24                list.add(node);
25            }
26        }
27
28        List<Tuple3<Long,Long,Long>> result =
29            new ArrayList<Tuple3<Long,Long,Long>>();
30        if (haveSeenSpecialNodeZero) {
31            if (list.isEmpty()) {
32                // no triangles found
33                return result;
34            }
35            // emit triangles
36            for (long node : list) {
37                long[] aTriangle = {key._1, key._2, node};
38                Arrays.sort(aTriangle);
39                Tuple3<Long,Long,Long> t3 =
40                    new Tuple3<Long,Long,Long>(aTriangle[0], aTriangle[1], aTriangle[2]);
41                result.add(t3);
42            }
43        }
44        else {
45            // no triangles found
46            return result;
47        }
48
49        return result;
50    }
51 });

```

---

To debug this step, we use `JavaRDD.collect()` method.

```
// STEP-8.1: debug4: print all triangles (includes duplicates)
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);
}

```

### 16.8.10 Step-9: Create Unique Triangles

Spark provides a very powerful API to find distinct elements for a given RDD. `JavaRDD<Tuple3<Long,Long,Long>>.distinct()` will create a new `JavaRDD<Tuple3<Long,Long,Long>>`, where are elements are distinct.

**Listing 16.10:** Step-9: eliminate duplicate triangles and create unique triangles

```

1  // Step-9: eliminate duplicate triangles and create unique triangles
2  JavaRDD<Tuple3<Long,Long,Long>> uniqueTriangles = trianglesWithDuplicates.distinct();
3
4  // Step-9.1: print unique triangles
5  System.out.println("== Unique Triangles ==");
6  List<Tuple3<Long,Long,Long>> output = uniqueTriangles.collect();
7  for (Tuple3<Long,Long,Long> t3 : output) {
8      //System.out.println(t3._1 + "," + t3._2+ "," + t3._3);
9      System.out.println("t3="+t3);
10 }

```

## 16.9 Spark Sample Run

Here we show a complete run of Spark's program for counting triangles.

### 16.9.1 Input

We use HDFS as an input medium:

```

# 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
```

### 16.9.2 Script

We use a convenient shell script to run our Spark program:

```
# cat run_count_triangles.sh  
#!/bin/bash  
source /home/hadoop/conf/env_2.3.0.sh  
export SPARK_HOME=/home/hadoop/spark-1.0.0  
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh  
source $SPARK_HOME/conf/spark-env.sh  
  
# system jars:  
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop  
#  
jars='find $SPARK_HOME -name *.jar'  
for j in $jars ; do  
    CLASSPATH=$CLASSPATH:$j  
done  
  
# app jar:  
export CLASSPATH=/home/hadoop/spark_mahmoud_examples/mp.jar:$CLASSPATH  
export SPARK_CLASSPATH=$CLASSPATH  
export SPARK_MASTER=spark://myserver100:7077  
export INPUT=triangles/sample_graph.txt  
export OPTIONS="-Dsun.lang.ClassLoader.allowArraySyntax=true -Dspark.master=$SPARK_MASTER"  
$JAVA_HOME/bin/java -cp $CLASSPATH $OPTIONS CountTriangles $SPARK_MASTER $INPUT
```

### 16.9.3 Running Script

The output log is trimmed to fit the page.

```
# ./run_count_triangles.sh  
...  
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]  
14/06/03 15:15:49 WARN spark.SparkConf: null jar passed to SparkContext constructor  
.  
14/06/03 15:15:59 INFO scheduler.DAGScheduler: Completed ResultTask(0, 0)  
14/06/03 15:15:59 INFO scheduler.TaskSetManager: Finished TID 1 in 112 ms on myserver100 (progress: 1/1)  
14/06/03 15:15:59 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 0.0, whose tasks have all completed, from pool  
14/06/03 15:15:59 INFO scheduler.DAGScheduler: Stage 0 (collect at CountTriangles.java:45) finished in 0.145 s  
14/06/03 15:15:59 INFO spark.SparkContext: Job finished: collect at CountTriangles.java:45, took 4.383103788 s  
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]
14/06/03 15:15:59 INFO spark.SparkContext: Starting job: collect at CountTriangles.java:92
...
14/06/03 15:15:59 INFO scheduler.DAGScheduler: Stage 2 (collect at CountTriangles.java:92) finished in 0.040 s
14/06/03 15:15:59 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 2.0, whose tasks have all completed, from pool
14/06/03 15:15:59 INFO spark.SparkContext: Job finished: collect at CountTriangles.java:92, took 0.05745656 s
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
14/06/03 15:15:59 INFO spark.SparkContext: Starting job: collect at CountTriangles.java:100
14/06/03 15:15:59 INFO scheduler.DAGScheduler: Registering RDD 7 (flatMapToPair at CountTriangles.java:51)
...
14/06/03 15:16:02 INFO scheduler.DAGScheduler: Stage 4 (collect at CountTriangles.java:100) finished in 2.552 s
14/06/03 15:16:02 INFO spark.SparkContext: Job finished: collect at CountTriangles.java:100, took 3.305817554 s
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 ===
14/06/03 15:16:02 INFO spark.SparkContext: Starting job: collect at CountTriangles.java:158
...
14/06/03 15:16:02 INFO scheduler.DAGScheduler: Stage 7 (collect at CountTriangles.java:158) finished in 0.050 s
14/06/03 15:16:02 INFO spark.SparkContext: Job finished: collect at CountTriangles.java:158, took 0.063394393 s
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 ===
14/06/03 15:16:02 INFO spark.SparkContext: Starting job: collect at CountTriangles.java:168
14/06/03 15:16:02 INFO scheduler.DAGScheduler: Registering RDD 14 (distinct at CountTriangles.java:165)
...
14/06/03 15:16:02 INFO scheduler.DAGScheduler: Stage 10 (collect at CountTriangles.java:168) finished in 0.044 s
14/06/03 15:16:02 INFO spark.SparkContext: Job finished: collect at CountTriangles.java:168, took 0.147241806 s
t3=(2,3,4)
t3=(2,4,5)

```

# Chapter 17

## K-mer Counting

### 17.1 Introduction to K-mers

A K-mer is a substring of length  $k$ , and counting the occurrences of all such substrings is a central step in many analyses of DNA sequence data. Counting K-mers for a DNA sequence means finding frequencies of K-mers for the entire sequence. In bioinformatics, counting K-mers is used for genome and transcriptome assembly, metagenomic sequencing, and for error correction of sequence reads. Although simple in principle, counting K-mers is a big data problem, since DNA sequences can total up to several billions DNA code. K-mer counting problem has been defined in <http://schatzlab.cshl.edu/teaching/exercises/hadoop/>.

The goal of this chapter is to provide a complete K-mer solution using Spark/Hadoop. Our implementation discovers all K-mers for a given  $K > 0$  and finds top-N K-mers for a given  $N > 0$ .

A K-mer is a DNA word of length  $k$ . For example, if a sample DNA sequence is "CACACACAGT," then the following are 3-mers and 5-mers for that sequence:

```
original sequence: CACACACAGT
3-mers:           CAC
                  ACA
                  CAC
```

```

ACA
CAC
ACA
CAG
AGT

original sequence: CACACACAGT
5-mers:
      CACAC
      ACACA
      CACAC
      ACACA
      CACAG
      ACAGT

```

Given a DNA sequence (call it string  $S$  – alphabet is  $\{A, C, G, T, \dots\}$ ), we are interested in counting the number of occurrences in  $S$  of every substring of length  $k$ . Typically DNA sequences are represented as a FASTQ file format (every 4 lines of input file is a single sequence). Note that a given DNA sample may have billions of sequences.

## What are the applications of counting K-mers?

Counting the K-mers in a DNA sequence is a very important step in many bioinformatics applications. The following are examples of applications of K-mer counting:

- Use K-mer frequencies to find out if a misalignment between reads is a sequencing error or a genuine difference in sequence.
- Use K-mer frequencies to detect repeated sequences, such as transposons, which is an important factor for biological role applications
- Use K-mer frequencies to correct short-read assembly errors
- Use K-mer frequencies to compute metrics such as **relatedness** and **unique enough** (useful in metagenomic applications)

## 17.2 K-mer counting using MapReduce

Conceptually, K-mer counter using MapReduce is similar to the "word count" program, but since there are no spaces in the human genome, we will count overlapping K-mers instead of discrete words.

### 17.2.1 K-mer counting using MapReduce: map()

The idea is if the genome sequence is: CACACACAGT and we are counting 3-mers, then the map() function will output the following (key, value) pairs:

CAC	1
ACA	1
CAC	1
ACA	1
CAC	1
ACA	1
CAG	1
AGT	1

**Listing 17.1:** K-mer Counting map() Function

```
1 /**
2  * @param key is generated by MapReduce, ignored here
3  * @param sequence (as value) is one line of input for a given genome sequence
4 */
5 map(Long key, String sequence) {
6     // we will get the value of K (K-mer) from setup()
7     // or broadcast which will be called once
8     for (int i=0; i < sequence.length()-K+1 ; i++) {
9         String kmer = sequence.substring(i, K+i);
10        emit(kmer, 1);
11    }
12 }
```

### 17.2.2 K-mer counting using MapReduce: reduce()

The sort and shuffle function will sort them (output of map()) so the same key comes right after each other:

```
ACA    1
ACA    1
ACA    1
CAC    1
CAC    1
CAC    1
CAG    1
AGT    1
```

Finally, the reduce() function will output:

```
ACA    3
CAC    3
CAG    1
AGT    1
```

#### Listing 17.2: K-mer Counting reduce() Function

```
1 /**
2  * @param key is the unique kmer generated by mappers
3  * @param value is a list of integers (partial count of a kmer)
4 */
5 reduce(String key, List<integer> value) {
6     int sum = 0;
7     for (int count : value) {
8         sum += count;
9     }
10    emit(key, sum);
11 }
```

### 17.2.3 K-mer Counting with MapReduce and Hadoop

I have provided a complete solution of K-mer counting by 4 small classes:

- KmerCountDriver (Mapreduce job driver)
- KmerCountMapper (defines map() function)
- KmerCountReduce (defines reduce() function)
- KmerUtil (generates K-mers for a given k)

## 17.3 Input Data for K-mer Counting

For input, we will use FASTQ format. FASTQ is a concise and compact format, which stores DNA sequences in a text file. Each consecutive 4 lines in a FASTQ file represent a single sequence. For example, the following 4 lines represent a sequence:

```
LINE-1: @EAS54_6_R7_30800
LINE-2: GTTGCTTCCCGCGTGGGTGGGTCGGGG
LINE-3: +EAS54_6_R7
LINE-4: ;;;;;;33;;;;9;7;;.7;393333
```

LINE-2 represents the sequence (the other lines are ignored for K-mer counting). For details on FASTQ files, see <http://maq.sourceforge.net/fastq.shtml>. To read FASTQ file format, we do need to understand the format specification. LINE-1 begins with @, LINE-3 begins with +, and LINE-4 begins with ! or ~. Therefore, we will discard all lines except LINE-2, which represents the sequence to be analyzed.

You may use the following sample data to test your K-mer counting.

- ecoli genome from: <http://schatzlab.cshl.edu/teaching/exercises/hadoop/data/ecoli.fa.gz>
- human genome from: <http://hgdownload.cse.ucsc.edu/goldenpath/hg19/chromosomes/>. Note that hg19 means human genome, build 19.

### 17.3.1 Sample Runs of K-mer Counting

Now, you may use the K-mer counting solution to answer the following questions:

- What are the top 10 most frequently occurring 9-mers in E coli?
- What are the top 10 most frequently occurring 9-mers in hg19?
- What are the top 10 most frequently occurring 21-mers in hg19?

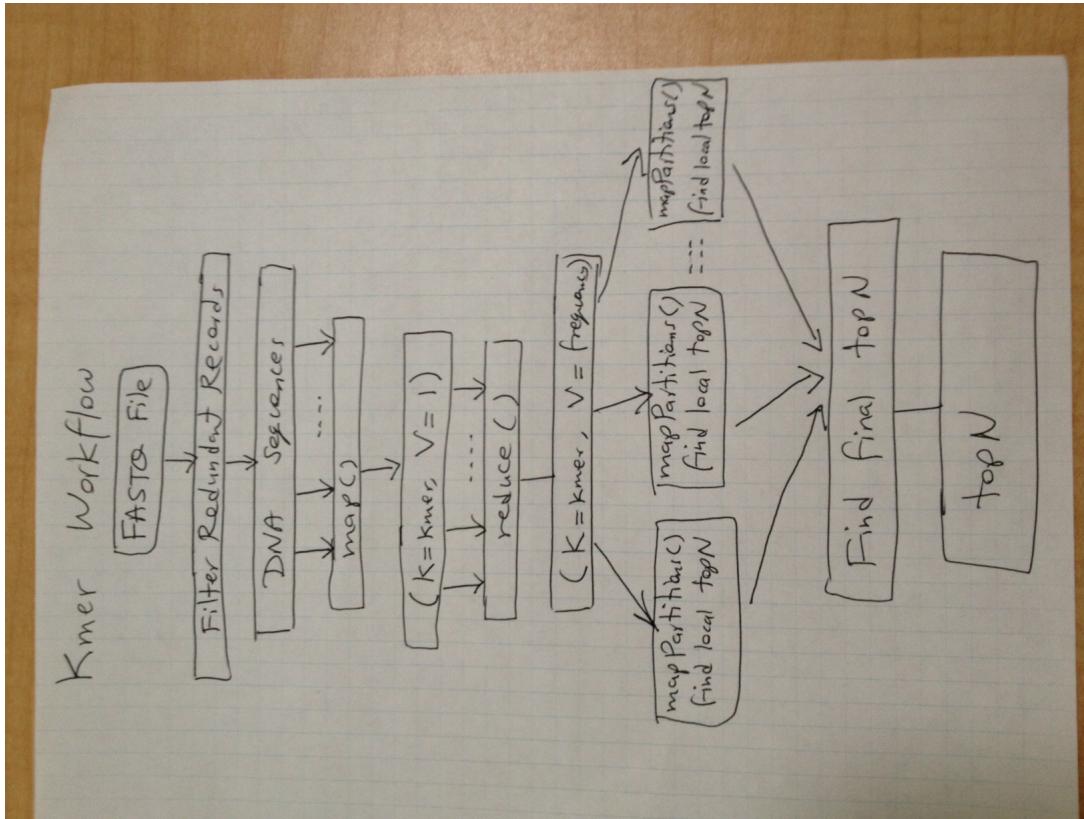


Figure 17.1: K-mer MapReduce WorkFlow

## 17.4 K-mer Implementation in Spark

K-mer MapReduce workflow implementation is provided below:

Assume that we want to discover K-mers (for a given  $K > 0$ ) and top-N (for a given  $N > 0$ ) for a set of FASTQ files. Since FASTQ file format is very well-defined, first we create a JavaRDD<String> for the given FASTQ file. Next, we remove the records, which are not sequences (lines similar to LINE-1, LINE-3, and LINE-4). This filtering is implemented by the JavaRDD.filter() function. Once, we have only sequences, then we create  $(K, 1)$  pairs, where  $K$  is a K-mer. Then, we find frequency of kmers. Finally, we can find top-N kmers for  $N > 0$ . Finding top-N is simple: we

assume that (K2,V2) are partitioned (K2 is a kmer and V2 is the frequency of K2), then we map each partition into top-N and then once we have a list of top-N (one top-N from each partition), then we may do the final reduction to find top-N.

We have 3 inputs to our Spark program:

- FASTQ file: stored in HDFS
- $K > 0$  to find K-mers
- $N > 0$  to find top-N K-mers

Spark solution is implemented a single Java driver class; this is possible due to high abstraction of Spark API. First, the entire solution is presented as a set of 10 high-level steps, then each step is implemented as a Java/Spark API.

#### 17.4.1 K-mer High-Level Solution in Spark

**Listing 17.3:** K-mer High-Level Solution in Spark

```
1 // STEP-0: import required classes and interfaces
2 public class Kmer {
3     static JavaSparkContext createJavaSparkContext() throws Exception {...}
4
5     public static void main(String[] args) throws Exception {
6         // STEP-1: handle input parameters
7         // STEP-2: create a Spark context object
8         // STEP-3: broadcast K and N as global shared objects,
9         // which can be accessed from all cluster nodes
10        // STEP-4: read FASTQ file from HDFS and create the first RDD
11        // STEP-5: filter redundant records
12        // STEP-6: generate K-mers
13        // STEP-7: combine/reduce frequent kmers
14        // STEP-8: create a local top-N for all partitions
15        // STEP-9: now collect all topN from all partitions
16        // and find final topN from all partitions
17        // STEP-10: emit final topN descending
18        System.exit(0);
19    }
20 }
```

## 17.4.2 STEP-0: import required classes and interfaces

**Listing 17.4:** STEP-0: import required classes and interfaces

```
1 // STEP-0: import required classes and interfaces
2 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;
```

## 17.4.3 createJavaSparkContext()

To create RDDs, you need an instance of a JavaSparkContext, which is a factory class for creating JavaRDD and JavaPairRDD objects. Note that here, I have hard-coded the hostname (as myserver100) for YARN's "resourcemanager". For deploying to production environments, you should read these values from a configuration file.

**Listing 17.5:** createJavaSparkContext()

```
1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     conf.set("yarn.resourcemanager.hostname", "myserver100");
4     conf.set("yarn.resourcemanager.scheduler.address", "myserver100:8030");
5     conf.set("yarn.resourcemanager.resource-tracker.address", "myserver100:8031");
6     conf.set("yarn.resourcemanager.address", "myserver100:8032");
7     conf.set("mapreduce.framework.name", "yarn");
8     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
9     conf.set("spark.executor.memory", "1g");
10    JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "Kmer", conf);
11    return ctx;
12 }
```

#### 17.4.4 STEP-1: handle input parameters

This step reads 3 required inputs (FASTQ file, K – size of kmer – , and N – for topN) from the command line.

##### **Listing 17.6:** STEP-1: handle input parameters

```
1 // STEP-1: handle input parameters
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 file as input
7 final int K = Integer.parseInt(args[1]); // to find K-mers
8 final int N = Integer.parseInt(args[2]); // to find top-N
```

#### 17.4.5 STEP-2: create a Spark context object

We use Kmer.createJavaSparkContext() to create a JavaSparkContext object.

##### **Listing 17.7:** STEP-2: create a Spark context object

```
1 // STEP-2: create a Spark context object
2 JavaSparkContext ctx = createJavaSparkContext();
```

#### 17.4.6 STEP-3: broadcast global shared objects

In Spark framework, to use shared objects in all cluster nodes, we can broadcast them and read these from map() and reduce() functions. To make an object sharable, we use the following API:

```
T t = <an-instance-of-T>;
Broadcast<T> broadcastT = ctx.broadcast(t);
```

To access the broadcasted object (shared object) from any cluster node, we use the following API:

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

Since K (K-mer size) and N (for top-N) are required from all cluster nodes, we broadcast these two values as:

**Listing 17.8:** STEP-3: broadcast K and N as global shared objects

```
1 // STEP-3: broadcast K and N as global shared objects,
2 // which can be accessed from all cluster nodes
3 final Broadcast<Integer> broadcastK = ctx.broadcast(K);
4 final Broadcast<Integer> broadcastN = ctx.broadcast(N);
```

#### 17.4.7 STEP-4: read FASTQ file from HDFS and create the first RDD

This step reads a FASTQ file and creates a JavaRDD<String> object, where each record of FASTQ file is represented as a String object.

**Listing 17.9:** STEP-4: read FASTQ file from HDFS and create the first RDD

```
1 // STEP-4: read FASTQ file from HDFS and create the first RDD
2 JavaRDD<String> records = ctx.textFile(fastqFileName, 1);
3 records.saveAsTextFile("/kmers/output/1");
```

Output of this step is shown below.

```
# hadoop fs -cat /kmers/output/1/part*
@EAS54_6_R1_2_1_413_324
CCCTTCTTGTCCCCAGCGTTCTCC
+
;;3;;;;;;7;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
+
;;;;;;7;;;-;;3;83
@EAS54_6_R1_2_1_443_348
GTTGCTTCTGGCGTGGGTGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;;;9;7;.7;393333
@EAS54_6_R1_2_1_413_324
CCCCCTTGTCTCAGCCCTTCTCC
```

```

+
;;3;;;;;;7;;88
@EAS54_6_R1_2_1_540_792
TTTCAGGCCAAGGCCGATGGATCA
+
;;;;7;--;;3;83
@EAS54_6_R1_2_1_443_348
GTTGTTCTGGCGTGGGTGGGGGG
+EAS54_6_R1_2_1_443_348
;;;;9;7;.7;393333
@EAS54_6_R1_2_1_443_348
GTTGTTCTGGCGTGGGTGGCCCCC
+EAS54_6_R1_2_1_443_348
;;;;9;7;.7;393333

```

#### 17.4.8 STEP-5: filter redundant records

This step uses a very powerful Spark API (`JavaRDD.filter()`) to filter out the redundant records before applying the `map()` function. In a FASTQ file, we just keep the records, which represent DNA sequences.

**Listing 17.10:** STEP-5: filter redundant records

```

1   // STEP-5: filter redundant records
2   // JavaRDD<T> filter(Function<T,Boolean> f)
3   // Return a new RDD containing only the elements that satisfy a predicate.
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; // do not return these records
13             }
14         else {
15             return true;
16         }
17     }
18 });
19 filteredRDD.saveAsTextFile("/kmers/output/1.5");

```

Output of this step is shown below.

```
# hadoop fs -cat /kmers/output/1.5/part*
CCCTTCTTGTCCCCAGCGTTCTCC
TTGGCAGGCCAAGGCCGATGGATCA
GTTGCTTCTGGCGTGGGTGGGGGG
CCCCCCTTGTCTTCAGCCCTTCTCC
TTTCAGGCCAAGGCCGATGGATCA
GTTGTTCTGGCGTGGGTGGGGGG
GTTGTTCTGGCGTGGGTGGCCCC
```

#### 17.4.9 STEP-6: generate K-mers

This step implements the mapper for K-mers by using the `JavaRDD.flatMapToPair()` function. The mapper accepts a sequence and K (size of K-mers) and then generates all (`kmer`, 1) pairs.

**Listing 17.11:** STEP-6: generate K-mers

```
1 // STEP-6: generate K-mers
2 // PairFlatMapFunction<T, K, V>
3 // T => Iterable<Tuple2<K, V>>
4 JavaPairRDD<String, Integer> kmers = filteredRDD.flatMapToPair(new PairFlatMapFunction<
5     String,           // T
6     String,           // K
7     Integer          // V
8 >() {
9     public Iterable<Tuple2<String, Integer>> call(String sequence) {
10         int K = broadcastK.value();
11         List<Tuple2<String, Integer>> list = new ArrayList<Tuple2<String, Integer>>();
12         for (int i=0; i < sequence.length()-K+1 ; i++) {
13             String kmer = sequence.substring(i, K+i);
14             list.add(new Tuple2<String, Integer>(kmer, 1));
15         }
16         return list;
17     }
18 });
19 kmers.saveAsTextFile("/kmers/output/2");
```

Partial output of this step is shown below.

```
# hadoop fs -cat /kmers/output/2/part*
(CC,1)
(CCT,1)
(CTT,1)
...
...
```

```
(GGC,1)  
(GCC,1)  
(CCC,1)  
(CCC,1)  
(CCC,1)
```

#### 17.4.10 STEP-7: combine/reduce frequent kmers

This step implements the reducer for K-mers by using the `JavaPairRDD.reduceByKey()` function. The reducer accepts a key (as kmer) and values (frequencies of K-mers) and then generates final count of kmers.

**Listing 17.12:** STEP-7: combine/reduce frequent kmers

```
1 // STEP-7: combine/reduce frequent kmers  
2 JavaPairRDD<String, Integer> kmersGrouped =  
3     kmers.reduceByKey(new Function2<Integer, Integer, Integer>() {  
4         public Integer call(Integer i1, Integer i2) {  
5             return i1 + i2;  
6         }  
7     });  
8     kmersGrouped.saveAsTextFile("/kmers/output/3");
```

Output of this step is shown below.

```
# 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)
```

#### 17.4.11 STEP-8: create a local top-N

This step partitions all (kmer, frequency) pairs into many partitions and then finds a top-N for every partition. For every partition, we just keep the top N (frequency, kmer) (as `SortedMap` – where `TreeMap` implements `SortedMap` interface). Finding local top-N per partition is implemented by `JavaPairRDD.mapPartitions()` method.

##### **Listing 17.13:** STEP-8: create a local top-N

```
1 // STEP-8: create a local top-N
2 // now, we have: (K=kmer, V=frequency)
3 // next step is find the top-N kmers
4 JavaRDD<SortedMap<Integer, String>> partitions = kmersGrouped.mapPartitions(
5     new FlatMapFunction<Iterator<Tuple2<String, Integer>>, SortedMap<Integer, String>>() {
6         @Override
```

```

7   public Iterable<SortedMap<Integer, String>> call(Iterator<Tuple2<String, Integer>> iter) {
8     int N = broadcastN.value();
9     SortedMap<Integer, String> topN = new TreeMap<Integer, String>();
10    while (iter.hasNext()) {
11      Tuple2<String, Integer> tuple = iter.next();
12      String kmer = tuple._1;
13      int frequency = tuple._2;
14      topN.put(frequency, kmer);
15      // keep only top N
16      if (topN.size() > N) {
17        topN.remove(topN.firstKey());
18      }
19    }
20    //System.out.println("topN="+topN);
21    return Collections.singletonList(topN);
22  }
23 });

```

---

### 17.4.12 STEP-9: Find Final top-N

This step aggregates all top-N's generated from all partitions and creates the final top-N for all partitions. For this step, we do need the value of N (for top-N), which we read it from a broadcasted variable (`broadcastN`).

**Listing 17.14:** STEP-9: Find Final top-N

```

1  // STEP-9: now collect all topN from all partitions
2  // and find final topN from all partitions
3  SortedMap<Integer, String> finaltopN = new TreeMap<Integer, String>();
4  List<SortedMap<Integer, String>> alltopN = partitions.collect();
5  for (SortedMap<Integer, String> localtopN : alltopN) {
6    // frequency = tuple._1
7    // kmer = tuple._2
8    for (Map.Entry<Integer, String> entry : localtopN.entrySet()) {
9      finaltopN.put(entry.getKey(), entry.getValue());
10     // keep only top N
11     if (finaltopN.size() > N) {
12       finaltopN.remove(finaltopN.firstKey());
13     }
14   }
15 }

```

---

### 17.4.13 STEP-10: Emit Final top-N

This step emits the final top-N for all kmers.

#### **Listing 17.15: STEP-10: Emit Final top-N**

```
1 // STEP-10: emit final topN descending
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" + finaltopN.get(frequencies.get(i)));
6 }
```

#### **17.4.14 YARN Script for Spark**

This script runs our Spark program in YARN environment.

#### **Listing 17.16: YARN Script for Spark**

```
1 # cat run_kmer.sh
2 #!/bin/bash
3 export SPARK_HOME=/usr/local/spark-1.0.0
4 export HADOOP_HOME=/usr/local/hadoop-2.4.0
5 export JAVA_HOME=/usr/java/jdk7
6 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
7 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
8 export MY_JAR=/home/mahmoud/mp.jar
9 export SPARK_JAR=$MP/spark-assembly-1.0.0-hadoop2.4.0.jar
10 #
11 export INPUT=/data/sample.fastq
12 export K=3
13 export N=5
14 $SPARK_HOME/bin/spark-submit --class Kmer \
15   --master yarn-cluster \
16   --num-executors 12 \
17   --driver-memory 3g \
18   --executor-memory 7g \
19   --executor-cores 12 \
20   $MY_JAR $INPUT $K $N
```

#### **17.4.15 HDFS Input**

```
# hadoop fs -cat /data/sample.fastq
@EAS54_6_R1_2_1_413_324
CCCTTCTTGCCCCAGCGTTCTCC
+
;;3;;;;;;7;;;;;;88
@EAS54_6_R1_2_1_540_792
TTGGCAGGCCAAGGCCGATGGATCA
```

#### 17.4.16 Output for Final Top-N

==== top 5 ====  
13 GGG  
12 TGG  
11 CCC  
9 GGC  
8 TTC

# Chapter 18

## DNA-Sequencing

### 18.1 Introduction

Today, genome sequencing machines (such as Illumina) are able to generate hundreds of gigabases of DNA and RNA sequencing data in a day for less than US\$1,000 (few years ago the price was over US\$100,000 and the first sequenced human genome cost about US\$3 billion – so the price has dropped considerably!). It is believed that success in the biology and life sciences will depend on our ability to properly analyze the big data sets that are generated by these technologies, which in turn requires us to adopt advances in informatics. MapReduce/Hadoop/Spark enable us to compute and analyze thousands of gigabytes of data in hours (rather than days or weeks).

In simple terms, DNA-sequencing is the sequencing of whole genomes (such as human genomes). Following <http://dnasequencing.com/>: "if finding DNA was the discovery of the exact substance holding our genetic makeup information, DNA sequencing is the discovery of the process that will allow us to READ that information." The main function of DNA sequencing is to find the precise order of nucleotides within a DNA molecule. Also, DNA sequencing process is used to determine the order of the four bases – adenine (A), guanine (G), cytosine (C), and thymine (T) – in a strand of DNA.

What are some of the challenges with DNA-Sequencing? There are many, but I am listing some of the important ones:

- There are several sequencing technology to generate FASTQ files (the length of DNA sequences are different per sequencing technology)
- Input data (FASTQ data) size is BIG (a single DNA-Seq sample size can be up to 400-000GB)
- Using a single powerful server, it takes too long (up to 80 hours) to process one DNA-Seq sample data and get SNP/variants out
- There are many algorithms and steps to run DNA-Seq (selecting proper combinations of open-source tools is a serious challenge). For example, there are quite few mapping/alignment algorithms and parameters
- Scalability (optimizing number of mappers and reducers)

A high-level DNA-Sequencing workflow is presented below. The focus of this chapter will be the implementation of "DNA Sequencing" as a MapReduce program, which will accept DNA data set as a FASTQ data and finally it generate a VCF file, which has variants for a given set of DNA data sets.

One of the major goal of DNA-Sequencing is to find variants (since most of our DNAs are identical and only very small percent is different from each other). One important objective is the identification of genetic variants such as single nucleotide polymorphisms (SNPs). The identification and extraction of SNPs from the raw genetic sequences involves many algorithms and the application of a diverse set of tools.

DNA-Sequencing pipeline includes

- Input data validation: quality control of input data such as FASTQ files
- Alignment: mapping of short reads to the reference genome
- Recalibration: visualization and post-processing of the alignment including base quality recalibration
- Variant detection: the SNP calling procedure along with filtering of SNP candidates

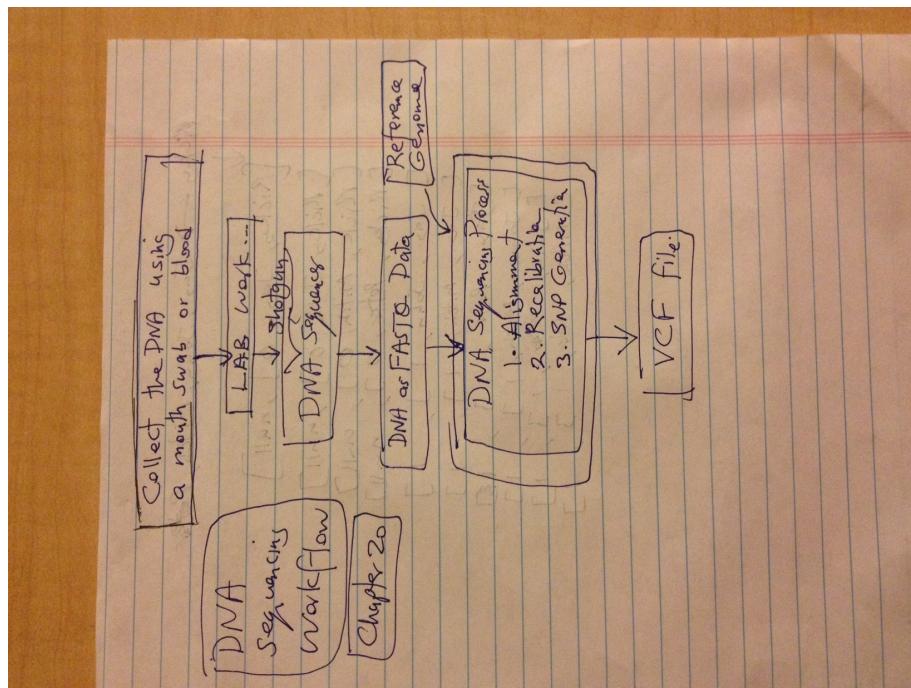


Figure 18.1: High-Level View of DNA Sequencing

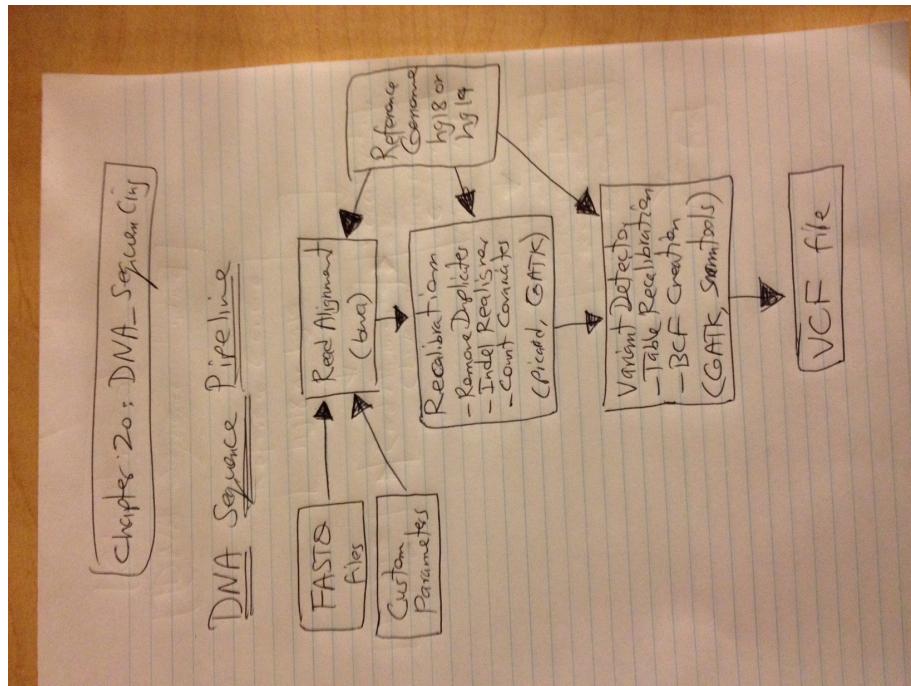


Figure 18.2: DNA Sequencing Pipeline

The DNA Sequencing pipeline is presented below.

There are plenty of data (<http://www.1000genomes.org/data>) to analyze and apply DNA-Sequencing and there are lots of open-source algorithms for these 4 steps outlined above. We do need to understand the choice of the open-source tools, which significantly affects the final results.

## 18.2 Input to DNA-Sequencing

The common format for DNA sequencing is FASTQ format. FASTQ format is a text-based format for storing both a biological sequence and its quality scores. For a given FASTQ file, each 4 record/lines represent a single DNA sequence. General syntax and an example is given below.

- Syntax:

```
<fastq>:= <block>+
<block>:=@<seqname>\n<seq>\n[<seqname>]\n<qual>\n
<seqname>:= [A-Za-z0-9_.:-]+
<seq>:= [A-Za-z\n.\~]+
<qual>:= [!-\~\n]+
```

- Example:

```
@NCYC361-11a03.q1k bases 1 to 1576
GCGTGCCCGAAAAAAATGCTTTGGAGGCCGCGCGTGAAT...
+NCYC361-11a03.q1k bases 1 to 1576
!))))))****(((*%((((*((+,**((+**+, -...
```

The FASTQ data can be paired or non-paired. If it is paired, then input for a DNA-Sequencing will be a pair of files: `left_file.fastq` and `right_file.fastq`. If it is non-paired, then there will be a single file `file.fastq`.

## 18.3 Input data validation

This step validates the format of fastq files. With validation, you want to make sure that quality of input files are guaranteed. Input data validation

tools help to provide a way to do some quality control checks on raw sequence data (for example in FASTQ file format) coming from high throughput sequencing pipelines.

There are lots of open-source tools for input data validation. For example, for FASTQ validation you may use the following open-source tools:

- FastQValidator (<http://genome.sph.umich.edu/wiki/FastQValidator>)
- FastQC (<http://www.bioinformatics.babraham.ac.uk/projects/fastqc>)

Input data validation step is very simple and straightforward and we will not cover it here. Our focus will be on mapping/alignment, recalibration, and variant detection algorithms using MapReduce algorithms.

## 18.4 DNA-Sequencing: Alignment

What is "sequence alignment"? Sequence alignment is the comparison of two or more DNA or protein sequences to each other. The main purpose of sequence alignment is to highlight similarity between the sequences.

For global sequence alignment, consider the following example with two input sequences over the same alphabet:

*Sequence1 : GCGCATGGATTGAGCGA*

*Sequence2 : TGCGCCATTGATGACCA*

Output: an alignment of the two sequences (a possible alignment):

*-GCGC-ATGGATTGAGCGA*

*TGCGCCATTGAT-GACC-A*

We can observe three elements in a possible alignment output:

- Perfect matches (Blue color)
- Mismatches (Red color)
- Insertions and deletions (indel)(Green color)

For alignment phase, we use MapReduce/Hadoop along with the following open-source tools:

- BWA: Burrows-Wheeler Aligner (BWA) is an efficient program that aligns relatively short nucleotide sequences against a long reference sequence such as the human genome (see <http://bio-bwa.sourceforge.net/>)
- SAM Tools: provide various utilities for manipulating alignments in the SAM format, including sorting, merging, indexing and generating alignments in a per-position format (see <http://samtools.sourceforge.net/>)

## 18.5 MapReduce Algorithms for DNA-Sequencing

Typical DNA-Sequencing for single sample data (about 300-400GB of FASTQ file format) might take 70+ hours using a very powerful single server. The goal of MapReduce algorithm is to find the answer in few hours and make the solution scalable.

Since most of open source tools (such as BWA, SAM Tools, and GATK) for alignment, recalibration, and variant detection have Linux command line interfaces, `map()` and `reduce()` at each MapReduce phases will call Linux shell scripts with passing proper parameters. To execute these shell scripts, we will use the FreeMarker<sup>1</sup> templating language, which will merge Java objects (and data structures) with a template to create a proper shell script. To identify one DNA-Seq from another DNA-seq, for each analysis, we assign and utilize a unique GUID called "analysis ID" (this helps us to keep input and output directories organized).

FreeMarker template engine usage is presented below:

We present 3 steps MapReduce solution (see figure):

DNA-Sequencing Data Flow for 3 steps MapReduce solution is presented below. It shows how data are partitioned and merged at different steps of MapReduce algorithms.

---

<sup>1</sup><http://freemarker.org/>

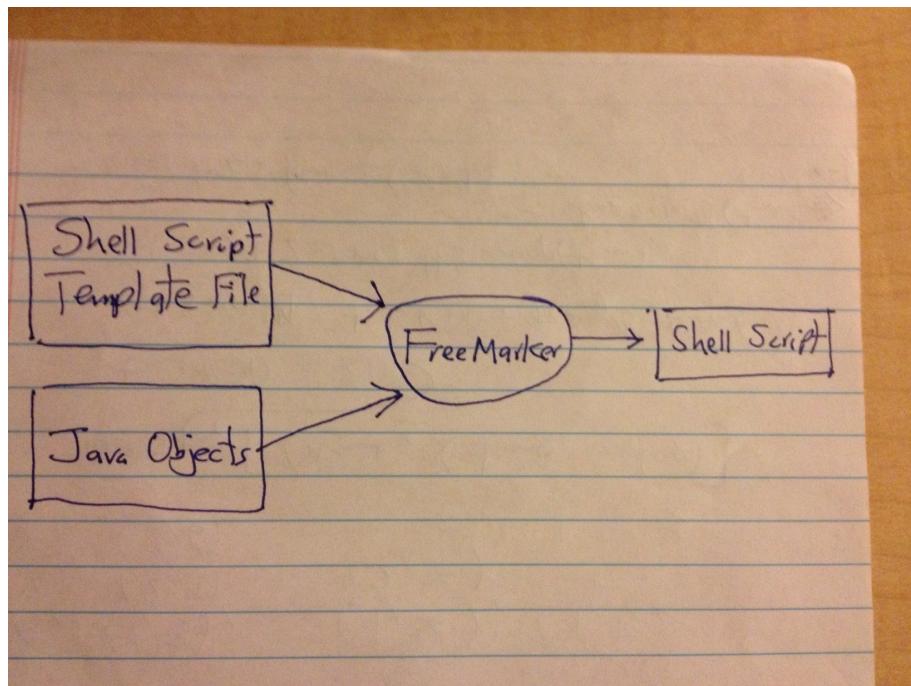


Figure 18.3: FreeMarker Template Engine

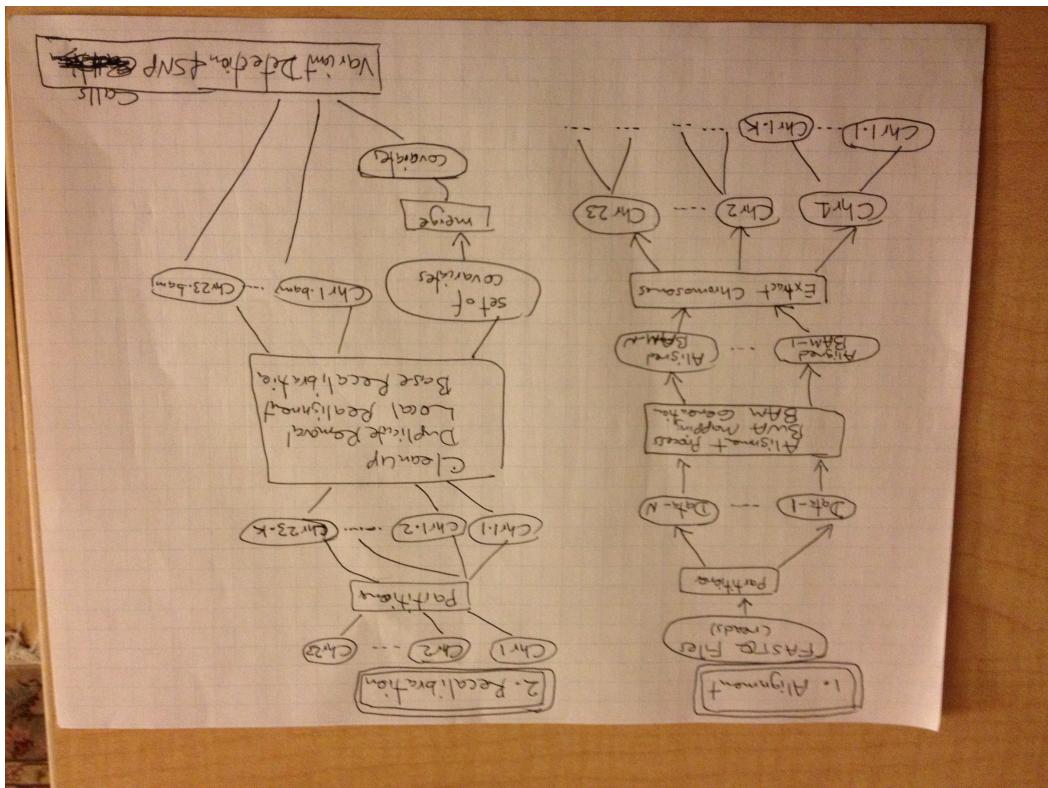


Figure 18.4: 3 Steps MapReduce Solution (Steps 1 and 2)

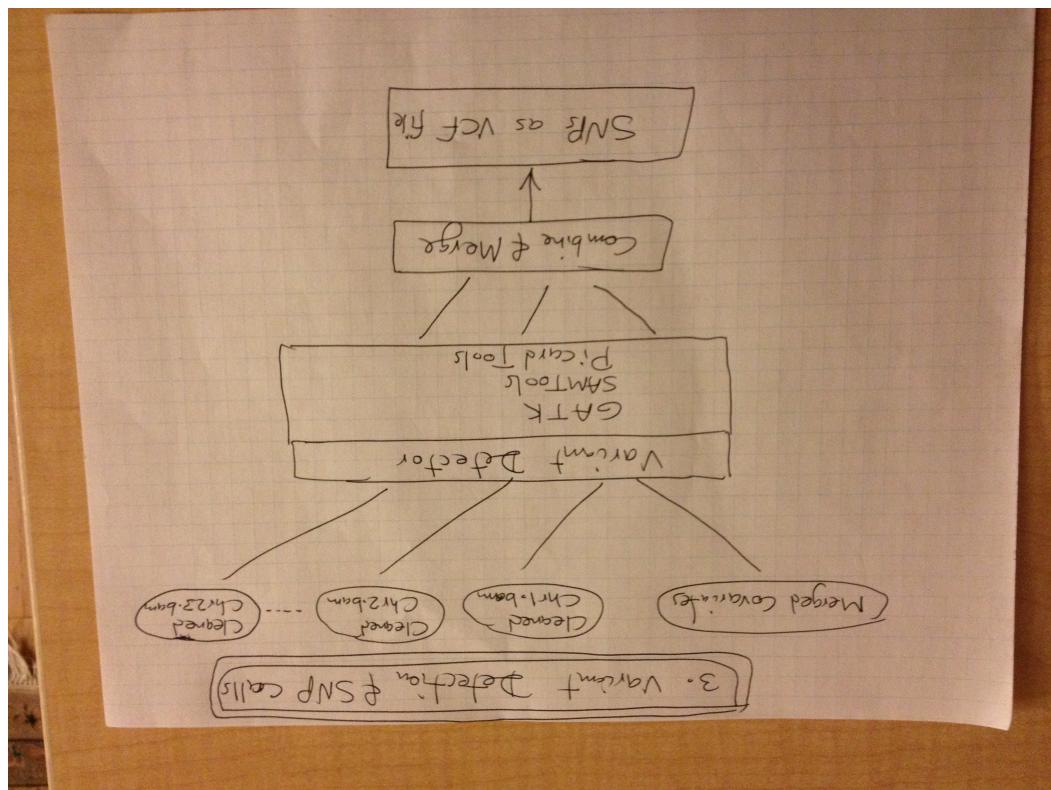


Figure 18.5: 3 Steps MapReduce Solution (Step 3)

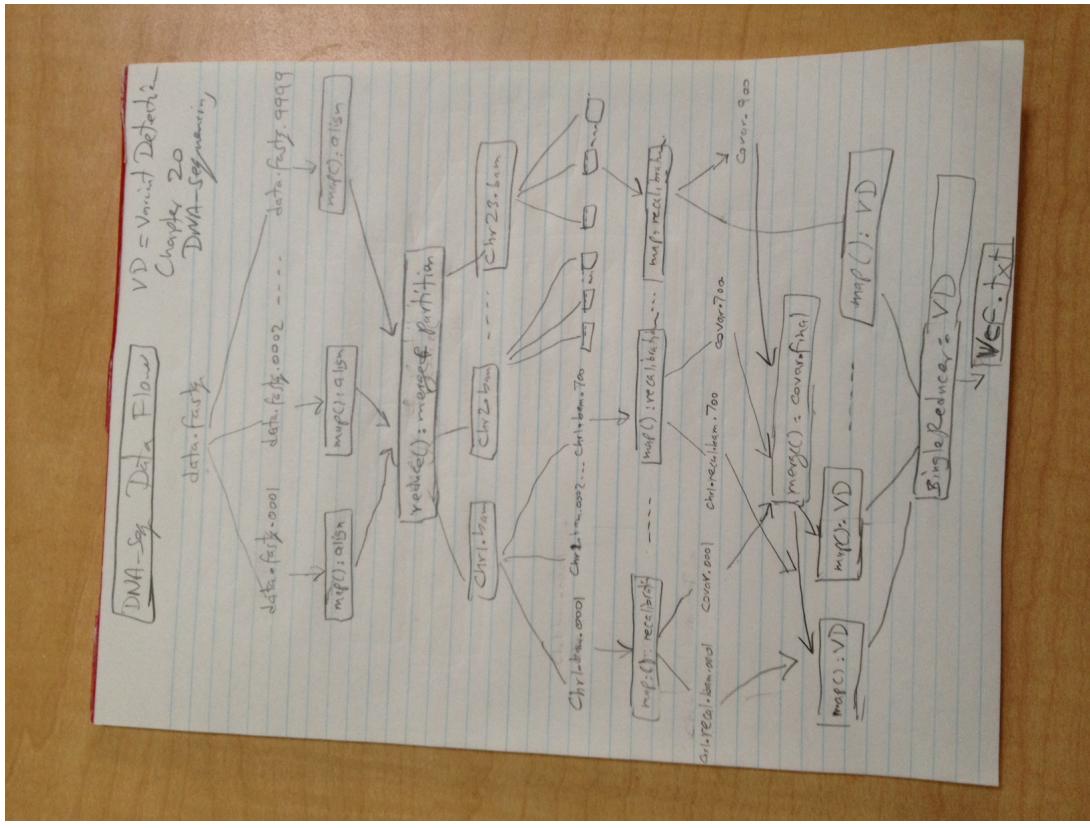


Figure 18.6: DNA-Sequencing Data Flow

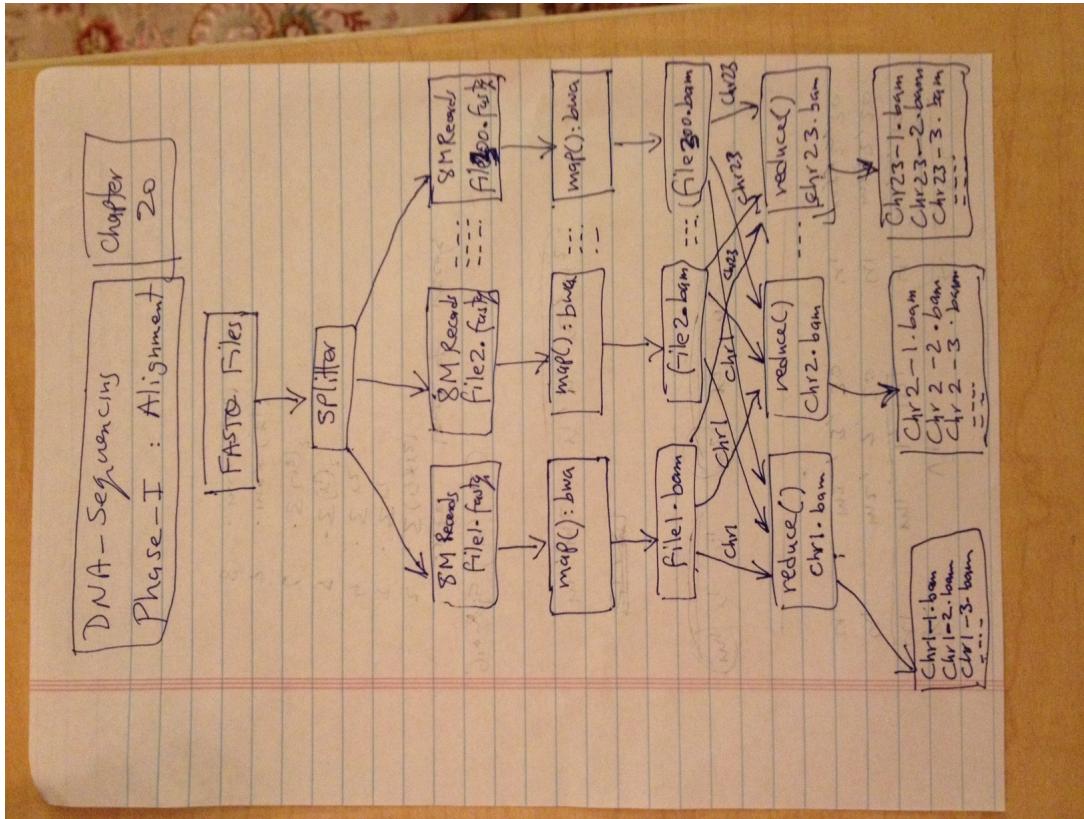


Figure 18.7: DNA-Sequencing: Alignment Workflow

## 18.6 MR Algorithms: Step-1: DNA-Sequencing: Alignment

A High-level Workflow for Alignment phase is depicted below:

Before alignment step starts, we partition DNA-Sequence FASTQ file(s) into 8M lines (or 2M sequences – in FASTQ format, each 4 lines represent a single DNA sequence data). If FASTQ data is paired, then our input is a pair of files: `left_file.fastq` and `right_file.fastq`. If it is non-paired, then input will be a single file `file.fastq`. For paired data, we will partition as (for paired data, an alignment `map()` function will process `left_file.fastq.NNNN` and `right_file.fastq.NNNN` together):

```
left_file.fastq.0000  right_file.fastq.0000  
left_file.fastq.0001  right_file.fastq.0001  
left_file.fastq.0002  right_file.fastq.0002  
...                 ...
```

For non-paired data, we will partition as (for non-paired data, an alignment `map()` function will process `file.fastq.NNNN`):

```
file.fastq.0000  
file.fastq.0001  
file.fastq.0002  
...
```

Each partition (so called a chunk) will be consumed by a `map()` function. The partitioning into 8M lines does depend on the size of your hadoop cluster. If you have a cluster of 50 nodes and each node can handle 4 mappers, then you should split your FASTQ file by 200. For example, if your total input size is about 400GB, then you should partition your input into 2GB chunks (this way you will maximize the usage of all your mappers). The `map()` function will read input file (one single chunk) and will generate an aligned file in BAM format. Here, the `map()` function uses the BWA (Burrows-Wheeler Aligner<sup>2</sup>) to perform the alignment process. Once alignment is done, then it will extract all chromosomes (1, 2, ..., 22, 23<sup>3</sup>) and save them in MapReduce file system (in Hadoop it is called HDFS). For example, if we had 800 partitions, then we have generated 800 files per chromosome (total of  $23 * 800 = 18,400$  files). There will be only 23 reducers (one per chromosome). The reducer will concatenate (merge-sort) all chromosomes for a specific chromosome ID. All chromosomes 1 will be concatenated into a single file called `chr1.bam` and all chromosomes 2 will be concatenated into a single file called `chr2.bam` and so on. Then each reducer will partition merged BAM file into small files, which will be used as input to the recalibration phase.

---

<sup>2</sup><http://bio-bwa.sourceforge.net/>

<sup>3</sup>There is no chromosome 23, but we concatenate chromosomes X, Y, and M and call it chromosome 23

### 18.6.1 Step-1: map(): Alignment

For alignment, our solution will accept FASTQ file format as input and will generate partitioned chromosomes (chr1, chr2, ..., chr22, chr23). Note that human chromosomes are numbered as 1, 2, ..., 22, X, Y, and M. There is no chromosome 23, but we concatenate chromosomes X, Y, and M and call it chromosome 23.

**Listing 18.1:** map(): Alignment

```
1 /**
2  * @param key a key generated by MapReduce framework
3  * @param value a partitioned FASTQ file (may be 8M lines = 2M sequences)
4 */
5 map(key, value) {
6     // note: chr23 = concat(chrX, chrY, chrM)
7     alignedBAMFile = alignByBWA(value);
8     (chr1File, chr2File, ..., chr22File, ch23File) = partitionByChromosome(alignedB
9     for (i=1, i < 24; i++) {
10         emit(chr<i>, chr<i>File);
11     }
12 }
```

The `alignByBWA()` function accepts a partitioned FASTQ file, performs the alignment, and finally partitions the alinged file by chromosome. All of these are done by a shell script template. Portions of this template are listed below:

**Listing 18.2:** Alignment Phase: For Non-Paired Input

```
1 #!/bin/bash
2 ...
3 export BWA=<bwa-install-dir>/bwa
4 export SAMTOOLS=<samtools-install-dir>/samtools
5 export BCFTOOLS=<bcftools-install-dir>/bcftools
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 # data directories
11 export TMP_HOME=<root-tmp-dir>/tmp
12 export BWA_INDEXES=<root-index-dir>/ref/bwa
13 ...
```

```

14 # define ref. genome
15 export REF=<root-reference-dir>/hg19.fasta
16
17 ### step 1: alignment
18 # the KEY uniquely identifies the input file
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 # start indexing aln.flt.bam file
30 $SAMTOOLS index aln.flt.bam
31
32 # partition aligned data
33 for i in 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
34 do
35     CHR=chr${i}
36     $SAMTOOLS view -b -o ${CHR}.bam aln.flt.bam ${CHR}
37     $HADOOP_HOME/bin/hadoop fs -put ${CHR}.bam /genome/dnaseq/output/$ANALYSIS_ID/${CHR}/${KEY}.${CHR}.bam
38 done
39
40 # do the same thing for X, Y and M chromosomes
41 $SAMTOOLS view -b -o chr23.bam aln.flt.bam chrX chrY chrM
42 $HADOOP_HOME/bin/hadoop fs -put chr23.bam /genome/dnaseq/output/$ANALYSIS_ID/chr23/${KEY}.chr23.bam
43
44 exit 0

```

---

The provided shell script handles non-paired data only. If the input files are paired, then lines 24-26 are replaced by:

**Listing 18.3: Alignment Phase: For Paired Input**

```

1 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE_1 > out1.sai
2 $BWA aln -t $NUM_THREAD $REF $INPUT_FILE_2 > out2.sai
3 $BWA sampe -r $INFO_RG $REF out1.sai out2.sai $INPUT_FILE_1 $INPUT_FILE_2 | \
4     $SAMTOOLS view -Su -F 4 - | $SAMTOOLS sort - aln.flt

```

---

### 18.6.2 Step-1: reduce(): Alignment

For Alignment phase, there will be exactly 23 reducers (one reducer per chromosome). The reducer key will be a composite key of <chrID><;><analysisID>, where chromosome ID (labeled as 01, 02, 03, ..., 23). Note the chromosome ID of 23 includes chrM, chrX, and chrY. Each reducer will merge all aligned .bam files into single merged.bam:

```

chr<i>.bam = merge the following files:
    chr<i>.bam.0000
    chr<i>.bam.0001
    ...
    chr<i>.bam.0437
    ...

```

After merging all files into a single `chr<i>.bam` file, we partition `chr<i>.bam` file into many small bam files to be fed to the Recalibration mapper of Step-2. The partitioned files will be:

```
chr<i>.bam.j (j = 1, 2, 3, ..., 100+)
```

The reduce() function for alignment phase is presented below:

#### **Listing 18.4: reduce(): Alignment**

```

1 /**
2 * @param key is a <chrID><;><analysis_id>
3 *   where chrID is in (1, 2, 3, ..., 23)
4 * @param value is ignored (not used)
5 */
6 reduce(key, value) {
7     DNASeq.mergeAllChromosomesAndPartition(key);
8 }

```

The bulk of work is with `DNASeq.mergeAllAndPartition()` method, which merges all aligned bam files for a specific chromosome and then the final merged file is partitioned for further processing by the recalibration phase (called Step-2).

#### **Listing 18.5: mergeAllChromosomesAndPartition(): method**

```

1 /**
2 *
3 * reducerKey=<chrID>;<analysis_id>
4 *   where chrID=1, 2, ..., 22, 23 (23 includes chrM, chrX, chrY)
5 */
6 public static void mergeAllChromosomesAndPartition(String reducerKey) throws Exception {
7     // split the line: each line has (fields are separated by ";")

```

```

8  String[] tokens = reducerKey.split(";");
9  String chrID = tokens[0];
10 String analysisID = tokens[1];
11 Map<String, String> templateMap = new HashMap<String, String>();
12 templateMap.put("chr_id", chrID);
13 templateMap.put("analysis_id", analysisID);
14 mergeAllChromosomesBamFiles(templateMap);
15 partitionSingleChromosomeBam(templateMap);
16 }

```

---

As you can observe from the mergeAllChromosomesAndPartition() method, both of the helper methods mergeAllChromosomesBamFiles() and partitionSingleChromosomeBam() use FreeMarker template engine to pass the required Java objects and then execute shell scripts on behalf of reducers. For example,

**Listing 18.6:** mergeAllChromosomesAndPartition(): method

```

1 /**
2 * This method will merge the following files and creates a single chr<i>.bam file
3 * where ( i is in {1, 2, ..., 23} ).  

4 *
5 * HDFS: /.../chr<i>/chr<i>.bam.0000
6 * HDFS: /.../chr<i>/chr<i>.bam.0001
7 * ...
8 * HDFS: /.../chr<i>/chr<i>.bam.0437
9 *
10 * Then merge all these (.0000, .0001, ..., .0437) files and save it in
11 * /data/tmp/<analysis_id>/chr<i>/chr<i>.bam
12 *
13 * Once chr<i>.bam is created, then we partition it into small bam files,
14 * which will be fed to RecalibrationDriver (Step-2 of DNA Sequencing)
15 *
16 */
17 public static void mergeAllChromosomesBamFiles(Map<String, String> templateMap)
18 throws Exception {
19 TemplateEngine.initTemplateEngine();
20 String templateFileName = <freemarker-template-file-as-a-bash-script>;
21 // create the actual script from a template file
22 String chrID = templateMap.get("chr_id");
23 String analysisID = templateMap.get("analysis_id");
24 String scriptFileName = createScriptFileName(chrID, analysisID);
25 String logFileName = createLogFileName(chrID, analysisID);
26 File scriptFile = TemplateEngine.createDynamicContentAsFile(templateFileName,
27                                         templateMap,
28                                         scriptFileName);
29 if (scriptFile != null) {
30     ShellScriptUtil.callProcess(scriptFileName, logFileName);
31 }
32 }

```

---

The TemplateEngine.createDynamicContentAsFile() method does the magic: it takes two inputs (templateFileName and templateMap) and produces a scriptFileName. Basically all parameters are passed to templateFileName and then a new shell script is generated as a scriptFileName, which is then executed ob behalf of a reducer. There are two important classes (TemplateEngine and ShellScriptUtil), which needs some discussion. The `ShellScriptUtil.callProcess()` method accepts a shell script file (first parameter), which executes it and then writes all log from the script execution to a log file (second parameter). Logging is asynchronous: it means that as you execute the script, the log file immediately becomes available.

The TemplateEngine class is defined below: it just implements the basic notion of a templating engine: it accepts a template (as a text file with key holders) and (key,value) pairs as a Java Map and then creates a brand new file in which all keys in template are replaced by values.

### **Listing 18.7:** TemplateEngine Class

```

1 import java.io.File;
2 import java.io.Writer;
3 import java.io.FileWriter;
4 import java.util.Map;
5 import java.util.concurrent.atomic.AtomicBoolean;
6 import freemarker.template.Template;
7 import freemarker.template.Configuration;
8 import freemarker.template.DefaultObjectWrapper;
9
10 /**
11  * This class uses FreeMarker (http://freemarker.sourceforge.net/).
12  * FreeMarker is a "template engine"; a generic tool to generate text
13  * output (anything from shell scripts to autogenerated source code)
14  * based on templates. It's a Java package, a class library for Java
15  * programmers. It's not an application for end-users in itself, but
16  * something that programmers can embed into their products.
17 *
18 * @author Mahmoud Parsian
19 *
20 */
21 public class TemplateEngine {
22
23     // You usually do it only once in the whole application life-cycle
24     private static Configuration TEMPLATE_CONFIGURATION = null;
25     private static AtomicBoolean initialized = new AtomicBoolean(false);
26
27     // the following template directories will be loaded from configuration file
28     private static String TEMPLATE_DIRECTORY = "/home/dnaseq/template";
29
30     public static void init() throws Exception {
31         if (initialized.get()) {
32             // it is already initialized and returning...

```

```

33         return;
34     }
35     initConfiguration();
36     initialized.compareAndSet(false, true);
37 }
38
39 static {
40     if (!initialized.get()) {
41         try {
42             init();
43         }
44         catch(Exception e) {
45             theLogger.error("TemplateEngine init failed at static initialization.", e);
46         }
47     }
48 }
49
50 // this supports single template directory
51 private static void initConfiguration() throws Exception {
52     TEMPLATE_CONFIGURATION = new Configuration();
53     TEMPLATE_CONFIGURATION.setDirectoryForTemplateLoading(new File(TEMPLATE_DIRECTORY));
54     TEMPLATE_CONFIGURATION.setObjectWrapper(new DefaultObjectWrapper());
55     TEMPLATE_CONFIGURATION.setWhitespaceStripping(true);
56     // if the following is set, then undefined keys will be set to "".
57     TEMPLATE_CONFIGURATION.setClassicCompatible(true);
58 }
59
60 public static File createDynamicContentAsFile(...){...}

```

---

The most important method of TemplateEngine class is defined below. This method accepts a template file, which has key holders and a set of (key,value) pairs and then generates a new file by substituting the given keys in key holders.

**Listing 18.8:** TemplateEngine.createDynamicContentAsFile() Method

```

1 /**
2  * @param templateFile is a template filename such as script.sh.template
3  * @param keyValuePairs set of (K,V) pairs
4  * @param outputFileName is a generated filename from templateFile
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        // create a template : example "cb_stage1.sh.template2"
17        Template template = TEMPLATE_CONFIGURATION.getTemplate(templateFile);

```

```

18     // Merge data-model with template
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 }
```

---

## 18.7 Step-2: DNA-Sequencing: Recalibration

Recalibration is the second phase of DNA sequencing pipeline. In recalibration step, each `map()` function will work on a specific aligned chromosomes. The mapper will do "marking duplicates", "local re-alignment", and recalibration. The goal of `map()` is to create a local "recalibration table" (called covariates), which will be merged by the single reducer. Then a single reducer will create the final single file (called "recalibration table"), which will be used by the `map()` of variant detection step (called Step-3 – final step of the DNA Sequencing).

Recalibration MapReduce algorithm (data flow) is presented below.

Following the alignment phase, we create a special metadata to be used by recalibration mappers. The metadata generated for recalibration mappers has the following format:

<counter><;><partitioned-bam-file><;><ref\_genome><;><analysis\_id>

where

<counter> is a auto-generated sequence numbers 0000, 0001, 0002, ...  
<partitioned-bam-file> is a chunk of partitioned aligned file  
<ref\_genome> refers to hg18 or hg19  
<analysis\_id> is a GUID for DNA Sequencing (to manage

### DNA-Sequencing : Recalibration Phase

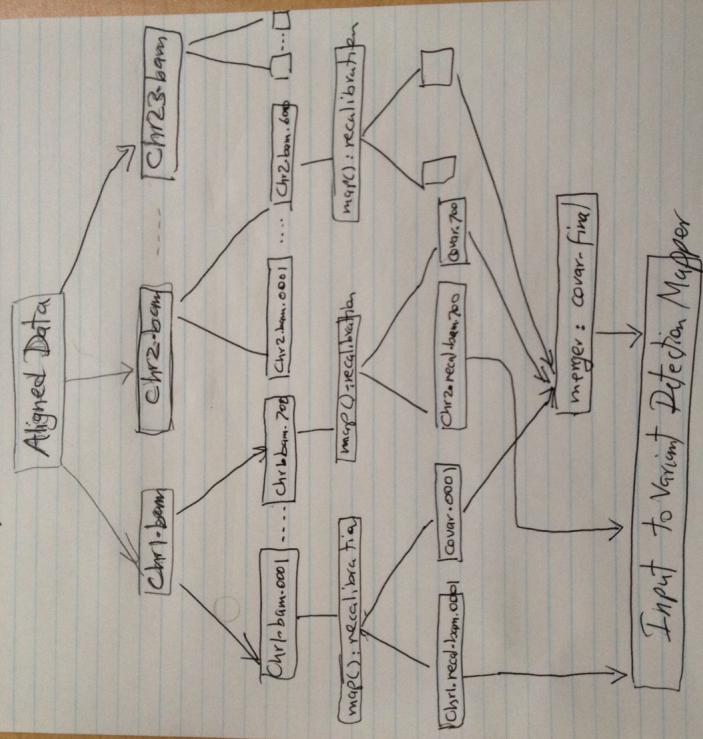


Figure 18.8: DNA-Sequencing: Recalibration

one analysis from another one)

Example of an input is given below:

```
0001;chr07.bam.0001;hg19;208  
0002;chr07.bam.0002;hg19;208  
0003;chr07.bam.0003;hg19;208  
...
```

Recalibration mapper is presented below:

#### Listing 18.9: Recalibration Mapper

```
1 // key is MR generated, ignored here  
2 // value is: <counter><;><partitioned-bam-file><;><ref_genome><;><analysis_id>  
3 map(key, value) {  
4     // actual file location will be: /data/dnaseq/align/ANALYSIS_ID/merged.bam.<KEY>  
5     Map<String, String> tokens = DNASEq.tokenizeRecalibrationMapperInput(value);  
6     String reducerKey = tokens.get("analysis_id");  
7     DNASEq.recalibrationMapper(tokens);  
8     emit(reducerKey, value);  
9 }  
10  
11 public static void recalibrationMapper(Map<String, String> templateMap)  
12 throws Exception {  
13     TemplateEngine.init();  
14     // create the actual script from a template file  
15     String key = templateMap.get("key");  
16     String analysisID = templateMap.get("analysis_id");  
17     String scriptFileName = createScriptFileName("recalibration_mapper", key, analysisID);  
18     String logFileName = createLogFileName("recalibration_mapper", key, analysisID);  
19     File scriptFile = TemplateEngine.createDynamicContentAsFile("recalibration_mapper.template",  
20                                         templateMap,  
21                                         scriptFileName);  
22     if (scriptFile != null) {  
23         ShellScriptUtil.callProcess(scriptFileName, logFileName);  
24     }  
25 }
```

There will be only ONE reducer per <analysis\_id>.

#### Listing 18.10: Recalibration Mapper

```
1 // key: analysisID  
2 // values: ignored  
3 reduce(key, Iterable<Object> values) {  
4     DNASEq.recalibrationReducer(key);
```

```
5     emit(key, key);
6 }
```

The recalibrationReducer() method is defined below:

#### Listing 18.11: recalibrationReducer() Method

```
1 public static void recalibrationReducer(String analysisID)
2     throws Exception {
3     TemplateEngine.init();
4     String[] tokens = valueAsString.split(",");
5     Map<String, String> templateMap = new HashMap<String, String>();
6     templateMap.put("key", "-"); // key is undefined
7     templateMap.put("analysis_id", key);
8     // create the actual script from a template file
9     String scriptFileName = createScriptFileName("recalibration_reducer", analysisID);
10    String logFileName = createLogFileName("recalibration_reducer", analysisID);
11    File scriptFile = TemplateEngine.createDynamicContentAsFile("recalibration_reducer.template",
12                                                               templateMap,
13                                                               scriptFileName);
14    if (scriptFile != null) {
15        ShellScriptUtil.callProcess(scriptFileName, logFileName);
16    }
17 }
```

Recalibration templates are defined below:

#### Listing 18.12: Recalibration Mapper Template

```
1#!/bin/bash
2
3###
4### Recalibration Mapper Template
5###
6### call.snp (get variants) up to calculation of ...recal.table.csv
7### once recal.table.csv is created, it will be saved in HDFS
8### input file: aligned bam file (partitioned from a chr<i>.bam)
9...
10##
11## input file: aligned bam file (partitioned from a chr<i>.bam)
12## copy HDFS_BAM_FILE to LOCAL_BAM_FILE
13HDFS_BAM_FILE=${hdbs_bam_file}
14BAM_FILE=`basename $HDFS_BAM_FILE`;
15$HADOOP_HOME/bin/hadoop fs -copyToLocal $HDFS_BAM_FILE .
16...
17#
18# put 4.recal.table.csv into GLOBAL/SHARED directory
19#
20export SHARED_RECAL_DIR=/dnaseq/recal/${analysis_id}
21...
22## marking duplicates
23java -Xmx4g\
```

```

24      -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
25      -jar $PICARD_JAR/MarkDuplicates.jar \
26      I=$BAM_FILE \
27      O=2.mark.out.bam \
28      M=2.mark.out.metrics \
29      AS=true
30
31 ## local realignment
32 samtools index 2.mark.out.bam
33 java      -Xmx4g \
34      -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
35      -jar $GATK_JAR/GenomeAnalysisTK.jar \
36      -T IndelRealigner \
37      -I 2.mark.out.bam \
38      -o 3.realigned.out.bam \
39      -R $REF \
40      -targetIntervals $DBSNP/dbsnp_indel.intervals \
41      -known $DBSNP/dbsnp_indel.vcf \
42      --consensusDeterminationModel KNOWNS_ONLY \
43      -LOD 0.4
44 ## base quality recalibration
45 java      -Xmx4g \
46      -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
47      -jar $GATK_JAR/GenomeAnalysisTK.jar \
48      -T CountCovariates \
49      -I 3.realigned.out.bam \
50      -recalFile 4.recal.table.csv \
51      -R $REF \
52      -knownSites $DBSNP/dbsnp.vcf \
53      -cov QualityScoreCovariate \
54      -cov ReadGroupCovariate \
55      -cov PositionCovariate \
56      -cov DinucCovariate
57 # copy result to shared directory
58 cp -f 4.recal.table.csv $SHARED_RECAL_DIR/$KEY.4.recal.table.csv
59
60 ##
61 ## we need also to save: 3.realigned.out.bam (which will be
62 ## needed in variant_detection_mapper.sh.template
63 ##
64 cp -f 3.realigned.out.bam $SHARED_RECAL_DIR/$BAM_FILE.3.realigned.out.bam
65 ###
66 ### so we will have:
67 ###   $SHARED_RECAL_DIR/$KEY.4.recal.table.csv      for KEY=1, 2, 3, ....
68 ###   $SHARED_RECAL_DIR/$BAM_FILE.3.realigned.out.bam for KEY=1, 2, 3, ....
69 ###   (will be input to variant_detection_mapper.sh.template)
70 ###

```

---

**Listing 18.13:** Recalibration Reducer Template

```

1#!/bin/bash
2###
3### Merge all -.4.recal.table.csv files (generated by individual .bam files)
4### into a single recal.table.merged.final.txt file.

```

```

5 ####
6 ### Once recal.table.merged.final.txt is created, it will be saved in
7 ### /dnaseq/recal/{analysis_id}/ and will be fed into VariantDetectionMapper.
8 ####
9 ...
10 #
11 # All -4.recal.table.csv files are in $SHARED_RECAL_DIR directory
12 #
13 export SHARED_RECAL_DIR=/dnaseq/recal/$ANALYSIS_ID/
14 recal_files='find $SHARED_RECAL_DIR -name *.4.recal.table.csv' | sort'
15 num_of_recal_files='find $SHARED_RECAL_DIR -name *.4.recal.table.csv' | wc -l'
16 ...
17 ### NOTE: all calculations will take place at $SHARED_RECAL_DIR
18 # prepare java input files
19 java_input_files=""
20 for file in $recal_files
21 do
22     echo "preparing java input file=$file"
23     java_input_files="$file $java_input_files"
24 done
25 ...
26 cd $SHARED_RECAL_DIR
27 current_dir=`pwd`
28 export MERGE_COVARIATES=JavaMergeCovariates
29 $JAVA_HOME/bin/java -Xms4g -Xmx12g $MERGE_COVARIATES -i "$java_input_files" -o recal.txt.unsorted
30 #
31 # sort the file accordingly
32 #
33 /bin/sort -t, -k 2,2n -k3,3n -k4,4 recal.txt.unsorted > recal.txt.sorted
34 #
35 # The recal.txt.sorted file will be used by the Variant Detection Mapper.

```

---

## 18.8 Step-3: DNA-Sequencing: Variant Detection

Variant Detection is the final step of DNA Sequencing. The goal of this step is to generate variants in VCF (variant call format – developed by 1000 Genomes Project) format. The `map()` function will use the BAM file (generated by the `map()` function of the recalibration step) and the final single "recalibration table" file. The `map()` function will use open-source tools (such as GATK<sup>4</sup> and SAM Tools) to generate partial variants (called raw BCF files). The reducer will concatenate (sort-merge) the raw BCF files to generate a single VCF file. Once VCF file is created, it can be used by many analytical algorithms such as "Allelic Frequency", "Family Analysis", and

---

<sup>4</sup><https://www.broadinstitute.org/gatk/>

”Cochran-Armitage Test”.

Variant Detection (also called SNP calling) is the process of finding bases in the NGS data that differ from the reference genome (such as hg18 or hg19 – these refer to the version of the human genome assembly and determines the version of the corresponding reference annotations – for details, see <http://hgdownload.cse.ucsc.edu/goldenPath/hg18/bigZips/>).

### 18.8.1 Variant Detection Mapper

The mapper accepts a chunked ”realigned bam” file and performs the following transformations on it:

- base quality recalibration
- variant calling and filtering

The bulk of work is done by calling DNaseq.`theVariantDetectionMapper()` method, which accepts the required parameters and creates a proper shell script from a given template. Finally it executes the shell script. The mapper for variant detection is provided below.

#### Listing 18.14: Variant Detection Mapper

```
1 // key: ignored, not used
2 // value: <counter><;><3.realigned.out.bam.<key>><;><ref_genome><;><analysis_id>
3 // index < 0 > < 1 > < 2 > < 3 >
4 // value example-1: 0001;/<dir>/realigned.out.bam.0001;hg19;208
5 // value example-2: 0007;/<dir>/realigned.out.bam.0007;hg19;208
6 // NOTE: THERE WILL BE ONE SINGLE REDUCER for variant detection:
7 // the key for output of reducer will be: <analysis_id>
8 map(key, value) {
9     Map<String, String> tokens = DNaseq.tokenizeTheVariantDetectionMapper(value);
10    String reducerKey = tokens.get("analysis_id");
11    DNaseq.theVariantDetectionMapper(tokens);
12    emit(reducerKey, reducerKey);
13 }
```

The `theVariantDetectionMapper()` method accepts `analysis_id` (which uniquely identifies all files dir directories for a specific DNA Seq. run).

#### Listing 18.15: theVariantDetectionMapper()

```

1 public static void theVariantDetectionMapper(Map<String, String> templateMap)
2     throws Exception {
3     TemplateEngine.init();
4     // create the actual script from a template file
5     String scriptFileName = "/dnaseq/variant_detection_mapper_" + templateMap.get("analysis_id") + "_" + templateMap.get("sample_id");
6     String logFileName =      "/dnaseq/variant_detection_mapper_" + templateMap.get("analysis_id") + "_" + templateMap.get("sample_id");
7     File scriptFile = TemplateEngine.createDynamicContentAsFile("variant_detection_mapper.template", templateMap, scriptFileName);
8     if (scriptFile != null) {
9         ShellScriptUtil.callProcess(scriptFileName, logFileName);
10    }
11 }

```

---

Portions of the `variant_detection_mapper.template` is provided below:

**Listing 18.16:** Variant Detection Mapper Template

```

1 cat variant_detection_mapper.template
2 #!/bin/bash
3 ...
4
5 # 1. perform base quality recalibration:
6 # GATK required that the BAM file extension has to .bam
7 samtools index $REALIGNED_OUT_BAM_FILE
8 #
9 java -Xmx4g \
10      -Djava.io.tmpdir=$JAVA_IO_TMPDIR \
11      -jar $GATK_JAR/GenomeAnalysisTK.jar \
12      -T TableRecalibration \
13      -I $REALIGNED_OUT_BAM_FILE \
14      -o 4.recal.out.bam \
15      -R $REF \
16      -recalFile $SHARED_RECAL_DIR/recal.table.merged.final.txt
17 ...
18 # 2. variant calling and filtering
19 samtools mpileup -Duf $REF -q 1 4.recal.out.bam | bcftools view -bvg - > $REALIGNED_OUT_BAM_FILE.raw.bcf

```

---

### 18.8.2 Variant Detection Reducer

There will be ONLY ONE REDUCER for all mappers. The reason for one reducer is that we will be merging values to create a single output: a VCF file. The reducer does only one thing: creates a VCF file.

**Listing 18.17:** Variant Detection Reducer

```

1 // key: <analysis_id>, which identifies all data uniquely
2 // values: ignored
3 reduce(key, values){

```

```
4     DNASEq.theVariantDetectionReducer(key);
5     emit(key, key);
6 }
```

### Listing 18.18: theVariantDetectionReducer()

```
1 public static void theVariantDetectionReducer(String analysisID)
2     throws Exception {
3     TemplateEngine.init();
4     Map<String, String> templateMap = new HashMap<String, String>();
5     templateMap.put("key", "_");
6     templateMap.put("analysis_id", analysisID);
7     // create the actual script from a template file
8     String scriptFileName = "/dnaseq/variant_detection_reducer_" + templateMap.get("analysis_id") + ".sh";
9     String logFileName = "/dnaseq/variant_detection_reducer_" + templateMap.get("analysis_id") + ".log";
10    File scriptFile = TemplateEngine.createDynamicContentAsFile("variant_detection_reducer.template", templateMap, scriptFileName);
11    if (scriptFile != null) {
12        ShellScriptUtil.callProcess(scriptFileName, logFileName);
13    }
14 }
```

Portions of the `variant_detection_mapper.template` is provided below:

### Listing 18.19: Variant Detection Reducer Template

```
1 #cat variant_detection_reducer.template
2 #!/bin/bash
3 ...
4 # call snp (get variants)
5 # concatenate all $KEY.raw.bcf files
6 #
7 FINAL_BCF_FILE=$FINAL_DIR/all.raw.bcf
8 VCF_FILE=$FINAL_DIR/var.flt.vcf
9 ...
10 ##
11 ## Concatenate BCF files. The input files are required to be
12 ## sorted and have identical samples appearing in the same order.
13 ##
14 ALL_BCF_FILES='find $RECAL_DIR/ -name \'*.raw.bcf\' | sort'
15 $BCFTOOLS cat $ALL_BCF_FILES > $FINAL_BCF_FILE
16 #
17 #begin bcftools & create final VCF file
18 $BCFTOOLS view $FINAL_BCF_FILE | $VCFUTILS varFilter > $VCF_FILE
```

# Chapter 19

## Cox Regression

### 19.1 Introduction to Survival Analysis using Cox Regression

In medical statistics, survival analysis is used to describe the effect on survival times of a continuous variable (such as [gene expression](#)). Cox proportional hazard regression is a very important and popular regression algorithm; its simplicity and no assumption about survival distribution provide relative risk for a unit change in the variable. For example, a unit change in the expression of a specific gene gives a 2-fold increase in relative risk. A simple example of Cox regression is given: do men and women have different risks of developing brain cancer based on drinking alcoholic beverages? By constructing a Cox Regression model, with alcohol usage (ounces drank per day) and gender entered as covariates, you can test hypotheses regarding the effects of gender and alcohol usage on time-to-onset for brain cancer.

What is a Cox Regression? A Cox Regression model is a statistical technique used to explore the relationship between the survival of a patient and several explanatory variables such as `time` and `censor`. Cox Regression model was developed by [Sir Professor David Cox](#). One important characteristic of Cox regression is that estimates relative rather than absolute risk and it does not assume any knowledge of absolute risk. By definition, Cox regression, which implements the proportional hazards model is designed for analysis of time until an event or time between events. Cox regression uses one or

more predictor variables (so called covariates – example is time variable), to predict a status (event such as survival) variable. For example, time is an example of covariate (time from diagnosis of illness until the event of death – called survival analysis).

Cox Proportional Hazard Model is the most popular model for survival analysis because of its simplicity and no assumption about survival distribution.

For details on Cox regression, see David Garson's book [10].

Cox Proportional Hazard Regression is used to describe the effect on survival times of a continuous variable (e.g., gene expression). For details on Cox Regression, refer to [http://en.wikipedia.org/wiki/Cox\\_regression](http://en.wikipedia.org/wiki/Cox_regression). The R's programming language (<http://www.r-project.org/>) implementation of Cox Regression is an industry-standard reliable implementation (called `coxph()` function) and we will use it in our MapReduce solution. Unfortunately, there is no good implementation of Cox Regression in Java.

Cox Regression is used extensively by clinical and medical fields. The following are some examples of Cox Regression in medical fields:

- Survival analysis using continuous variables such as gene expression or White Blood Count ([27])
- Combining Gene Signatures Improves Prediction of Breast Cancer Survival ([31]).

The goal of this chapter is to provide a two-phase MapReduce solution to Cox regression:

- The PHASE-1 prepares proper input for Cox Regression
- The PHASE-2 uses output of PHASE-1 and applies R's `coxph()` to solve Cox Regression (there is no proper Java package for Cox regression algorithms – R provides defacto standards for Cox regression)

## 19.2 Cox Model in a Nutshell

Cox Regression is a relatively complex time-based statistical function. The main function of Cox Regression is to build a predictive model for time-to-event data. In a nutshell, the Cox regression model produces a survival

function that predicts the probability that the event of interest has occurred at a given time  $t$  for given values of the predictor variables (for example, in lung cancer analysis predictor variables can be the number of cigarettes smoked by different genders – men and women). The Cox regression is estimated from observed subjects and events during some time period.

In a nutshell, Cox Proportional Hazard Model can be expressed as a model

$$h_i(t) = h_0(t) \exp(\beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_n x_{in})$$

where

- $(x_{i1}, x_{i2}, \dots, x_{in})$  are the predictor variables and time independent
- $(\beta_1, \beta_2, \dots, \beta_n)$  is a vector of regression parameters and are estimated by Cox regression
- $h_0(t)$  is the baseline hazard, which is an unspecified function, making it a semi-parametric model ( $h_0(t)$  depends on time but not the covariates)
- $\exp(\beta X)$  depends on the covariates but not time.
- The hazard ratio for two observations is independent of time  $t$ , which defines the **proportional hazards** property as:

$$\frac{h_i(t)}{h_j(t)} = \frac{h_0(t)e^{\theta_i}}{h_0(t)e^{\theta_j}} = \frac{e^{\theta_i}}{e^{\theta_j}}$$

We can divide the both sides of the equation by  $h_0(t)$  taking logarithms, we obtain:

$$\ln\left(\frac{h_i(t)}{h_0(t)}\right) = (\beta_1 x_{i1} + \beta_2 x_{i2} + \dots + \beta_n x_{in})$$

we call  $\frac{h_i(t)}{h_0(t)}$  the hazard ratio.

### 19.3 MapReduce Solution for Cox Regression

We implement Cox Regression by 2-phase MapReduce algorithms:

- **MapReduce PHASE-1:** Aggregate, Group By, and Generate data ready for calling R's `coxph()` function. The `coxph()`<sup>1</sup> function fits a Cox proportional hazards regression model.
- **MapReduce PHASE-2:** Use output of PHASE-1 and Call R's `coxph()`. Finally, analyze generated results.

### 19.3.1 Cox Regression Basic Terminology

To use, Cox Regression in the context of survival analysis, we need to understand several key definitions. We borrow these definitions from Michael Walker (<http://walkerbioscience.com/pdfs/Survival%20analysis.pdf>).

- **Event:** This is an application dependent concept. Examples are: death, disease recurrence, or recovery.
- **Time:** This parameter for sure plays an important role in Cox regression definition. Events are time based. For example, the time from the beginning of an observation period (e.g., surgery) to (a) an event, or (b) end of the study, or (c) loss of contact or (d) withdrawal from the study.
- **Censoring/Censored observation:** When a test subject does not have an event during the observation time, they are described as censored, meaning that we cannot observe what has happened to them subsequently.
- **Censored subject:** a censored subject may or may not have an event after the end of observation time.
- **Survivor function:**  $S(t)$  = the probability that a subject survives longer than time  $t$ .

---

<sup>1</sup>Fits a Cox proportional hazards regression model. Time dependent variables, time dependent strata, multiple events per subject, and other extensions are incorporated using the counting process formulation of Andersen and Gill. (source: <http://stat.ethz.ch/R-manual/R-patched/library/survival/html/coxph.html>)

## 19.4 Cox Regression by using R Language

The R programming language provides an implementation of Cox Proportional Hazards Regression Model by *coxph{survival}*.

### Survival data

```
>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)
> censor=c(1,1,0,1,1,1,0,1,1,0,1,0,1,1,0,0,1,1,1,1,1,1,0,1,1,0,0)
```

### Expression data

The data for a specific gene is expressed as a foldchange below.

```
>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
```

### Cox regression

The R's Surv() function accepts two parameters: Surv(time, censor), where

- time is a vector of event time

- censor is a vector of indicator (denoting if the event was observed or censored)

```
> 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
```

From coxph function, we are only interested in two values: `coef` =  $-0.0247$  and `pvalue` =  $0.758$ .

## 19.5 Problem Statement

We have a set of patients. Each patient has a set of biosets and each bioset has a set of genes (for example, genes in RNA Gene-Expression data type) and its associated values (so called foldchange). What is a bioset? Individually analyzed data signatures are referred to as "biosets." "Biosets" encompass data in the form of experimental sample comparisons (for transcriptomic, epigenetic, and copy number variation data), as well as genotype signatures (for GWAS and mutational data). A bioset is most commonly referred to as a "gene signature" (for details, see <sup>2</sup>).

The number of genes per bioset can be up to 60,000. This number can be differentiated by the type of bioset, for example, for methylation biosets, this can be up to 20,000 genes and for gene-expression type this can be up to 50,000 genes). The problem can be stated as such: given a set of biosets, time, and censor, find "coef" (coefficient) and "value" for all genes contained in all biosets. If we are doing the Cox Regression for a set of 100,000 samples, then we have to analyze  $100,000 \times 60,000 = 6$  billion data records. This is a good candidate for MapReduce.

---

<sup>2</sup>[http://en.wikipedia.org/wiki/Gene\\_signature](http://en.wikipedia.org/wiki/Gene_signature)

## 19.6 Cox Regression POJO Solution

To understand Cox Regression, we present a non-MapReduce solution. For each bioset data type ("gene expression," "copy number variation," "methylation") there are fixed number of genes, which are identified by geneIdList. The objective of algorithm is to create input like:

```
geneID_1 bioset1_value11 bioset2_value12, ..., biosetN_value1N  
geneID_2 bioset1_value21 bioset2_value22, ..., biosetN_value2N  
...  
geneID_M bioset1_valueM1 bioset2_valueM2, ..., biosetN_valueMN
```

and then to pass this input to R's *coxph()* function. For example, for copy number variation (cnv) type, we can express our algorithm as:

**Listing 19.1:** Cox Regression Algorithm without MapReduce

```
1 input: double[] time;  
2 input: int[] censor;  
3 input: List<Long> biosetIDs;  
4 input: List<Long> cnvGeneIdList; // pre built (key, value) database  
5 output: List<Tuple3<String, Double, Double>>  
6 // as List<Tuple3<geneID, coef, pvalue>>  
7 //  
8 // iterate through all genes  
9 for (geneID : cnvGeneIdList) {  
10    // there is a key-value store per geneID  
11    // where key is the biosetID and value is the gene value  
12    mapDB = getMapDB(geneID);  
13    int index = 0;  
14    double[] geneValues = new double[time.length];  
15    for (biosetID : biosetIDs) {  
16        double geneValue = mapDB.get(biosetID);  
17        geneValues[index++] = geneValue;  
18    }  
19  
20    // call R's Cox Regression as coxph() function  
21    double[] result = coxph(time, censor, geneValues);  
22    double coef = result[0];  
23    double pvalue = result[1];  
24    emit(geneID, (coef, pvalue));  
25 }
```

For example, if we have 5,000 biosets, then we want to calculate *coxph()* for all genes (up to 40,000) contained in these biosets. This will require 40,000 *coxph()* calls and each call will have three arrays of time[5,000], censor[5,000] and foldchange[5,000], where foldchange refers to gene values).

Running this algorithm will generate the following output:

```
geneID_1 coef_1 pvalue_1  
geneID_2 coef_2 pvalue_2  
...  
geneID_M coef_M pvalue_M
```

## Discussion of Cox Regression Solution without MapReduce

This solution does not use MapReduce. Each bioset has a set of geneIDs (it can be up to 40,000 gene IDs per bioset) and its associated value. To make this happen, we create a persistent (*key, value*) store for each gene: the database name is the geneID, *key* is the biosetID, and *value* is the value associated with geneID. For persistent store, we use MapDB<sup>3</sup>.

This is an efficient solution when the number of biosets are in the hundreds. When the number of biosets are in the thousands, MapReduce/Hadoop solution will scale much better than the non-MapReduce solution.

## 19.7 Input for MapReduce

Typically, a patient has a set of biosets (and each bioset can be a different type defined by copy number variation, methylation, gene-expression). Each bioset has thousands of records and each record contains a "gene-id" and "value: associated with that "gene-id."

For example, a biosetID 8800 has the following data:

```
$ head 8800.csv  
  
13972,r1;2.45  
4082,r1;1.8  
40583,r1;1.8  
16422,r1;1.8
```

---

<sup>3</sup>MapDB (<https://github.com/jankotek/MapDB>) provides concurrent TreeMap and HashMap backed by disk storage or off-heap-memory. It is a fast, scalable and easy to use embedded Java database engine. It is tiny (160KB jar), yet packed with features such as transactions, space efficient serialization, instance cache and transparent compression/encryption. It also has outstanding performance rivaled only by native embedded db engines. MapDB is developed by Jan Kotek (<https://github.com/jankotek/>).

```
21602,r1;1.8  
45735,r1;1.8  
43936,r1;1.8  
26446,r1;1.8  
16030,r1;-3  
828,r1;0
```

where each row/record represents gene-id-and-ref (for first row, 13972 is the gene-id, r1 is a normal reference and 2.45 is an associated foldchange (gene-value) value. The reference can be `r1 = normal`, `r2 = disease`, `r3 = paired`, `r4 = unknown`. Given that the order of values per gene-id is very important in Cox Regression (`coxph()` function call – it is a time and event-based algorithm), we generate another set of data from existing data to include the biosetID as well. Below is our desired input for biosetID of 8800 to be used in MapReduce/Hadoop implementation.

```
$ 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
```

## Input Format for MapReduce

Each bioset record will have the following format:

Record Format:

```
<geneID><,><referenceID><;><biosetID><,><geneValue>
```

where

```
referenceID = r1 (normal)  
referenceID = r2 (disease)
```

```
referenceID = r3 (paired)
referenceID = r4 (unknown)
```

Example for biosetID of 8800:  
13972,r1;8800,2.45

Note that we used two different delimiters (";" and ",") in our data, which will help us in tokenizing data in MapReduce's `t map()` and `t reduce()` functions.

## 19.8 Cox Regression by MapReduce

We implemented Cox Regression in 2-phase MapReduce algorithms:

- **MapReduce PHASE-1:** Generate data ready for calling `coxph()` function
  - **map() PHASE-1:** Group data by <geneID><referenceID>
  - **reduce() PHASE-1:** Per reducer (key of <geneID><referenceID>), generate <geneID><referenceID> followed by all values of gene value in proper order of biosets (order has to be preserved, otherwise `coxph()` call will be meaningless).
- **MapReduce PHASE-2:** Call `coxph()` and analyze generated results
  - **map() PHASE-2:** Call `coxph()` for data generated by `t reduce()` phase of Phase1.
  - **reduce() PHASE-2:** NONE

### 19.8.1 Cox Regression PHASE-1: `map()`

The mapper is an identity mapper. It accepts a (key, value) pairs and emits the same (key, value) pairs.

**Listing 19.2:** Cox Regression Algorithm PHASE-1: `map()`

```

1 /**
2 * Each input record represents a (key, value) pair
3 * @param key is: <geneID>,><referenceID>
4 * @param value is: <biosetID>,><geneValue>
5 */
6 map(key, value) {
7     String[] tokens =StringUtil.split(value, ",");
8     String reducerKey = tokens[0]; // <geneID>,><referenceID>
9     String reducerValue = tokens[1]; // <biosetID>,><geneValue>
10    emit(reducerKey, reducerValue);
11 }

```

---

### 19.8.2 Cox Regression PHASE-1: reduce()

Each reducer accepts a (key, values) pair, where key is a pair of (geneID, referenceID) and values is a list of pair of (biosetID, geneValue). The reducer orders geneValue(s) according to the order of biosets received at the setup() of the reducer class. For example if values = {(B1, G1), (B3, G3), (B4, G4), (B2, G2)} and biosets are received as (B1, B2, B3, B4), then reducer will generate sorted gene values as (G1, G2, G3, G4).

**Listing 19.3:** Cox Regression Algorithm PHASE-1: reduce()

```

1 /**
2 * @param key key is the <geneID><referenceID>
3 * @param values is a list of {<biosetID>,><geneValue>}
4 * NOTE: bioset IDs (List<Long>) will be passed from MapReduce Driver and
5 * will be saved at the reducer's setup() function (done once before reduce() starts)
6 */
7 reduce(key, values) {
8     String[] doubleValues = new String[biosets.size()];
9     int numberOfValues = 0;
10    for (pair : values) {
11        String[] tokens = StringUtil.split(pair, ",");
12        String biosetID = tokens[0];
13        String geneValue = tokens[1];
14
15        int index = biosets.indexOf(biosetID);
16        if (index == -1) {
17            // biosetID not found
18            return;
19        }
20
21        // biosetID found at location "index"
22        doublevalues[index] = geneValue;
23        numberOfValues++;
24    }
25
26    // values are candidate for cox regression analysis

```

```

27     if (numberOfValues != biosets.size()) {
28         // there are not enough gene values for these biosets,
29         // can not perform cox regression
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         // check to see if we need to add comma
40         if (i < (doublevalues.length -1)) {
41             builder.append(",");
42         }
43     }
44
45     // prepare reducer for output
46     String reducerValue = builder.toString();
47     // reducerValue = <geneID>,><referenceID>, value1, value2, ..., valueN
48     context.write(null, reducerValue);
49 }
```

---

### 19.8.3 Cox Regression PHASE-2: map()

The mapper accepts an HDFS input file and then executes *coxph()* with this input. Output is then transferred to HDFS for final analysis. For this mapper, we do need a *setup()* function which will be executed once.

**Listing 19.4:** Cox Regression Algorithm PHASE-2: *setup()*

```

1 /**
2 * map(key) = LongWritable (generated by hadoop, ignored here)
3 * map(value) = Text (name of a HDFS file: /biomarker/output/rnae/0/part-r-00000)
4 *               which will be used as an input to cox regression analysis
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    // will be run only once
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    // make sure /hadoop/home/cox_regression.r.template file does exist
31    // in local unix file system, if it does not exist then copy it from
32    // hdfs:/biomarker/template/cox_regression.r.template
33    //
34    if (IOUtil.fileExists(CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME)) {
35        THE_LOGGER.info("template file does exist: " +
36        CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME);
37    }
38    else {
39        // copy it from HDFS
40        copyHDFSFileToLocal(CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_HDFS_PATH,
41                            CoxRegressionUsingR.COX_REGRESSION_TEMPLATE_FILE_FULL_NAME);
42    }
43}
44
45 // map() defined next

```

---

### Listing 19.5: Cox Regression Algorithm PHASE-2: map()

```

1 /**
2 * @param key key is the key generated by MapReduce framework (not used here)
3 * @param value is the name of HDFS input file generated by reduce() of PHASE-1
4 * (for example: HDFS file like: </biomarker/output/rnae/0><,><part-r-00019>)
5 * NOTE: the following variables (timeAsCommaSeparatedString, censorAsCommaSeparatedString,
6 * and.hadoopOutputPathAsString) are initialized in reducer's setup() function (done once).
7 */
8 map(key, value) {
9     String coxRegressionInputFileName = "/tmp/" + HadoopUtil.getRandomUUID();
10    // coxRegressionInputFileName = /tmp/a99817a0-c149-4cb2-a771-dbc7da86b56a
11    String coxRegressionOutputFileName = coxRegressionInputFileName + ".out.txt";
12    File coxRegressionInputFile = new File(coxRegressionInputFileName);
13
14    Path hdfsPartFile = new Path(valueAsString);
15    FileUtil.copy(fs,                                // FileSystem
16                  hdfsPartFile,                      // src file
17                  coxRegressionInputFile,           // destination
18                  false,                          // boolean deleteSource,
19                  conf);
20
21    int coxCallStatus = -1; // failure
22    // perform Cox Regression
23    coxCallStatus = CoxRegressionUsingR.callCoxRegression(
24        coxRegressionInputFileName,
25        coxRegressionOutputFileName,

```

```

26         timeAsCommaSeparatedString,
27         censorAsCommaSeparatedString);
28
29     if (coxCallStatus == 0) {
30         // it is success! then copy coxRegressionOutputFileName to HDFS (to hadoopOutputPathAsString)
31         File coxRegressionOutputFile = new File(coxRegressionOutputFileName);
32         if (coxRegressionOutputFile.exists()) {
33             FileUtil.copy(coxRegressionOutputFile, fs, new Path(hadoopOutputPathAsString), false, conf);
34         }
35     }
36
37     // send it to reducer (indication, we are done for logging purposes)
38     context.write(key, value);
39 }
```

---

#### 19.8.4 Sample Output Generated by PHASE-2 reduce()

Output files generated by reduce() of PHASE-1:

```
$ hadoop fs -ls /biomarker/output/rnae/0/part*
Found 30 items
-rw-r--r-- 3 hadoop hadoop 1047918 ... /biomarker/output/rnae/0/part-r-00000
-rw-r--r-- 3 hadoop hadoop 1053398 ... /biomarker/output/rnae/0/part-r-00001
-rw-r--r-- 3 hadoop hadoop 1022627 ... /biomarker/output/rnae/0/part-r-00002
...
-rw-r--r-- 3 hadoop hadoop 1026116 ... /biomarker/output/rnae/0/part-r-00025
-rw-r--r-- 3 hadoop hadoop 1030567 ... /biomarker/output/rnae/0/part-r-00026
-rw-r--r-- 3 hadoop hadoop 1028483 ... /biomarker/output/rnae/0/part-r-00027
```

If our test example has 5000 biosets, then we have:

```
[hadoop@hnnode01337 ~]$ 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>
```

### 19.8.5 Sample Output Generated by PHASE-2 map()

The first column is <geneID\_and\_rreference>}, the second column is `coef`, and the last column is `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
```

### 19.8.6 Cox Regression by MapReduce: How Does It Work

The `CoxRegressionUsingR` class has the core functionality for Cox regression. This class creates an `Rscript` from a template file. An instance of a `Rscript` is given below:

**Listing 19.6:** Cox Regression Algorithm by Rscript

```
1 #!/usr/local/bin/Rscript
2
3 input_file = "/tmp/f11d10f4-e0a6-4796-9a1c-34c6914be1e7"
4 cat("input_file=", input_file, "\n")
5 output_file = "/tmp/f11d10f4-e0a6-4796-9a1c-34c6914be1e7.out.txt"
6 cat("output_file=", output_file, "\n")
7
8 library("survival")
9
10 # a function for conversion
11 convert=function(x){return(log2(max(x, -1/x)))}
12
13 # a function to perform Cox regression
14 cox_regression <- function(line){
15     #cat(line, "\n")
16     # each line has this format:
17     # <geneID><;><V1,V2,...,Vn><;><T1,T2,...,Tn><;><C1,C2,...,Cn>
18     items = unlist(strsplit(line, ","))
19     #
20     geneID = items[[1]]
21     #
22     value = as.double(unlist(strsplit(items[[2]], ",")))
23     #
```

```

24     time = as.double(unlist(strsplit(items[[3]], ",")))
25     #
26     censor = as.double(unlist(strsplit(items[[4]], ",")))
27     #
28     coxphoutput = coxph(Surv(time,censor) ~ value)
29     pvalue = summary(coxphoutput)$waldtest[3]
30     cat(geneID, coxphoutput$coef, pvalue, "\n", file=output_file, append=TRUE)
31 }
32
33 # driver section of Rscript
34 conn <- file(input_file, open="r")
35 while(length(line <- readLines(conn, 1)) > 0) {
36   try.output <- try( cox_regression(line) )
37 }
38 close.connection(conn)

```

---

Rscript code description is given below:

**Lines 34-37** This is the driver for the Rscript. It opens the file, reads line-by-line, and applies the `cox_regression` function. Finally, it closes the input file (to release the resources).

**Lines 14-31** Defines our custom function called `cox_regression()`, which calls R's `coxph()` function. Line 18 tokenizes the input record. Finally, `coxph()` produces "coef" and "pvalue".

**Line 30** Appends geneID, "coef" and "pvalue" to output file.

Sample output is given below:

```
$ 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
```

# Chapter 20

## Cochran-Armitage Test for Trend

### 20.1 Introduction

Cochran-Armitage Test for Trend (CATT) is used in analyzing germline<sup>1</sup> data. For example, variants expressed as a VCF (Variant Call Format) and generated by DNA-Sequencing can be labeled as a germline data. The CATT is a statistical method of directing chi squared tests toward narrow alternatives. Let  $R$  be a set of response variables and  $E$  be a set of experimental variables, then the CATT is sensitive to the linearity between  $R(s)$  and  $E(s)$  and detects trends. The CATT can be expressed another way: let  $B$  be a binary outcome of some events {PASSED, FAILED} and  $C$  be a set of ordered categories  $\{C_1, \dots, C_n\}$ , then CATT can be used as a linear trend in proportions on  $B$  across levels of  $C$ . To apply CATT, we build a contingency table: 2 rows with outcome values {PASSED, FAILED} and  $n$  columns as  $\{C_1, \dots, C_n\}$ . The contingency table for CATT is explained in the next sections.

According to Wikipedia<sup>2</sup>: "The Cochran-Armitage test for trend, named for William Cochran and Peter Armitage, is used in categorical data analysis when the aim is to assess for the presence of an association between a variable with two categories and a variable with  $k$  categories. It modifies the Pearson chi-squared test to incorporate a suspected ordering in the effects of the  $k$  categories of the second variable. For example, doses of a treatment can be

---

<sup>1</sup><https://en.wikipedia.org/wiki/Germline>

<sup>2</sup> [https://en.wikipedia.org/wiki/Cochran%20%80%93Armitage\\_test\\_for\\_trend](https://en.wikipedia.org/wiki/Cochran%20%80%93Armitage_test_for_trend)

ordered as 'low', 'medium', and 'high', and we may suspect that the treatment benefit cannot become smaller as the dose increases. The trend test is often used as a genotype-based test for case-control genetic association studies.<sup>3</sup> And according to Wellek and Ziegler<sup>3</sup>: "The Cochran-Armitage trend test based on the linear regression model has become a standard procedure for association testing in case-control studies."

## 20.2 Cochran-Armitage Algorithm

The CATT is applied when the data take the form of a  $2 \times k$  contingency table. The number of rows (2) indicate outcome of an experiment and the number of columns indicate variable number ( $k$ ) of experiments. For example, if  $k = 3$ , then the contingency will be:

Group \ Count	$B = 1$	$B = 2$	$B = 3$
$A = 1$	$N_{11}$	$N_{12}$	$N_{13}$
$A = 2$	$N_{21}$	$N_{22}$	$N_{23}$

This contingency table can be completed with the marginal totals of the two variables:

Group \ Count	$B = 1$	$B = 2$	$B = 3$	Sum
$A = 1$	$N_{11}$	$N_{12}$	$N_{13}$	$R_1$
$A = 2$	$N_{21}$	$N_{22}$	$N_{23}$	$R_2$
Sum	$C_1$	$C_2$	$C_3$	$N$

where

- $R_1 = N_{11} + N_{12} + N_{13}$
- $R_2 = N_{21} + N_{22} + N_{23}$
- $C_1 = N_{11} + N_{21}$

---

<sup>3</sup> Cochran-Armitage test versus logistic regression in the analysis of genetic association studies. Wellek S, Ziegler A. Department of Biostatistics, Central Institute of Mental Health Mannheim, University of Heidelberg, Mannheim, Germany.

- $C_2 = N_{12} + N_{22}$
- $C_3 = N_{13} + N_{23}$
- $N = R_1 + R_2 = C_1 + C_2 + C_3 = N_{11} + N_{12} + N_{13} + N_{21} + N_{22} + N_{23}$

The trend test statistic is

$$T \equiv \sum_{i=1}^k w_i(N_{1i}R_2 - N_{2i}R_1)$$

where the  $w_i$  are weights.

The hypothesis of no association (known as the null hypothesis) can be expressed as:

$$\Pr(A = 1|B = 1) = \dots = \Pr(A = 1|B = k)$$

Assuming that the null hypothesis holds, then, using iterated expectation we can write:

$$E(T) = E(E(T|R_1, R_2)) = E(0) = 0$$

Given two discrete random variables X and Y, you can define the conditional expectation as:

$$E[X|Y = y] = \sum_x x \cdot P(X = x|Y = y)$$

Now, using all these definitions and formulas, we are ready to write Cochran-Armitage Algorithm in Java. One major goal of CATT is compute the p-value (a probability value between 0.00 and 1.00). Using the algorithm defined in Wikipedia, we implement CATT as a POJO class `CochranArmitage`. This Java class will be used in our MapReduce solution.

**Listing 20.1:** Cochran-Armitage Algorithm

```

1 import java.io.BufferedReader;
2 import java.io.BufferedWriter;
3 import java.io.FileReader;
4 import java.io.FileWriter;
5 import java.io.IOException;
6
```

```

7 import org.apache.log4j.Logger;
8 import org.apache.commons.math3.distribution.NormalDistribution;
9
10 /**
11 * Class that calculates the Cochran-Armitage test for trend
12 * on a 2x3 contingency table. Used to estimate association
13 * in additive genetic models of genotype data.
14 */
15 public class CochranArmitage {
16
17     private static final Logger THE_LOGGER = Logger.getLogger(CochranArmitage.class);
18
19     // use weights corresponding to additive/codominant model
20     private static final int[] WEIGHTS = { 0, 1, 2 };
21
22     // dimensions of passed contingency table - must be 2 rows x 3 columns
23     private static final int NUMBER_OF_ROWS = 2;
24     private static final int NUMBER_OF_COLUMNS = 3;
25
26     // variables to hold variance, raw statistic, standardized statistic, and p-value
27     private double stat = 0.0;
28     private double standardStatistics = 0.0;
29     private double variance = 0.0;
30     private double pValue = -1.0; // range is 0.0 to 1.0 (-1.0 means undefined)
31
32     // NormalDistribution class from Apache used to calculate p-values
33     private static NormalDistribution normDist = new NormalDistribution();
34
35     /**
36      * Get the variance
37      */
38     public double getVariance() {
39         return variance;
40     }
41
42     /**
43      * Get the Stat
44      */
45     public double getStat() {
46         return stat;
47     }
48
49     /**
50      * Get the StandardStatistics
51      */
52     public double getStandardStatistics() {
53         return standardStatistics;
54     }
55
56     /**
57      * Get the pvalue
58      */
59     public double getPValue() {
60         return pValue;
61     }
62

```

```

63  /**
64   * Computes the Cochran-Armitage test for trend for the passed
65   * 2 row by 3 column contingency table
66   * @param countTable 2x3 contingency table
67   * @return the p-value of the Cochran-Armitage statistic of the passed table
68   */
69  public double callCochranArmitageTest(int[][] countTable) {
70      // defined in next sections ...
71  }
72
73
74  /**
75   * @param args input/output files for testing/debugging
76   * args[0] as input file
77   * args[1] as output file
78   */
79  public static void main(String[] args) throws IOException {
80      // defined in next sections ...
81  }
82
83 }

```

---

The following method is the core of the Cochran-Armitage Algorithm:

**Listing 20.2: Cochran-Armitage Algorithm: callCochranArmitageTest()**

```

1 /**
2  * Computes the Cochran-Armitage test for trend for the passed
3  * 2 row by 3 column contingency table
4  * @param countTable 2x3 contingency table
5  * @return the p-value of the Cochran-Armitage statistic of the passed table
6  */
7 public double callCochranArmitageTest(int[][] countTable) {
8
9     if (countTable == null) {
10         throw new IllegalArgumentException("contingency table cannot be null/empty.");
11     }
12
13     if ( (countTable.length != NUMBER_OF_ROWS) || (countTable[0].length != NUMBER_OF_COLUMNS) ) {
14         throw new IllegalArgumentException("contingency table must be 2 rows by 3 columns");
15     }
16
17     int totalSum=0;
18     int[] rowSum = new int[NUMBER_OF_ROWS];
19     int[] colSum = new int[NUMBER_OF_COLUMNS];
20
21     // calculate marginal and overall sums for the contingency table
22     for (int i=0; i<NUMBER_OF_ROWS; i++) {
23         for (int j=0; j<NUMBER_OF_COLUMNS; j++) {
24             rowSum[i] += countTable[i][j];
25             colSum[j] += countTable[i][j];
26             totalSum += countTable[i][j];
27         }

```

```

28     }
29
30     // calculate the test statistic and variance based on the formulae at
31     // http://en.wikipedia.org/wiki/CochranArmitage_test_for_trend
32     stat = 0.0;
33     variance = 0.0;
34     for (int j=0; j<NUMBER_OF_COLUMNS; j++) {
35         stat += WEIGHTS[j] * (countTable[0][j]*rowSum[1] - countTable[1][j]*rowSum[0]);
36         variance += WEIGHTS[j]*WEIGHTS[j]*colSum[j]*(totalSum-colSum[j]);
37
38         if (j!=NUMBER_OF_COLUMNS-1) {
39             for (int k=j+1;k<NUMBER_OF_COLUMNS;k++) {
40                 variance -= 2*WEIGHTS[j]*WEIGHTS[k]*colSum[j]*colSum[k];
41             }
42         }
43     }
44     variance *= rowSum[0]*rowSum[1]/totalSum;
45
46     // standardized statistic is stat divided by SD
47     standardStatistics = stat/Math.sqrt(variance);
48
49     // use Apache commons normal distribution to calculate two tailed p-value
50     pValue = 2*normDist.cumulativeProbability(-Math.abs(standardStatistics));
51
52     // return the p-value
53     return pValue;
54 }
```

The following program tests the Cochran-Armitage Algorithm:

**Listing 20.3: Cochran-Armitage Algorithm: main()**

```

1 /**
2  * @param args input/output files for testing/debugging
3  * args[0] as input file
4  * args[1] as output file
5 */
6 public static void main(String[] args) throws IOException {
7     if (args.length != 2) {
8         THE_LOGGER.info("usage: java CochranArmitage <input-filename> <output-filename>");
9         throw new IOException("must provide input and output files for testing.");
10    }
11
12    long startTime = System.currentTimeMillis();
13    String inputFileName = args[0];
14    String outputFileName = args[1];
15    BufferedWriter outfile = new BufferedWriter(new FileWriter(outputFileName));
16    outfile.write("score\tp-value\n");
17    BufferedReader infile = new BufferedReader(new FileReader(inputFileName));
18
19    int[][] countTable = new int[2][3];
20    String line = null;
21    while ((line = infile.readLine()) != null) {
```

```

22     String[] tokens = line.split("\t");
23     int index=0;
24
25     // populate 2x3 contingency table
26     for(int i=0; i<2; i++) {
27         for(int j=0; j<3; j++) {
28             countTable[i][j] = Integer.parseInt(tokens[index++]);
29         }
30     }
31
32     CochranArmitage catest = new CochranArmitage();
33     double pValue = catest.callCochranArmitageTest(countTable);
34     outfile.write(String.format("%f\t%f\n", catest.getStandardStatistics(), pValue));
35 }
36
37 long elapsedTime = System.currentTimeMillis() - startTime;
38 THE_LOGGER.info("run time (in milliseconds): " + elapsedTime);
39
40 infile.close();
41 outfile.close();
42 }

```

---

The following is a sample run of the Cochran-Armitage Algorithm:

```

# Compile the algorithm
$ javac CochranArmitage.java

# Define input
$ cat test3.txt
1386  1565  401   1342  1579  434
2716   672    13   2689   695     9
2062  1144  151   2021  1184  173

# Run algorithm
$java CochranArmitage test3.txt test3.txt.out
Jul 13 2013 04:36:31 [main] [INFO ] [CochranArmitage] - run time (in milliseconds)

# Expected output
$ cat test3.txt.out
score      p-value
-1.414843  0.157114
-0.488857  0.624943
-1.555344  0.119864

```

## 20.3 Application of Cochran-Armitage

In genome analysis, the CATT is applied for statistical tests for difference in "Genotype Frequency". For differences in genotype frequency, each individual should be coded as  $\{0, 1, 2\}$  based on the number of the particular variant allele in that individual. These counts can be used to assemble a  $2 \times 3$  (2 rows and 3 columns: each row represent a specific group and each column represent outcome of an experiment such as an allele count) contingency table that can be analyzed by standard statistical methods. The CATT is used to approximate an additive genetic model.

The real example in genomic analysis can be stated as follows: let Group-A denotes a set of biosets (generated from VCF – variant call format – files) for a set of patients and let Group-B denotes another set of biosets (generated from VCF files) for another set of patients. The goal is to apply CATT to a set of alleles found at a specific chromosome<sup>4</sup> with a given "start position" and "stop position" (for details on chromosomes, you may visit National Institute of Health<sup>5</sup>). This data can be very huge. Each bioset generated from a (human sample) VCF file may have over 4000,000 chromosomes. If Group-A has 3,000 samples and Group-B has 5,000 samples to compare (genotype frequency), then we have to analyze 32 billion records (obviously this is a big data problem).

$$\text{Group-A records} = 3,000 * 4,000,000 = 12,000,000,000$$

$$\text{Group-B records} = 5,000 * 4,000,000 = 20,000,000,000$$

$$\text{Total records} = 12,000,000,000 + 20,000,000,000 = 32,000,000,000$$

To use the CATT for genotype frequency we form of a  $2 \times 3$  contingency table for each allele found at the common key of a specific chromosome (identified by its "start position" and "end position").

---

<sup>4</sup>Humans normally have 46 chromosomes in each cell, divided into 23 pairs. Two copies of chromosome 1, one copy inherited from each parent, form one of the pairs. (source: <http://ghr.nlm.nih.gov/chromosome/1>)

<sup>5</sup> <http://ghr.nlm.nih.gov/handbook/basics/chromosome>

Group	Allele Count	Count of 0	Count of 1	Count of 2
		$N_{11}$	$N_{12}$	$N_{13}$
Group-A		$N_{11}$	$N_{12}$	$N_{13}$
Group-B		$N_{21}$	$N_{22}$	$N_{23}$

I will demonstrate building a  $2 \times 3$  contingency table for each allele by the following example. Let Group-A be a set of 6 biosets identified by  $\{B_1, B_2, B_3, B_4, B_5, B_6\}$  and let Group-B be a set of 5 biosets identified by  $\{B_7, B_8, B_9, B_{10}, B_{11}\}$ . Note that these data are for a very specific chromosome at a defined position (identified by "chromosome start position" and "chromosome stop position")

Group-A Biosets		
Bioset ID	Allele1	Allele2
$B_1$	A	C
$B_2$	A	A
$B_3$	A	C
$B_4$	G	G
$B_5$	A	A
$B_6$	AC	T

Group-B Biosets		
Bioset ID	Allele1	Allele2
$B_7$	A	A
$B_8$	C	C
$B_9$	A	C
$B_{10}$	A	A
$B_{11}$	A	A

Before generating/building contingency tables, we do need to build some data structures:

Genotype Frequency Table						
Bioset ID	Group	"A" count	"C" count	"G" count	"T" count	"AC" count
$B_1$	Group-A	1	1	0	0	0
$B_2$	Group-A	2	0	0	0	0
$B_3$	Group-A	1	1	0	0	0
$B_4$	Group-A	0	0	2	0	0
$B_5$	Group-A	2	0	0	0	0
$B_6$	Group-A	0	0	0	1	1
$B_7$	Group-B	2	0	0	0	0
$B_8$	Group-B	0	2	0	0	0
$B_9$	Group-B	1	1	0	0	0
$B_{10}$	Group-B	2	0	0	0	0
$B_{11}$	Group-B	2	0	0	0	0

Now, we can generate a contingency table for each allele ("A", "C", "G", "T", and "AC"), which then we may apply the CATT algorithm. In MapReduce algorithm, each reducer for  $(Key_2, Value_2)$  will generate a set of contingency tables (where  $Key_2$  is a composite key of `chromosomeID:start:stop`).

Contingency Table for allele "A"			
Count Group	Count of 0	Count of 1	Count of 2
Group-A	2	2	2
Group-B	1	1	3

Contingency Table for allele "C"			
Count Group	Count of 0	Count of 1	Count of 2
Group-A	4	2	0
Group-B	3	1	1

Contingency Table for allele "G"			
Count Group	Count of 0	Count of 1	Count of 2
Group-A	5	0	1
Group-B	5	0	0

Contingency Table for allele "T"				
Group	Count	Count of 0	Count of 1	Count of 2
Group-A		5	1	0
Group-B		5	0	0

Contingency Table for allele "AC"				
Group	Count	Count of 0	Count of 1	Count of 2
Group-A		5	1	0
Group-B		5	0	0

## 20.4 MapReduce Solution

We present a MapReduce algorithm for CATT, which can be implemented by Hadoop and Spark. Our implementation is based on MapReduce/Hadoop.

### 20.4.1 Input

Since the same bioset can be selected for both group of biosets ( $A$  and  $B$ ), we will generate two types of data (the only difference will be the **GROUP-NAME**, for group  $A$ , **GROUP-NAME** will be  $a$  and for group  $B$ , **GROUP-NAME** will be  $b$ , this will enable us to identify one group from another). Therefore, **GROUP-NAME** can be in  $\{a, b\}$ .

Each bioset record will have the following format:

```

<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<;>
<GROUP-NAME>
<:>
<allele1>
<:>
```

```

<allele2>
<:>
<referece>
<:>
<snp-id>
<:>
<mutation-class-ID>
<:>
<gene-id>
<:>
<bioset_Id>

```

For example, if we select 6 biosets for group A, then we will have:

```

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

```

For example, if we select 5 biosets for group B, then we will have:

```

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

```

## 20.4.2 Expected Output

Each result record (p-value is generated by Cochran-Armitage Test) will have the following format:

```

<pValue> (result of CATT)
<:>
<chromosome-ID>
<:>
<chromosome-start-position>
<:>

```

```

<chromosome-stop-position>
<:>
<gene-id>
<:>
<mutation-class-ID>
<:>
<reference>
<:>
<alternate-allele>
<:>
<N11>
<:>
<N12>
<:>
<N13>
<:>
<N21>
<:>
<N22>
<:>
<N23>
<:>
<snp-id>

```

### 20.4.3 Mapper

The mapper will generate a  $(key, value)$  pair for each record, where  $key$  will be:

```
<chromosome-ID><:><chromosome-start-position><:><chromosome-stop-position>
```

and the  $value$  will be the remaining attributes:

```

<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>

```

```

<referece>
<:>
<snp-id>
<:>
<mutation-class-ID>
<:>
<gene-id>
<:>
<bioset_Id>

```

**Listing 20.4:** map() for Cochran-Armitage

```

1 /**
2 * @param key the key generated by hadoop, ignored here
3 * @param value has the following format:
4 *   <chr-ID><:><start><:><stop><:><GROUP></><allele1></><allele2></>
5 *   <ref></><snp-id></><mutation-class></><gene-id>/<bioset_Id>
6 *   where GROUP can be in {"a" , "b"}
7 *
8 * reducerKey = Text:  <chr-ID><:><start><:><stop>
9 * reducerValue = Text: <GROUP></><allele1></><allele2></><ref></>
10 *                      <snp-id></><mutation-class></><gene-id>/<bioset_Id>
11 */
12 map(Object key, Text value) {
13     String[] tokens = StringUtils.split(line, ";");
14     if (tokens.length == 2) {
15         String reducerKey = tokens[0];
16         String reducerValue = tokens[1];
17         emit(reducerKey, reducerValue);
18     }
19 }

```

#### 20.4.4 Reducer

The main task for CATT is done by reducers. Each reducer's key will be the key generated by mappers. The value of each reducer will a list of values generated by mappers. Therefore, a reducer which has a key of "7:10005296:10005296", will have the following values (by iteration of one by one):

```

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

```

For implementation of `reduce()`, we do need 3 data structures:

```

// all alleles in Group-A and Group-B
Map<String, String[]> allAlleles = new HashMap<String, String[]>();

// all alleles in Group-A
List<String[]> groupA = new ArrayList<String[]>();

// all alleles in Group-B
List<String[]> groupB = new ArrayList<String[]>();

```

After all these 3 data structures are populated, we will generate contingency tables and then apply Cochran-Armitage trend test.

#### Listing 20.5: `reduce()` for Cochran-Armitage

```

1 /**
2  * @param key the key: Text: <chr-ID><:><start><:><stop>
3  * @param values list of {value}, where each value has the following format:
4  *   <GROUP><:><allele1><:><allele2><:><ref><:><snp-id><:><mutation-class><:><gene-id><:><bioset_Id>
5  *   where GROUP can be in {"a", "b"}
6 *
7  * outputKey = null
8  * outputValue = built by buildOutputValue() method
9 */
10 reduce(Text key, Iterable<Text> values) {
11     // tokens[index]          0      1      2      3      4
12     // reduce(value) = Text: <GROUP><:><allele1><:><allele2><:><ref><:><snp-id><:>
13     // tokens[index]          5      6      7
14     // reduce(value) = Text: <mutation-class><:><gene-id><:><bioset_Id>
15
16     Map<String, String[]> allAlleles = new HashMap<String, String[]>();
17     List<String[]> groupA = new ArrayList<String[]>();
18     List<String[]> groupB = new ArrayList<String[]>();
19
20     for (Text valueAsText : values) {

```

```

21     String value = valueAsText.toString();
22     String[] tokens = StringUtils.split(value, " | ");
23     if (tokens.length != 8) {
24         continue;
25     }
26
27     String group = tokens[0];
28     String allele1 = tokens[1];
29     String allele2 = tokens[2];
30     if (!allAlleles.containsKey(allele1)) {
31         allAlleles.put(allele1, tokens);
32     }
33     if (!allAlleles.containsKey(allele2)) {
34         allAlleles.put(allele2, tokens);
35     }
36
37     if (group.equals("a")) {
38         // it is either group "a"
39         groupA.add(tokens);
40     }
41     else {
42         // or it is group "b"
43         groupB.add(tokens);
44     }
45 } // end-for
46
47 if ( (groupA.isEmpty()) & (groupB.isEmpty()) ){
48     return;
49 }
50
51 // iterate through allAlleles and do analysis
52 int[][] contingencyTable = new int[2][3]; // create a contingency table
53 for (Map.Entry<String, String[]> entry : allAlleles.entrySet()) {
54     String allele = entry.getKey();           // key
55     String[] tokens = entry.getValue();        // value
56
57     // -----
58     // create the following table:
59     //
60     //      0      1      2      -- counts
61     //      -----
62     // groupA    n10    n11    n12    (index 0 is groupA)
63     // groupB    n20    n21    n22    (index 1 is groupB)
64     //
65     // The pass this array of 2x3 to CochranArmitage Test
66     // -----
67
68     clearContingencyTable(contingencyTable);
69     fillContingencyTable(groupA, groupB, contingencyTable);
70     double pValue = CochranArmitage.callTrendTest(contingencyTable);
71     String outputValue = buildOutputValue(pValue, key, allele, tokens, contingencyTable);
72     // prepare reducer for output
73     emit(null, outputValue);
74 }
75 }

```

### Listing 20.6: Helper Functions for Cochran-Armitage

```
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         // here count = 0, 1, or 2
15         contingencyTable[0][count]++;
16     }
17
18     for (String[] tokensB: groupB) {
19         int count = countAlleles(tokensB, allele);
20         // here count = 0, 1, or 2
21         contingencyTable[1][count]++;
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     // here is count = 0, 1, or 2
34     return count;
35 }
```

The following method builds final output to be emitted by reducers.

### Listing 20.7: buildOutputValue() for Cochran-Armitage

```
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>/<biostudy_Id>
5     // tokens[index]   5      6      7
6     StringBuilder outputValue = new StringBuilder();
7     outputValue.append(pValue);           // pValue
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);             // alternate allele
18   outputValue.append(":");
19   outputValue.append(contingencyTable[0][0]); // n10 (group A, count 0)
20   outputValue.append(":");
21   outputValue.append(contingencyTable[0][1]); // n11 (group A, count 1)
22   outputValue.append(":");
23   outputValue.append(contingencyTable[0][2]); // n12 (group A, count 2)
24   outputValue.append(":");
25   outputValue.append(contingencyTable[1][0]); // n20 (group B, count 0)
26   outputValue.append(":");
27   outputValue.append(contingencyTable[1][1]); // n21 (group B, count 1)
28   outputValue.append(":");
29   outputValue.append(contingencyTable[1][2]); // n22 (group B, count 2)
30   outputValue.append(":");
31   outputValue.append(tokens[4]);           // snp-id
32   return outputValue.toString();
33 }

```

---

## 20.5 MapReduce/Hadoop Implementation

The Hadoop implementation for CATT is comprised of the following classes:

Hadoop Implementation Classes	
Class Name	Class Description
CochranArmitage.java	Cochran-Armitage Trend test Algorithm
CochranArmitageAnalyzer.java	Read and analyze the result generated by MapReduce program
CochranArmitageClient.java	Simple class to submit MapReduce jobs
CochranArmitageDriver.java	Actual Driver to submit MapReduce jobs
CochranArmitageItem.java	Output record is mapped into this bean class
CochranArmitageItemFactory.java	Factory class to generate CochranArmitageItem object
CochranArmitageItemImpl.java	Implementation of CochranArmitageItem
CochranArmitageMapper.java	Hadoop map() for Cochran-Armitage Trend test Algorithm
CochranArmitageReducer.java	Hadoop reduce() for Cochran-Armitage Trend test Algorithm
PaginatedObject.java	How to paginate result for front-end
ResultObject.java	Result object for Cochran-Armitage Trend Test Algorithm

## 20.5.1 Sample Run of MapReduce/Hadoop Implementation

### 20.5.1.1 Input

```
$ cat run_CochranArmitage.sh
#!/bin/bash
client=CochranArmitageClient
java $client interactive Oout 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

```

### 20.5.1.2 Sample Run

```

$ ./run_CochranArmitage.sh
Jul 18 2013 16:34:40 [main] [INFO ] [CochranArmitageClient] - executionType: inter
Jul 18 2013 16:34:40 [main] [INFO ] [CochranArmitageClient] - requestID: Oout
Jul 18 2013 16:34:40 [main] [INFO ] [CochranArmitageClient] - biosetIdFilenameGro
Jul 18 2013 16:34:40 [main] [INFO ] [CochranArmitageClient] - biosetIdFilenameGro
...
Jul 18 2013 16:34:41 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - Running
Jul 18 2013 16:34:42 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 0%
Jul 18 2013 16:34:58 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100%
Jul 18 2013 16:35:10 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100%
Jul 18 2013 16:35:11 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100%
Jul 18 2013 16:35:12 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100%
Jul 18 2013 16:35:13 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100%
Jul 18 2013 16:35:18 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - Job com
...
Jul 18 2013 16:35:18 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] -      Map
Jul 18 2013 16:35:18 [main] [INFO ] [CochranArmitageDriver] - run(): jobSucceeded=
Jul 18 2013 16:35:18 [main] [INFO ] [CochranArmitageDriver] - run(): Job Finished
Jul 18 2013 16:35:18 [main] [INFO ] [CochranArmitageDriver] - run(): jobid=job_201
Jul 18 2013 16:35:18 [main] [INFO ] [CochranArmitageDriver] - submitJob(): runStat

```

```
Jul 18 2013 16:35:18 [main] [INFO ] [CochranArmitageClient] - main() jobid=job_201
```

### 20.5.1.3 Generated Output

```
$ 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
```

# Chapter 21

## Allelic Frequency

### 21.1 Introduction

Allelic Frequency is a technique used to find frequency of alleles for genomic<sup>1</sup> data (especially for germline<sup>2</sup> data type). An "allelic frequency"<sup>3</sup> is defined as "the percentage of a population of a species that carries a particular allele on a given chromosome locus." After all genomic data is aggregated per desired key (comprised of [chromosome, start-position, stop-position]) then Fisher's Exact Test is applied. Fisher's Exact Test is a statistical test used to determine if there are non-random associations between two groups of variables (these two groups of variables can be patient's biosets, which will be discussed shortly). We will then analyze and plot the output of the MapReduce program. The input for Allelic Frequency comes from VCF files, which are generated by DNA-Sequencing pipelines. Typically each VCF record includes: chromosome, start-position, stop-position, genome-reference, and two alleles (labeled as allele1 and allele2 – one from mother and one from father). These information will be sufficient to perform an allelic frequency for two sets of data.

The main goal of this chapter is to present a MapReduce solution, comprised of three MapReduce jobs, to Allelic Frequency using Fisher's Exact Test<sup>4</sup>.

<sup>1</sup><http://en.wikipedia.org/wiki/Genomics>

<sup>2</sup><http://en.wikipedia.org/wiki/Germline>

<sup>3</sup><http://www.thefreedictionary.com/allele+frequency>

<sup>4</sup>[http://en.wikipedia.org/wiki/Fisher%27s\\_exact\\_test](http://en.wikipedia.org/wiki/Fisher%27s_exact_test)

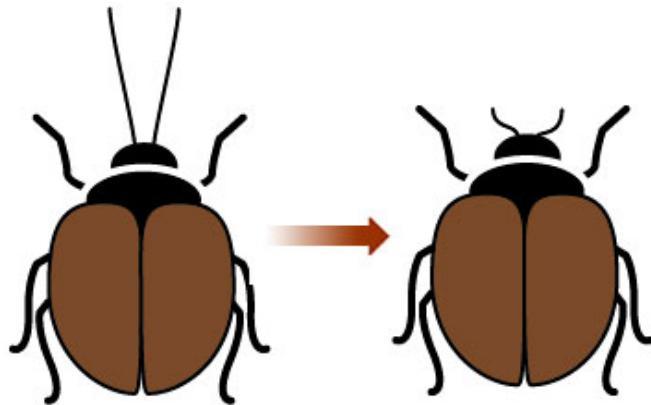


Figure 21.1: Mutation Example

To comprehend the importance and the impact of Allelic Frequency, it is better to understand the meaning of "mutations", "migrations", and "selections". For details on these concepts, see <sup>5</sup> and <sup>6</sup>. Calculating an allelic frequency is very important for biology scientists: if over time, there are changes in allele frequencies then this can implicate that genetic drift is occurring or that new mutations have been introduced into the population.

Example of a mutation<sup>7</sup> is illustrated below: the image shows that how do populations change genetically away from the Hardy-Weinberg equilibrium.

In this chapter we will:

1. Find the p-value (probability value in the range of 0.00 to 1.00) for every triplet of (**chromosome**, **start**, **stop**) between two given groups (A and B) of biosets (defined below). This will be accomplished by a MapReduce Phase-1. For efficiency of run-time processing, for a given bioset, we create two identical copies (they differ just by groupID value) of the original data by adding a groupID as "a" and "b"

---

<sup>5</sup><http://163.16.28.248/bio/activelearner/18/ch18c3.html>

<sup>6</sup><http://www.ndsu.edu/pubweb/~mcclean/plsc431/popgen/popgen4.htm>

<sup>7</sup>source of image is from <http://163.16.28.248/bio/activelearner/18/images/ch18c3.jpg>

2. Find the first smallest 100 p-values `among all chromosomes`, which are closest to 0.00 — as p-values get closer to 0.00, they become interesting ones). This means that if we sort all p-values in ascending order, then our desired solution will be the first 100 p-values. This is implemented by another MapReduce Phase-2 program. The output of Phase-1 will be used as input to Phase-2.
3. Find the first smallest-100 p-values `per chromosome`, which are closest to 0.00. This is implemented by another MapReduce Phase-3 program. The output of Phase-1 will be used as input to Phase-3.

## 21.2 Basic Definitions

### 21.2.1 Chromosome

A chromosome is an organized structure of DNA, protein, and RNA found in cells. Human cells have 23 pairs of chromosomes labeled as  $\{1, 2, \dots, 22, X, Y\}$ . For details, see [Wikipedia: Chromosome](#).

### 21.2.2 Bioset

We have a set of patients. Each patient has a set of biosets (biosets are created directly from VCF files – each record of VCF file contains information about a position in the genome) and each bioset has a set of genes. For example, a bioset may have genes in RNA Gene-Expression data type, and its associated values or "fold change." What is a bioset? Individually analyzed data signatures are referred to as "biosets." "Biosets" encompass data in the form of experimental sample comparisons (for transcriptomic, epigenetic, and copy number variation data), as well as genotype signatures (for GWAS and mutational data). A bioset most commonly referred to as a "gene signature." A sample record of a bioset will contain a chromosome, its start and stop positions, two alleles, and other related information. A bioset has an associated data type. A bioset data type can be "gene-expression," "protein-expression," "methylation," "copy-number-variation," "miRNA," or "somatic mutation." The number of entries/records per bioset does depend on the data type of a bioset. For example, a germline bioset can have 4.3 million records while a gene-expression bioset can have up to 60,000 records.

### 21.2.3 Allele and Allelic Frequency

What is an "allele" and "allelic frequency"? An allele<sup>8</sup> is a viable DNA (deoxyribonucleic acid) coding that occupies a given locus (position) on a chromosome. There are two alleles per chromosome position and they are called `allele1` and `allele2`. An **allelic frequency** is defined as "the percentage of a population of a species that carries a particular allele on a given chromosome locus." Alternatively, "allele frequency" can be defined as the frequency of an allele relative to that of other alleles of the same gene in a population. Typically, the Fisher's Exact Test is used to calculate the "pvalue" (probability value) for Allelic Frequency. The Fisher's Exact Test uses a  $2 \times 2$  contingency table, which represents how different treatments have produced different outcomes. Furthermore, the Fisher's Exact Test is based on "null hypothesis," which assumes that treatments do not affect outcomes, that the two are independent.

### 21.2.4 Source of Data for Allelic Frequency

Shotgun sequencing machines generate DNA-Sequence data (in FASTQ<sup>9</sup> and other well known formats) from biological (including human) samples. Then these FASTQ data (size of these can be up to 200GB) are analyzed by DNA sequencing<sup>10</sup> software and workflows, which eventually generate VCF (Variant Call Format)<sup>11</sup> files. VCF is the result of DNA sequencing on germline data. VCF has a text file format and partitioned into meta-data (what kind of data it represents, quality of data, ...) and data. Once a VCF file is available, then we can create biosets and biomarkers (which are used in further analysis of genomes contained in groups of biosets/biomarkers). One such analysis is an "Allelic Frequency", which we will focus in this chapter. A workflow diagram for Allelic Frequency is presented below.

Allelic Frequency for germline data is a big data problem: each germline bioset has about 4.3 millions records (genomic information, such as chromosome ID, chromosome start position, chromosome end position, reference

---

<sup>8</sup><http://www.sciencedaily.com/articles/a/allele.htm>

<sup>9</sup>[http://en.wikipedia.org/wiki/FASTQ\\_format](http://en.wikipedia.org/wiki/FASTQ_format)

<sup>10</sup>[https://en.wikipedia.org/wiki/DNA\\_sequencing](https://en.wikipedia.org/wiki/DNA_sequencing)

<sup>11</sup>[http://en.wikipedia.org/wiki/Variant\\_Call\\_Format](http://en.wikipedia.org/wiki/Variant_Call_Format)

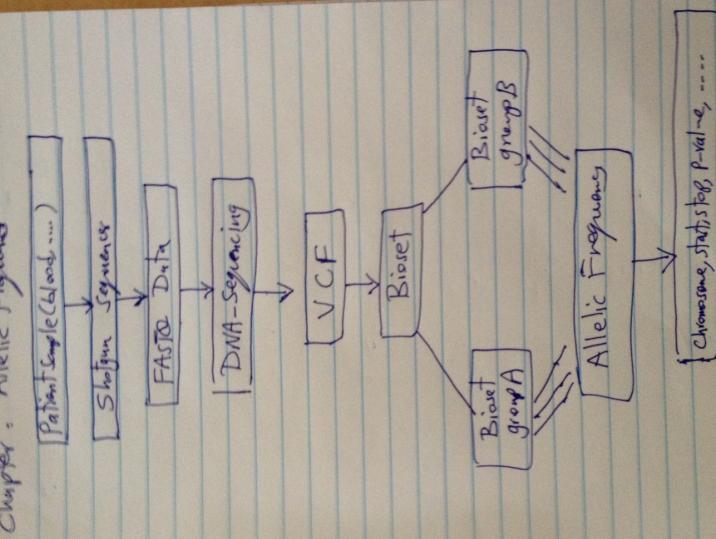


Figure 21.2: Allelic Frequency WorkFlow

genome, allele1, and allele2, ...). Imagine we have two group of germline biosets: A and B. If group A has 2000 biosets and group B has 4000 biosets and we want to find an "Allelic Frequency" of "group A biosets" with "group B biosets", then we have to analyze

$$4,300,000 \times (2000 + 4000) = 25,800,000,000$$

records and on top of it we have to do "Allelic Frequency" analysis (such as Fisher's Exact Test) for all alleles (allele1, and allele2) in groups A and B. With 15 nodes Hadoop cluster, "allele frequency" analysis takes about 20 minutes and with 100 nodes this might take just under 4 minutes.

### 21.2.5 Allelic Frequency Analysis by Fisher's Exact Test

Typically, after we find the Allelic Frequency for a given chromosome (identified by its chromosome ID, start position, and end stop position), we apply Fisher's Exact Test to find the `pvalue` where ( $0.0 \leq pvalue \leq 1.0$ ). As `pvalue` is getting closer to 1.0, it indicates that the association between rows (groups) and columns (outcomes) is considered to be not statistically significant; and as `pvalue` is moving closer to 0.0, it indicates that the association between rows (groups) and columns (outcomes) is statistically significant.

### 21.2.6 Fisher's Exact Test

What is the Fisher's Exact Test? Fisher's<sup>12</sup> exact test is "A statistical test of independence much used in medical research. It tests the independence of rows and columns in a  $2 \times 2$  contingency table (with 2 horizontal rows crossing 2 vertical columns creating 4 places for data) based on the exact sampling distribution of the observed frequencies. Hence it is an *exact* test." The Fisher's Exact Test was devised by Sir Ronald Aylmer Fisher<sup>13</sup>.

Typically, Fisher's test is conducted for two groups (Group-1, Group-2) with two outcomes (Outcome-1, Outcome-2). Actual data values are shown in blue color.

---

<sup>12</sup><http://www.medterms.com/script/main/art.asp?articlekey=3469>

<sup>13</sup>[http://en.wikipedia.org/wiki/Ronald\\_Fisher](http://en.wikipedia.org/wiki/Ronald_Fisher)

	<i>Outcome-1</i>	<i>Outcome-2</i>	Row Totals
<i>Group-1</i>	a	b	$a + b$
<i>Group-2</i>	c	d	$c + d$
Column Totals	$a + c$	$b + d$	$n = a+b+c+d$

Fisher's Exact Test formula is given below:

$$p = \frac{\binom{a+b}{a} \binom{c+d}{c}}{\binom{n}{a+c}}$$

$$= \frac{(a+b)!}{a!} \frac{(c+d)!}{b!} \frac{(a+c)!}{c!} \frac{(b+d)!}{d! n!}$$

where  $\binom{n}{k}$  is a Binomial coefficient and  $n!$  is a factorial of  $n$ .

What do we determine with Fisher's Exact Test? "Fisher's exact test can be used to statistically determine whether there is any association between two categorical variables with two levels each. The p-value for this test is the probability that the observed data or more extreme data occur under the null hypothesis of no difference between the two levels." (source: [http://rfd.uoregon.edu/files/rfd/StatisticalResources/lec\\_05a.txt](http://rfd.uoregon.edu/files/rfd/StatisticalResources/lec_05a.txt)).

Below we provide two simple examples:

- An example of  $2 \times 2$  contingency table is given below:

	<i>Success</i>	<i>Failure</i>	Row Totals
<i>Group-1</i>	4	12	16
<i>Group-2</i>	8	10	18
Column Totals	12	22	$n = 34$

Fisher's exact test: the two-tailed p-value equals to 0.2966. The association between rows (groups) and columns (outcomes) is considered to be not statistically significant. A p-value can be calculated with either one or two tails. It is recommended to use two-tailed (also called two-sided) P values (for details on one-tail vs. two-tail P values see [2]).

For example, in R programming language, we can perform Fisher's Exact Test as:

```
# 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
```

- An example of  $2 \times 2$  contingency table is given below:

	<i>Success</i>	<i>Failure</i>	Row Totals
<i>Group-1</i>	12	2	14
<i>Group-2</i>	2	10	12
Column Totals	14	12	n = 26

Fisher's exact test: the two-tailed p-value equals 0.0011. The association between rows (groups) and columns (outcomes) is considered to be very statistically significant.

## 21.3 Formal Problem Statement

Let  $A$  (denoted by groupID of "a") and  $B$  (denoted by groupID of "b") be a set of biosets for germline data (these biosets are created from VCF files generated by DNA-Sequencing). Size of set  $A$  is  $m$  and size of set  $B$  is  $n$ . Note that  $m$  and  $n$  can be the same or different sizes). Given  $A$  and  $B$ , we want to find the Allele Frequency for these two groups by using Fisher's Exact Test. Each bioset record represents a variant and will be identified by the following attributes. Note that a bioset record may have over 50 attributes, but for Allele Frequency the following attributes will be sufficient):

- Bioset's Key Attributes

- Chromosome ID: 1, 2, 3, ...
- Chromosome Start position
- Chromosome Stop position
- Bioset's Value Attributes
  - Group ID: "a", "b"
  - Allele1
  - Allele2
  - Reference
  - SNP ID
  - Mutation Class ID
  - Gene ID
  - Bioset ID

Therefore, we need to implement:

1. Find the p-value for every triplet of (chromosomeID, start, stop) between two given groups of A and B. This is solved by [MapReduce-Phase-1](#).
2. Find the first 100 p-values, which are closest to 0.00 — as p-values get closer to 0.00, they become interesting ones). This means that if we sort all p-values in ascending order, then our desired solution will be the first 100 p-values. This is solved by [MapReduce-Phase-2](#). Output of Phase-1 will be an input to Phase-2.
3. Find the first smallest 100 p-values per chromosome, which are closest to 0.00. This is solved by [MapReduce-Phase-3](#). Output of Phase-1 will be an input to Phase-3.

## 21.4 MapReduce Phase-1

This phase finds the p-value for every triplet of (chromosome, start, stop) between two given groups (A and B) of biosets. The map() function will group input records by key of (chromosome, start, stop). The reduce() function

will construct a  $2 \times 2$  contingency table and finally calls the Fisher's Exact Test. It is not possible provide a `combine()` function, since we need all alleles per reducer.

### 21.4.1 Input

Since the same bioset can be selected for both sets ( $A$  and  $B$ ), we will generate two types of data (the only difference will be the `group-name`, for group  $A$ , `group-name` will be `a` and for group  $B$ , `group-name` will be `b`, this will enable us to identify one group from another).

Each bioset record will have the following format:

```
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<;>
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<referece>
<:>
<snp-id>
<:>
<mutation-class-ID>
<:>
<gene-id>
<:>
<bioset_Id>
```

For example, if we select 6 biosets for group A, then for `chromosome 7` we might have the following data:

```
7:10005296:10005296;a:A:C:A:snpid:mc:geneid:1000
7:10005296:10005296;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
```

For example, if we select 5 biosets for group B, then for `chromosome 7` we might have the following data:

```
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
```

#### 21.4.2 Output/Result

Each output/result record (`pValue` is generated by Fisher's Exact Test per `chromosome-ID`, `chromosome-start-position`, and `chromosome-stop-position`) will have the following format:

```

<pValue> (result of Fisher's Exact Test)
<:>
<chromosome-ID>
<:>
<chromosome-start-position>
<:>
<chromosome-stop-position>
<:>
<gene-id>
<:>
<mutation-class-ID>
<:>
<reference>
<:>
<alternate-allele>
<:>
<N11>
<:>
<N12>
<:>
<N21>
<:>
<N22>
<:>
<snp-id>

```

Note that  $N_{11}$ ,  $N_{12}$ ,  $N_{21}$ ,  $N_{22}$  refers to the values that we generate as a  $2 \times 2$  contingency table shown below:

	<i>Known</i>	<i>Others</i>	Row Totals
<i>Group-A</i>	$N_{11}$	$N_{12}$	$N_{11} + N_{12}$
<i>Group-B</i>	$N_{21}$	$N_{22}$	$N_{21} + N_{22}$
Column Totals	$N_{11} + N_{21}$	$N_{12} + N_{22}$	$n = N_{11} + N_{12} + N_{21} + N_{22}$

Allelic Frequency analysis by Fisher's Exact Test will generate the following output:

```

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

```

### 21.4.3 MapReduce Solution for Allelec Frequency

We implement Allelec Frequency by 3-phase MapReduce algorithms:

- **MapReduce Phase-1:** Aggregate, Group By, and Generate p-values by calling Fisher's Exact Test
- **MapReduce Phase-2:** Use output of Phase-1 and find the bottom 100 smallest p-values among all chromosomes. We have designed this algorithm in such a way that we will be able to find bottom-N smallest for any N — for example, bottom 50 or bottom 500).
- **MapReduce Phase-3:** Find the first smallest-100 p-values per chromosome which are closest to 0.00. For example, what is the smallest-100 p-values for chromosome 5?

### 21.4.4 Phase-1 Mapper

The mapper will generate a  $(key, value)$  pair for each record, where  $key$  will be:

```
<chromosome-ID><:><chromosome-start-position><:><chromosome-stop-position>
```

and the  $value$  will be the remaining attributes:

```
<GROUP-NAME>
<:>
<allele1>
<:>
<allele2>
<:>
<reference>
<:>
<snpID>
<:>
<mutationClassID>
<:>
<geneID>
<:>
<biosetID>
```

**Listing 21.1:** map() for Allelec Frequency

```

1 /**
2  * @param key the key generated by hadoop, ignored here
3  * @param value has the following format:
4  *   <chrID><:><start><:><stop><;><GROUP></><allele1></><allele2></><ref></>
5  *   <snpID></><mutationClass></><geneID>/<biosetID>
6  * where GROUP can be in {"a" , "b"}
7  *
8  * reducerKey = Text: <chrID><:><start><:><stop>
9  * reducerValue = Text: <GROUP></><allele1></><allele2></><ref></>
10    <snpID></><mutationClass></><geneID>/<biosetID>
11 */
12 map(Text key, Text value) {
13     String[] tokens = StringUtils.split(line, ";");
14     if (tokens.length == 2) {
15         String reducerKey = tokens[0];
16         String reducerValue = tokens[1];
17         emit(reducerKey, reducerValue);
18     }
19 }
```

---

The mapper function generates (key, value) pairs, where key is comprised of { chrID, start, stop } and value is comprised of { GROUP, allele1, allele2, ref, snpID, mutationClass, geneID, biosetID } . Therefore, each reducer will receive a key as <chrID><:><start><:><stop> and values (as a list of mappers's emitted values).

#### 21.4.5 Phase-1 Reducer

The main task is done by reducers. Each reducer's key will be the key generated by mappers. The value of each reducer will be a list of values generated by mappers. Therefore, a reducer which has a key of "7:10005296:10005296" will have the following values (by iteration of one by one):

```

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: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

```

In our implementation of a reducer, we do need 3 basic hash tables:

- groupA: alleles, which are only in group A

```
Map<String, Integer> groupA = new HashMap<String, Integer>();
```

- groupB: alleles, which are only in group B

```
Map<String, Integer> groupB = new HashMap<String, Integer>();
```

- globalMap: alleles, which are group A or group B

```
Map<String, String[]> globalMap = new HashMap<String, String[]>();
```

After all these 3 tables are populated per key (as <chrID><:><start><:><stop>), we will iterate globalMap to find all Allele Frequencies.

### **Listing 21.2: reduce() for Allele Frequency**

```

1 /**
2  * @param key the key: Text: <chrID><:><start><:><stop>
3  * @param values list of {value}, where each value has the following format:
4  *   <GROUP><:><allele1><:><allele2><:><ref><:><snpID><:>
5  *   <mutationClass><:><geneID>:<biosetID>
6  *   where GROUP can be in {"a" , "b"}
7  *
8  * outputKey = null (no key is required)
9  * outputValue = built by buildOutputValue() method
10 */
11 reduce(Text key, Iterable<Text> values) {

```

```

12
13     int totalNumOfBiosetsInGroupA = 0;
14     int totalNumOfBiosetsInGroupB = 0;
15
16     Map<String, String[]> globalMap = new HashMap<String, String[]>();
17     Map<String, Integer> groupA = new HashMap<String, Integer>();
18     Map<String, Integer> groupB = new HashMap<String, Integer>();
19
20     <STEP-1: POPULATE-TABLES-BY-ITERATING-VALUES>
21
22     // STEP-2: create an instance of the Fisher's Exact Test
23     FisherExactTest theFisherExactTest = new FisherExactTest();
24
25     <STEP-3: FIND-ALLELIC-FREQUENCIES-BY-ITERATING-TABLES-BUILT>
26 }
```

---

**Listing 21.3:** STEP-1: POPULATE-TABLES-BY-ITERATING-VALUES

```

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) = <mutationClass><:><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          // it is group "a"
17          totalNumOfBiosetsInGroupA++;
18          // handle 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     // handle 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     // handle allele1 for group "b"
39     totalNumOfBiosetsInGroupB++;
40     Integer count = groupB.get(allele1);
41     if (count == null) {
42         groupB.put(allele1, 1);
43         // check to see if we want to add this allele or not
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     // handle allele2
52     Integer count2 = groupB.get(allele2);
53     if (count2 == null) {
54         groupB.put(allele2, 1);
55         // check to see if we want to add this allele or not
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 } // end-while
66
67 if ( (groupA.size() == 0) || (groupB.size() == 0) ){
68     return;
69 }

```

---

**Listing 21.4:** STEP-3: FIND-ALLELIC-FREQUENCIES-BY-ITERATING-TABLES-BUILT

```

1   // now all of our 3 required hash tables are created:
2   //      groupA<String, Integer>
3   //      groupB<String, Integer>
4   //      globalMap<allele, tokens[]>
5   //
6   // next, iterate through globalMap<allele, tokens[]> and do analysis
7   // note that globalMap<allele, tokens[]> has all alleles
8   for (Map.Entry<String, String[]> entry : globalMap.entrySet()) {
9       String allele = entry.getKey(); // key
10      String[] tokens = entry.getValue();
11
12      int n11 = 0;
13      int n12 = 0;
14      int n21 = 0;
15      int n22 = 0;
16
17      // check to see if groupA has this allele:
18      Integer countOfAllelesInGroupA = groupA.get(allele);
19      Integer countOfAllelesInGroupB = groupB.get(allele);
20      if (countOfAllelesInGroupA == null) {

```

```

21         // then this allele must be only in groupB
22         n11 = 0;
23         n12 = (2 * totalNumOfBiosetsInGroupA);
24         n21 = countOfAllelesInGroupB;
25         n22 = (2 * totalNumOfBiosetsInGroupB) - n21;
26     }
27     else if (countOfAllelesInGroupB == null) {
28         // then this allele must be only in groupA
29         n11 = countOfAllelesInGroupA;
30         n12 = (2 * totalNumOfBiosetsInGroupA) - n11;
31         n21 = 0;
32         n22 = (2 * totalNumOfBiosetsInGroupB);
33     }
34     else {
35         // then this allele must be in both (groupA and groupB)
36         n11 = countOfAllelesInGroupA;
37         n12 = (2 * totalNumOfBiosetsInGroupA) - n11;
38         n21 = countOfAllelesInGroupB;
39         n22 = (2 * totalNumOfBiosetsInGroupB) - n21;
40     }
41
42     theFisherExactTest.init(n11, n12, n21, n22);
43     double pValue = theFisherExactTest.get2Tail();
44     String outputValue = buildOutputValue(pValue, key, allele, tokens, n11, n12, n21, n22);
45     // prepare reducer for output
46     emit(null, outputValue);
47 } // end for-loop
48 }

```

---

**Listing 21.5:** buildOutputValue() for Allele Frequency

```

1 String buildOutputValue(pValue, key, allele, tokens, n11, n12, n21, n22) {
2     StringBuilder outputValue = new StringBuilder();
3     outputValue.append(pValue);           // pValue
4     outputValue.append(":");
5     outputValue.append(key.toString()); // key = <chrID><:><start><:><stop>
6     outputValue.append(":");

```

```

7     outputValue.append(tokens[6]);           // <geneID>
8     outputValue.append(":");
9     outputValue.append(tokens[5]);           // <mutationClass>
10    outputValue.append(":");
11    outputValue.append(tokens[3]);           // <ref>
12    outputValue.append(":");
13    outputValue.append(allele);             // alternate allele
14    outputValue.append(":");
15    outputValue.append(n11);                // n11
16    outputValue.append(":");
17    outputValue.append(n12);                // n12
18    outputValue.append(":");
19    outputValue.append(n21);                // n21
20    outputValue.append(":");
21    outputValue.append(n22);                // n22
22    outputValue.append(":");
23    outputValue.append(tokens[4]);           // snpID
24    return outputValue.toString();
25 }

```

---

## 21.4.6 Sample Run of MapReduce/Hadoop Implementation

### 21.4.6.1 The Script

```

# 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

```

### 21.4.6.2 Input

```
# 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
```

### 21.4.6.3 Running The Script

```
# . env.sh
JAVA_HOME=/usr/java/jdk6
# ./run_test.sh
Jul 18 2013 16:01:50 [main] [INFO ] [AllelicFrequencyClient] - executionType: interactive
Jul 18 2013 16:01:50 [main] [INFO ] [AllelicFrequencyClient] - requestID: 0
Jul 18 2013 16:01:50 [main] [INFO ] [AllelicFrequencyClient] - biosetIDsFilenameGroupA: groupA.txt
Jul 18 2013 16:01:50 [main] [INFO ] [AllelicFrequencyClient] - biosetIDsFilenameGroupB: groupB.txt
...
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - GroupA biosetID: 0
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - GroupA biosetPath::/bioset/input/groupA::
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - GroupB biosetID: 0
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - GroupB biosetPath::/bioset/input/groupB::
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - hadoopOutputPathAsString: /bioset/output
Jul 18 2013 16:01:51 [main] [INFO ] [AllelicFrequencyDriver] - run() wait=wait
...
Jul 18 2013 16:01:51 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - Running job: job_201307110717_0278
Jul 18 2013 16:01:52 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 0% reduce 0%
Jul 18 2013 16:02:06 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100% reduce 0%
Jul 18 2013 16:02:16 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100% reduce 6%
...
Jul 18 2013 16:03:07 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - map 100% reduce 100%
Jul 18 2013 16:03:12 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - Job complete: job_201307110717_0278
...
Jul 18 2013 16:03:12 [main] [INFO ] [org.apache.hadoop.mapred.JobClient] - Map output records=11
Jul 18 2013 16:03:12 [main] [INFO ] [AllelicFrequencyDriver] - run(): jobSucceeded=true
Jul 18 2013 16:03:12 [main] [INFO ] [AllelicFrequencyDriver] - run(): Job Finished in 80.862 seconds
Jul 18 2013 16:03:12 [main] [INFO ] [AllelicFrequencyDriver] - run(): jobid=job_201307110717_0278
Jul 18 2013 16:03:12 [main] [INFO ] [AllelicFrequencyDriver] - submitJob(): runStatus=0
Jul 18 2013 16:03:12 [main] [INFO ] [AllelicFrequencyClient] - main() jobid=job_201307110717_0278
```

#### 21.4.6.4 Generated Output

```
# 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
```

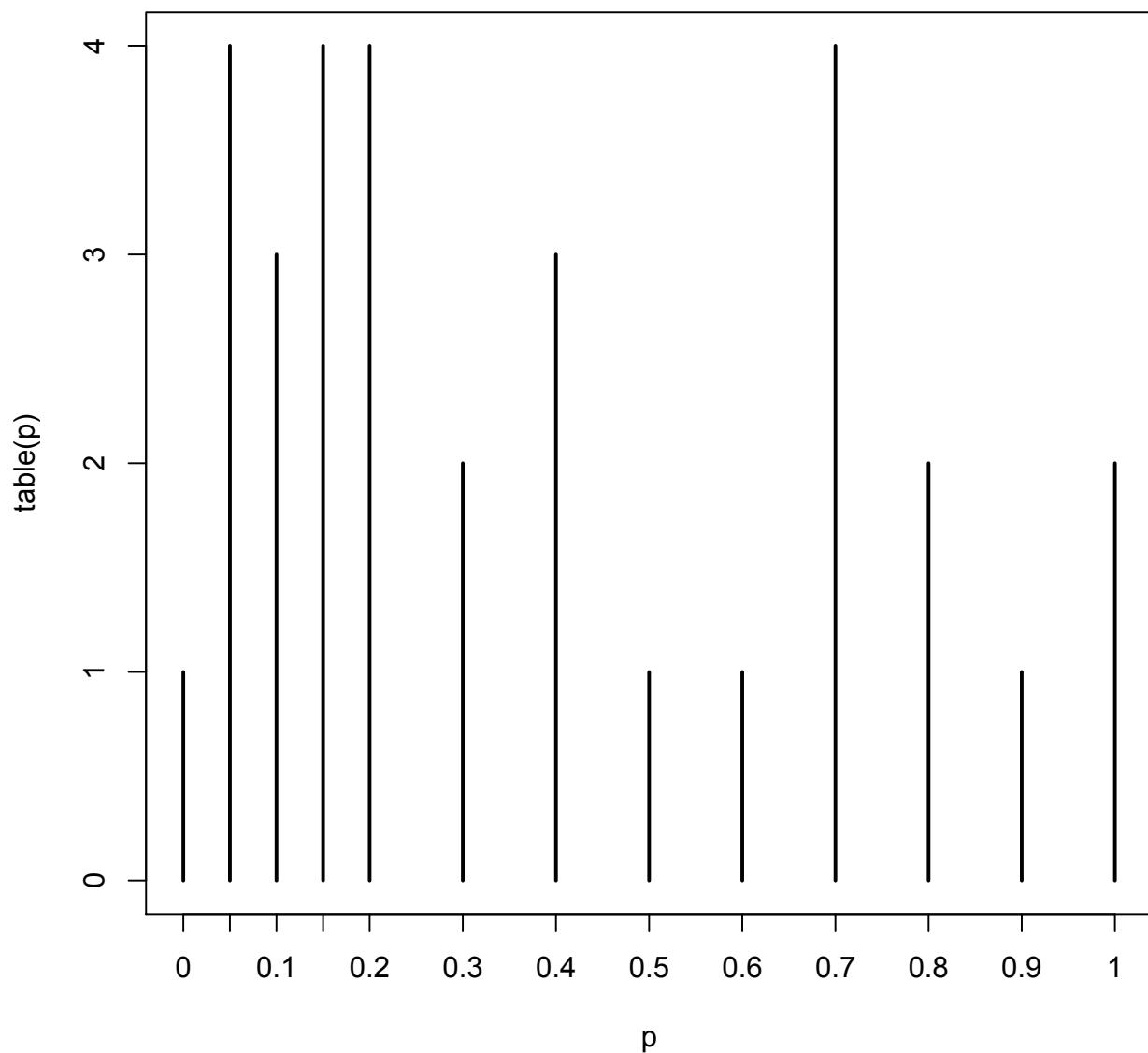
#### 21.4.7 Sample Plot of Pvalues

Let's say we have extracted pvalues from the Allelic Frequency output (into a file called `pvalue.txt` and we want to plot these values (I have just extracted about 40 pvalues here — out of 4.3 million). Using the R programming language, it is straightforward:

```
# 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();
```

The output will be a PDF file illustrated below:

## Allelic Frequency



## 21.5 MapReduce Phase-2

The goal of this phase is to find the first smallest 100 p-values among all chromosomes, which are closest to 0.00. The output of Phase-1 will be used as input to Phase-2. The easiest solution is to sort (by ascending order) the output of Phase-1 by p-value and then output the first 100 records (this is the desired output). Since the output of Phase-1 may be big, sorting is not a good option, instead, we solve this by another MapReduce job called as Phase-2. Using relational databases (such as Oracle or MySQL), this question can be answered as SQL query (assuming that `allele_frequency_table` is a table which contains `pvalue` and `<other>` columns and data is loaded to this table from the output of Phase-1):

```
SELECT *
  FROM allele_frequency_table
    ORDER BY pvalue LIMIT 100;
```

Using allele frequency algorithm (Phase-1), we were able to generate (`pvalue`, `chromosomeID`, `start`, `stop`, ...) for all chromosomes for all given biosets in both groups. Now we need to do sorting (in ascending order) and find the smallest-100 pvalues among generated records. Sorting all the output generated by "allele frequency algorithm" is not a good solution at all. If the output of Phase-1 is huge, then we might ran out of memory. The other option is to use another MapReduce program, which reads the output of the "allele frequency" and then generates the smallest-100 pvalues among all chromosomes. How do we do this MapReduce for finding the smallest-100 pvalues? The new MapReduce algorithm is pretty simple:

1. Mapper:

- a. Each mapper finds its local bottom-100 pvalues and sends that bottom-100 list to the reducer.
- b. We will use many mappers.

2. Reducer

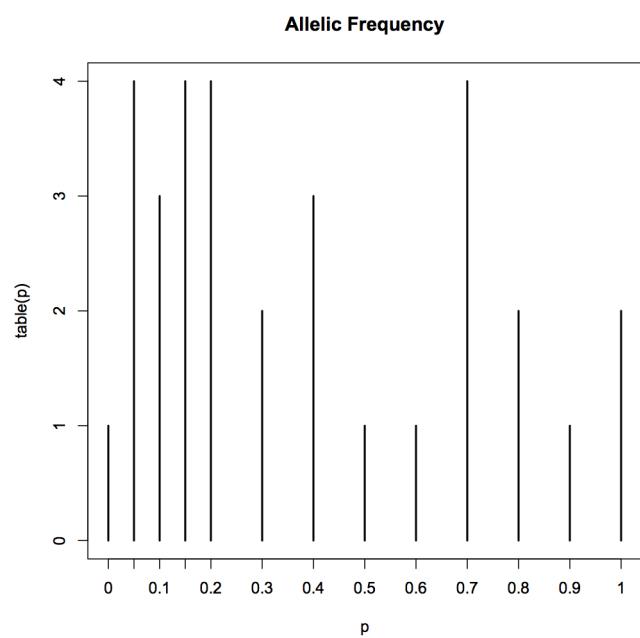


Figure 21.3: p-value for Allelic Frequency

- a. The reducer finds the final bottom-100 pvalues from the bottom-100 lists sent from the mappers.
- b. We will use a single reducer for final bottom-100.

### 21.5.1 Phase-2: Mapper for Bottom-100

The mappers will consume the outputs generated by allele frequency algorithm. Each mapper will read (pvalue, chromosomeID, start, stop, ...) and then emit a bottom-100 list of pvalues. Each mapper will use a bottom-100 list data structure. We will use the Java's `TreeMap`<sup>14</sup> data structure to keep track of bottom-100 list: the bottom-100 sorted list will be preserved by the pvalue (as a Double key) ( $0 \leq \text{pvalue} \leq 1$ ).

```
import java.util.TreeMap;
...
TreeMap<Double, String> bottom100 = new TreeMap<Double, String>();
```

We have to make sure that our bottom100 data structure just holds "bottom 100" pvalues. Here is the map() function inside Bottom100Mapper class: the map() function itself will not emit any key-value pairs, but it will maintain and update the bottom100 data structure. When the map() has completed its function, then the cleanup() function will emit the "bottom 100 list".

**Listing 21.6:** Bottom100Mapper

```
1 public class Bottom100Mapper ... {
2
3     private SortedMap<Double, String> bottom100 =
4         new TreeMap<Double, String>();
5
```

---

<sup>14</sup>The `java.util.TreeMap` is a Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used. This implementation provides guaranteed  $\log(n)$  time cost for the `containsKey`, `get`, `put` and `remove` operations.

```

6   // key is the pvalue of double type and range is 0.00 to 1.00
7   // value is the entire record of allelic frequency output (includes pvalue)
8   map(Double key, String entireRecord) {
9     bottom100.put(key, value); // sort by pvalue
10    if (bottom100.size() > 100) {
11      // remove the greatest pvalue
12      bottom100.remove(bottom100.lastKey());
13    }
14  }
15
16 // called once at the end of the mapper task.
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 key will send all key-value
23     // pairs to a single reducer only
24     emit(NULL, outputValue);
25   }
26 }
27 }
```

---

### 21.5.2 Phase-2: Reducer for "Bottom 100"

We designed "bottom100" algorithm in such a way so as to have only one reducer to aggregate all bottom 100 lists generated by all mappers. The key for our single reducer is NULL (in Hadoop this will be an instance of `NullWritable`. To guarantee that we will have only one reducer, additionally we will set the number of reduce tasks (`setNumReduceTasks()`) to one (1) in our job driver class:

**Listing 21.7:** Bottom100Driver

```

1 public class Bottom100Driver {
2   ...
3   void run(String[] args) {
4     ...
5     Job job = new Job(...);
6     ...
7     job.setNumReduceTasks(1);
8     ...
```

```
9     }
10    ...
11 }
```

The job of the single Reducer is to generate the final "bottom 100" list from all "bottom 100" lists generated by all mappers. So, the reducer functionality is very similar to the mappers.

**Listing 21.8:** Bottom100Reducer

```
1 public class Bottom100Reducer ... {
2
3     reduce(NullWritable key, Iterable<pair<Double, String>> values) {
4         SortedMap<Double, String> finalBottom100 =
5             new TreeMap<Double, String>();
6
7         for (pair(Double, String) value : values) {
8             Double pvalue = value.pvalue;
9             String entireRecord = value.entireRecord;
10            finalBottom100.put(pvalue, entireRecord);
11
12            if (finalBottom100.size() > 100) {
13                // remove the greatest pvalue
14                finalBottom100.remove(finalBottom100.lastKey());
15            }
16        }
17
18        // now, we have the final bottom 100 list
19        for (Map.Entry<Double, String> entry : finalBottom100.entrySet()) {
20            Double pvalue = entry.getKey();
21            String entireRecord = entry.getValue();
22            emit(pvalue, entireRecord);
23        }
24    }
25 }
```

## 21.6 Is Bottom 100 List A Monoid?

The question is whether our bottom100 is a monoid? If it is, then we can provide a combiner for our bottom100 list. As we will see below, bottom100 is a commutative monoid:

- The bottom100 Definition:

$\text{bottom100} = \text{MONOID}(S, e, f)$ ,  
where

$S = \{(p, w), 0 \leq p \leq 1, w \text{ is a unique chromosome record}\}$   
 $e = \{\}$  is an empty set  
 $f = \text{bottom100}$  function

- The bottom100 Commutative Property:

$\text{bottom100}(a) = \{(p_1, w_1), \dots, (p_{100}, w_{100}) \text{ where } 0 \leq p_1 \leq \dots \leq p_{100} \leq 1\}$   
 $\text{bottom100}(a, b) = \text{bottom100}(a \cup b) = \text{bottom100}(b \cup a) = \text{bottom100}(b, a)$   
 $\text{bottom100}(a, b) = c \in S$

- The bottom100 Identity Element Property:

Let  $a, b \in S$ , then  
 $\text{bottom100}(a, e) = a$   
 $\text{bottom100}(e, a) = a$

And in mathematical notation we can write these as

- ✓ Closure:  $\forall a, b \in S : \text{bottom100}(a, b) \in S$
- ✓ Associativity:  $\forall a, b, c \in S :$   
 $\text{bottom100}(a, \text{bottom100}(b, c)) = \text{bottom100}(\text{bottom100}(a, b), c)$
- ✓ Identity element:  $\exists e \in S : \forall a \in S : \text{bottom100}(a, e) = a = \text{bottom100}(e, a)$
- ✓ Commutative:  $\forall a, b \in S : \text{bottom100}(a, b) = \text{bottom100}(b, a)$

### 21.6.1 Hadoop Solution for Bottom 100 List

The Java classes used in Bottom 100 List MapReduce solution are listed below. In our solution, we utilized the `edu.umd.cloud9.io.pair.PairOfDoubleString`<sup>15</sup>

---

<sup>15</sup> <http://lintool.github.io/Cloud9/docs/api/edu/umd/cloud9/io/pair/PairOfDoubleString.html>  
(Cloud<sup>9</sup>, developed by Jimmy Lin, is a collection of Hadoop tools that tries to make working with big data a bit easier.)

class to represent a Hadoop's WritableComparable representing a pair consisting of a `Double` and an `String`.

<i>Class name</i>	<i>Description</i>
<code>Bottom100Driver</code>	A driver program to submit Hadoop jobs
<code>Bottom100Mapper</code>	Defines <code>map()</code>
<code>Bottom100Combiner</code>	Defines <code>combine()</code>
<code>Bottom100Reducer</code>	Defines <code>reduce()</code>
<code>HadoopUtil</code>	Defines some utility functions
<code>PairOfDoubleString</code>	WritableComparable representing a pair consisting of a double and a String.

## 21.6.2 Sample Run of Bottom 100 List

### 21.6.2.1 Prepare HDFS's Input/Output

### 21.6.2.2 Running the Job

### 21.6.2.3 Viewing Output

### 21.6.2.4 Sample Run for Bottom 50

## 21.7 MapReduce Phase-3

In Phase-2 we found the first smallest 100 p-values `among all chromosomes`, which are closest to 0.00. In Phase-3, we will find the first smallest 100 p-values `per chromosome`, which are closest to 0.00. That is, we want to find the

- first smallest 100 p-values for chromosome 1
- first smallest 100 p-values for chromosome 2
- ...
- first smallest 100 p-values for chromosome 22
- first smallest 100 p-values for chromosome X
- first smallest 100 p-values for chromosome Y

Therefore, we want to generate 24 outputs. The output of Phase-1 will be used as input to Phase-3. The easiest solution is to sort (by ascending order) the output of Phase-1 by p-value and then group by a chromosome, and finally output the first 100 records (this is the desired output) per chromosome. Given that the output of Phase-1 may be big, sorting is not a good option. Instead, we solve this by another MapReduce job called as Phase-3. Using relational databases (such as Oracle or MySQL), this question can be answered as SQL query (assuming that `allele_frequency_table` is a table which contains `pvalue` and `<other>` columns and data is loaded to this table from the output of Phase-1): we may run this SQL query per chromosome

```

for (id in (1, 2, 3, ..., 21, 22, X, Y)) {
    SELECT *
    FROM allele_frequency_table
    WHERE chromosome_id = id
    ORDER BY pvalue LIMIT 100;

}

```

The MapReduce Phase-3 algorithm is pretty simple:

1. Mapper:

a. Each mapper finds its local bottom-100 pvalues per chromosome and sends that bottom-100 list to the reducer (one reducer per chromosome ---)

total of 24 reducers).

b. We will use many mappers.

## 2. Reducer

- a. Each reducer finds the final bottom-100 pvalues from the bottom-100 lists sent from the mappers.
- b. We will use a single reducer per chromosome for final bottom-100.

### 21.7.1 Phase-3: Mapper for Bottom-100

The mappers will consume the outputs generated by allele frequency algorithm. Each mapper will read (pvalue, chromosomeID, start, stop, ...) and then emit a bottom-100 list of pvalues. Each mapper will use a bottom-100 list data structure for each chromosome. We will use the Java's TreeMap data structure to keep track of bottom-100 list: the bottom-100 sorted list will be preserved by the pvalue (as a Double key) ( $0 \leq \text{pvalue} \leq 1$ ).

**Listing 21.9:** Required Data Structures

```
1 import java.util.Map;
2 import java.util.HashMap;
3 import java.util.SortedMap;
4 import java.util.TreeMap;
5 ...
6 // map<chromosomeID, treemap<pvalue, record>>
7 private Map<String, SortedMap<Double, String>> chromosomes =
8     new HashMap<String, TreeMap<Double, String>>();
```

We have to make sure that our bottom100 data structure just holds "bottom 100" pvalues. Here is the map() function inside Bottom100MapperPhase3 class: the map() function itself will not emit any key-value pairs, but it will maintain and update the bottom100 data structure. When the map() is done its function, then the cleanup() function will emit the "bottom 100 list" per chromosome.

### Listing 21.10: Bottom100MapperPhase3

```
1 public class Bottom100MapperPhase3 ... {
2     // map<chromosomeID, treemap<pvalue, record>>
3     private Map<String, SortedMap<Double, String>> chromosomes =
4         new HashMap<String, TreeMap<Double, String>>();
5
6     // key is the pvalue of double type and range is 0.00 to 1.00
7     // value is the entire record of allelic frequency output (includes pvalue)
8     map(Double key, String entireRecord) {
9         String chromosomeID = extract(entireRecord);
10        SortedMap<Double, String> bottom100 = chromosomes.get(chromosomeID);
11        if (bottom100 == null) {
12            bottom100 = new TreeMap<Double, String>();
13        }
14        bottom100.put(key, value); // sort by pvalue
15        if (bottom100.size() > 100) {
16            // remove the greatest pvalue
17            bottom100.remove(bottom100.lastKey());
18        }
19    }
20
21    // called once at the end of the mapper task.
22    cleanup() {
23        for (Map.Entry<String, SortedMap<Double, String>>> entry : chromosomes.entrySet()) {
24            String chromosomeID = entry.getKey();
25            SortedMap<Double, String> bottom100 = entry.getValue();
26            for (Map.Entry<Double, String> row : bottom100.entrySet()) {
27                Double pvalue = row.getKey();
28                String entireRecord = row.getValue();
29                String outputValue = pair(pvalue, entireRecord);
30                // pairs to a specific chromosome
31                emit(chromosomeID, outputValue);
32            }
33        }
34    }
35 }
```

## 21.7.2 Phase-3: Reducer for "Bottom 100"

We designed "bottom100" algorithm in such a way to have only one reducer per chromosome to aggregate all bottom 100 lists generated by all mappers (note that each reducer will work on bottom-100 per chromosome — as op-

posed to all chromosomes in Phase-2). The key for our single reducer is chromosomeID.

The job of each Reducer is to generate the final "bottom 100" list from all "bottom 100" lists generated by all mappers (per chromosome). So, the reducer functionality is very similar to the mappers.

**Listing 21.11:** Bottom100ReducerPhase3

```
1 public class Bottom100ReducerPhase3 ... {
2
3     // key is a chromosomeID in {1, 2, ..., 22, X, Y}
4     reduce(String key, Iterable<pair<Double, String>> values) {
5         SortedMap<Double, String> finalBottom100 =
6             new TreeMap<Double, String>();
7
8         for (pair(Double, String) value : values) {
9             Double pvalue = value.pvalue;
10            String entireRecord = value.entireRecord;
11            finalBottom100.put(pvalue, entireRecord);
12
13            if (finalBottom100.size() > 100) {
14                // remove the greatest pvalue
15                finalBottom100.remove(finalBottom100.lastKey());
16            }
17        }
18
19        // now, we have the final bottom 100 list
20        for (Map.Entry<Double, String> entry : finalBottom100.entrySet()) {
21            Double pvalue = entry.getKey();
22            String entireRecord = entry.getValue();
23            emit(key, (pvalue + entireRecord));
24        }
25    }
26 }
```

### 21.7.3 Hadoop Solution for Bottom 100 List Per Chromosome

The Java classes used in Bottom 100 List Per Chromosome MapReduce solution are listed below. In our solution, we utilized the `edu.umd.cloud9.io.pair.PairOfDoubleString` class to represent a Hadoop's WritableComparable representing a pair consisting of a Double and an String.

<i>Class name</i>	<i>Description</i>
<code>Bottom100DriverPhase3</code>	A driver program to submit Hadoop jobs
<code>Bottom100MapperPhase3</code>	Defines <code>map()</code>
<code>Bottom100CombinerPhase3</code>	Defines <code>combine()</code>
<code>Bottom100ReducerPhase3</code>	Defines <code>reduce()</code>
<code>HadoopUtil</code>	Defines some utility functions
<code>PairOfDoubleString</code>	WritableComparable representing a pair consisting of a double and a String.

### 21.7.4 Sample Run of Bottom 100 List Per Chromosome

#### 21.7.4.1 Prepare HDFS's Input/Output

#### 21.7.4.2 Running the Job

#### 21.7.4.3 Viewing Output

---

<sup>16</sup> <http://lintool.github.io/Cloud9/docs/api/edu/umd/cloud9/io/pair/PairOfDoubleString.html>  
(Cloud<sup>9</sup>, developed by Jimmy Lin, is a collection of Hadoop tools that tries to make working with big data a bit easier.)

# Chapter 22

## The T-Test

### 22.1 Introduction

The T-Test (known as two-sample t-test) is used in clinical applications and genome analysis as a statistical hypothesis test. The t-test for independent samples compares the means of two samples. In statistics, to compare two data sets, we convert the data to a simpler form such as means of data and then compare the means. We can compare these two sets of data by computing an average, or mean ( $\mu$ ). Since we are comprising random sample, there is a room for a random error (usually denoted by the sample's standard deviation  $\sigma = \sqrt{\mu_2}$ ). In factoring a random error, therefore, we might be comparing  $\mu \pm \sigma$ . According to Statistics in a Nutshell<sup>1</sup> "The purpose of this test (T-test) is to determine whether the means of the populations from which the samples were drawn are the same. The subjects in the two samples are assumed to be unrelated and to have been independently selected from their populations."

The main goal of this chapter is to provide a MapReduce/Hadoop and Spark/Hadoop solutions for T-Test. The MapReduce algorithm presented here is generic and can be used for any high volume of data.

In genome analysis and especially in "somatic mutations," they use the T-Test for a pair of samples drawn for the same gene (identified as a GENE-ID).

---

<sup>1</sup>Statistics in a Nutshell, Second Edition by Sarah Boslaugh, Published by O'Reilly Media, Inc., 2013

This is how T-Test is done: given a set of biosets/biomarkers identified by  $B_1, B_2, \dots, B_n$  (where each bioset has a set of "pair of GENE-ID and GENE-VALUE"<sup>2</sup>) and given "survival time" identified by  $S_1, S_2, \dots, S_n$  (note that ,  $S_i$  corresponds to  $B_i$ ), the goal is create two sample sets: **Exist-Set** and **Non-Exist-Set**. Given these input, we want to create  $G_1, G_2, \dots, G_m$  where  $G_i$  represents a set of biosets and  $G_i$  is a GENE-ID in that bioset ( $G_i$  acts as a reverse index to biosets).

- **Exist-Set** Sample:  $\{S_i | B_i \in G_k\}$
- **Non-Exist-Set** Sample:  $\{S_i | B_i \notin G_k\}$

Then we apply *ttest* function to **Exist-Set** and **Non-Exist-Set**. Here is a concrete example with 4 biosets ( $B_i$  is a GID and  $T_i$  is primitive double data type):

Sample Data				
Biosets ID:	$B_1$	$B_2$	$B_3$	$B_4$
Survival Time:	$T_1$	$T_2$	$T_3$	$T_4$

Further, assume that the biosets have the following GENE-ID and GENE-VALUE pairs:

---

<sup>2</sup>We note that the number of pairs does depend on the "bioset type". For example, for RNA Gene Expression data type we might have 40,000+ pairs and for Methylation data type we might have 20,000+ pairs.

Bioset Data		
Bioset	GENE-ID	GENE-VALUE
$B_1$	G1	V11
	G2	V12
	G3	V13
$B_2$	G1	V21
	G2	V22
	G4	V24
$B_3$	G1	V31
	G3	V32
	G4	V33
	G5	V34
	G6	V35
$B_4$	G2	V41
	G5	V42
	G6	V43

Then we will have a containment table as:

GENE-ID \ Time	T1	T2	T3	T4
G1	B1	B2	B3	-
G2	B1	B2	-	B4
G3	B1	-	B3	-
G4	-	B2	B3	-
G5	-	-	B3	B4
G6	-	-	B3	B4

For example, the first two indicates that G1 is contained in biosets  $\{B_1, B_2, B_3\}$ , but not contained in  $\{B_4\}$ . This leads us to call up the following functions:

```

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})

```

For a single *ttest* implementation, we will use the following objects from the Apache's Common Math project:

- Interface: `org.apache.commons.math.stat.inference.TTest`
- Implementation class: `org.apache.commons.math.stat.inference.TTestImpl`

The implementation of *ttest* (used in `TTestImpl` class) can be described as follows. This statistic can be used to perform a two-sample (sample1 and sample2) t-test to compare sample means. The returned t-statistic is

$$t = \frac{(m_1 - m_2)}{\sqrt{\frac{v_1}{n_1} + \frac{v_2}{n_2}}}$$

where

- $n_1$  is the size of the first sample (sample1)
- $n_2$  is the size of the second sample (sample2)
- $m_1$  is the mean of the first sample
- $m_2$  is the mean of the second sample
- $v_1$  is the variance of the first sample
- $v_2$  is the variance of the second sample

How do we use these objects for implementation of *ttest*? Here is a simple example:

#### **Listing 22.1: TTest Implementation**

```
1 import org.apache.commons.math.stat.inference.TTest;
2 import org.apache.commons.math.stat.inference.TTestImpl;
3 import org.apache.commons.math.MathException;
4
5 public class MathUtil {
6     /**
7      * @param sample1 array of sample data values
8      * @param sample2 array of sample data values
9      * @return Returns the observed significance level,
10     * or p-value, associated with a two-sample, two-tailed
11     * t-test comparing the means of the input arrays.
```

```

12     */
13     public static double ttest(double[] sample1, double[] sample2) {
14         if ( (sample1 == null) ||
15             (sample2 == null) ||
16             (sample1.length == 0) ||
17             (sample2.length == 0) ) {
18             // return a non-existent-value
19             return Double.MAX_VALUE;
20         }
21
22         if ((sample1.length == 1) && (sample2.length == 1)) {
23             // return a non-existent-value
24             return Double.MAX_VALUE;
25         }
26         return calculateTtest(sample1, sample2);
27     }
28
29     private static double calculateTtest(double[] sample1, double[] sample2) {
30     ...
31 }
32 }
```

---

Implementation of `calculateTtest()` method is given below:

**Listing 22.2: The calculateTtest() Method**

```

1 /**
2 * @param sample1 array of sample data values
3 * @param sample2 array of sample data values
4 * @return Returns p-value, associated with sample1 and sample2
5 */
6 private static double calculateTtest(double[] sample1, double[] sample2) {
7     double pvalue = 0.0d;
8     TTest ttest = new TTestImpl();
9     try {
10         if (sample1.length == 1) {
11             pvalue = ttest.tTest(sample1[0], sample2);
12         }
13         else if (sample2.length == 1) {
14             pvalue = ttest.tTest(sample2[0], sample1);
15         }
16         else {
17             pvalue = ttest.tTest(sample1, sample2);
18         }
19     }
20     catch(MathException me) {
21         System.out.println("ttest() failed. +" , me.getMessage());
22         pvalue = 0.0d;
23     }
24
25     return pvalue;
26 }
```

---

## 22.2 MapReduce Problem Statement

Given a set of biosets<sup>3</sup> with data in the millions and tens of millions, and where a bioset may have up to 40,000 to 100,000 genes, we want to apply T-Test (*ttest*) function to each gene for all biosets. Typically, in genome analysis, the T-Test is applied to "gene expression," "methylation," "somatic mutation," and "copy number variation" bioset data types. This kind of analysis involves reading and processing hundreds of millions of records. For example, to process 200,000 biosets (the normal number of biosets in patient centric applications) with each having 50,000 unique GENE-IDs, the MapReduce solution has to handle 10 billion (10,000,000,000) records (this volume of data and *ttest* processing cannot be handled by a single server).

## 22.3 Input

Our input for T-Test has two parts ( $B_i$  corresponds to  $S_i$ ):

- List of biosets:  $B_1, B_2, \dots, B_n$
- Survival time for these biosets:  $S_1, S_2, \dots, S_n$

Each record of a bioset will have the following format:

```
format:  
<geneID><,><biosetID><,><geneValue>  
  
example-1:  
    7562135,778800,1.04  
example-2:  
    7570769,778800,-1.09
```

The bioset files persists in HDFS and will be read by MapReduce framework (during the *map()* execution). How will we pass these two additional dynamic parameters ("list of bioset IDs", and "survival time for biosets")

---

<sup>3</sup>For details on biosets, please refer to [Appendix A](#)

from MapReduce driver to `map()` and `reduce()` functions. Using MapReduce/Hadoop Configuration object, we can set these values by `Configurtion.set()` and retrieve them (by `Configurtion.get()`) in the `setup()` function of `map()` or `reduce()`. An alternative method is to use Hadoop's `DistributedCache`<sup>4</sup> class, which can distribute application-specific large, read-only files efficiently.

## 22.4 Expected Output

For each GENE-ID (contained in all of biosets), we do need to generate:

```
format:  
<geneID><,><RESULT-of-TTEST-FUNCTION>  
  
where <RESULT-of-TTEST-FUNCTION> is a p-value
```

## 22.5 MapReduce Solution

The goal of MapReduce solution is to apply *ttest* to all genes contained in these biosets. The mapper gets `<geneID><,><biosetID><,><geneValue>` and emits a (*key, value*) pair, where *key* is the GENE-ID and *value* is the BIOSET-ID. The reducer function's job is to create two lists (EXIST and NON-EXIST) and to then apply *ttest* to these two lists.

The `map()` function is a simple function, which emits pairs of (geneID, biosetID).

**Listing 22.3:** Mapper for T-Test

```
1 /**
2 * @param key is generated by MapReduce framework, ignored here
3 * @param value as a String has the the following format:
4 *   <geneID><,><biosetID><,><geneValue>
5 * Note that <geneValue> is not used in T-Test.
6 */
```

---

<sup>4</sup>The full name is `org.apache.hadoop.filecache.DistributedCache`. DistributedCache is a class provided by the Hadoop's MapReduce framework to cache files needed by applications.

```

7 map(key, value) {
8     String[] tokens = value.split(",");
9     String geneID = tokens[0];
10    String biosetID = tokens[1];
11    emit(geneID, biosetID);
12 }

```

---

The main function of a reducer is to create a reverse index genes based on bioset-IDs and finally apply the T-Test algorithm to two sets: EXIST and NON-EXIST. The `reduce()` function does T-Test implementation and preserves the order of bioset IDs with survival time.

#### **Listing 22.4:** Reducer for T-Test

```

1 public class TtestReducer {
2
3     // instance variables
4     private Configuration conf = null;
5     // biosetIDs as Strings : NOTE oreder of biosets are VERY IMPORTANT
6     public List<String> biosets = null;
7     // survival time : NOTE oreder of time items are VERY IMPORTANT
8     public List<Double> time = null;
9
10    // will be run only once
11    public void setup(Context context) {
12        this.conf = context.getConfiguration();
13
14        // get parameters from Hadoop's configuration
15        String biosetsAsString = conf.get("biosets");
16        this.biosets = DataStructuresUtil.splitOnToListOfString(biosetsAsString, ",");
17
18        String timeAsCommaSeparatedString = conf.get("time");
19        this.time = DataStructuresUtil.toListOfDouble(timeAsCommaSeparatedString);
20    }
21
22    // key = geneID
23    // values = list of { biosetID }
24    public void reduce(Text key, Iterable<Text> values) {
25        // solution is given next
26    }
27 }

```

---

The `reduce()` function for T-Test is presented below:

#### **Listing 22.5:** `reduce()` function for T-Test

```

1 // key = geneID
2 // values = list of { biosetID }
3 public void reduce(Text key, Iterable<Text> values) {

```

```

4     Iterator<Text> iter = values.iterator();
5     List<Double> exist = new ArrayList<Double>();
6     List<Double> notexist = new ArrayList<Double>();
7     List<String> genebiosets = new ArrayList<String>();
8     while (iter.hasNext()) {
9         String biosetId = iter.next();
10        genebiosets.add(biosetId);
11    }
12
13    for (int i=0; i < biosets.size(); i++) {
14        // check to see if this biosetId does exist in genes
15        int index = genebiosets.indexOf(biosets.get(i));
16        if (index == -1) {
17            // biosetId not found
18            notexist.add(time.get(i));
19        }
20        else {
21            // biosetId found
22            exist.add(time.get(i));
23        }
24    }
25
26    if (exist.isEmpty()) {
27        // no need to include it
28        return;
29    }
30
31    if (notexist.isEmpty()) {
32        // no need to include it
33        return;
34    }
35
36    //
37    // here we have:
38    //      (exist.size() > 0) && (notexist.size() > 0)
39    //
40    // prepare final value(s) for reducer
41    //
42    double pvalue = MathUtil.ttest(exist, notexist);           // calls ttest()
43    emit(key, pvalue);
44 }

```

---

## 22.6 Hadoop Implementation

This section implements a MapReduce solution for T-Test using Hadoop. Our Hadoop solution is comprised of the following Java classes:

<i>Class Name</i>	<i>Description</i>
TtestDriver.java	The driver class for submitting MapReduce jobs
TtestMapper.java	Defines the map() function
TtestReducer.java	Defines the reduce() function
TtestAnalyzer.java	Reads output data generated by reducers

## 22.7 Spark Implementation

This section implements a MapReduce solution for T-Test using Spark. The Spark soultion is a single driver Java class using RDDs. Three types of input are involved for T-Test:

- List of bioset IDs as  $\{B_1, B_2, \dots, B_n\}$ , where  $n > 0$  and each  $B_i$  is a bioset ID
- List of survival time as  $\{T_1, T_2, \dots, T_n\}$ , where each  $T_i$  is a double data type. Each  $B_i$  corrsponds to  $T_i$ .
- List of  $n$  files (one file per bioset):  $\{B1.txt, B2.txt, \dots, Bn.txt\}$ . Each record of bioset files has the following format (note that for ttest, geneValue will not be used):

```
<geneID><,><biosetID><,><geneValue>
```

Since each  $B_i$  corrsponds to  $T_i$ , we will combine these two data into a single file, called `timetable.txt` where each record will have two columns: `<bioset_id>` followed by `<survival_time>`. Therefore, we assume the following input files:

```
/<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
```

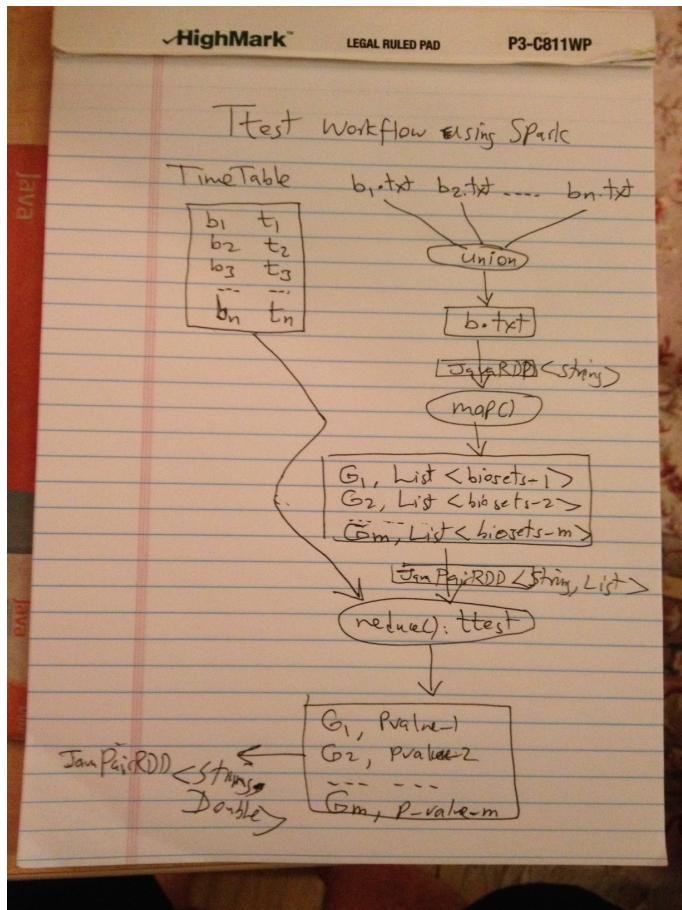


Figure 22.1: Ttest Spark WorkFlow

Using these input files, Ttest Spark workflow is presented below:

The high-level implementation of workflow in Spark is presented below:

**STEP-1** : Concatenate all bioset files (`B1.txt`, `B2.txt`, ..., `Bn.txt`) and create a `JavaRDD<String>`, where each item (as a `String` object) of this RDD will be as:

```
<geneID><,><biosetID><,><geneValue>
```

**STEP-2** : Map `JavaRDD<String>` into `JavaRDD<String, Iterable<String>>`, where key (as a `String`) is a gene-id and value is a list of biosets.

**STEP-3** : Read timetable data (as pairs of (biosetID, survival-time)) and reduce `JavaRDD<String, Iterable<String>>` using T-test algorithm into a final `JavaRDD<String, Double>` where key (as a `String`) is a gene-id and value is a p-value (output of ttest algorithm).

**STEP-4** : Save the final `JavaPairRDD` in HDFS.

### 22.7.1 High Level Steps

#### Listing 22.6: Spark Solution: High Level Steps

```
1 // STEP-0: import required classes and interfaces
2 public class SparkTtest {
3     static JavaSparkContext createJavaSparkContext() {...}
4     static Map<String, Double> createTimeTable(String filename) {...}
5     static JavaRDD<String> readBiosetFiles(JavaSparkContext ctx, String biosetFiles) {...}
6
7     public static void main(String[] args) throws Exception {
8         // STEP-1: handle input parameters
9         // STEP-2: Create time table data structure
10        // STEP-3: create a spark context object
11        // STEP-4: broadcast shard variables used by all cluster nodes
12        // STEP-5: create RDD for all biosets
13        // each RDD element has: <geneID><,><biosetID><,><geneValue>
14        // STEP-6: map bioset records into JavaPairRDD(K,V) pairs
15        // where K = <GeneID>, V = <Bioset-ID>
16        // STEP-7: group biosets by GENE-ID
17        // now, for each GENE-ID we have a List<BIOSET-ID>
18        // STEP-8: perform Ttest for every GENE-ID
```

```
19     System.exit(0);
20 }
21 }
```

---

## 22.7.2 STEP-0: import required classes and interfaces

**Listing 22.7:** Spark Solution: High Level Steps

```
1 // STEP-0: import required classes and interfaces
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;
```

---

## 22.7.3 Create JavaSparkContext

**Listing 22.8:** Spark Solution: High Level Steps

```
1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     conf.set("yarn.resourcemanager.hostname", "hnode01319.nextbiosystem.net");
4     conf.set("yarn.resourcemanager.scheduler.address", "hnode01319.nextbiosystem.net:8030");
5     conf.set("yarn.resourcemanager.resource-tracker.address", "hnode01319.nextbiosystem.net:8031");
6     conf.set("yarn.resourcemanager.address", "hnode01319.nextbiosystem.net:8032");
7     conf.set("mapreduce.framework.name", "yarn");
8     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
9     conf.set("spark.executor.memory", "7g");
10    JavaSparkContext ctx = new JavaSparkContext("yarn-cluster", "SparkTtest", conf);
11    return ctx;
12 }
```

---

## 22.7.4 Create TimeTable Data Structure

**Listing 22.9:** Spark Solution: High Level Steps

```
1  static Map<String, Double> createTimeTable(String filename) throws Exception {
2      Map<String, Double> map = new HashMap<String, Double>();
3      BufferedReader in = null;
4      try {
5          in = new BufferedReader(new FileReader(filename));
6          String line = null;
7          while ((line = in.readLine()) != null) {
8              String value = line.trim();
9              String[] tokens = value.split("\t");
10             String biosetID = tokens[0];
11             Double time = new Double(tokens[1]);
12             map.put(biosetID, time);
13         }
14         System.out.println("createTimeTable() map=" + map);
15     }
16     finally {
17         if (in != null) {
18             in.close();
19         }
20     }
21     return map;
22 }
```

## 22.7.5 Create RDD for All Biosets

**Listing 22.10:** Spark Solution: High Level Steps

```
1  static JavaRDD<String> readBioSetFiles(JavaSparkContext ctx,
2                                              String bioSetFiles)
3  throws Exception {
4  StringBuilder unionPath = new StringBuilder();
5  BufferedReader in = null;
6  try {
7      in = new BufferedReader(new FileReader(bioSetFiles));
8      String singleBioSetFile = null;
9      while ((singleBioSetFile = in.readLine()) != null) {
10          singleBioSetFile = singleBioSetFile.trim();
11          unionPath.append(singleBioSetFile);
12          unionPath.append(",");
13      }
14      //System.out.println("readBioSetFiles() unionPath=" + unionPath);
15  }
16  finally {
17      if (in != null) {
18          in.close();
19      }
20  }
```

```

19         }
20     }
21     // remove the last comma ","
22     String unionPathAsString = unionPath.toString();
23     unionPathAsString = unionPathAsString.substring(0, unionPathAsString.length()-1);
24     // create RDD
25     JavaRDD<String> allBiosets = ctx.textFile(unionPathAsString);
26     JavaRDD<String> partitioned = allBiosets.coalesce(14);
27     return partitioned;
28 }
```

## 22.7.6 STEP-1: handle input parameters

**Listing 22.11:** Spark Solution: High Level Steps

```

1   // STEP-1: handle input parameters
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);
```

## 22.7.7 Create time table data structure

**Listing 22.12:** Spark Solution: High Level Steps

```

1   // STEP-2: Create time table data structure
2   Map<String, Double> timetable = createTimeTable(timetableFileName);
```

## 22.7.8 STEP-3: create a spark context object

**Listing 22.13:** Spark Solution: High Level Steps

```

1   // STEP-3: create a spark context object
2   JavaSparkContext ctx = createJavaSparkContext();
```

## 22.7.9 High Level Steps

### Listing 22.14: Spark Solution: High Level Steps

```
1  // STEP-4: broadcast shard variables used by all cluster nodes
2  // we need this shared data structure when we want to find
3  // "existSet" and "notExistSet" sets after grouping data by GENE-ID.
4  final Broadcast<Map<String, Double>> broadcastTimeTable = ctx.broadcast(timetable);
```

## 22.7.10 STEP-5: create RDD for all biosets

### Listing 22.15: Spark Solution: High Level Steps

```
1  // STEP-5: create RDD for all biosets
2  // each RDD element has: <geneID>,<,><biosetId>,<,><geneValue>
3  JavaRDD<String> biosets = readBioSetFiles(ctx, bioSetFileNames);
4  biosets.saveAsTextFile("/ttest/output/1");
```

## 22.7.11 STEP-6: map bioset records into JavaPair-RDD(K,V) pairs

### Listing 22.16: Spark Solution: High Level Steps

```
1  // STEP-6: map bioset records into JavaPairRDD(K,V) pairs
2  // where K = <GeneID>
3  //       V = <Bioeset-ID>
4  // Note that for TTest, <geneValue> is not used (ignored here)
5  //
6  JavaPairRDD<String, String> pairs = biosets.mapToPair(new PairFunction<String, String, String>() {
7      public Tuple2<String, String> call(String bioSetRecord) {
8          String[] tokens = StringUtil.split(bioSetRecord, ",");
9          String geneID = tokens[0]; // K
10         String biosetId = tokens[1]; // V
11         return new Tuple2<String, String>(geneID, biosetId);
12     }
13 });
14 pairs.saveAsTextFile("/ttest/output/2");
```

## 22.7.12 STEP-7: group biosets by GENE-ID

### **Listing 22.17: Spark Solution: High Level Steps**

```
1 // STEP-7: group biosets by GENE-ID
2 JavaPairRDD<String, Iterable<String>> grouped = pairs.groupByKey();
3 // now, for each GENE-ID we have a List<BIOSSET-ID>
4 grouped.saveAsTextFile("/ttest/output/3");
```

### **22.7.13 STEP-8: perform Ttest for every GENE-ID**

### **Listing 22.18: Spark Solution: High Level Steps**

```
1 // STEP-8: perform Ttest for every GENE-ID
2 // mapValues[U](f: (V) => U): JavaPairRDD[K, U]
3 // Pass each value in the key-value pair RDD through a map function without
4 // changing the keys; this also retains the original RDD's partitioning.
5 JavaPairRDD<String, Double> ttest = grouped.mapValues(
6     new Function<Iterable<String>, Double> {
7         // input
8         // output (result of ttest)
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                 // now we do need shared Map data structure to iterate over its items
17                 Map<String, Double> timetable = broadcastTimeTable.value();
18                 // the following two lists are needed for ttest(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()) {
22                     String biosetId = entry.getKey();
23                     Double time = entry.getValue();
24                     if (geneBiosets.contains(biosetId)) {
25                         exist.add(time);
26                     } else {
27                         notexist.add(time);
28                     }
29                 }
30
31                 // perform the ttest(exist, notexist)
32                 double ttest = MathUtil.ttest(exist, notexist);
33                 return ttest;
34             }
35         });
36
37         ttest.saveAsTextFile("/ttest/output/4");
```

## 22.7.14 Ttest Algorithm

**Listing 22.19:** Spark Solution: High Level Steps

```
1 import java.util.List;
2 import java.util.ArrayList;
3 import org.apache.commons.math.stat.inference.TTest;
4 import org.apache.commons.math.stat.inference.TTestImpl;
5
6 public class MathUtil {
7
8     private static final TTest ttest = new TTestImpl();
9
10    public static double ttest(double[] arrA, double[] arrB) {
11        if ((arrA.length == 1) && (arrB.length == 1)) {
12            // return a NULL value for score (does not make sense)
13            return Double.NaN;
14        }
15
16        double score = Double.NaN;
17        try {
18            if (arrA.length == 1) {
19                score = ttest.tTest(arrA[0], arrB);
20            }
21            else if (arrB.length == 1) {
22                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    }
```

```

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
86     return arr;
87 }

```

---

### 22.7.15 Input for Spark Program

```

# 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

```

### 22.7.16 Spark on YARN Script

```

# cat ./run_ttest.sh
#!/bin/bash
/bin/date
source /home/hadoop/conf/env_2.4.0.sh
export HADOOP_HOME=/usr/local/hadoop/hadoop-2.4.0
export SPARK_HOME=/home/hadoop/spark-1.0.0
source /home/hadoop/spark_mahmoud_examples/spark_env_yarn.sh
source $SPARK_HOME/conf/spark-env.sh

# system jars:
CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
#
jars='find $SPARK_HOME -name *.jar'
for j in $jars ; do
    CLASSPATH=$CLASSPATH:$j
done

# app jar:
export MP=/home/hadoop/spark_mahmoud_examples
export CLASSPATH=$MP/mp.jar:$CLASSPATH
export CLASSPATH=$MP/commons-math3-3.0.jar:$CLASSPATH
export CLASSPATH=$MP/commons-math-2.2.jar:$CLASSPATH
export SPARK_CLASSPATH=$CLASSPATH
export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
export JAVA_HOME=/usr/java/jdk7
export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
export MY_JAR=$MP/mp.jar
export SPARK_JAR=$MP/spark-assembly-1.0.0-hadoop2.4.0.jar
export YARN_APPLICATION_CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
export THEJARS=$MP/commons-math-2.2.jar,$MP/commons-math3-3.0.jar
$SPARK_HOME/bin/spark-submit --clas SparkTtest \
    --master yarn-cluster \
    --num-executors 12 \
    --driver-memory 3g \
    --executor-memory 7g \
    --executor-cores 12 \
    --jars $THEJARS \
    $MY_JAR $MP/ttestinput/timetable.txt $MP/ttestinput/biosets.txt

```

## 22.7.17 Sample Run of Script

```

# ./run_ttest.sh
<timetable-file>/home/hadoop/spark_mahmoud_examples/ttestinput/timetable.txt
<bioset-file>/home/hadoop/spark_mahmoud_examples/ttestinput/biosets.txt
createTimeTable() map={b1=0.9, b3=0.5, b2=0.75, b4=1.1}
...
14/07/06 22:26:13 INFO yarn.Client: Uploading file:/home/hadoop/spark_mahmoud_examples/mp.jar to
  hdfs://myserver100:8020/user/hadoop/.sparkStaging/application_1403538869175_0253/mp.jar
14/07/06 22:26:14 INFO yarn.Client: Uploading file:/home/hadoop/spark_mahmoud_examples/spark-assembly-1.0.0-hadoop2.4.0.jar
  to hdfs://myserver100:8020/user/hadoop/.sparkStaging/application_1403538869175_0253/spark-assembly-1.0.0-hadoop2.4.0.jar
...
14/07/06 22:26:16 INFO impl.YarnClientImpl: Submitted application application_1403538869175_0253
...
      appTrackingUrl: http://myserver100:8088/proxy/application_1403538869175_0253/
...

```

## 22.7.18 Generated Outputs

### 22.7.18.1 Outputs for Debugging

```

# 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])

```

### 22.7.18.2 Final Output

```

# hadoop fs -cat /ttest/output/4/part*
(G3,0.08146847659449757)
(G4,0.14994028688738084)
(G5,0.48769401782708255)
(G1,0.05441315690970782)
(G2,0.0)

```

# Chapter 23

## Computing Pearson Correlation

What is Pearson Correlation? The Pearson Correlation measures how well two sets of data are related (linear relationship). The common measure of correlation in Math. and Statistics is the Pearson Correlation. In a nutshell, Pearson Correlation answers this question: is it possible to draw a line graph to represent the data? According to [onlinestatbook<sup>1</sup>](http://onlinestatbook.com/chapter4/pearson.html): "the Pearson product-moment correlation coefficient is a measure of the strength of the linear relationship between two variables. It is referred to as Pearson's correlation or simply as the correlation coefficient. If the relationship between the variables is not linear, then the correlation coefficient does not adequately represent the strength of the relationship between the variables."

The goal of this chapter is to provide two MapReduce solutions for Pearson<sup>2</sup> Correlation:

- Pearson Correlation using MapReduce/Hadoop: this will be a simple case using classical MapReduce/Hadoop
- Pearson Correlation using Spark/Hadoop: this will correlate "all-vs-all" using Java/Spark/Hadoop.

The algorithms presented for Pearson correlations can be easily adapted to Spearman ranked correlations. To perform Spearman ranked correlation, I have provided a Java wrapper class called `Spearman.java`.

<sup>1</sup> <http://onlinestatbook.com/chapter4/pearson.html>

<sup>2</sup>Karl Pearson (27 March 1857 – 27 April 1936) was an English mathematician who has been credited with establishing the discipline of mathematical statistics. For details on understanding Pearson Correlation , see [http://en.wikipedia.org/wiki/Karl\\_Pearson](http://en.wikipedia.org/wiki/Karl_Pearson)

By end of this chapter you will be able to replace Pearson correlation with Spearman's<sup>3</sup> ranked correlation algorithm.

## 23.1 Pearson Correlation Formula

Formula for Pearson Correlation can be written in many different equivalent forms. Let  $x = (x_1, x_2, \dots, x_n)$  and  $y = (y_1, y_2, \dots, y_n)$ , then Pearson Correlation for  $x$  and  $y$  can be expressed as:

$$r = \frac{\Sigma(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\Sigma(x_i - \bar{x})^2 \Sigma(y_i - \bar{y})^2}}$$

$$r = \frac{\Sigma xy - \frac{\Sigma x \Sigma y}{n}}{\sqrt{(\Sigma x^2 - \frac{(\Sigma x)^2}{n})(\Sigma y^2 - \frac{(\Sigma y)^2}{n})}}$$

where

$$\bar{x} = \frac{\Sigma x}{n}$$

$$\bar{y} = \frac{\Sigma y}{n}$$

Pearson Correlation has the following properties:

- Range:  $-1 \leq r \leq 1$
- Correlation coefficient is a unitless index of strength of association between two variables:

$r > 0$  means positive association

$r < 0$  means negative association

$r = 0$  means no association

- Measures the linear relationship between  $x$  and  $y$

---

<sup>3</sup>[http://en.wikipedia.org/wiki/Spearman%27s\\_rank\\_correlation\\_coefficient](http://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient)

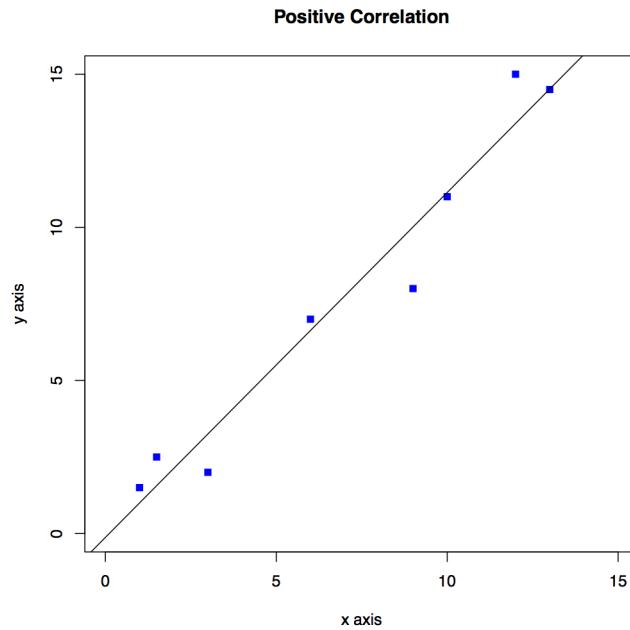


Figure 23.1: Positive Correlation

Given two sets of data, what are the possible values for the Pearson correlation? The results will be between -1.00 and 1.00. The values can be classified as:

- No correlation: close to 0.00
- High/Positive correlation: 0.6 to 1.0 or -0.6 to 1.0
- Medium correlation: 0.3 to 0.6 or -0.3 to 0.6
- Low/Negative correlation: 0.1 to 0.3 or -0.1 to -0.3

The following images shows the meaning of positive, negative, and no correlations.

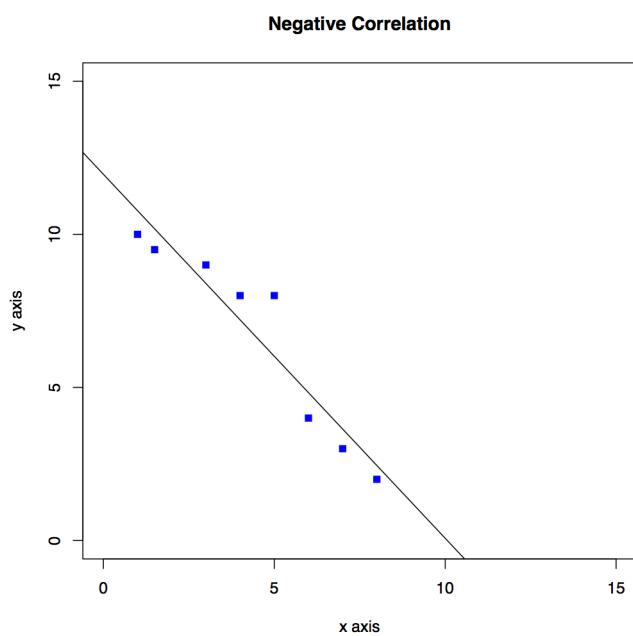


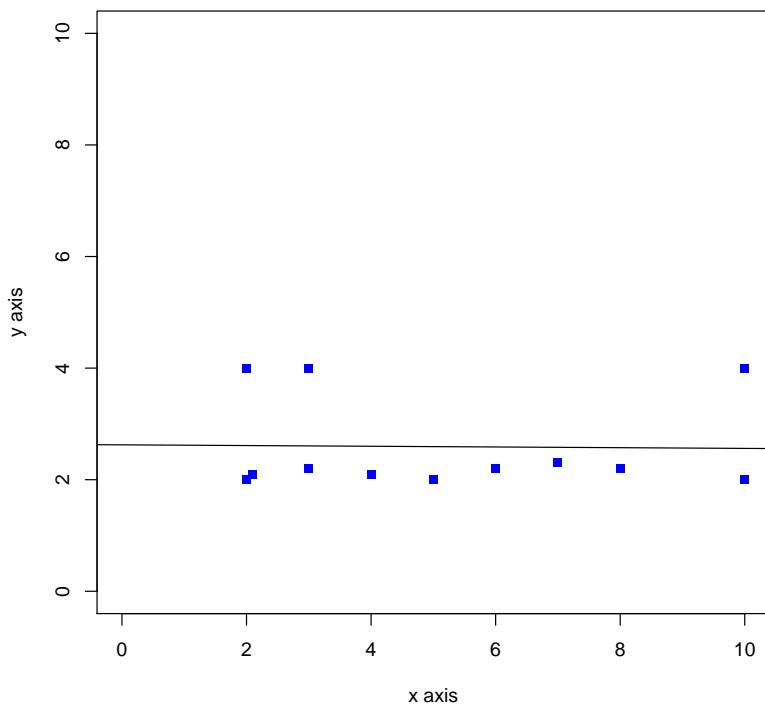
Figure 23.2: Negative Correlation

**Positive Correlation**

**Negative Correlation**

**No Correlation**

**No Correlation**



## 23.2 Pearson Correlation by Example

Below are the data for seven participants giving their number of years in college ( $x$ ) and their subsequent yearly income ( $y$ ). Income here is in thousands of dollars, but this fact does not require any changes in our computations.

Participant	$x$	$y$	$x^2$	$y^2$	$xy$
Alex	0	15	0	225	0
Mary	1	18	0	225	15
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	1225	210
	$\Sigma x = 18$	$\Sigma y = 140$	$\Sigma x^2 = 78$	$\Sigma y^2 = 3600$	$\Sigma xy = 505$

Plugging the values into the Pearson Correlation will result in  $r = 0.95$ .

## 23.3 Data Set for Pearson Correlation

We want to process a generalized data set as a matrix (in CSV format), where the columns correspond to variables (such as  $x$ ,  $y$ ,  $z$ , ...) and rows correspond to instances. The sample data might look like the following:

```
ROW-1: 1, 1, 3, -1  
ROW-2: 2, 2, 1, -2  
ROW-3: 3, 3, 8, -3  
...
```

## 23.4 POJO Solution for Pearson Correlation

Let  $data$  be a 2-dimensional array of doubles, then we can write:

**Listing 23.1:** Computing Pairwise Correlations

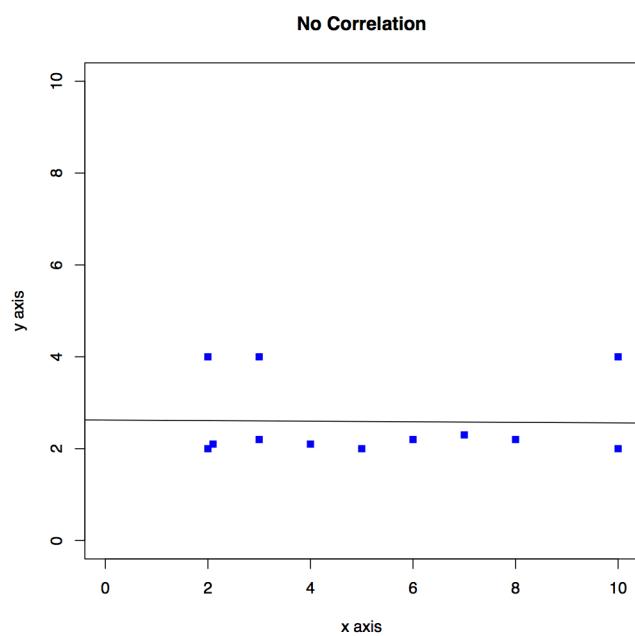


Figure 23.3: No Correlation

```

1 public void computeAllPairwiseCorrelations(double[][] data) {
2     int numColumns = data[0].length;
3     for(int i=0; i < numColumns; i++) {
4         for(int j=i+1; j < numColumns; j++) {
5             //compute the correlation between the i-th and j-th columns/variables
6             double correlation = computeCorrelation(i, j, data);
7             System.out.println("i="+i+" j="+j+" correlation="+correlation);
8         }
9     }
10 }
```

The Java method to compute the pairwise Pearson correlation might look like the following.

### **Listing 23.2:** Computing Pearson Correlation

```

1 public double computePearsonCorrelation(int i, int j, double[][] data) {
2     double x = 0;
3     double y = 0;
4     double xx = 0;
5     double yy = 0;
6     double xy = 0;
7     double n = data.length;
8     for(int row=0; row < data.length; row++) {
9         x += data[row][i];
10        y += data[row][j];
11        xx += Math.pow(data[row][i], 2.0d);
12        yy += Math.pow(data[row][j], 2.0d);
13        xy += data[row][i] * data[row][j];
14    }
15    double numerator = xy - ((x * y) / n);
16    double denominator1 = xx - (Math.pow(x, 2.0d) / n);
17    double denominator2 = yy - (Math.pow(y, 2.0d) / n);
18    double denominator = Math.sqrt(xx * yy);
19    double correlation = numerator / denominator;
20    return correlation;
21 }
```

## **Test Drive POJO Solution**

To test drive Pearson correlation, a small sample is used to show how it is done:

### **Listing 23.3:** Test Drive Pearson Correlation

```

1 public class PearsonCorrelation {
2
3     public void computeAllPairwiseCorrelations(double[][] data) {
4         ...
5     }
6
7     public double computeCorrelation(int i, int j, double[][] data) {
8         ...
9     }
10
11    public static void main(String[] args) throws Exception {
12        double[][] matrix = new double[][] {
13            {1, 1, 3, -1},
14            {2, 2, 1, -2},
15            {3, 3, 8, -3}
16        };
17        PearsonCorrelation pc = new PearsonCorrelation();
18        pc.computeAllPairwiseCorrelations(matrix);
19        System.exit(0);
20    }
21 }
```

---

Sample run is provided below::

```

$ 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
```

From this output we can conclude that our MapReduce solution has to generate six unique reducer keys.

## 23.5 MapReduce Solution for Pearson Correlation

For MapReduce solution, we will focus on map() and reduce()functions.

### 23.5.1 map() for Pearson Correlation

**Listing 23.4:** Mapper for Pearson Correlation

```
1 /**
2 * key is MapReduce generated, ignored here
3 * value is one row of the matrix
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 }
```

Now let's see what each `map()` function will generate:

`map(ROW-1):` will generate:  
K2=(0,1) and V2=(1,1)  
K2=(0,2) and V2=(1,3)  
K2=(0,3) and V2=(1,-1)  
K2=(1,2) and V2=(1,3)  
K2=(1,3) and V2=(1,-1)  
K2=(2,3) and V2=(3,-1)

`map(ROW-2):` will generate:  
K2=(0,1) and V2=(2,2)  
K2=(0,2) and V2=(2,1)  
K2=(0,3) and V2=(2,-2)  
K2=(1,2) and V2=(2,1)  
K2=(1,3) and V2=(1,-2)  
K2=(2,3) and V2=(1,-2)

`map(ROW-3):` will generate:  
K2=(0,1) and V2=(3,3)  
K2=(0,2) and V2=(3,8)  
K2=(0,3) and V2=(3,-3)

```
K2=(1,2) and V2=(3,8)
K2=(1,3) and V2=(3,-3)
K2=(2,3) and V2=(8,-3)
```

### 23.5.2 reduce() for Pearson Correlation

Six unique keys are generated for reducers:

```
K2=(0,1) and List_of_V2=[(1,1), (2,2), (3,3)]
K2=(0,2) and List_of_V2=[(1,3), (2,1), (3,8)]
K2=(0,3) and List_of_V2=[(1,-1), (2,-2), (3,-3)]
K2=(1,2) and List_of_V2=[(1,3), (2,1), (3,8)]
K2=(1,3) and List_of_V2=[(1,-1), (1,-2), (3,-3)]
K2=(2,3) and List_of_V2=[(3,-1), (1,-2), (8,-3)]
```

### 23.5.3 reduce() for Pearson Correlation

**Listing 23.5:** Reducer for Pearson Correlation

```
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 }
```

## 23.6 Hadoop Implementation for Pearson Correlation

For Hadoop implementation, we will provide a driver, `map()`, `reduce()`, and some custom data structures and classes to hold intermediate values for Pearson Correlation.

Class name	Description
PearsonCorrelationDriver	A driver program to submit Hadoop jobs
PearsonCorrelationMapper	Defines <code>map()</code>
PearsonCorrelationReducer	Defines <code>reduce()</code>
PearsonCorrelation	Implements Pearson correlation algorithm
HadoopUtil	Defines some utility functions
TestDataGeneration	Generates test data for Pearson correlation
PairOfWritables<L,R>	edu.umd.cloud9.io.pair.PairOfWritables<L,R>

In our Hadoop implementation, we use a `PairOfWritables<L,R>` class, which can be instantiated to generate `PairOfWritables<Long,Long>` and `PairOfWritables<Double,Double>`. The definition of `PairOfWritables` (a class representing pair of `Writables`) is given below:

**Listing 23.6:** `PairOfWritables` Class

```
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,
9                         R extends Writable>
10    implements Writable {
11
12    private L leftElement;
13    private R rightElement;
14    ...
15 }
```

We use `PairOfWritables.getLeftElement()` and `PairOfWritables.getRightElement()` to retrieve `leftElement` and `rightElement` respectively.

## 23.7 Pearson Correlation using Spark/Hadoop

This goal of this section is to correlate "all-vs-all". What does this mean? Let's define a specific problem case and then explain correlation of "all-vs-all". Note that you may easily adapt the algorithm presented here to your specific input data. For sure input data formats will be different per different application domains. Let  $P = \{P_1, P_2, \dots, P_n\}$  be a set of patients (each patient is identified by a Patient-ID). Let each patient to have a finite set of biomarker data, which has the following format:

```
<Gene-ID><,><Reference><,><Patient-ID><,><Biomarker-Value-As-Double-Data-Type>
```

```
where Reference = {  
    r1, // normal  
    r2, // disease  
    r3 // paired  
    r4 // unknown  
}
```

To achieve correct results of correlation, only one type of "reference" is used (the reference will be used as a filter). For example, two sample biomarker rows might look like:

```
G1234,r3,P100,0.04  
G1345,r1,P200,0.90  
G2155,r2,P200,0.86
```

Let  $G = \{G_1, G_2, \dots, G_m\}$  be a set of genes (expresses as Gene-IDs). For example for RNA-Expression data type the number of unique genes might be about 40,000. Then the correlation of all-vs-all will generate

$$40,000 \times 40,000 = 1600,000,000$$

Pearson correlations. But since correlation of  $(G_i, G_j)$  is the same as  $(G_j, G_i)$ , then this calculation can be reduced to (where  $N = 40,000$ ).

$$\frac{N \times (N - 1)}{2}$$

Therefore, for example, if  $G = \{G_1, G_2, G_3, G_4, G_5\}$ , then we will generate the Pearson correlation for the following pairs:

$$\begin{aligned} & (G_1, G_2) \\ & (G_1, G_3) \\ & (G_1, G_4) \\ & (G_1, G_5) \\ & (G_2, G_3) \\ & (G_2, G_4) \\ & (G_2, G_5) \\ & (G_3, G_4) \\ & (G_3, G_5) \\ & (G_4, G_5) \end{aligned}$$

To generate correct pair of genes for correlation, we will make sure that

$$G_i < G_j$$

this will guarantee that we do not generate both of these pairs:  $(G_i, G_j)$  and  $(G_j, G_i)$ .

### 23.7.1 Input

We will read biomarker data from HDFS. We will assume that the following data exist in HDFS (biomarker data are in text format and identified by text files  $\{b1, b2, b3, \dots\}$ ):

```
/biomarker/input/b1
/biomarker/input/b2
/biomarker/input/b3
...

```

where each record in these biomarker files has the following format:

```
<Gene-ID><, ><Reference><, ><Patient-ID><, ><Biomarker-Value-As-Double-Data-Type>
```

### 23.7.2 Output

Let  $G = \{G_1, G_2, \dots, G_m\}$  be a set of gene IDs. The goal is to generate the following set of output records for every pair of genes  $(G_i, G_j)$ :

$$(G_i, G_j), (\text{pearson-correlation}, \text{p-value})$$

where

- $G_i < G_j$
- $G_i \in \{G_1, G_2, \dots, G_m\}$
- $G_j \in \{G_1, G_2, \dots, G_m\}$
- $0.00 \leq \text{p-value} \leq 1.00$
- $-1.00 \leq \text{pearson-correlation} \leq +1.00$

### 23.7.3 Spark Solution

We present our Spark solution as a single Java driver class, which uses rich Spark API (such as `filter()` and `cartesian()` functions) to find correlation of all-vs-all genes. Note that, here we present a Spark solution for all-genes-vs-all-genes, but you may apply this technique to any kind of data. The input to Spark program will be a set of biomarker files and a `reference` (where the value will be in  $\{r1, r2, r3, r4\}$ ). All biomarker inputs are read from HDFS into a single `JavaRDD` and then partitioned by the following Spark API:

```
public JavaRDD<T> coalesce(int number_of_partitions)
// Description: return a new RDD that is
// reduced into number_of_partitions partitions.
```

After `JavaRDD` is created and partitioned, then we filter all biomarkers by `reference` value. For example, if `reference = r2`, then all records, which does not have reference of `r2` are tossed out. This is accomplished by `JavaRDD.filter()` function. Next we map all filtered records by using `JavaRDD.mapToPair()` function, which generates the following `JavaPairRDD<K, V>`:

```

K = Gene-ID (as String)
V = Tuple2<String, Double>(patientID, geneValue)

```

The next obvious thing to do is group all biomarker data by Gene-ID, which generates the following JavaPairRDD (let's call this `grouped`):

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

where

```

K = Gene-ID
V = Iterable<Tuple2(patientID, geneValue)>

```

To correlate all-vs-all genes, we have to create a cartesian product of `grouped` by `grouped`: this will generate all possible combinations of  $(G_i, G_j)$ . Let's call the result of the cartesian product as `cart`. Before doing any correlation, we will filter the cartesian product result (`cart`) and keep only pairs where  $G_i < G_j$ . Filtering is accomplished by

```
cart.filter(...);
```

After filtering the cartesian product, we are ready to calculate Pearson correlation and its associated p-value for all possible combinations of  $(G_i, G_j)$ . To compute Pearson correlation for  $(G_i, G_j)$ , we will make the following matrix:

	PatientID <sub>1</sub>	PatientID <sub>2</sub>	PatientID <sub>3</sub>	...
$G_i$	avg(values)	avg(values)	avg(values)	...
$G_j$	avg(values)	avg(values)	avg(values)	...

Now, we are ready to present a complete solution using Spark API. First, we present the entire structure of the Spark program as a set of high-level steps and then each step is presented in detail.

### 23.7.4 Spark Solution: High-Level Steps

A workflow for Spark solution is presented below.

**Listing 23.7:** Spark Solution: High-Level Steps

```
1 // STEP-0: import required classes and interfaces
2 /**
3  * What does All-vs-All correlation means?
4  *
5  * Let selected genes be: G = (g1, g2, g3, g4).
6  * Then all-vs-all will correlate between the following pair of genes:
7  *
8  * (g1, g2)
9  * (g1, g3)
10 * (g1, g4)
11 * (g2, g3)
12 * (g2, g4)
13 * (g3, g4)
14 *
15 * Note that pairs (Ga, Gb) are generated if and only if (Ga < Gb).
16 * Correlation of (Ga, Gb) and (Gb, Ga) are the same, so there is no
17 * need to calculate correlation for duplicate genes.
18 *
19 * Biomarker record example:
20 *      format: <geneID><,><r{1,2,3,4}><,><patientID><,><value>
21 *      example: 37761,r2,p10001,1.287
22 *
23 * @author Mahmoud Parsian
24 *
25 */
26 public class AllVersusAllCorrelation implements java.io.Serializable {
27
28     static boolean smaller(String g1, String g2) {...}
29     static class MutableDouble implements java.io.Serializable {...}
30     static Map<String, MutableDouble> toMap(List<Tuple2<String,Double>> list) {...}
31     static List<String> toListOfString(Path hdfsFile) throws Exception {...}
32     static JavaRDD<String> readBiosets(JavaSparkContext ctx, List<String> biosets) {...}
33
34     public static void main(String[] args) throws Exception {
35         // STEP-1: handle input parameters
36         // STEP-2: create a Spark context object
37         // STEP-3: create list of input files/biomarkers
38         // STEP-4: broadcast reference as global shared object
39         //          which can be accessed from all cluster nodes
40         // STEP-5: read all biomarkers from HDFS and create the first RDD
41         // STEP-6: filter biomarkers by reference
42         // STEP-7: create (Gene-ID, (Patient-ID, Gene-Value) pairs
43         // STEP-8: group biomarkers by geneID
44         // STEP-9: create Cartesian product of all genes
45         // STEP-10: filter redundant pairs of genes
46         // STEP-11: calculate Pearson Correlation and p-value
47         System.exit(0);
}
```

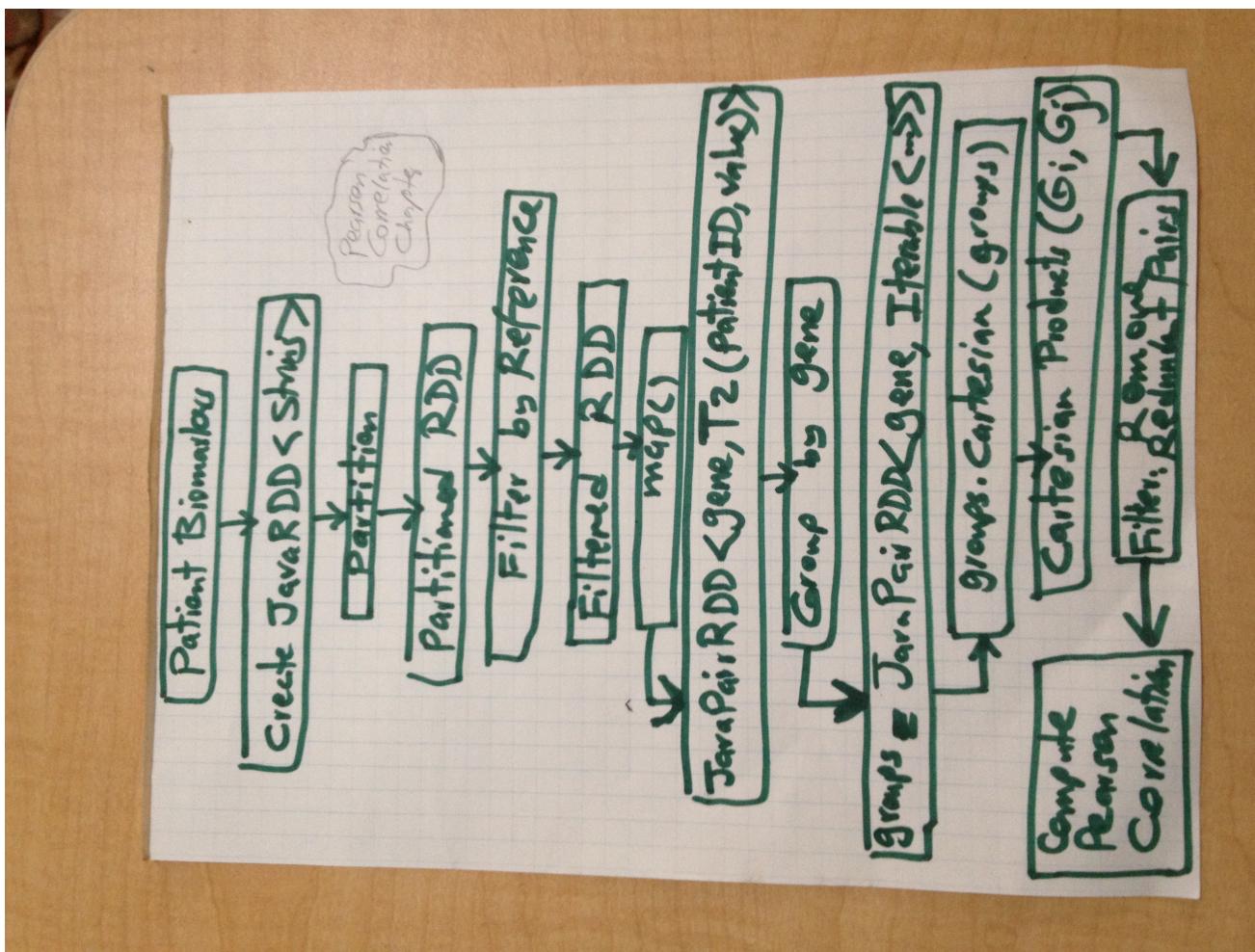


Figure 23.4: Pearson Correlation of All-vs-All

```
48    }
49 }
```

---

### 23.7.5 STEP-0: import required classes and interfaces

This step import the required classes and interfaces. The classes and interfaces we need for our solution are contined in the following Spark packages:

- `org.apache.spark.api.java`, which we use the following classes: `JavaPairRDD`, `JavaRDD`, and `JavaSparkContext`.
- `org.apache.spark.api.java.function`, which we use the following interfaces: `Function` and `PairFunction`,

**Listing 23.8:** STEP-0: import required classes and interfaces

```
1 // STEP-0: import required classes and interfaces
2 import java.util.Map;
3 import java.util.HashMap;
4 import java.util.List;
5 import java.util.ArrayList;
6 import java.io.BufferedReader;
7 import java.io.InputStreamReader;
8 import scala.Tuple2;
9 import org.apache.spark.api.java.JavaRDD;
10 import org.apache.spark.api.java.JavaPairRDD;
11 import org.apache.spark.api.java.JavaSparkContext;
12 import org.apache.spark.api.java.function.PairFunction;
13 import org.apache.spark.api.java.function.Function;
14 import org.apache.spark.broadcast.Broadcast;
15 import org.apache.hadoop.fs.Path;
16 import org.apache.hadoop.fs.FileSystem;
17 import org.apache.hadoop.fs.FSDataInputStream;
18 import org.apache.hadoop.conf.Configuration;
```

---

### 23.7.6 Method smaller()

The `smaller()` method enable us to avoid generating duplicate pair of genes. The pairs  $(g_1, g_2)$  and  $(g_2, g_1)$  will generate the sample correlation, so we will only generate a single pair  $(g_1, g_2)$  where  $g_1$  is smaller than  $g_2$ . After Cartesian product of genes are generated (by using the `cartesian()` function), we use the `smaller()` method to filter out the non-needed pair of genes.

### **Listing 23.9:** Method smaller()

```
1 static boolean smaller(String g1, String g2) {
2     if (g1.compareTo(g2) < 0){
3         return true;
4     }
5     else {
6         return false;
7     }
8 }
```

### **23.7.7 MutableDouble Class**

The MutableDouble class is used to store all biomarker values for a single patient. This class just sums up the double values and keeps the count and it provides a convenient `avg()` method. This class will hold all values for single gene and a single patient.

### **Listing 23.10:** MutableDouble Class

```
1 static class MutableDouble implements java.io.Serializable {
2     private double value = 0.0;
3     private double count = 0.0;
4
5     public MutableDouble(double d) {
6         value = d;
7         count = 1.0;
8     }
9     public MutableDouble(Double d) {
10        if (d == null) {
11            value = 0.0;
12        }
13        else {
14            value = d;
15        }
16        count = 1.0;
17    }
18
19    public void increment(Double d) {
20        if (d == null) {
21            // value does not change
22        }
23        else {
24            value += d;
25        }
26        count++;
27    }
28
29    public void increment(double d) {
30        count++;
```

```

31         value += d;
32     }
33
34     public double avg() {
35         return value/count;
36     }
37 }
```

---

### 23.7.8 Method toMap()

This method converts a list of `Tuple2<PatientID, BiomakerValue>` into a `Map<PatientID, MutableDouble>` (basically aggregates values for all patients for a single gene).

**Listing 23.11:** Method toMap()

```

1 static Map<String, MutableDouble> toMap(Iterable<Tuple2<String,Double>> list) {
2     Map<String, MutableDouble> map = new HashMap<String, MutableDouble>();
3     for (Tuple2<String,Double> entry : list) {
4         MutableDouble md = map.get(entry._1);
5         if (md == null) {
6             map.put(entry._1, new MutableDouble(entry._2));
7         }
8         else {
9             md.increment(entry._2);
10        }
11    }
12    return map;
13 }
```

---

### 23.7.9 Method toListOfString()

This method create a `List<String>`, where each element is an HDFS file represneting a biomarker file for a patient (each patient may have any number of biomarker files). The input to this method is an HDFS file, which contains all biomarker files involved for the Pearson correlation. To understand this method, I provide the following example used for a demo.

```
# 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
...

```

For this example, the result is a `List<String>`, which contains the following list of Strings:

```

/biomarker/input/b1
/biomarker/input/b2
...
/biomarker/input/b14
/biomarker/input/b15

```

### **Listing 23.12: Method `toListOfString()`**

```

1 static List<String> toListOfString(Path hdfsFile) throws Exception {
2     FSDataInputStream fis = null;
3     BufferedReader br = null;
4     FileSystem fs = FileSystem.get(new Configuration());
5     List<String> list = new ArrayList<String>();
6     try {
7         fis = fs.open(hdfsFile);
8         br = new BufferedReader(new InputStreamReader(fis));
9         String line = null;
10        while ((line = br.readLine()) != null) {
11            String value = line.trim();
12            list.add(value);
13        }
14    }
15    finally {
16        if (br != null) {

```

```
17         br.close();
18     }
19 }
20 return list;
21 }
```

---

### 23.7.10 Method readBiosets()

This method reads all biomarkers (files in HDFS) and generates the first RDD for further processing. Each RDD element is a record of biomarker file. You may further partition this RDD by using the following Spark API:

```
public JavaRDD<T> coalesce(int numPartitions)
// Description: return a new RDD that is reduced
// reduced into numPartitions partitions.
```

The question is how to select the right number of partitions for an RDD. The answer depends on the number of cluster nodes, number of cores per server, and the amount of RAM. There is no silver bullet formula for finding the the right number of partitions for an RDD (you may find and set this for your environment by some trial and error).

**Listing 23.13:** Method readBiosets()

```
1 static JavaRDD<String> readBiosets(JavaSparkContext ctx,
2                                     List<String> biosets) {
3     int size = biosets.size();
4     int counter = 0;
5     StringBuilder paths = new StringBuilder();
6     for (String biosetFile : biosets) {
7         counter++;
8         paths.append(biosetFile);
9         if (counter < size) {
10             paths.append(",");
11         }
12     }
13     JavaRDD<String> rdd = ctx.textFile(paths.toString());
14     return rdd;
15 }
```

---

### 23.7.11 STEP-1: handle input parameters

This step reads 3 input:

- The first parameter is a YARN's resource manager host name, which is used to create JavaSparkContext object
- The second parameter is a reference value, which can be any of {"r1", "r2", "r3", "r4"}
- The last parameter is an HDFS file, which contains the list of all biomarker files (persisted as HDFS files) required for Pearson correlation.

**Listing 23.14:** STEP-1: handle input parameters

```
1 // STEP-1: handle input parameters
2 if (args.length < 3) {
3     System.err.println("Usage: AllVersusAllCorrelation <resource-manager-host-name> " +
4                         "<reference> <all-bioset-ids-as-filename>");
5     System.err.println("Usage: OneVersusAllCorrelation myserver100 r2 " +
6                         " <all-bioset-ids-as-filename>");  

7     System.exit(1);
8 }
9 final String yarnResourceManagerHostName = args[0]; // myserver100
10 final String reference = args[1]; // {"r1", "r2", "r3", "r4"}
11 final String biomarkersFileName = args[2];
```

### 23.7.12 STEP-2: create a Spark context object

This step creates a JavaSparkContext object by using YARN's resource manager. An instance of JavaSparkContext can be created many different ways; the SparkUtil class provides two convenient methods to create an instance of JavaSparkContext object. SparkUtil's methods are:

```
public class SparkUtil {
    /**
     * Create a JavaSparkContext object from a given YARN's resource manager host
     *
     * @param yarnResourceManagerHost the YARN's resource manager host
     * @return a JavaSparkContext
     */
}
```

```

public static JavaSparkContext
    createJavaSparkContext(String yarnResourceManagerHost)
    throws Exception {...}

/**
 * Create a JavaSparkContext object from a given Spark's master URL
 *
 * @param sparkMasterURL Spark master URL as
 *           "spark://<spark-master-host-name>:7077"
 * @param description program description
 * @return a JavaSparkContext
 *
 */
public static JavaSparkContext
    createJavaSparkContext(String sparkMasterURL, String description)
    throws Exception {...}
}

```

**Listing 23.15: STEP-2: create a Spark context object**

```

1 // STEP-2: create a Spark context object
2 JavaSparkContext ctx = SparkUtil.createJavaSparkContext(yarnResourceManagerHostName);

```

### 23.7.13 STEP-3: create list of input files/biomarkers

This step reads an HDFS file, which contains all biomarker files required for Pearson correlation. The `biomarkersFileName` is a text HDFS file, which contains all input biomarker files (one biomarker file per line).

**Listing 23.16: STEP-3: create list of input files/biomarkers**

```

1 List<String> list = toListOfString(new Path(biomarkersFileName));

```

### 23.7.14 STEP-4: broadcast "reference" as global shared object

Since "reference" (values are in {"r1", "r2", "r3", "r4"}) value is used for filtering RDD elements, it should be broadcasted to all cluster nodes. In Spark, to broadcast (as a read only shared object) a data structure, we can use the `Broadcast` class. This is how we can use a `Broadcast` class to broadcast a shared data structure:

- To broadcast a shared data structure of type T

```
T t = <create-object-of-type-T>;
final Broadcast<T> broadcastT = ctx.broadcast(t);
```

- To read/access a broadcasted shared data structure of type T

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

In MapReduce/Hadoop, you may broadcast a shared data to `map()` or `reduce()` functions by using Hadoop's `Configuration` object. You may use `Configuration.set(...)` to broadcast and use `Configuration.get(...)` to read/access broadcasted objects. Spark's API is much richer than Hadoop's API since you may broadcast any type of data structures.

#### **Listing 23.17: STEP-4: broadcast reference as global shared object**

```
1 // broadcast reference which can be accessed from all cluster nodes
2 final Broadcast<String> REF = ctx.broadcast(reference); // "r2"
```

### 23.7.15 STEP-5: read all biomarkers from HDFS and create the first RDD

This step reads all biomarker files and create a single `JavaRDD<String>`, which can be partitioned further by the following method:

```
JavaRDD<T> coalesce(int numPartitions)
```

#### **Listing 23.18: STEP-5: read all biomarkers from HDFS and create the first RDD**

```
1 // STEP-5: read all biomarkers from HDFS and create the first RDD
2 JavaRDD<String> biosets = readBiosets(ctx, list);
3 biosets.saveAsTextFile("/output/1");
```

To debug, a sample output of this step is provided:

```
# 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
...
...
```

Note that the methods `JavaRDD.saveAsTextFile()` and `JavaPairRDD.saveAsTextFile()` are not required for our Spark solution (these methods saves the RDDs in HDFS). These methods are provided for debugging and understanding of the provided steps.

### 23.7.16 STEP-6: filter biomarkers by reference

Spark provides a very simple and powerful API for filtering RDDs. To filter elements, we just need to implement a `filter()` function: return `true` for the records you want to keep and return `false` for the records you want to toss out.

**Listing 23.19:** STEP-6: filter biomarkers by reference

```
1 // JavaRDD<T> filter(Function<T,Boolean> f)
2 // Return a new RDD containing only the elements that satisfy a predicate.
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; // do return these records
9     }
10 }
```

```

9         }
10        else {
11            return false; // do not retrun these records
12        }
13    }
14 });
15 filtered.saveAsTextFile("/output/2");

```

---

To debug, a sample output of this step is provided: note that only "r2" references will be present in the output (all other references {r1, r3, r4} are dropped from the resulting RDD).

```
# 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
...

```

### 23.7.17 STEP-7: create (Gene-ID, (Patient-ID, Gene-Value) pairs

This step implements a map() function, which transforms (note that from this point on "reference" is not needed for subsequent steps.)

<GeneID><,><reference><,><PatientID><,><BiomarkerValue>

into

(K, V) pair  
 where  
 K = GeneID  
 V = Tuple2<PatientID, BiomarkerValue>

**Listing 23.20:** STEP-7: create (Gene-ID, (Patient-ID, Gene-Value) pairs

```

1 // STEP-7: create (Gene-ID, (Patient-ID, Gene-Value) pairs
2 // PairMapFunction<T, K, V>
3 // T => Tuple2<K, V> = Tuple2<gene, Tuple2<patientID, value>>
4 //
5 JavaPairRDD<String,Tuple2<String,Double>> pairs = filtered.mapToPair(new PairFunction<
6     String,                                     // T
7     String,                                     // K = g1234 (as GeneID)
8     Tuple2<String,Double>                      // V = <patientID, value>
9   >() {
10   public Tuple2<String,Tuple2<String,Double>> call(String rec) {
11     String[] tokens = rec.split(",");
12     // tokens[0] = 1234
13     // tokens[1] = 2 (this is a ref in {"1", "2", "3", "4"})
14     // tokens[2] = patientID
15     // tokens[3] = value
16     Tuple2<String,Double> V =
17       new Tuple2<String,Double>(tokens[2], Double.valueOf(tokens[3]));
18     return new Tuple2<String,Tuple2<String,Double>>(tokens[0], V);
19   }
20 });
21 pairs.saveAsTextFile("/output/3");

```

To debug, a sample output of this step is provided:

```
# 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))
...
```

### 23.7.18 STEP-8: group by gene

This step groups data by GeneID. The result of this grouping is a new RDD as:

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

where

K = GeneID

V = Iterable<Tuple2<PatientID, BiomarkerValue>>

**Listing 23.21:** STEP-8: group by gene

```
1 // STEP-8: group by gene
2 JavaPairRDD<String, Iterable<Tuple2<String,Double>>> grouped = pairs.groupByKey();
3 grouped.saveAsTextFile("/output/4");
4 // grouped = (K, V)
5 // where
6 //       K = gene
7 //       V = Iterable<Tuple2<patientID,value>>
8 grouped.saveAsTextFile("/output/5");
```

To debug, a partial sample output of this step is provided: the output is formatted to fit the page.

```
# 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)])
...
```

### 23.7.19 STEP-9: create Cartesian product of all genes

To perform all-genes-vs-all-genes correlation of all genes, we have to create a Cartesian product of all genes. This is accomplished by `JavaPairRDD.cartesian()` function.

### Listing 23.22: STEP-9: create Cartesian product of all genes

```
1 // STEP-9: create Cartesian product of all genes
2 // <U> JavaPairRDD<T,U> cartesian(JavaRDDLike<U,?> other)
3 // Return the Cartesian product of this RDD and another one,
4 // that is, the RDD of all pairs of elements (a, b)
5 // where a is in this and b is in other.
6 JavaPairRDD< Tuple2<String, Iterable<Tuple2<String,Double>>>,
7           Tuple2<String, Iterable<Tuple2<String,Double>>>
8           > cart = grouped.cartesian(grouped);
9 cart.saveAsTextFile("/output/6");
10 // cart =
11 //      (g1, g1), (g1, g2), (g1, g3), (g1, g4)
12 //      (g2, g1), (g2, g2), (g2, g3), (g2, g4)
13 //      (g3, g1), (g3, g2), (g3, g3), (g3, g4)
14 //      (g4, g1), (g4, g2), (g4, g3), (g4, g4)
```

### 23.7.20 STEP-10: filter redundant pairs of genes

Let  $g_1$  and  $g_2$  be two genes; since Pearson correlation for  $(g_1, g_2)$  is the same as  $(g_2, g_1)$ , therefore, to reduce computation time, we will filter duplicate pairs. We only keep the gene pairs of  $(g_1, g_2)$  if and only if  $g_1 < g_2$ .

### Listing 23.23: STEP-10: filter redundant pairs of genes

```
1 // STEP-10: filter redundant pairs of genes
2 // filter it and keep the ones ( $G_a$ ,  $G_b$ ) if and only if ( $G_a < G_b$ ).
3 // after filtering, we will have:
4 // filtered2 =
5 //      (g1, g2), (g1, g3), (g1, g4)
6 //      (g2, g3), (g2, g4)
7 //      (g3, g4)
8 //
9 // JavaRDD<T> filter(Function<T,Boolean> f)
10 // Return a new RDD containing only the elements that satisfy a predicate.
11 JavaPairRDD< Tuple2<String, Iterable<Tuple2<String,Double>>>,
12           Tuple2<String, Iterable<Tuple2<String,Double>>> filtered2 =
13           cart.filter(new Function< Tuple2< Tuple2<String, Iterable< Tuple2<String,Double>>>, 
14           Tuple2<String, Iterable< Tuple2<String,Double>>>
15           >,
16           Boolean>() {
17           public Boolean call(Tuple2< Tuple2<String, Iterable< Tuple2<String,Double>>>,
18                               Tuple2<String, Iterable< Tuple2<String,Double>>> pair) {
19               // pair._1 = Tuple2<String, Iterable< Tuple2<String,Double>>>
20               // pair._2 = Tuple2<String, Iterable< Tuple2<String,Double>>>
21               if (smaller(pair._1._1, pair._2._1)) {
22                   return true; // do return these records
23               }
24               else {
25                   return false; // do not retrun these records
26               }
27           }
28       }
```

```
27     }
28 });
29 filtered2.saveAsTextFile("/output/7");
```

To debug this step, partial output of this step is displayed below. Output is formatted to fit the page.

```
# 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)])
)
...
...
```

### 23.7.21 STEP-11: calculate Pearson Correlation and p-value

Now, our generated pairs are aggregated with proper data and it is time to perform a Pearson correlation and find its associated p-value. If there is no sufficient data for correlation, then we return `Double.NaN` for correlation and p-value. The Pearson correlation is implemented by using Apache's Common Math3 package (provided in the following sections).

**Listing 23.24:** STEP-11: calculate Pearson Correlation and p-value

```

1 // STEP-11: calculate Pearson Correlation and p-value
2 // next iterate through all mappedValues
3 // JavaPairRDD<String, List<Tuple2<String,Double>>> mappedvalues
4 // create (K,V), where
5 //      K = Tuple2<String, String>(g1, g2)
6 //      V = Tuple2<Double, Double>(corr, pvalue)
7 //
8 JavaPairRDD<Tuple2<String, String>, Tuple2<Double, Double>> finalresult =
9         filtered2.mapToPair(new PairFunction<
10             Tuple2<Tuple2<String, Iterable<Tuple2<String, Double>>>,
11             Tuple2<String, Iterable<Tuple2<String, Double>>>>, // input
12             Tuple2<String, String>, // K
13             Tuple2<Double, Double> // V
14         >() {
15     public Tuple2<Tuple2<String, String>, Tuple2<Double, Double>>
16         call(Tuple2<Tuple2<String, Iterable<Tuple2<String, Double>>>,
17               Tuple2<String, Iterable<Tuple2<String, Double>>> t) {
18         Tuple2<String, Iterable<Tuple2<String, Double>>> g1 = t._1;
19         Tuple2<String, Iterable<Tuple2<String, Double>>> g2 = t._2;
20         //
21         Map<String, MutableDouble> g1map = toMap(g1._2);
22         Map<String, MutableDouble> g2map = toMap(g2._2);
23         // now perform a correlation(one, other)
24         // make sure we order the values accordingly by patientID
25         // each patientID may have one or more values
26         List<Double> x = new ArrayList<Double>();
27         List<Double> y = new ArrayList<Double>();
28         for (Map.Entry<String, MutableDouble> g1Entry : g1map.entrySet()) {
29             String g1PatientID = g1Entry.getKey();
30             MutableDouble g2MD = g2map.get(g1PatientID);
31             if (g2MD != null) {
32                 // both one and other for patientID have values
33                 x.add(g1Entry.getValue().avg());
34                 y.add(g2MD.avg());
35             }
36         }
37         System.out.println("x="+x);
38         System.out.println("y="+y);
39         // K = pair of genes
40         Tuple2<String, String> K = new Tuple2<String, String>(g1._1, g2._1);
41         if (x.size() < 3) {
42             // not enough data to perform correlation
43             return new Tuple2<Tuple2<String, String>, Tuple2<Double, Double>>
44                 (K, new Tuple2<Double, Double>(Double.NaN, Double.NaN));
45         }
46     }
47     else {
48         // Pearson
49         double correlation = Pearson.getCorrelation(x, y);
50         double pvalue = Pearson.getPValue(correlation, (double) x.size() );
51         return new Tuple2<Tuple2<String, String>, Tuple2<Double, Double>>
52             (K, new Tuple2<Double, Double>(correlation, pvalue));
53     }
54 }
55 }
56 });

```

```
57 finalresult.saveAsTextFile("/output/corr");
```

To debug, a complete output of final step is provided. You should note that if two genes do have not enough data to correlate, then for a lack of proper value, we have emitted 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))
```

### 23.7.22 Pearson Class

This is a wrapper class, which provides two methods: `getCorrelation()` and `getPvalue()`. The underlying implementation uses Apache's Commons

Math3 package<sup>4</sup>.

#### Listing 23.25: Pearson Wrapper Class

```
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  * Class for calculating the Pearson Correlation Coefficient and p-value
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 }
```

### 23.7.23 Test Pearson Class

Here we test our wrapper class.

---

<sup>4</sup><http://commons.apache.org/proper/commons-math/>

### **Listing 23.26:** Pearson Wrapper Class

```
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      * test/debug
11     */
12    public static void test(String[] args) {
13        // index          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 }
```

### **23.7.24 Pearson Correlation Using R**

Here we show how to use R language for Pearson correlation. You may use this to compare Java implementation with R.

### **Listing 23.27:** Pearson Correlation using R Language

```
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
14 cor
15 0.9012503
```

### 23.7.25 YARN Script to Run Spark Program

To run our Spark program, AllVersusAllCorrelation, the following shell script is used to run it on YARN.

**Listing 23.28:** YARN Script to Run Spark Program

```
1 # cat run_all_vs_all.sh
2 #!/bin/bash
3 export SPARK_HOME=/usr/local/spark-1.0.0
4 # app jar:
5 export MY_JAR=/home/mahmoud/mp.jar
6 prog=AllVersusAllCorrelation
7 yarnResourceManagerHostName=myserver100
8 reference=r2
9 biomarkers=/home/hadoop/spark_mahmoud_examples/biomarkers.txt
10 $SPARK_HOME/bin/spark-submit --class $prog \
11     --master yarn-cluster \
12     --num-executors 12 \
13     --driver-memory 3g \
14     --executor-memory 7g \
15     --executor-cores 12 \
16     $MY_JAR $yarnResourceManagerHostName $reference $biomarkers
```

## 23.8 Spearman Correlation

To calculate Spearman correlation instead of Pearson, you just need to replace the following two lines:

```
// Pearson
double correlation = Pearson.getCorrelation(x, y);
double pvalue = Pearson.getPvalue(correlation, (double) x.size() );
```

with

```
// Spearman
double correlation = Spearman.getCorrelation(x, y);
double pvalue = Spearman.getPvalue(correlation, (double) x.size() );
```

and use the Spearman wrapper class provided below.

### 23.8.1 Spearman Correlation Wrapper Class

This is a wrapper class, which provides two methods: `getCorrelation()` and `getPvalue()`. The underlying implementation uses Apache's Commons Math3 package.

**Listing 23.29:** Spearman Wrapper Class

```
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  * Class for calculating Spearman's Rank Correlation between two vectors.
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 }
```

### 23.8.2 Test Spearman Correlation Wrapper Class

This class tests Spearman's wrapper class.

### Listing 23.30: TestSpearman Class

```
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 }
```

# Chapter 24

## DNA Base Count

### 24.1 Introduction

The purpose of this chapter is to count DNA<sup>1</sup> bases. Human DNA's code is written in only four letters: A, C, T and G; and when we cannot recognize the code, we label it as N. The meaning of this DNA code lies in the sequence of the letters A, T, C and G in the same way that the meaning of a word lies in the sequence of English alphabet letters (A-Z).

The goal of this chapter is to find frequencies (or percentages) of A, T, C, G, and N (anything other than A, T, C, or G) in a given set of DNA sequences. Another goal of this chapter is to provide custom record readers for Hadoop's input files. What does ATCG stand for when talking about DNA? It does refer to 4 of the nitrogenous bases associated with DNA:

- A = Adenine
- T = Thymine
- C = Cytosine

---

<sup>1</sup>"Deoxyribonucleic acid (DNA) is a molecule that encodes the genetic instructions used in the development and functioning of all known living organisms and many viruses. Along with RNA and proteins, DNA is one of the three major macromolecules essential for all known forms of life. Genetic information is encoded as a sequence of nucleotides (guanine, adenine, thymine, and cytosine) recorded using the letters G, A, T, and C." (source: <http://en.wikipedia.org/wiki/DNA>)

- G = Guanine

For example, ACGGGTACGAAT is a very small DNA sequence. DNA sequences can be huge<sup>2</sup>. DNA base counting for our example will generate the following table:

Table 24.1: DNA Base Count Example

Base	Count
a	4
t	2
c	2
g	4
n	0

DNA sequences can be represented in many different formats including FASTA and FASTQ popular text-based formats. Our solution will handle FASTA and FASTQ formats. Note that Hadoop's default record reader reads records line-by-line and therefore we can not use the Hadoop's default record reader for reading files in FASTQ format. We do need to plugin (inject custom record readers) `FastaInputFormat` (to read fasta file formats) and `FastqInputFormat` (to read fastq file formats).

## 24.2 FASTA Format

A sequence file in FASTA format can contain several sequences. Each sequence in FASTA format begins with a single-line description, followed by one or many lines of sequence data. The description line must begin with a greater-than (">") symbol in the first column.

---

<sup>2</sup>"The haploid human genome (contained in egg and sperm cells) consists of three billion DNA base pairs, while the diploid genome (found in somatic cells) has twice the DNA content." source: [http://en.wikipedia.org/wiki/Human\\_genome](http://en.wikipedia.org/wiki/Human_genome)

### 24.2.1 FASTA Format Example

An example sequence in FASTA format is (this file has 4 sequences and case (upper-case or lower-case) of characters are irrelevant):

```
cat test.fasta
>seq1
cGTAAccaataaaaaacaagcttaacctaattc
>seq2
agcttagTTTGGatctggccgggg
>seq3
gcggatttactcCCCCCAAAAANaggggagagcccagataaatggagtctgtgcgtccaca
gaattcgcacca
AATAAACCTCACCCAT
agagcccagaatttactcCCC
>seq4
gcggatttactcaggggagagccagGGataaatggagtctgtgcgtccaca
gaattcgcacca
```

## 24.3 FASTQ Format

FASTQ<sup>3</sup> format is a text-based format for storing both a biological sequence (usually nucleotide sequence) and its corresponding quality scores. Both the sequence letter and quality score are encoded with a single ASCII character for brevity. A FASTQ file normally uses four lines per sequence defined as:

- Line 1 begins with a '@' character and is followed by a sequence identifier and an optional description
- Line 2 is the raw sequence letters (A, T, C, G, ...)
- Line 3 begins with a '+' character and is optionally followed by the same sequence identifier (and any description) again.
- Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

---

<sup>3</sup>source: [http://en.wikipedia.org/wiki/FASTQ\\_format](http://en.wikipedia.org/wiki/FASTQ_format)

### 24.3.1 FASTQ Format Example

A FASTQ file containing a single sequence might look like this:

```
@SEQ_ID
GATTGGGGTTCAAAGCAGTATCGATCAAATAGTAAATCCATTGTTCAACTCACAGTT
+
! ''*((((***+))%%%++) (%%%).1***-+*'')**55CCF>>>>CCCCCCC65
```

## 24.4 MapReduce Solution: FASTA Format

For DNA base counting, we use a mapper, a reducer, and custom FASTA format reader (FASTA reader is accomplished by injecting `FastaInputFormat` class to `Job.setInputFormatClass()` method).

### 24.4.1 Reading FASTA Files

How did we instruct MapReduce/Hadoop framework to read FASTA format data? The default reader in Hadoop reads input records line-by-line, but a FASTA record may span many (one or more) lines. Hadoop provides a plug-in framework for "input format" and "record reader". We developed two custom plug-in classes `FastaInputFormat` and `FastaRecordReader` to enable us to read FASTA format (one or more lines is a single record). In Hadoop, default input format is provided by the `TextInputFormat` class. Our custom class `FastaInputFormat` extends this class and overrides two methods: `createRecordReader()` and `isSplitable()`. The override `createRecordReader()` method returns a new custom "record reader", called `FastaRecordReader`.

### 24.4.2 MapReduce Solution: map()

To find count of DNA bases, the `map()` will get a record of FASTA file which can be one or more lines of DNA sequences. The `map()` will then tokenize each line and count the bases. To make our mapper more efficient, we will not `emit(letter, 1)` for each letter, but we will emit a total for each base by using a hash table (`Map < Character, Long >`). Finally, by using the Hadoop's `cleanup()` method, we will iterate the hash table and `emit(letter, countOfLetter)`. The hash table is a very efficient solution, since the number of keys are limited to the very small number of DNA letters.

Furthermore this smart map() algorithm reduces the network traffic by just emitting the minimum number of (*letter*, *countOfLetter*) to the MapReduce framework. The mapper class, **FastaCountBaseMapper**, will have the following structure:

Listing 24.1: FastaCountBaseMapper Class

```
public class FastaCountBaseMapper ... {  
  
    Map<Character, Long> dnaBasesCounter = null;  
  
    // this function is called once at the beginning of the map task.  
    setup() {  
        dnaBasesCounter = new HashMap<Character, Long>();  
    }  
  
    map(Object key, String value) {  
        ...  
    }  
  
    // called once at the end of the map task.  
    cleanup() {  
        // now iterate the baseCounter and emit <key, value>  
        for (Map.Entry<Character, Long> entry : dnaBaseCounter.entrySet())  
        {  
            emit(entry.getKey(), entry.getValue());  
        }  
    }  
}
```

The **map()** function is listed below:

Listing 24.2: DNA Base Count map()

```
/**  
 * @param key is the key generated by Hadoop (ignored here)  
 * @param value is one line of input for a given document  
 */  
map(Object key, String value) {  
    // fasta is a string comprised of many DNA-seq lines  
    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 {
        v++;
        dnaBaseCounter.put(c, v);
    }
}
}
```

---

#### 24.4.3 MapReduce Solution: reduce()

Since the number of DNA alphabet is very limited, we might just set the number of reducers to 5 (one for every DNA letter). The reducer is almost identical to the word-count reducer. It just sums up the counters for each DNA letter.

Listing 24.3: DNA Base Count reduce() Function

```
/**
 * @param key is the unique DNA letter generated by mapper
 * @param value is a list of integers (partial count of a unique word)
 */
reduce(String key, List<long> value) {
    long sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}
```

---

## 24.5 Hadoop Implementation: FASTA Format

### 24.5.1 Hadoop Sample Run

```
# ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Note: src/CountBaseMapper.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
added manifest
adding: BaseComparator.class(in = 622) (out= 374)(deflated 39%)
adding: BaseComparator.java(in = 530) (out= 242)(deflated 54%)
...
adding: FastaInputFormat.class(in = 4744) (out= 1976)(deflated 58%)
adding: FastaInputFormat.java(in = 3908) (out= 1191)(deflated 69%)
adding: FastaRecordReader.class(in = 4683) (out= 2181)(deflated 53%)
adding: FastaRecordReader.java(in = 4060) (out= 1373)(deflated 66%)
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 09:16:22 INFO input.FileInputFormat: Total input paths to process : 1
13/03/15 09:16:23 INFO mapred.JobClient: Running job: job_201303150852_0002
13/03/15 09:16:24 INFO mapred.JobClient: map 0% reduce 0%
13/03/15 09:16:38 INFO mapred.JobClient: map 14% reduce 0%
...
13/03/15 09:17:53 INFO mapred.JobClient: map 100% reduce 100%
13/03/15 09:17:58 INFO mapred.JobClient: Job complete: job_201303150852_0002
...
13/03/15 09:17:58 INFO mapred.JobClient: Map-Reduce Framework
13/03/15 09:17:58 INFO mapred.JobClient: Map output materialized bytes=948
13/03/15 09:17:58 INFO mapred.JobClient: Map input records=7
13/03/15 09:17:58 INFO mapred.JobClient: Reduce shuffle bytes=828
13/03/15 09:17:58 INFO mapred.JobClient: Spilled Records=88
13/03/15 09:17:58 INFO mapred.JobClient: Map output bytes=440
13/03/15 09:17:58 INFO mapred.JobClient: Total committed heap usage (bytes)=21
13/03/15 09:17:58 INFO mapred.JobClient: Combine input records=0
```

```

13/03/15 09:17:58 INFO mapred.JobClient: SPLIT_RAW_BYTES=868
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: Combine output records=0
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

# 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

```

### 24.5.2 What If 1

What if you wanted to send "a" and "A" to the same reducer and wanted to sum up "a"'s with "A"'s (since they are the same DNA letter). There are at least two ways to accomplish this:

- Option-1: Convert all input of map() to lowercase letters. This can be done very easily in the mapper code (map() function):

---

```

Replace the following line
String fasta = value.toString();
with this one:
String fasta = value.toString().toLowerCase();

```

---

- Option-2: Provide a outcome sorter, which will be injected to the MapReduce framework. We can define the comparator that controls how the keys are sorted before they are passed to the reducer. This is how we do it:

```
job.setSortComparatorClass(BaseComparator.class);
```

And this is how we implement our comparator:

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

public class BaseComparator extends WritableComparator {
protected BaseComparator(){
super(Text.class, true);
}

@Override
public int compare(WritableComparable w1, WritableComparable w2) {
Text t1 = (Text) w1;
Text t2 = (Text) w2;
String s1 = t1.toString().toUpperCase();
String s2 = t2.toString().toUpperCase();
int cmp = s1.compareTo(s2);
return cmp;
}
}
```

In Hadoop, for writing a "custom comparator," you just need to extend the `org.apache.hadoop.io.WritableComparator` class and implement the `compare()` method. Here, the "custom comparator" (`BaseComparator` class) sorts the keys ("a", "A", "t", "T", "c", "C", "g", "G", "n", "N") ignoring cases so that the same DNA base with different cases will be next to each other. By default, classes injected to methods `setSortComparatorClass()` and `setGroupingComparatorClass()` use the same comparator, for example, if we set `setSortComparatorClass()` and leave `setGroupingComparatorClass()` unsetted, `setGroupingComparatorClass()` will use the same comparator as we set for `setSortComparatorClass()`, but the opposite case (vice versa) is not true. You can use this example to test and verify it.

### 24.5.3 What If 2

What if we wanted to send all of our keys just to only 5 reducers (one reducer per DNA letter: A, T, C, G, N). The answer is to provide a custom partitioner. The default partitioner in Hadoop is the `HashPartitioner`, which hashes a mapper generated key to determine which partition (which reducer) the (key, value) belongs in. The number of partition is then equal to the number of reduce tasks for the job. So the main question is when a `map()` generates a (key, value), does it have to be sent to a reducer? If so, which one? We can use `Job.setPartitionerClass()` method to enable our custom partitioner. This is how: we send all keys with letter a/A to reducer 1, b/B to reducer 2 , and so on.

**Listing 24.1:** Custom Partitioner

```
1 import org.apache.hadoop.mapreduce.Partitioner;
2
3 /**
4  * Partition keys by bases{A,T,G,C,a,t,g,c}. */
5 public class BasePartitioner<K, V> extends Partitioner<K, V> {
6     public int getPartition(K key, V value, int numReduceTasks) {
7         String base = key.toString();
8         if (base.compareToIgnoreCase("A") == 0) {
9             return 0;
10        }
11        else if (base.compareToIgnoreCase("C") == 0) {
12            return 1;
13        }
14        else if (base.compareToIgnoreCase("G") == 0) {
15            return 2;
16        }
17        else if (base.compareToIgnoreCase("T") == 0) {
18            return 3;
19        }
20        else{
21            return 4;
22        }
23    }
}
```

Then in our driver class inject our custom classes as:

**Listing 24.2: DNA Base Count Driver**

```
1 public class CountBaseDriver extends Configured implements Tool {  
2  
3     public int run(String[] args) throws Exception {  
4         Job job = new Job(getConf(), "count-dns-bases");  
5         job.setJarByClass(CountBaseMapper.class);  
6         job.setMapperClass(CountBaseMapper.class);  
7         job.setReducerClass(CountBaseReducer.class);  
8         job.setNumReduceTasks(5);  
9         //job.setCombinerClass(CountBaseCombiner.class);  
10        job.setInputFormatClass(FastaInputFormat.class);  
11        job.setPartitionerClass(BasePartitioner.class);  
12        job.setSortComparatorClass(BaseComparator.class);  
13        job.setGroupingComparatorClass(BaseComparator.class);  
14        job.setOutputKeyClass(Text.class);  
15        job.setOutputValueClass(LongWritable.class);  
16        FileInputFormat.addInputPath(job, new Path(args[0]));  
17        FileOutputFormat.setOutputPath(job, new Path(args[1]));  
18  
19        boolean status = job.waitForCompletion(true);  
20        theLogger.info("run(): status="+status);  
21        return status ? 0 : 1;  
22    }  
23    ...  
24 }
```

Now, when we run our job, we will have:

```
mahmoud@mahmouds-macbook:~/zmp/map_reduce_book/hadooptests/dna-base-count# ./run.s  
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home  
Note: src/CountBaseMapper.java uses unchecked or unsafe operations.  
Note: Recompile with -Xlint:unchecked for details.  
added manifest  
adding: BaseComparator.class(in = 622) (out= 374)(deflated 39%)  
adding: BaseComparator.java(in = 530) (out= 242)(deflated 54%)  
adding: BasePartitioner.class(in = 758) (out= 463)(deflated 38%)
```

```

adding: BasePartitioner.java(in = 553) (out= 265)(deflated 52%)
...
adding: FastaInputFormat.class(in = 4744) (out= 1976)(deflated 58%)
adding: FastaInputFormat.java(in = 3908) (out= 1191)(deflated 69%)
adding: FastaRecordReader.class(in = 4683) (out= 2181)(deflated 53%)
adding: FastaRecordReader.java(in = 4060) (out= 1373)(deflated 66%)
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 09:44:43 INFO mapred.JobClient: map 14% 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: Counters: 25
...
13/03/15 09:46:03 INFO mapred.JobClient: Map-Reduce Framework
13/03/15 09:46:03 INFO mapred.JobClient: Map output materialized bytes=948
13/03/15 09:46:03 INFO mapred.JobClient: Map input records=7
13/03/15 09:46:03 INFO mapred.JobClient: Reduce shuffle bytes=828
13/03/15 09:46:03 INFO mapred.JobClient: Spilled Records=88
13/03/15 09:46:03 INFO mapred.JobClient: Map output bytes=440
13/03/15 09:46:03 INFO mapred.JobClient: Total committed heap usage (bytes)=21
13/03/15 09:46:03 INFO mapred.JobClient: Combine input records=0
13/03/15 09:46:03 INFO mapred.JobClient: SPLIT_RAW_BYTES=868
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: Combine output records=0
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

```

Now we have only 5 reducers:

```
$ hadoop fs -ls /dna-base-count/output/p*
```

```

-rw-r--r-- 1 mahmoud staff 6 2013-03-15 10:06 /dna-base-count/output/pa
-rw-r--r-- 1 mahmoud staff 5 2013-03-15 10:06 /dna-base-count/output/pa
-rw-r--r-- 1 mahmoud staff 5 2013-03-15 10:06 /dna-base-count/output/pa
-rw-r--r-- 1 mahmoud staff 5 2013-03-15 10:06 /dna-base-count/output/pa
-rw-r--r-- 1 mahmoud staff 4 2013-03-15 10:06 /dna-base-count/output/pa

$ hadoop fs -cat /dna-base-count/output/p*
a 117
c 89
g 90
t 87
n 5

```

## 24.6 MapReduce Solution: FASTQ Format

For DNA base counting, we use a mapper and a reducer. This solution is very similar to the FASTA solution, with the difference of injecting a different class for handling input format. For FASTQ format, we will inject `FastqInputFormat` class:

```
job.setInputFormatClass(FastqInputFormat.class);
```

Using `FastqInputFormat`, the mapper will get a value as `Text` object, which has the following format (the line delimiter is ",;,"):

```
<line-1><,;,><line-2><,;,><line-3><,;,><line-4>
```

We have 4 lines per mapper value, since each FASTQ record corresponds to lines of data. The mapper class for FASTQ format will be different from FASTA classes, due to the fact that input formats are different, but the reducer class will not change.

### 24.6.1 MapReduce Solution: map()

To find count of DNA bases, the `map()` will get a record of FASTQ file, which is exactly 4 lines of text and only the second line contains a DNA sequence. The `map()` will then tokenize the second line which contains DNA sequences and counts the bases. To make our mapper more efficient, we will not `emit(letter, 1)` for each letter, but we will emit total of

each base by using a hash table ( $Map < Character, Long >$ ). Finally, by using the Hadoop's `cleanup()` method, we will iterate the hash table and `emit(letter, countOfLetter)`. The hash table is a very efficient solution, since the number of keys are limited to the very small number of DNA letters. Furthermore this smart map() algorithm reduces the network traffic by just emitting the minimum number of `(letter, countOfLetter)` to the MapReduce framework.

Listing 24.4: DNA Base Count map() using FASTQ Format

```
Map<Character, Long> dnaBasesCounter = null;
/**
 * this function is called once at the beginning of the map task.
 */
setup() {
    dnaBasesCounter = new HashMap<Character, Long>();
}

/**
 * @param key is the key generated by Hadoop (ignored here)
 * @param value is one line of input for a given document
 */
map(Object key, String value) {
    String fastq = value.toString();
    // 4 lines lines are separated by a special delimiter: ";;,"
    String[] lines = fastq.split(";;,");
    // 2nd line = lines[1] = the DNA sequence
    char[] array = lines[1].toCharArray();
    for(char c : array){
        Long v = dnaBaseCounter.get(c);
        if (v == null) {
            dnaBaseCounter.put(c, 1);
        }
        else {
            v++;
            dnaBaseCounter.put(c, v);
        }
    }
}

/**
 * Called once at the end of the map task.
 */
```

```

cleanup() {
    // now iterate the baseCounter and emit <key, value>
    for (Map.Entry<Character, Long> entry : dnaBaseCounter.entrySet()) {
        emit(entry.getKey(), entry.getValue());
    }
}

```

---

### 24.6.2 MapReduce Solution: reduce()

Since the number of DNA alphabet is very limited, we might just set the number of reducers to 5 (one for every DNA letter). The reducer is almost identical to the word-count reducer. It just sums up the counters for each DNA letter.

Listing 24.5: DNA Base Count reduce() Function using FASTQ Format

```

/**
 * @param key is the unique DNA letter generated by mapper
 * @param value is a list of integers (partial count of a unique word)
 */
reduce(String key, List<long> value) {
    long sum = 0;
    for (int count : value) {
        sum += count;
    }
    emit(key, sum);
}

```

---

## 24.7 Hadoop Implementation

The Hadoop implementation consists of the following classes:

- BaseComparator.java
- BasePartitioner.java
- FastqCountBaseDriver.java
- FastqCountBaseMapper.java
- FastqCountBaseReducer.java
- FastqInputFormat.java
- FastqRecordReader.java

### 24.7.1 Sample Run of Hadoop Implementation

```
mahmoud@Mahmouds-MacBook:~/zmp/map_reduce_book/hadooptests/dna-base-count# ./run_f
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Note: Some input files use unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
added manifest
adding: BaseComparator.class(in = 622) (out= 374)(deflated 39%)
...
adding: FastqRecordReader.java(in = 3297) (out= 1046)(deflated 68%)
Deleted hdfs://localhost:9000/dna-base-count/fastq/output
13/03/19 09:24:45 INFO FastqCountBaseDriver: inputDir=/dna-base-count/fastq/input
13/03/19 09:24:45 INFO FastqCountBaseDriver: outputDir=/dna-base-count/fastq/output
13/03/19 09:24:45 INFO FastqCountBaseDriver: run(): input args[0]=/dna-base-count
13/03/19 09:24:45 INFO FastqCountBaseDriver: run(): output args[1]=/dna-base-count
13/03/19 09:24:46 INFO input.FileInputFormat: Total input paths to process : 1
13/03/19 09:24:46 INFO mapred.JobClient: Running job: job_201303190908_0003
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: Job complete: job_201303190908_0003
...
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
mahmoud@Mahmouds-MacBook:~/zmp/map_reduce_book/hadooptests/dna-base-count# hadoop
Found 12 items
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:26 /dna-base-count/fastq/out
drwxr-xr-x - mahmoud staff          0 2013-03-19 09:24 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          5 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          5 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff          0 2013-03-19 09:25 /dna-base-count/fastq/out
```

```

-rw-r--r-- 1 mahmoud staff      5 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff      5 2013-03-19 09:25 /dna-base-count/fastq/out
-rw-r--r-- 1 mahmoud staff      0 2013-03-19 09:25 /dna-base-count/fastq/out
mahnoud@Mahmouds-MacBook:~/zmp/map_reduce_book/hadooptests/dna-base-count# hadoop
c 36
g 22
t 67
a 55

```

### 24.7.2 Reading FASTQ Files

How did we instruct MapReduce/Hadoop framework to read FASTQ format data? The default reader in Hadoop reads input records line-by-line, but for FASTQ format we need to read 4-lines-by-4-lines (each 4 lines is a FASTQ record). Hadoop provides a plug-in framework for "input format" and "record reader". We developed two custom plug-in classes `FastqInputFormat` and `FastqRecordReader` to enable us to read FASTQ format (each 4 lines is a single record). In Hadoop, default input format is provided by the `TextInputFormat` class. Our custom class `FastqInputFormat` extends this class and overrides two methods: `createRecordReader()` and `isSplitable()`. The override `createRecordReader()` method returns a new custom "record reader", called `FastqRecordReader`. This is how `FastqInputFormat` class is implemented:

Listing 24.6: FastqInputFormat Custom Class

```

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;

/**
 * This class define an InputFormat for FASTQ
 * files for the Hadoop MapReduce framework.

```

```

*/
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;
    }
}

```

---

The other custom class is the `FastqRecordReader`, which extends the `RecordReader<LongWritable, Text>` class. The `FastqRecordReader` class breaks the input data into (key, value) pairs for input to the Mapper. In this case, `FastqRecordReader` create values (as a `String` object), which can hold the 4 lines of a FASTQ record:

Listing 24.7: FastqRecordReader Custom Class

```

import ...
import org.apache.hadoop.mapreduce.RecordReader;

/**
 * This class define a RecordReader for FASTQ files
 * for the Hadoop MapReduce framework.
 */
public class FastqRecordReader extends RecordReader<LongWritable, Text> {
    ...
}

```

---

# Chapter 25

## RNA-Sequencing

### 25.1 Introduction

In recent years, RNA sequencing has revolutionized the exploration of gene expression. The improvements of RNA-sequencing methods enable researchers to rapid profile and investigate the transcriptome.

RNA stands for ribonucleic acid (*RiboNucleic Acid*). Dr. Ananya Mandel<sup>1</sup> defines RNA as: "It is an important molecule with long chains of nucleotides. A nucleotide contains a nitrogenous base, a ribose sugar, and a phosphate. Just like DNA, RNA is vital for living beings." What is the main function of RNA? The main function of RNA is to transfer the genetic code need for the creation of proteins from the nucleus to the ribosome. "This process prevents the DNA from having to leave the nucleus. This keeps the DNA and genetic code protected from damage. Without RNA, proteins could never be made."<sup>2</sup>

Our main focus in this chapter is to provide a complete MapReduce solution for a computational pipeline for analyzing RNA-Sequencing (RNA-Seq) data for differential gene expression. In our implementation, we will utilize two open-source packages:

- TopHat<sup>3</sup> (TopHat is a fast splice junction mapper for RNA-Seq reads. It aligns RNA-Seq reads to mammalian-sized genomes using the ul-

<sup>1</sup><http://www.news-medical.net/health/What-is-RNA.aspx>

<sup>2</sup><http://www.news-medical.net/health/What-is-RNA.aspx>

<sup>3</sup><http://tophat.cbcn.umd.edu/>

tra high-throughput short read aligner Bowtie, and then analyzes the mapping results to identify splice junctions between exons.

- Cufflinks<sup>4</sup> Cufflinks assembles transcripts, estimates their abundances, and tests for differential expression and regulation in RNA-Seq samples. It accepts aligned RNA-Seq reads and assembles the alignments into a parsimonious set of transcripts. Cufflinks then estimate the relative abundances of these transcripts based on how many reads support each one, taking into account biases in library preparation protocols.

## 25.2 Data Size and Format

RNA-Seq data can be presented in FASTA, FASTQ, and many other formats (like BAM format). The TopHat package handles FASTA and FASTQ data formats. How big is an RNA-Seq data? Each sample of RNA-Seq data can be from 20GB to 300GB of data (a single analysis of RNA-Seq can have up to 500GB of data).

## 25.3 MapReduce Solution

RNA-Sequencing pipeline includes the following main steps:

- Input data validation (quality control of input data such as FASTQ files)
- Alignment (mapping of short reads to the reference genome)
- Assemble transcripts (using Cufflink and Cuffdiff tools)

RNA-Seq workflow is presented below:

### 25.3.1 Input data validation

This step validates the format of FASTQ files. With validation, you want to make sure that quality of input files are guaranteed. Input data validation tools help to provide a way to do some quality control checks on raw sequence

---

<sup>4</sup><http://cufflinks.cbcn.umd.edu/>

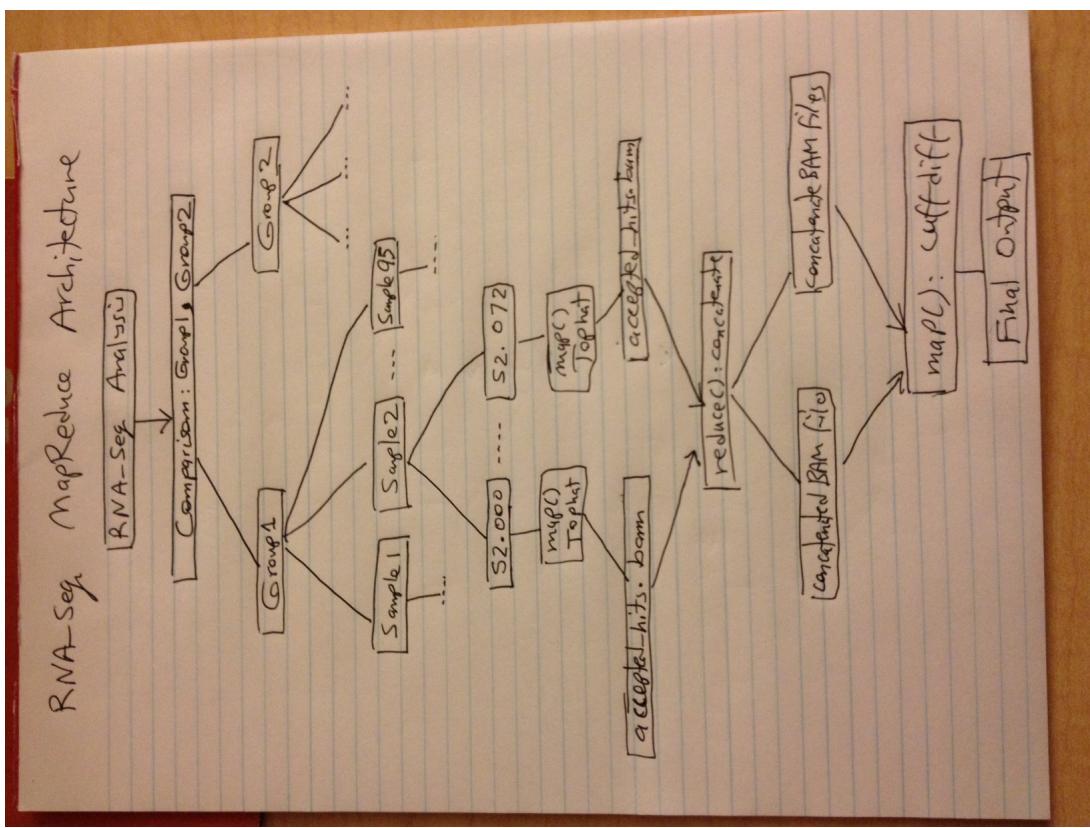


Figure 25.1: RNA-Seq WorkFlow

data (for example in FASTQ file format) coming from high throughput sequencing pipelines.

There are lots of open-source tools for input data validation. For example, for FASTQ validation you may use the following open-source tools:

- FastQValidator (<http://genome.sph.umich.edu/wiki/FastQValidator>)
- FastQC (<http://www.bioinformatics.babraham.ac.uk/projects/fastqc>)

The input data validation step is very simple and straightforward and we will not cover it here. Our focus will be on core of RNA-Sequencing, on mapping/alignment and tests for differential expression and regulation in RNA-Seq samples.

## 25.4 MapReduce Algorithms for RNA-Sequencing

Before we delve into MapReduce Algorithms for RNA-Sequencing, we need to understand how data/samples are used in RNA-Seq analysis. To perform an RNA-Seq analysis, we do need to compare two groups of RNA-Seq data samples (this is a single comparison; an RNA-Seq analysis may have one or more comparisons):

- Group-1: set of samples, where each sample may have one or more FASTQ files. For example, this group might be set of "normal" samples.
- Group-2: set of samples, where each sample may have one or more FASTQ files. For example, this group might be set of "cancer" (disease) samples.

The goal is to find significant changes in transcript expression, splicing, and promoter use between samples of Group-1 and Group-2 (typically, we will use Cufflink's Cuffdiff program to perform RNA-Seq analysis between two groups). Typically, an input to an RNA-Seq has one or more comparisons. Each comparison has exactly two groups (Group-1 and Group-2, which are discussed above).

For example, the following is a sample input for RNA-Seq analysis. This analysis has 3 comparisons (Comparison-1, Comparison-2, and Comparison-3) and each comparison has exactly two groups: Group-1 and Group-2.

Comparison-1's Group-1 has 4 samples and Group-2 has 3 samples (sample data for Comparison-2 and Comparison-3 are not shown here):

#### **Comparison-1:**

Group-1: {Sample-1, Sample-2, Sample-3, Sample-4}

Group-2: {Sample-7, Sample-8, sample-9}

where

Sample-1 = {file11.fastq, file12.fastq, file13.fastq}

Sample-2 = {file21.fastq, file22.fastq}

Sample-3 = {file31.fastq, file32.fastq, file33.fastq, file34.fastq}

Sample-4 = {file41.fastq}

Sample-7 = {file71.fastq, file72.fastq}

Sample-8 = {file81.fastq, file82.fastq, file83.fastq}

Sample-9 = {file91.fastq, file92.fastq, file93.fastq}

#### **Comparison-2:**

Group-1: {...}

Group-2: {...}

#### **Comparison-3:**

Group-1: {...}

Group-2: {...}

RNA-Seq data structures are presented below:

Input data for RNA-Seq can easily be represented by the following Java class and data structures:

```
import java.util.Arrays;
import java.util.Map;
import java.util.HashMap;
import java.util.List;
```

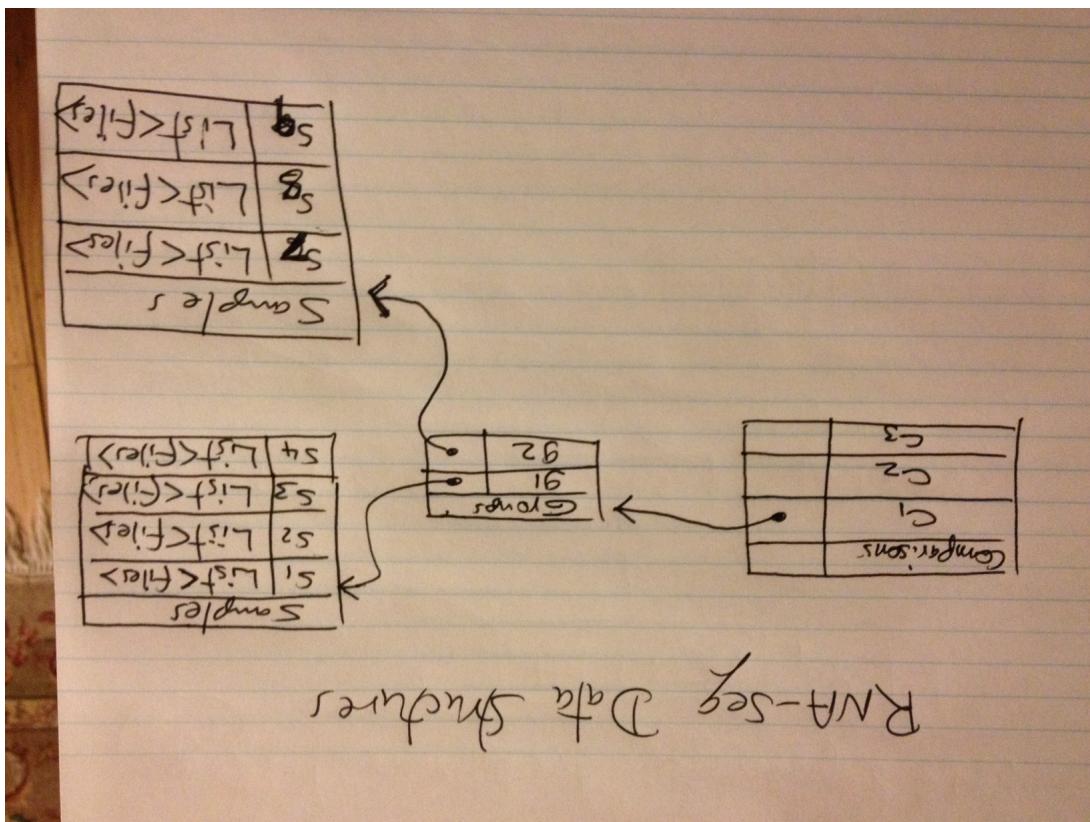


Figure 25.2: RNA-Seq Data Structures

```

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;
    }
}

```

Now we may use the `Comparison` class to create proper RNA-Seq data structures:

```

// create all 3 comparisons:
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);

// prepare Group-1 for comparison_1
Map<String, List<String>> group1 = new HashMap<String, List<String>>()
group1.put("Sample-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")));

// prepare Group-2 for comparison_1
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>(
    Arrays.asList("file81.fastq", "file82.fastq", "file83.fastq")));
group2.put("Sample-9", new ArrayList<String>(
    Arrays.asList("file91.fastq", "file92.fastq", "file93.fastq")));
comparison_1.setGroup1(group1);
comparison_1.setGroup2(group2);
...

```

We further split the <filename>.fastq files (using Linux's `split` function) into 8 million records (exactly 2 million FASTQ records, since each 4 lines is one FASTQ record). The reason for this split is to make sure that every mapper will get fair share of input records.

Our solution for RNA-Seq analysis has 2 MapReduce algorithm steps:

- STEP-1: MapReduce Tophat mapping
- STEP-2: MapReduce Cuffdiff Calling

### 25.4.1 STEP-1: MapReduce Tophat mapping

This step is comprised on a `map()` and `reduce()` functions: the mapper uses Tophat to align input data and then reducers concatenate the output of mappers.

The driver program for STEP-1, partitions data into small chunks and passes each input file to a mapper. For example, an input for an RNA-Seq might be the HDFS directory: `/rnaseq/input/2397/` (2397 is an analysisID, which uniquely identifies each RNA Seq. analysis) where

```
# hadoop fs -ls /rnaseq/input/2397/
0000.txt
0001.txt
0002.txt
..
0090.txt
0091.txt
```

Each input file has one single record comprised of many tokens: (NOTE: since each record is too long, it is formatted – each line contains one token with delimiter of ";"):

```
# 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
```

#### 25.4.1.1 The map() function

The `map()` function calls a BASH shell script<sup>5</sup>, which apply Tophat and generate intermediate BAM files. The output of each Tophat mapper will be

---

<sup>5</sup> For generating shell scripts we used FreeMarker. FreeMarker is a "template engine"; a generic tool to generate text output (anything from HTML to autogenerated source code) based on templates. It's a Java package, a class library for Java programmers. It's not an application for end-users in itself, but something that programmers can embed into their products (source: <http://freemarker.sourceforge.net/>)

an accepted\_hits.bam file, which is a list of read alignments in SAM format. SAM is a compact short read alignment format that has been increasingly adopted.

**Listing 25.1:** STEP-1: map()

```
1 // key is MR generated, ignored here.
2 // value: represents one record of input
3 map(key, value) {
4     Map<String, String> tokens = tokenizeMapperRecord(value);
5     // The reducer key will be:
6     // "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
7     String reducerKey = getReducerKey(tokens);
8     RNASEq.tophat(tokens);
9     emit(reducerKey, value);
10 }
11
12 public static void tophat(Map<String, String> tokens)
13     throws Exception {
14     TemplateEngine.init();
15     Map<String, String> templateMap = new HashMap<String, String>();
16     templateMap.put("key", tokens.get("counter"));
17     templateMap.put("counter", tokens.get("counter"));
18     templateMap.put("comparison_id", tokens.get("comparison_id"));
19     templateMap.put("group_id", tokens.get("group_id"));
20     templateMap.put("sample_id", tokens.get("sample_id"));
21     templateMap.put("rna_seq_input_type", tokens.get("rna_seq_input_type"));
22     templateMap.put("analysis_id", tokens.get("analysis_id"));
23     templateMap.put("rna_seq_tophat_segment_mismatches",
24         tokens.get("rna_seq_tophat_segment_mismatches"));
25     templateMap.put("rna_seq_tophat_library_type",
26         tokens.get("rna_seq_tophat_library_type"));
27     templateMap.put("rna_seq_tophat_segment_length",
28         tokens.get("rna_seq_tophat_segment_length"));
29     templateMap.put("rna_seq_tophat_gene_model",
30         tokens.get("rna_seq_tophat_gene_model"));
31     templateMap.put("rna_seq_tophat_reference_genome",
32         tokens.get("rna_seq_tophat_reference_genome"));
```

```

33 // create the actual script from a template file
34 String scriptFileName = "/rnaseq/scripts/tophat." +
35     tokens.get("analysis_id") + "." + tokens.get("counter") + ".sh";
36 String logFileName = "/rnaseq/logs/tophat." +
37     tokens.get("analysis_id") + "." + tokens.get("counter") + ".log";
38 File scriptFile = TemplateEngine.createDynamicContentAsFile("tophat.template",
39     templateMap, scriptFileName);
40 if (scriptFile != null) {
41     ShellScriptUtil.callProcess(scriptFileName, logFileName);
42 }
43 }

```

---

#### 25.4.1.2 The reduce() Function

The `reduce()` Function calls a BASH shell script, which uses samtools' `cat` function to concatenate accepted hits (`accepted_hits.bam`) BAM files. Then the script uses samtools' `sort` function to generate sorted BAM file. So for our example, for Comparison-1, the reducers will generate the following sorted BAM files in HDFS:

```

/<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

```

The reducer is defined as:

**Listing 25.2: STEP-1: reduce()**

```

1 // key as : "<analysis_id><;><comparison_id><;><group_id><;><sample_id>"
2 // values: not used
3 reduce(key, values) {

```

```

4     RNASEqTophat.catenatePartitionedBamFiles(key);
5     emit(key, key);
6 }
7
8 /**
9  * This method catenates small/partitioned accepted_hits.bam files
10 * and creates a single sorted.bam file.
11 * @param key as : "<analysis_id><;><comparison_id><;><group_id><;><sample_id>""
12 */
13 public static void catenatePartitionedBamFiles(String key) {
14     throws Exception {
15     TemplateEngine.init();
16     // split the key (fields are separated by ";")
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     // create a template map to be passes tp FreeMarker template
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     // create the actual script from a template file
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 }

```

---

## 25.4.2 STEP-2: MapReduce Calling Cuffdiff

This step is comprised on a map() and reduce() functions (the reducer is optional – may be used for further analysis). The mapper calls `cuffdiff` function for two groups for any given comparison, and then reducers generate the final desired output as a bioset/biomarker for further analysis.

### 25.4.2.1 map() function

The mapper will call cuffdiff function for every comparison (in our example, we have 3 comparisons, so cuffdiff will be called 3 times). For example, for Comparison-1, cuffdiff will be called as:

```
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 generates the following output files, which will be saved in HDFS:

- `isoform_exp.diff`
- `isoforms.read_group_tracking`
- `read_groups.info`

The mapper for STEP-2 is presented below:

**Listing 25.3:** STEP-2: map()

```

1 /**
2  * The mapper performs cuffdiff function for two groups of data.
3 *
4  * @param key will be: "<analysis_id><;><comparison_id><;><group_id><;><sample_i
5  * @param value is a String of 11 tokens:
6  *   index content
7  *   -----
8  *   0   <counter-as-key>; (such as 0000, 0001, )
9  *   1   <comparison_id>;
10 *   2   <group1_id>;
11 *   3   <group2_id>;
12 *   4   constant "cuffdiff"; (ignored)
13 *   5   <analysis_id>;
14 *   6   <segment_mismatches>;
15 *   7   <library_type>;
16 *   8   <segment_length>;
17 *   9   <gene_model>;
18 *   10  <reference_genome>;
19 *
20 */
21 map(key, value){
22     Map<String, String> tokens = tokenizeMapperRecord(value);
23     // reducerKey = <analysis_id><;><comparison_id>
24     String reducerKey = getReducerKey(tokens);
25     RNASeqCuffDiff.cuffdiff(tokens);
26     emit(reducerKey, value);
27 }
28
29 public static void cuffdiff(Map<String, String> tokens) throws Exception {
30     TemplateEngine.init();
31     String scriptFileName = "/rnaseq/scripts/run_cuffdiff." +
32         tokens.get("analysis_id") + "." + tokens.get("comparison_id") + ".sh";
33     String logFileName = "/rnaseq/logs/run_cuffdiff." +
34         tokens.get("analysis_id") + "." + tokens.get("comparison_id") + ".log";
35     File scriptFile = TemplateEngine.createDynamicContentAsFile("cuffdiff.sh.templ
36         tokens, scriptFileName);
37     if (scriptFile != null) {

```

```
38     ShellScriptUtil.callProcess(scriptFileName, logFileName);
39 }
40 }
```

---

#### 25.4.2.2 `reduce()` function

There is no reducer for the STEP-2. But, reducers may be used to read the output of `cuffdiff` function and generate biosets/biomarkers, which can be used in further analysis and evaluation of RNA-Seq samples.

# Chapter 26

## Gene Aggregation

### 26.1 Introduction

This chapter provides a MapReduce solution to gene aggregation. We provide two distinct solutions in MapReduce/Hadoop and MapReduce/Spark. The input data for gene aggregation are patients biosets. A bioset, also called gene signatures, encompass data in the form of experimental sample comparisons (for transcriptomic, epigenetic, and copy number variation data), as well as genotype signatures (for GWAS and mutational data). In simple terms, a bioset is a list of (key, value) pairs, where key is a GENE-ID and value is a list of associated attributes. Gene aggregation is used in clinical applications to identify transcriptional signatures and patterns of gene expression data. Also, gene aggregation is used to see how genes are grouped together into sets of genes and how this affects the overall analysis. It is proven that gene aggregation is an evolutionarily method and depends on chromosomal folding and higher-order structures.

Gene aggregation is achieved by using three metrics:

- **Reference Type** refers to type of patient data and they are:

```
r1=normal  
r2=disease  
r3=paired  
r4=unknown
```

- **Gene Filter Type** refers to type of filter applied to the data. The filter type indicates how gene values to be grouped and analyzed. For example, if a filter type is "up", then only gene values which are greater than (has "up" value) a Filter Value Threshold will be considered for further analysis. There are 3 gene filter types:

```
"absolute value (abs)"
"greater than (up)"
"less than (down)"
```

- **Filter Value Threshold** used by **Gene Filter Type** to exclude the genes, which does not satisfy the threshold value).

Each bioset belong to a patient identified by a "patient ID". Given a set of biosets (in thousands and tens of thousands), where a bioset may have up to 50,000 genes, we want to find frequency of each gene which satisfies three metrics: (1) "reference type", (2) "gene filter type", and (3) "filter value threshold". Filter type indicates how (should we use absolute value, greater than or less than operator) the gene value should be compared to the "filter value threshold". The number of genes per bioset depends on the bioset data type – bioset data types are "copy number variation", "gene expression", "methylation", and "somatic mutation"). Before we delve into MapReduce algorithm for gene aggregation, we discuss the input/output data.

## 26.2 Input

Each bioset may have 20,000 to 50,000 records and each record will have the following format (patients are identified by p100 and p200):

```
format:
<geneID><,><referenceType><;><patientID><,><geneValue>
example-1:
    7562135,r1;p100,1.04
example-2:
    7570769,r1;p200,-1.09
```

## 26.3 Output

The goal of Gene Aggregation is to find frequency of each gene along with its reference type (r1, r2, r3, r4). The MapReduce solution will generate the following output format in MapReduce's distributed file system (for example, Hadoop's distributed file system):

```
format:  
    <geneID><,><referenceType><TAB><frequency-count>  
example-1:  
    7562135,r1    1205  
example-2:  
    7570769,r3    14067
```

Eventually, this output can be read into a Java object such as `GeneAggregationFrequencyCount`, which will be discussed in Analysis of Output section.

## 26.4 MapReduce Solution

The question in gene aggregation algorithm is this: per patient, do we filter by "individual gene values" (patient ID is ignored) or by an "average of gene values" (patient ID is used). For example, if we have 10 values for GENE-1 (using real data, the number of values per geneID can be in thousands – to understand algorithms, I use a toy data) for three patients (identified by p100, p200, p300) with "reference type" of "r1", "gene filter type" as "up", and "filter value threshold" equal to 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

Then we will get the following different values (based on whether we filter by "individual gene values" or by "average of gene values"):

- Filter by "individual gene values": this will produce the following output:

GENE-1,r1 6

since only 6 records' gene values are greater than or equal to 1.04, which pass the filter.

- Filter by "average of gene values": this will produce the following output:

GENE-1,r1 2

since average of values ( $\frac{1.00+1.06+1.10+1.20}{4}$ ) for patient p100 is 1.09, which is greater than 1.04 (passes the filter), average of values ( $\frac{1.00+1.02+1.04}{3}$ ) for patient p200 is 1.02 (which does not pass the filter), and average of values ( $\frac{1.01+1.06+1.08}{3}$ ) for patient p300 is 1.05 (which passes the filter). Therefore only two patients passed the test by average, so the frequency will be 2.

We provide two distinct MapReduce solutions, which handles filter by "individual gene values" as well as filter by "average of gene values".

How will we pass these three dynamic parameters ("reference type", "gene filter type", and "filter value threshold") from MapReduce driver to `map()` and `reduce()` functions. Using `MapReduce Configuration` object, we can set these values by `Configurtion.set()` and retrieve them (by `Configurtion.get()`) in the `setup()` function of `map()` or `reduce()`.

The mapper gets one record of a bioset and tokenizes the input into a `<geneID><,><referenceType>` and `geneValue`. If `<geneID><,><referenceType>` contains the desired "reference type", then we check the `geneValue` to see if it satisfies the "filter value threshold". If both conditions are satisfied, then we emit (key, value) where key is the `<geneID><,><referenceType>` and the value is an integer 1 (one). The reducer sums up the frequencies for a specific `<geneID><,><referenceType>` and emits (key, value), where key is the `<geneID><,><referenceType>` and the value is the "final frequency count".

### 26.4.1 Mapper: Filter by Individual

The mapper's `setup()` will retrieve the desired parameters (metrics needed for gene aggregation) to be used in the `map()` function.

**Listing 26.1:** Gene Aggregator's mapper `setup()` Function

```

1 public class GeneAggregatorMapperByIndividual ... {
2
3     private String referenceType = null;
4     private String filterType = null;
5     private double filterValueThreshold;
6
7     /**
8      * will be run only once
9      */
10    public void setup(Context context) {
11        Configuration conf = context.getConfiguration();
12        this.referenceType = conf.get("gene.reference.type");
13        this.filterType = conf.get("gene.filter.type");
14        this.filterValueThreshold =
15            Double.parseDouble(conf.get("gene.filter.value.threshold"));
16    }

```

**Listing 26.2:** Gene Aggregator's mapper `map()` Function

```

1 /**
2  * @param key is the key generated by MapReduce partitioner (ignored here)
3  * @param value is one record of bioset as:
4  *   <geneID><,><referenceType><;><patientID><,><geneValue>
5  */
6 map(Long key, String value) {
7     String[] tokens = StringUtil.split(value, ";");
8     String geneIDAndReferenceType = tokens[0];
9     String patientIDAndGeneValue = tokens[1];
10    String[] val = StringUtil.split(patientIDAndGeneValue, ",");
11    // val[0] = patientID
12    // val[1] = geneValue

```

```

13     double geneValue = Double.parseDouble(val[1]);
14
15     String[] arr = StringUtil.split(geneIDAndReferenceType, ",");
16     // arr[0] = geneID
17     // arr[1] = referenceType
18     // check referenceType
19     if (arr[1].equals(this.referenceType)) {
20         if (checkFilter(geneValue)) {
21             // then create a counter for reducer,
22             // otherwise nothing will be written
23             // prepare key-value for reducer and send it to reducer
24             emit(geneIDAndReferenceType, 1);
25         }
26     }
27 }
```

**Listing 26.3:** Gene Aggregator's checkFilter() Function

```

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 }
```

Since *filterType* is passed from MapReduce driver to *map()*, therefore we know the value of *filterType* before any *map()* starts. For this reason, in our Mapreduce/Hadoop implementation, we will provide custom mappers (to avoid the **if-statement** so many times per *map()*). Note that in our MapReduce solution, for readability purposes, we have avoided all exception

checking and handling.

### 26.4.2 Reducer: Filter by Individual

Reducer will receive a (*key*, *List < integer >*) where *key* is the unique geneAndReferenceType and *List < integer >* is a partial frequencies of the unique geneAndReferenceAsString.

The reducer job is to sum up the number of occurrences of unique geneAndReferenceType.

Here is the reduce() function:

**Listing 26.4:** Gene Aggregator's reduce() Function

```
1 /**
2 * @param key is the unique geneAndReferenceAsString generated by the mapper
3 * @param values is a list of integers
4 *   (partial count of a unique geneIDAndReferenceType)
5 */
6 reduce(String key, List<Integer> values) {
7     int sum = 0;
8     for (int count : values) {
9         sum += count;
10    }
11    emit(key, sum);
12 }
```

### 26.4.3 Mapper: Filter by Average

The mapper's *setup()* will retrieve the desired parameters (metrics needed for gene aggregation) to be used in the *map()* function. since we are filtering by average of gene values per patient, we have to pass the patient IDs along the gene values to reducers.

**Listing 26.5:** Gene Aggregator's mapper setup() Function

```
1 public class GeneAggregatorMapperByIndividual ... {
2
3     private String referenceType = null;
4     private String filterType = null;
5     private double filterValueThreshold;
6
7     /**
8      * will be run only once
9     */
10    public void setup(Context context) {
```

```

11     Configuration conf = context.getConfiguration();
12     this.referenceType = conf.get("gene.reference.type");
13     this.filterType = conf.get("gene.filter.type");
14     this.filterValueThreshold =
15         Double.parseDouble(conf.get("gene.filter.value.threshold"));
16 }

```

---

**Listing 26.6:** Gene Aggregator's mapper map() Function

```

1 /**
2 * @param key is the key generated by MapReduce partitioner (ignored here)
3 * @param value is one record of bioset as:
4 *   <geneID><,><referenceType><;><patientID><,><geneValue>
5 */
6 map(Long key, String value) {
7     String[] tokens = StringUtil.split(value, ";");
8     String geneIDAndReferenceType = tokens[0];
9     String patientIDAndGeneValue = tokens[1];
10    String[] arr = StringUtil.split(geneIDAndReferenceType, ",");
11    // arr[0] = geneID
12    // arr[1] = referenceType
13    // check referenceType
14    if (arr[1].equals(this.referenceType)) {
15        // prepare key-value for reducer and send it to reducer
16        emit(geneIDAndReferenceType, patientIDAndGeneValue);
17    }
18 }
19

```

---

Since *filterType* is passed from MapReduce driver to *map()*, therefore we know the value of *filterType* before any *map()* starts. For this reason, in our Mapreduce/Hadoop implementation, we will provide custom mappers (to avoid the **if-statement** so many times per *map()*). Note that in our MapReduce solution, for readability purposes, we have avoided all exception checking and handling.

#### 26.4.4 Reducer: Filter by Average

Reducer will receive a (*K*, *List* < *V* >) where *K* is the unique geneIDAndReferenceType and *V* is a patientIDAndGeneValue.

The reducer job is to sum up the number of occurrences of unique geneIDAndReferenceType by filtering average of gene values.

Here is the reduce() function:

### **Listing 26.7:** Gene Aggregator's reduce() Function

```
1 /**
2 * @param key is the unique geneIDAndReferenceType generated by mappers
3 * @param values is a List<V> and V is a 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    // emit the output of reducer
15    emit(key, passedTheTest);
16 }
```

## 26.5 Computing Gene Aggregation

Computing gene aggregation is handled by a utility class called `GeneAggregatorUtil`. This class has two `static` methods:

- `buildPatientsMap()`: this method accepts a `List<Tuple2<patientID, geneValue>>` and builds a `Map<patientID, sumOfGeneValues>`.
- `getNumberOfPatientsPassedTheTest()`: counts to see how many patients passed the test (satisfied the filter criteria). This is achieved by examining each entry of Map built by `buildPatientsMap()` method.

Also, we use a simple class `PairOfDoubleInteger` to represent a `Tuple2<Double, Integer>` and be able to update its values.

### **Listing 26.8:** GeneAggregatorUtil.buildPatientsMap()

```
1 /**
2 * GeneAggregatorUtil.buildPatientsMap() method:
3 *
4 * @param values = List<Tuple2<patientID, geneValue>>
5 *
6 * THE RESULT Map:
7 *   make sure we do not count more than once for the same patient
8 *   patients.key = patientID
9 *   patients.value = Tuple2<D, I>
10 *     where D = sum of values for the patientID
11 *           I = number of values (counter)
```

```

12     */
13     public static Map<String, PairOfDoubleInteger> buildPatientsMap(Iterable<Text> values)
14         throws IOException, InterruptedException {
15         Map<String, PairOfDoubleInteger> patients = new HashMap<String, PairOfDoubleInteger>();
16         for (Text patientIdAndGeneValue : values) {
17             String[] tokens = StringUtils.split(patientIdAndGeneValue.toString(), ",");
18             String patientID = tokens[0];
19             //tokens[1] = geneValue
20             double geneValue = Double.parseDouble(tokens[1]);
21             PairOfDoubleInteger pair = patients.get(patientID);
22             if (pair == null) {
23                 pair = new PairOfDoubleInteger(geneValue, 1);
24                 patients.put(patientID, pair);
25             }
26             else {
27                 pair.increment(geneValue);
28             }
29         }
30     }
31     return patients;
32 }
```

---

**Listing 26.9:** GeneAggregatorUtil.getNumberOfPatientsPassedTheTest()

```

1 /**
2  * GeneAggregatorUtil.getNumberOfPatientsPassedTheTest() method:
3 */
4 public static int getNumberOfPatientsPassedTheTest(
5     Map<String, PairOfDoubleInteger> patients,
6     double filterValue,
7     String filterType) { // filterType = {"up", "down", "abs"}
8     if (patients == null) {
9         return 0;
10    }
11
12    // now, we will average the values and see which patient passes the threshold
13    int passedTheTest = 0;
14    for (Map.Entry<String, PairOfDoubleInteger> entry : patients.entrySet()) {
15        //String patientID = entry.getKey();
16        PairOfDoubleInteger pair = entry.getValue();
17        double avg = pair.avg();
18        if (filterType.equals("up")) {
19            if (avg >= filterValue) {
20                passedTheTest++;
21            }
22        }
23        if (filterType.equals("down")) {
24            if (avg <= filterValue) {
25                passedTheTest++;
26            }
27        }
28        else if (filterType.equals("abs")) {
29            if (Math.abs(avg) >= filterValue) {
30                passedTheTest++;
31            }
32        }
33    }
34    return passedTheTest;
35 }
```

```
31         }
32     }
33 }
34
35     return passedTheTest;
36 }
37 }
```

---

## 26.6 Hadoop Implementation

We have a simple MapReduce solution for Gene Aggregation. The following classes are used to implement the MapReduce solution in Hadoop.

- GeneAggregationDriverByIndividual: a driver for setting input/output and launching the job
- GeneAggregationMapperByIndividual: a mapper to filter by individual values
- GeneAggregationReducerByIndividual: a reducer to filter by individual values
- GeneAggregationDriverByAvergae: a driver for setting input/output and launching the job
- GeneAggregationMapperByAvergae: a mapper to filter by averages of values
- GeneAggregationReducerByAverage: a reducer to filter by averages of values

In order to achieve better performance, it is better to have a separate classes for handling filter type ("up", "down", "abs"). Therefore, GeneAggregationMapperByIndividual can be replaced by three plug-in classes:

- GeneAggregationMapperByIndividualUP: when filterType = "up"
- GeneAggregationMapperByIndividualDOWN when filterType = "down"
- GeneAggregationMapperByIndividualABS: when filterType = "abs"

Note that for performance efficiency purposes, we provided three mappers (one for each "gene filter type"). This enables us to avoid "gene filter type" checking by an **if-statement**. For example, if we are processing 20,000 biosets and each bioset has over 40,000 geneIDs, then we have avoided executing/calling 800 million non-needed **if-statement**.

Also, GeneAggregationReducerByAverage reducer classes can be optimized by three plug-in classes:

- GeneAggregationReducerByAverageUP: when filterType = "up"
- GeneAggregationReducerByAverageDOWN when filterType = "down"
- GeneAggregationReducerByAverageABS: when filterType = "abs"

Similar to plug-in mapper classes, custom plug-in reducers avoid unnecessary **if-statement** checks millions of times.

We set the plug-in mapper classes in the driver classes when we check gene-values individually:

#### Listing 26.10: GeneAggregationDriverByIndividual Class

```

1 public class GeneAggregationDriverByIndividual extends Configured implements Tool {
2     String filterType = ...;
3
4     public int run(String[] args) throws Exception {
5         Job job = ...;
6
7         // set the optimized mapper class
8         if (filterType.equals("up")) {
9             job.setMapper(GeneAggregationMapperByIndividualUP.class);
10            else if (filterType.equals("down")) {
11                job.setMapper(GeneAggregationMapperByIndividualDOWN.class);
12            }
13            else if (filterType.equals("abs")) {
14                job.setMapper(GeneAggregationMapperByIndividualABS.class);
15            }
16            else {
17                throw new Exception("filterType is undefined");
18            }
19
20         // set the reducer class
21         job.setReducer(GeneAggregationReducerByIndividualDOWN.class);
22         ...
23     }
24 }
```

We set the plug-in reducer classes in the driver classes when we check gene-values by average:

### Listing 26.11: GeneAggregationDriverByAverage Class

```
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         // set the mapper class
8         job.setMapper(GeneAggregationMapperByAverage.class);
9
10        // set the optimized reducer class
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 }
```

Therefore, the optimized set of implementation classes will be:

- GeneAggregationDriverByIndividual: a driver for setting input/output and launching the job
- GeneAggregationMapperByIndividualUP: a mapper to filter by individual values
- GeneAggregationMapperByIndividualDOWN: a mapper to filter by individual values
- GeneAggregationMapperByIndividualABS: a mapper to filter by individual values
- GeneAggregationReducerByIndividual: a reducer to filter by individual values
- GeneAggregationDriverByAverage: a driver for setting input/output and launching the job
- GeneAggregationMapperByAverage: a mapper to filter by average values

- GeneAggregationReducerByAverageUP: a reducer to filter by average of values
- GeneAggregationReducerByAverageDOWN: a reducer to filter by average of values
- GeneAggregationReducerByAverageABS: a reducer to filter by average of values

Therefore the `checkFilter()` method is broken down into 3 segments (one segment per mapper class). Since we have a custom plug-in mapper class per `filterType`, the `if-statement(s)` are dropped. Now, each mapper class knows what is the exact `filterType` value.

**Listing 26.12:** `checkFilter()` for `GeneAggregationMapperByIndividualUP` Class

```

1 // GeneAggregationMapperByIndividualUP class:
2 public boolean checkFilter(double value) {
3     //if (filterType.equals("up")) {
4         if (value >= this.filterValueThreshold) {
5             return true;
6         }
7         else {
8             return false;
9         }
10    //}
11 }
```

**Listing 26.13:** `checkFilter()` for `GeneAggregationMapperByIndividualDOWN` Class

```

1 // GeneAggregationMapperByIndividualDOWN class:
2 public boolean checkFilter(double value) {
3     //if (filterType.equals("down")) {
4         if (value <= this.filterValueThreshold) {
5             return true;
6         }
7         else {
8             return false;
9         }
10    //}
11 }
```

**Listing 26.14:** `checkFilter()` for `GeneAggregationMapperByIndividualABS` Class

```

1 // GeneAggregationMapperByIndividualABS class:
2 public boolean checkFilter(double value) {
3     //if (filterType.equals("abs")) {
4         if (Math.abs(value) >= this.filterValueThreshold) {
5             return true;
6         }
7     else {
8         return false;
9     }
10    //}
11 }

```

---

## 26.7 Analysis of Output

The output (saved in Hadoop's HDFS system as a SequenceFile – (key,value) as a binary data) of our MapReduce/Hadoop solution will have the following format:

```

format:
<geneID><,><referenceType><TAB><frequency-count>
example-1:
    7562135,r1      1205
example-2:
    7570769,r1      14067

```

We can easily read these output files and pass the values into Java objects.

```

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();
}

```

Then we invoke the following, where *dir* is the Hadoop output path:

```
List<FrequencyItem> list = readDirectoryIntoFrequencyItem(dir);
```

The `GeneAggregatorAnalyzerUsingFrequencyItem` class is defined as:

```
/**  
 * This class analyzes output directory of Hadoop distributed file system (HDFS)  
 * and returns the result as a List<FrequencyItem> object.  
 *  
 * @author Mahmoud Parsian  
 */  
public class GeneAggregatorAnalyzerUsingFrequencyItem {  
  
    static List<FrequencyItem> readDirectoryIntoFrequencyItem(String pathAsString)  
        return readDirectoryIntoFrequencyItem(new Path(pathAsString));  
    }  
  
    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;
}

@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;
}

```

```

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);
    return item;
}

static void closeAndIgnoreException(SequenceFile.Reader reader) {
    if (reader == null) {
        return;
    }

    try {
        reader.close();
    }
    catch(Exception ignore) {
        THE_LOGGER.error("Error closing SequenceFile.Reader.", ignore);
    }
}

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());
}

}

```

## 26.8 Gene Aggregation in Spark

This section presents a MapReduce Spark solution for Gene Aggregation. From HDFS input files, we will create RDDs and manipulate them in Spark. Spark's main data representation is an RDD, which enables parallel (such as `map()`, `reduce()`, and `groupByKey()` functions) operations on the data. Spark can read and save RDDs from many different sources (including HDFS data source). Another way to create RDDs are from Java collection objects.

Similar to hadoop silution, we provide two distict spark solutions for gene aggregation (the entire Spark solution is presented in one Java driver class):

- Filter by "individual gene values": this is implmented by the `SparkGeneAggregationByIndividual` class.
- Filter by "average of gene values": this is implmented by the `SparkGeneAggregationByAverage` class.

## 26.9 Gene Aggregation in Spark: Filter by Individual

The first step is to create a `JavaRDD<String>` for all input biosets, where each item of RDD will be a record of a bioset. The second step is to filter out the values, which do not meet the Reference Type and Filter Value Threshold values. The third step will be to map RDD (generated by the second step) items into `JavaPairRDD<K,V>`, where K is a `<geneID><,,><referenceType>` and V is an integer one (similar to the word count). The final step will be to group by K (as `<geneID><,,><referenceType>`).

### Sharing Data Between cluster Nodes

The question is how to pass the three metic values (referenceType, filterType, and filterValueThreshold) to Spark's actions and transformations for manipulating RDDs? Spark provides `Broadcast<T>`<sup>1</sup> class for sharing read-

---

<sup>1</sup>`org.apache.spark.broadcast.Broadcast`. Spark's `Broadcast` variables allow the programmer to keep a read-only variable cached on each cluster node rather than shipping a copy of it with MapReduce tasks. `Broadcast` variables provide every cluster node a copy of a input dataset in an efficient manner. Spark attempts to distribute `Broadcast` variables using efficient broadcast algorithms to reduce communication cost.

only variables among all cluster nodes. `Broadcast<T>` variables allow the programmer to keep a read-only variable cached on each cluster node rather than shipping a copy of it with tasks. In MapReduce/Hadoop, sharing variables among mappers and reducers are accomplished by using the Hadoop's `Configuration` object. The driver class (which submits jobs to the MapReduce framework), sets/defines shared variables and the `map()` or `reduce()` use these shared variables. Sharing is achieved by mapper's or reducer's `setup()` function, which is called once for each mapper or reducer.

In Spark, for our three metric values, we define three `Broadcast` objects and when we need them for manipulating RDDs (using `map` and `reduce` functions), we read them. This is how to define `Broadcast` objects

```

1 JavaSparkContext ctx = createJavaSparkContext();
2 ...
3 Broadcast<String> broadcastVarReferenceType = ctx.broadcast(referenceType);
4 Broadcast<String> broadcastVarFilterType = ctx.broadcast(filterType);
5 Broadcast<Double> broadcastVarFilterValueThreshold = ctx.broadcast(filterValueThreshold);
6
7

```

and this is how we use/read them when needed:

```

1 String referenceType = (String) broadcastVarReferenceType.value();
2 String filterType = (String) broadcastVarFilterType.value();
3 Double filterValueThreshold = (Double) broadcastVarFilterValueThreshold.value();
4
5

```

The general form of using `Broadcast<T>` is defined below:

```

1 DEFINITION:
2     JavaSparkContext ctx = <create-context-object>;
3     T t = <some-data-structure>;
4     final Broadcast<T> broadcastVariable = ctx.broadcast(t);
5
6 USAGE:
7     T t = (T) broadcastVariable.value();
8
9

```

Spark API indicates that after the broadcast variable is created, it should be used instead of the value `t` in any functions run on the cluster so that `t` is not shipped to the nodes more than once.

### 26.9.1 High Level Solution

The Spark solution is a single Java driver class manipulating several RDDs. This section provides main steps of the Java driver class and then we will provide details for each high-level step.

#### Listing 26.15: SparkGeneAggregationByIndividual Class

```
1 //STEP-0: import required classes and interfaces
2 public class SparkGeneAggregationByIndividual {
3
4     public static void main(String[] args) throws Exception {
5         //STEP-1: handle input parameters
6
7         //STEP-2: create a spark context object
8
9         //STEP-3: broadcast shard variables
10
11        //STEP-4: create a single JavaRDD from all bioset files
12
13        //STEP-5: map bioset records into JavaPairRDD(K,V) pairs
14        // where K = "<geneID><,><referenceType>", V = 1
15
16        //STEP-6: filter out the redundant RDD elements
17
18        //STEP-7: reduce by Key and sum up the frequency count
19
20        //STEP-8: prepare the final output
21
22        System.exit(0);
23    }
24 }
```

### 26.9.2 High Level Solution

To use RDD's, we need Spark's org.apache.spark.api.java package. The org.apache.spark.api.java.function package defines functions for actions and transformations. Also, org.apache.spark.broadcast.Broadcast class is needed to define/broadcast shared global data structures among all cluster nodes.

#### Listing 26.16: STEP-0: import required classes and interfaces

```
1 //STEP-0: import required classes and interfaces
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 import org.apache.spark.SparkConf;
12
13 import java.util.Arrays;
14 import java.util.List;
15 import java.util.ArrayList;
16 import java.io.FileReader;
17 import java.io.BufferedReader;

```

---

### 26.9.3 STEP-1: handle input parameters

This section reads three metric parameters (referenceType, filterType, filterValueThreshold) for gene aggregation and all biosets involved in the analysis. Each bioset is identified by an HDFS file.

**Listing 26.17:** STEP-1: handle input parameters

```

1 //STEP-1: handle input parameters
2 if (args.length != 4) {
3     System.err.println("Usage: SparkGeneAggregationByIndividual <referenceType>" +
4                         "<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);

```

---

### 26.9.4 STEP-2: Create a Spark Context Object

This step creates a `JavaSparkContext`, which is needed for creation of RDDs (fundamental data abstraction and parallelism in Spark). Context object can be created in many different ways. Here, we create a `JavaSparkContext` object by using `SparkConf` object. `SparkConf` is a configuration class for a

Spark application. SparkConf has set(K, V) method for parameter definitions.

#### **Listing 26.18: STEP-2: create a spark context object**

```
1 //STEP-2: create a spark context object
2 JavaSparkContext ctx = createJavaSparkContext();
```

#### **26.9.5 STEP-3: Broadcast Shard Variables**

Spark enables sharing variables and data structures among cluster nodes by Broadcast<T> class, where T is a type of shared data. Spark has an efficient mechanism for broadcasting shared variables and objects. Shared variables can be accessed in Sparks's map() and reduce() functions.

#### **Listing 26.19: STEP-3: broadcast shard variables**

```
1 //STEP-3: broadcast shard variables
2 final Broadcast<String> broadcastVarReferenceType = ctx.broadcast(referenceType);
3 final Broadcast<String> broadcastVarFilterType = ctx.broadcast(filterType);
4 final Broadcast<Double> broadcastVarFilterValueThreshold = ctx.broadcast(filterValueThreshold);
```

#### **26.9.6 STEP-4: Create a JavaRDD For Biosets**

The main data for gene aggregation analysis comes from bioset files. This step creates a single JavaRDD<String> for all records of biosets involved in the analysis. A debugging step has been included as well.

#### **Listing 26.20: STEP-4: create a single JavaRDD from all bioset files**

```
1 //STEP-4: create a single JavaRDD from all bioset files
2 JavaRDD<String> records = readInputFiles(ctx, biosets);
3
4 // debug1
5 List<String> debug1 = records.collect();
6 for (String rec : debug1) {
7     System.out.println("debug1 => "+ rec);
8 }
```

## 26.9.7 STEP-5: Map Biosets into JavaPairRDD(K,V)

This step creates JavaPairRDD(K,V) object from all bioset records, where K = "<geneID><,><referenceType>" and V = 1. This will enable us to count frequency of genes for all patients represented by biosets. When a bioset record does not match our metric criteria, we create a dummy object (Tuple2("null", 0)), which will be filtered out before finalizing the output. In Spark, RDD elements cannot be Java null objects and this is the reason for creating objects such as Tuple2Null (to be a replacement for Java null object).

**Listing 26.21:** STEP-5: map bioset records into JavaPairRDD(K,V) pairs

```
1 //STEP-5: map bioset records into JavaPairRDD(K,V) pairs
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        // check referenceType and geneValue
17        String referenceType = (String) broadcastVarReferenceType.value();
18        String filterType = (String) broadcastVarFilterType.value();
19        Double filterValueThreshold = (Double) broadcastVarFilterValueThreshold.value();
20
21        if ( (arr[1].equals(referenceType)) &&
22            (checkFilter(geneValue, filterType, filterValueThreshold)) ){
23            // prepare key-value for reducer and send it to reducer
24            return new Tuple2<String, Integer>(geneIDAndReferenceType, 1);
25        }
26        else {
27            // otherwise nothing will be counted
28            // later we will filter out these "null" keys
29            return Tuple2Null;
30        }
31    }
32});
```

The following debugs STEP-5:

```

// debug2
List<Tuple2<String, Integer>> debug2 = genes.collect();
for (Tuple2<String, Integer> pair : debug2) {
    System.out.println("debug2 => key="+ pair._1 + "\tvalue="+pair._2);
}

```

## 26.9.8 STEP-6: filter out the redundant RDD elements

Step-5 created redundant (K,V) = ("null", 0) pairs. This step filters non-necessary data and provides a clean JavaPairRDD<String, Integer>. This is accomplished by implementing Spark's filter() function. The filter() is defined as:

```

public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)
DESCRIPTION: Return a new RDD containing only the
elements that satisfy a predicate;

```

Here is the filter implementation:

### Listing 26.22: STEP-6: filter out the redundant RDD elements

```

1 //STEP-6: filter out the redundant RDD elements
2 // If a counter (i.e., V) is 0, then exclude them
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) {
8                 return true;
9             }
10            else {
11                return false;
12            }
13        }
14    });

```

## 26.9.9 STEP-7: reduce by Key and sum up the frequency count

This step implements `reduce()` function. It basically groups up the genes by GENE-ID and sums up the frequency counts.

**Listing 26.23:** STEP-7: reduce by Key and sum up the frequency count

```
1 //STEP-7: reduce by Key and sum up the frequency count
2 JavaPairRDD<String, Integer> counts =
3     filteredGenes.reduceByKey(new Function2<Integer, Integer, Integer>() {
4         public Integer call(Integer i1, Integer i2) {
5             return i1 + i2;
6         }
7    });
```

## 26.9.10 STEP-8: prepare the final output

This final step emits the final desired output for gene aggregation analysis.

**Listing 26.24:** STEP-8: prepare the final output

```
1 //STEP-8: prepare the final output
2 List<Tuple2<String, Integer>> output = counts.collect();
3 for (Tuple2<String, Integer> tuple : output) {
4     System.out.println("final output => "+ tuple._1 + ":" + tuple._2);
5 }
```

## 26.9.11 Utitlity Functions

The `toList()` method accepts a file, which contains all biosets involved in the analysis phase. Each record of input file is an HDFS file, which represents a single bioset.

**Listing 26.25:** Utitlity Function: `toList()`

```
1 /**
2  * Convert all bioset files into a List<biosetFileName>
3  *
4  * @param biosets is a filename, which holds all bioset files as HDFS entries
5  * an example will be:
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 aBiosetErrorFile = line.trim();
17             biosetFiles.add(aBiosetErrorFile);
18         }
19         in.close();
20         return biosetFiles;
21     }

```

---

**Listing 26.26:** Utility Function: readInputFiles()

```

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 biosetErrorFileName : biosetFiles) {
8       System.out.println("readInputFiles(): biosetErrorFileName=" + biosetErrorFileName);
9       JavaRDD<String> record = ctx.textFile(biosetErrorFileName);
10      rdds[counter] = record;
11      counter++;
12    }
13    JavaRDD<String> allBiosets = ctx.union(rdds);
14    return allBiosets;
15  }

```

---

**Listing 26.27:** Utility Function: checkFilter()

```

1   static boolean checkFilter(double value, String filterType, Double filterValueThreshold) {
2     if (filterType.equals("abs")) {
3       if (Math.abs(value) >= filterValueThreshold) {
4         return true;
5       }
6     else {
7       return false;
8     }
9   }
10  if (filterType.equals("up")) {
11    if (value >= filterValueThreshold) {
12      return true;
13    }
14  else {
15    return false;
16  }
17}
18  if (filterType.equals("down")) {

```

```

19         if (value <= filterValueThreshold) {
20             return true;
21         }
22         else {
23             return false;
24         }
25     }
26     return false;
27 }
28
29 }

```

---

## 26.9.12 Running Spark on YARN

### 26.9.12.1 Input

```

1 # cat biosets.txt
2 /biosets/b1.txt
3 /biosets/b2.txt
4 /biosets/b3.txt
5 # hadoop fs -cat /biosets/b1.txt
6 GENE-1,r1;p100,1.00
7 GENE-1,r1;p100,1.06
8 GENE-1,r1;p100,1.10
9 GENE-1,r1;p100,1.20
10 # hadoop fs -cat /biosets/b2.txt
11 GENE-1,r1;p200,1.00
12 GENE-1,r1;p200,1.02
13 GENE-1,r1;p200,1.04
14 # hadoop fs -cat /biosets/b3.txt
15 GENE-1,r1;p300,1.01
16 GENE-1,r1;p300,1.06
17 GENE-1,r1;p300,1.08

```

### 26.9.12.2 Script

```

1 # cat run_gene_aggregation_by_individual_yarn.sh
2 #!/bin/bash
3 export HADOOP_HOME=/usr/local/hadoop/hadoop-2.4.0
4 export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
5 export JAVA_HOME=/usr/java/jdk7
6 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
7 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
8 export SPARK_HOME=/home/hadoop/spark-1.0.0
9 export MP=/home/hadoop/spark_mahmoud_examples
10 export MY_JAR=$MP/mp.jar
11 export SPARK_JAR=$MP/spark-assembly-1.0.0-hadoop2.4.0.jar
12 export YARN_APPLICATION_CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
13 prog=SparkGeneAggregationByIndividual

```

```

14 biosets=/home/hadoop/spark_mahmoud_examples/biosets.txt
15 referenceType=r1
16 #{"r1", "r2", "r3", "r4"}
17 filterType=up
18 # {"up", "down", "abs"}
19 filterValueThreshold=1.04
20 $SPARK_HOME/bin/spark-submit --class $prog \
21   --master yarn-cluster \
22   --num-executors 6 \
23   --driver-memory 1g \
24   --executor-memory 1g \
25   --executor-cores 12 \
26   $MY_JAR $referenceType $filterType $filterValueThreshold $biosets

```

#### 26.9.12.3 Log of Script Run

```

1 ./run_gene_aggregation_by_individual_yarn.sh
2
3 args[0]: <referenceType>=r1
4 args[1]: <filterType>=up
5 args[2]: <filterValueThreshold>=1.04
6 args[3]: <biosets>/=home/hadoop/spark_mahmoud_examples/biosets.txt
7
8 readInputFiles(): biosetFileName=/biosets/b1.txt
9 readInputFiles(): biosetFileName=/biosets/b2.txt
10 readInputFiles(): biosetFileName=/biosets/b3.txt
11
12 debug2 => key=null value=0
13 debug2 => key=GENE-1,r1 value=1
14 debug2 => key=GENE-1,r1 value=1
15 debug2 => key=GENE-1,r1 value=1
16 debug2 => key=null value=0
17 debug2 => key=null value=0
18 debug2 => key=GENE-1,r1 value=1
19 debug2 => key=null value=0
20 debug2 => key=GENE-1,r1 value=1
21 debug2 => key=GENE-1,r1 value=1
22
23 final output => GENE-1,r1: 6

```

## 26.10 Gene Aggregation in Spark: Filter by Average

Gene Aggregation using "Filter by Average" algorithm is implemented in Spark. The first step is to create a JavaRDD<String> for all input biosets, where each item of RDD will be a record of a bioset. The second step is to

create a `JavaPairRDD<K, V>`, where K is a `<geneID><, ><referenceType>` and V is a `<patientID><, ><geneValue>`. The third step will be to group by K (as `<geneID><, ><referenceType>`). The final step will be to find frequencies of GENE-ID by average value of genes (per patient), which satisfies all given metrics. The core of Gene Aggregation using "Filter by Average" algorithm is presented by an example: let

```
K = Tuple2(geneID, referenceType)
V = List(Tuple2(patientID, geneValue))
```

To find average per patient, using V, we build a hash table as:

```
Map(patientID, Tuple2(sum(geneValue), count)).
```

Next, we find the average per patient as: `Map(patientID, average)`, where `average = sum(geneValue)/count`. If the `average` passes the "filter value threshold", then that will be counted as 1 (means passed the test), otherwise it will be counted as 0 (means did not pass the test).

First, we present the main steps for Spark solution and in subsequent sections, we present the details of each step.

#### **Listing 26.28: Gene Aggregation by Average: High-Level Steps**

```
1 // STEP-0: import required classes and interfaces
2
3 public class SparkGeneAggregationByAverage {
4
5     // used as a dummy object for filtering
6     static final Tuple2<String, String> Tuple2Null =
7         new Tuple2<String, String>("n", "n");
8
9     public static void main(String[] args) throws Exception {
10        //STEP-1: handle input parameters
11
12        //STEP-2: create a Java Spark context object
13
14        //STEP-3: share global variables in all cluster nodes
15
16        //STEP-4: read all bioset records and create an RDD as
17        //          JavaRDD<String> records = readInputFiles(ctx, biosets);
18
19        //STEP-5: map bioset records and create JavaPairRDD<K, V>
```

```

20     //    where K = "<geneID><,><referenceType>",
21     //          V = "<patientID><,><geneValue>"
22
23     //STEP-6: filter redundant records created by STEP-5
24
25     //STEP-7: group biosets by "<geneID><,><referenceType>"
26     // genesByID = JavaPairRDD<K, List<V>>
27     //           where K = "<geneID><,><referenceType>"
28     //           V = "<patientID><,><geneValue>"
29
30     //STEP-8: prepare the final desired output
31
32     //STEP-9: emit the final output
33
34     System.exit(0);
35 }
36 }
```

---

### 26.10.1 STEP-0: import required classes and interfaces

**Listing 26.29:** STEP-0: import required classes and interfaces

```

1 //STEP-0: import required classes and interfaces
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.broadcast.Broadcast;
10 import org.apache.spark.SparkConf;
11
12 import java.io.FileReader;
13 import java.io.BufferedReader;
14 import java.util.Map;
15 import java.util.HashMap;
16 import java.util.List;
17 import java.util.ArrayList;
```

---

### 26.10.2 STEP-1: handle input parameters

### **Listing 26.30: STEP-1: handle input parameters**

```
1 //STEP-1: handle input parameters
2 if (args.length != 4) {
3     System.out.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);
```

### **26.10.3 STEP-2: create a Java Spark context object**

You need to create a JavaSparkContext in order to create and manipulate your RDDs. There are many ways to create this context object. One way to create it is to create a Spark's configuration object (as `SparkConf` and then inject it into the `JavaSparkContext`'s constructor.

### **Listing 26.31: STEP-2: create a Java Spark context object**

```
1 //STEP-2: create a Java Spark context object
2 JavaSparkContext ctx = createJavaSparkContext();
```

Note that I have hard coded the YARN resource manager server (as myserver100), you need to replace it accordingly by some configuration files.

### **Listing 26.32: createJavaSparkContext() Method**

```
1 static JavaSparkContext createJavaSparkContext() throws Exception {
2     SparkConf conf = new SparkConf();
3     conf.set("yarn.resourcemanager.hostname", "myserver100");
4     conf.set("yarn.resourcemanager.scheduler.address", "myserver100:8030");
5     conf.set("yarn.resourcemanager.resource-tracker.address", "myserver100:8031");
6     conf.set("yarn.resourcemanager.address", "myserver100:8032");
7     conf.set("mapreduce.framework.name", "yarn");
8     conf.set("spark.serializer", "org.apache.spark.serializer.KryoSerializer");
9     conf.set("spark.executor.memory", "1g");
10    JavaSparkContext ctx =
11        new JavaSparkContext("yarn-cluster", "SparkGeneAggregationByAverage", conf);
```

```
12     return ctx;
13 }
```

#### 26.10.4 STEP-3: share global variables in all cluster nodes

In Spark, the `Broadcast<T>` class enable us to share data structures of type T among all cluster nodes. Basically, you can share/broadcast your data structures in your driver program (the class submitting the job to the Spark or YARN cluster) and then you may read them from any cluster node inside `map()` and `reduceByKey()` functions. For our three global variables, we do it as:

**Listing 26.33:** STEP-3: share global variables in all cluster nodes

```
1 //STEP-3: share global variables in all cluster nodes
2 final Broadcast<String> broadcastVarReferenceType = ctx.broadcast(referenceType);
3 final Broadcast<String> broadcastVarFilterType = ctx.broadcast(filterType);
4 final Broadcast<Double> broadcastVarFilterValueThreshold = ctx.broadcast(filterValueThreshold);
```

#### 26.10.5 STEP-4: read all bioset records and create an RDD

This step reads all bioset files from HDFS and create a single RDD (`JavaRDD<String>`). This functionality is explained in detail when we discuss the `readInputFiles()` method.

**Listing 26.34:** STEP-4: read all bioset records and create an RDD

```
1 //STEP-4: read all bioset records and create an RDD
2 JavaRDD<String> records = readInputFiles(ctx, biosets);
3
4 // debug
5 List<String> debug1 = records.collect();
6 for (String rec : debug1) {
7     System.out.println("debug1 => "+ rec);
8 }
```

## 26.10.6 STEP-5: map bioset records and create JavaPairRDD(K, V)

This step creates JavaPairRDD(K, V) object from all bioset records, where K = "<geneID><,><referenceType>" and V = "<patientID><,><geneValue>". This will enable us to count frequency of genes by average for all patients represented by patients for all biosets. When a bioset record does not match our metric criteria, we create a dummy (marked as null object) object (Tuple2("n", "n")), which will be filtered out before finalizing the output. In Spark, RDD elements cannot be Java `null` objects and this is the reason for creating objects such as `Tuple2Null` (to be a replacement for Java `null` object). Note that the Spark's approach for ignoring some RDD elements is different from MapReduce/Hadoop's approach, which just ignores them without emitting any redundant (key, value) pairs. For example, in MapReduce/Hadoop, if your mapper/reducer input is not your desired criteria, then you may ignore to emit any new (key, value) pairs. But in Spark, `mapToPair()` method has to return a non-null (key, value) pair.

**Listing 26.35:** STEP-5: map bioset records and create JavaPairRDD(K, V)

```
1 //STEP-5: map bioset records and create JavaPairRDD<K, V>
2 // JavaPairRDD<K, V>,
3 //   where K = "<geneID><,><referenceType>",
4 //         V = "<patientID><,><geneValue>"
5 JavaPairRDD<String, String> genes =
6     records.mapToPair(new PairFunction<String, String, String>() {
7     public Tuple2<String, String> call(String record) {
8         String[] tokens = StringUtils.split(record, ";");
9         String geneIDAndReferenceType = tokens[0];
10        String patientIDAndGeneValue = tokens[1];
11        String[] arr = StringUtils.split(geneIDAndReferenceType, ",");
12        // arr[0] = geneID
13        // arr[1] = referenceType
14        // check referenceType and geneValue
15        String referenceType = (String) broadcastVarReferenceType.value();
16        if (arr[1].equals(referenceType)) {
17            // prepare key-value for reducer and send it to reducer
18            return new Tuple2<String, String>(geneIDAndReferenceType,
19                                              patientIDAndGeneValue);
20        }
21        else {
22            // otherwise nothing will be counted
23            // later we will filter out these "null" keys
24            return Tuple2Null;
25        }
26    }
27});
```

---

### 26.10.7 STEP-6: filter redundant records created by STEP-5

Step-5 created redundant  $(K, V) = \text{Tuple2}("n", "n")$  pairs (as a replacement for Java `null` values, since Spark is unable to have `null` RDD elements). This step filters out redundant data and provides a clean `JavaPairRDD<String, String>`. This is accomplished by implementing Spark's `filter()` function. The `filter()` is defined as:

```
public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)
DESCRIPTION: Return a new RDD containing only the elements that
            satisfy a predicate defined by function f.
```

Here is the filter implementation:

**Listing 26.36:** STEP-6: filter redundant records created by STEP-5

```
1 //STEP-6: filter redundant records created by STEP-5
2 // public JavaPairRDD<K,V> filter(Function<Tuple2<K,V>,Boolean> f)
3 // Return a new RDD containing only the elements that satisfy a predicate;
4 // If K = "n", then exclude them
5 JavaPairRDD<String, String> filteredGenes =
6     genes.filter(new Function<Tuple2<String, String>, Boolean>() {
7         public Boolean call(Tuple2<String, String> s) {
8             String value = s._1;
9             if (value.equals("n")) {
10                 // exclude null entries
11                 return false;
12             }
13             else {
14                 return true;
15             }
16         }
17     });

```

---

### 26.10.8 STEP-7: group biosets by geneID and referenceType

This step groups all bioset records by `<geneID><,><referenceType>`. The grouped values will be ready for the final reductionm which we will calculate

gene aggregation by average per patient.

#### **Listing 26.37: STEP-7: group biosets by "`<geneID><,><referenceType>`"**

```
1 // STEP-7: group biosets by "<geneID><,><referenceType>"  
2 // genesByID = JavaPairRDD<K, List<V>>  
3 //   where K = "<geneID><,><referenceType>"  
4 //       V = "<patientID><,><geneValue>"  
5 JavaPairRDD<String, Iterable<String>> genesByID = filteredGenes.groupByKey();
```

#### **26.10.9 STEP-8: prepare the final desired output**

This step applies the core gene aggregation by average algorithm to all bioset records. Now we have a list of `<patientID><,><geneValue>` per `<geneID><,><referenceType>`. This step is implemented by `mapValues()` method. This is how it works:

```
mapValues[U](f: (V) => U): JavaPairRDD[K, U]  
Description: Pass each value in the key-value  
pair RDD through a map function  
without changing the keys; this also  
retains the original RDD's partitioning.
```

This step uses Spark's broadcasted global variables (so called shared variables). You can define shared data structures by creating `Broadcast<T>` (where T is your data structure). To use shared/broadcasted variables, you may use `Broadcast.value()` method (this is how we read shared variables in all cluster nodes – the `value()` method returns data structure of type T).

#### **Listing 26.38: STEP-8: prepare the final desired output**

```
1 //STEP-8: prepare the final desired output  
2 JavaPairRDD<String, Integer> frequency =  
3   genesByID.mapValues(new Function<Iterable<String>, Integer> {  
4     // input  
5     // output  
6     >() {  
7       Map<String, PairOfDoubleInteger> patients = buildPatientsMap(values);  
8       String filterType = (String) broadcastVarFilterType.value();  
9       Double filterValueThreshold = (Double) broadcastVarFilterValueThreshold.value();  
10      int passedTheTest = getNumberOfPatientsPassedTheTest(patients,
```

```

11                               filterType,
12                               filterValueThreshold);
13             return passedTheTest;
14         }
15     });

```

---

### 26.10.10 STEP-9: emit the final output

This step prints the final desired output.

**Listing 26.39:** STEP-9: emit the final output

```

1 //STEP-9: emit the final output
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 }

```

---

The following sections provide explantion of some of the important methods used in solving gene aggregation by average algorithm.

### 26.10.11 toList() Method

This method collects all HDFS bioset files and saves them as a `List<String>`, where each list element is an HDFS bioset file.

**Listing 26.40:** Support Method toList()

```

1 /**
2  * Convert all bioset files into a List<biosetFileName>
3  *
4  * @param biosets is a filename, which holds all bioset files as
5  * HDFS entries; an example will be:
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 }
```

### 26.10.12 readInputFiles() Method

This method reads all HDFS bioset files and creates a new RDD as `JavaRDD<String>`, where each element is a bioset record. All bioset records are merged by using `JavaContextObject.union()` method. Then we partition this RDD by applying `JavaRDD.coalesce()` method for further parallel processing.

**Listing 26.41:** Support Method `readInputFiles()`

```
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 } // readInputFiles
```

### 26.10.13 buildPatientsMap() Method

This method builds a `Map<String, PairOfDoubleInteger>`, where key is a patientID and value is a `PairOfDoubleInteger` object, which keeps track of sum of gene values and its count.

**Listing 26.42:** Support Method `buildPatientsMap()`

```
1 private static Map<String,PairOfDoubleInteger> buildPatientsMap(Iterable<String> values) {
2     Map<String, PairOfDoubleInteger> patients = new HashMap<String, PairOfDoubleInteger>();
3     for (String patientIdAndGeneValue : values) {
4         String[] tokens = StringUtils.split(patientIdAndGeneValue, ",");
5         String patientID = tokens[0];
6         //tokens[1] = geneValue
7         double geneValue = Double.parseDouble(tokens[1]);
8         PairOfDoubleInteger pair = patients.get(patientID);
9         if (pair == null) {
```

```

10         pair = new PairOfDoubleInteger(geneValue, 1);
11         patients.put(patientID, pair);
12     }
13     else {
14         pair.increment(geneValue);
15     }
16 }
17 return patients;
18 }

```

---

### 26.10.14 buildPatientsMap() Method

This method iterates the `Map<String, PairOfDoubleInteger>` built by the `buildPatientsMap()` method. Each entry of this map denotes a single patient. If each patient passes the desired metrics (i.e., the average of gene values passes the defined metrics – metrics are `filterType` and `filterValueThreshold`), then that is counted as the one, which passes the test.

**Listing 26.43:** Support Method `getNumberOfPatientsPassedTheTest()`

```

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     // now, we will average the values and see which
10    // patient passes the threshold
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 }
```

---

## 26.10.15 Running Spark on YARN

### 26.10.15.1 Input

```
1 # cat biosets.txt
2 /biosets/b1.txt
3 /biosets/b2.txt
4 /biosets/b3.txt
5 # hadoop fs -cat /biosets/b1.txt
6 GENE-1,r1;p100,1.00
7 GENE-1,r1;p100,1.06
8 GENE-1,r1;p100,1.10
9 GENE-1,r1;p100,1.20
10 # hadoop fs -cat /biosets/b2.txt
11 GENE-1,r1;p200,1.00
12 GENE-1,r1;p200,1.02
13 GENE-1,r1;p200,1.04
14 # hadoop fs -cat /biosets/b3.txt
15 GENE-1,r1;p300,1.01
16 GENE-1,r1;p300,1.06
17 GENE-1,r1;p300,1.08
```

### 26.10.15.2 Script

```
1 # cat run_gene_aggregation_by_average_yarn.sh
2 #!/bin/bash
3 export HADOOP_HOME=/usr/local/hadoop/hadoop-2.4.0
4 export SPARK_LIBRARY_PATH=$HADOOP_HOME/lib/native
5 export JAVA_HOME=/usr/java/jdk7
6 export HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
7 export YARN_CONF_DIR=$HADOOP_HOME/etc/hadoop
8 export SPARK_HOME=/home/hadoop/spark-1.0.0
9 export MP=/home/hadoop/spark_mahmoud_examples
10 export MY_JAR=$MP/mp.jar
11 export SPARK_JAR=$MP/spark-assembly-1.0.0-hadoop2.4.0.jar
12 export YARN_APPLICATION_CLASSPATH=$CLASSPATH:$HADOOP_HOME/etc/hadoop
13 #export THEJARS=$MP/commons-math-2.2.jar,$MP/commons-math3-3.0.jar
14 prog=SparkGeneAggregationByAverage
15 biosets=/home/hadoop/spark_mahmoud_examples/biosets.txt
16 referenceType=r1
17 #{"r1", "r2", "r3", "r4"}
18 filterType=up
19 # {"up", "down", "abs"}
20 filterValueThreshold=1.04
21 $SPARK_HOME/bin/spark-submit --class $prog \
```

```
22   --master yarn-cluster \
23   --num-executors 6 \
24   --driver-memory 1g \
25   --executor-memory 1g \
26   --executor-cores 12 \
27   $MY_JAR $referenceType $filterType $filterValueThreshold $biosets
```

#### 26.10.15.3 Log of Script Run

```
1 ./run_gene_aggregation_by_average_yarn.sh
2
3 args[0]: <referenceType>=r1
4 args[1]: <filterType>=up
5 args[2]: <filterValueThreshold>=1.04
6 args[3]: <biosets>=/home/hadoop/spark_mahmoud_examples/biosets.txt
7
8 debug1 biosetLabelName=/biosets/b1.txt
9 debug1 biosetLabelName=/biosets/b2.txt
10 debug1 biosetLabelName=/biosets/b3.txt
11
12
13 final output => GENE-1,r1: 2
14
```

# Chapter 27

## Linear Regression

### 27.1 Introduction

In this chapter we present a very important statistical concept, linear regression<sup>1</sup>, which is used by many applications including clinical applications such as genome analysis using patient sample data. What are the main applications of linear regression? "Linear regression is widely used in biological, behavioral and social sciences to describe possible relationships between variables. It ranks as one of the most important tools used in these disciplines."<sup>2</sup> Of course implementing linear regression for small data is very straightforward (we might use many existing Java classes such as the one from Apache common: SimpleRegression<sup>3</sup>, but these classes and packages can not handle huge amount of data due to limitation of memory and CPU in a single server). Our primary goal in this chapter is to solve linear regression for huge data sets (such as genomic data represented by biosets for many patients samples data).

---

<sup>1</sup>The linear regression model analyzes the relationship between the response or dependent variable and a set of independent or predictor variables. This relationship is expressed as an equation that predicts the response variable as a linear function of the parameters. These parameters are adjusted so that a measure of fit is optimized. Much of the effort in model fitting is focused on minimizing the size of the residual, as well as ensuring that it is randomly distributed with respect to the model predictions (source: [http://en.wikipedia.org/wiki/Predictive\\_analytics](http://en.wikipedia.org/wiki/Predictive_analytics))

<sup>2</sup>source: [http://en.wikipedia.org/wiki/Linear\\_regression](http://en.wikipedia.org/wiki/Linear_regression)

<sup>3</sup> org.apache.commons.math3.stat.regression.SimpleRegression

The most common form of linear regression is least squares fitting<sup>4</sup>. Before getting into details of linear regression, let's see what is Linear Regression and What Does it Tell You. In simple terms, we are trying to fit an equation to a real set of data (kind of how to predict the future by observing a real set of data).

This chapter provides two distinct MapReduce/Hadoop solutions for linear regression:

- The first solution utilizes Apache's Commons `SimpleRegression`
- The second solution implements MapReduce by using R's `Linear Model`

## 27.2 Simple Facts about Linear Regression

We borrow some of the following facts from [http://www.biddle.com/documents/bcg\\_comp\\_chapter3.pdf](http://www.biddle.com/documents/bcg_comp_chapter3.pdf).

1. Linear regression is modeled by a linear equation  $y = ax + b$ . In a nutshell, regression analysis is used to find equations that fit data (data is represented by a set of  $(x, y)$ ).
2. Linear regression uses the fact that there is a statistically significant correlation between two variables to allow you to make predictions about one variable based on your knowledge of the other.
3. You should not do linear regression unless your correlation coefficient is statistically significant
4. For linear regression to work there needs to be a linear relationship between the variables

Next we need some definitions for regression and linear Regression. A regression is a statistical analysis assessing the association between two variables (such as  $x$  and  $y$  in regression equation  $y = ax + b$ , in the simplest form, it is used to find the relationship between two variables  $x$  and  $y$ , but you may use more than two variables). Therefore, linear regression defines/estimates the relation between variables when the regression equation is linear: e.g.,

---

<sup>4</sup><http://mathworld.wolfram.com/LeastSquaresFitting.html>

$y = ax + b$ , where  $a$  is the the intercept point of the regression line and the  $y$  axis and  $b$  is the slope of the regression line. So, the main goal of the linear regression is that to find  $a$  (intercept - the value of  $y$  when  $x = 0$ ) and  $b$  (slope) for a given set of  $x, y$ . Typically variable  $x$  is the explanatory variable and  $y$  is the dependent variable.

## 27.3 Simple Example

Here we present a simple example to demonstrate the basic concept of a linear regression. The goal of Linear Regression analysis is to find a linear equation(s) that fit data. Once we have the linear equation ( $y = ax + b$ ), then you can use the Linear Regression model (expressed as a linear equation) to make predictions. Here, we shows how to use a sample data, calculate linear regression, and find the equation  $y = ax + b$ . Our two variables are "age" and "glucose level" for a set of very small patients: first we make a chart of our data:

Subject	Age (x)	Glucose Level (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
5	25	78	1950	625	6084
6	40	71	2840	1600	5041
7	58	88	5104	3364	7744
8	60	86	5160	3600	7396
sum	329	668	28144	14883	56714

Next, we use the following formulas to calculate  $a$  and  $b$ :

$$a = \frac{(\sum y)(\sum x^2) - (\sum x)(\sum xy)}{n(\sum x^2) - (\sum x)^2}$$

$$b = \frac{n(\sum xy) - (\sum x)(\sum y)}{n(\sum x^2) - (\sum x)^2}$$

Note that  $n$  is the number of samples for regression (in our example  $n = 8$ ). Now by plugging the pre-calculated values, we will have:

$$a =$$

$$b =$$

Next, we plot linear regression by R programming language:

```

1 # R
2 R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
3 > x <- c(41, 42, 43, 20, 25, 40, 58, 60)
4 > y <- c(90, 93, 98, 64, 78, 71, 88, 86)
5 > mod1 <- lm(y ~ x)
6 > plot(x, y, xlim=c(min(x)-5, max(x)+5), ylim=c(min(y)-10, max(y)+10))
7 > abline(mod1, lwd=2)
```

As you can observe from the plot, linear regression attempts to model the relationship between two variables ( $x$  and  $y$ ) by fitting a linear equation to observed data (the most common algorithm for fitting a regression line is the method of least-squares).

## 27.4 Problem Statement

Assume a set of patient data for a clinical trial of some medication or treatment. We are assuming that each patient may have a set of biosets/biomarkers and each bioset/biomarker has a set of  $(key, value)$  where  $key$  is the GENE-ID and  $value$  can represent "segment value", "copy change number", or other related genomic data value (here value represent variable  $x$  and variable  $y$  can be the survival time of that specific patient for this treatment). Number of  $(key, value)$  pairs for each biomarker depends on the type of bioset (bioset types are "methylation", "copy number variation", "gene expression", and "somatic mutation"). For example for "gene expression" type,

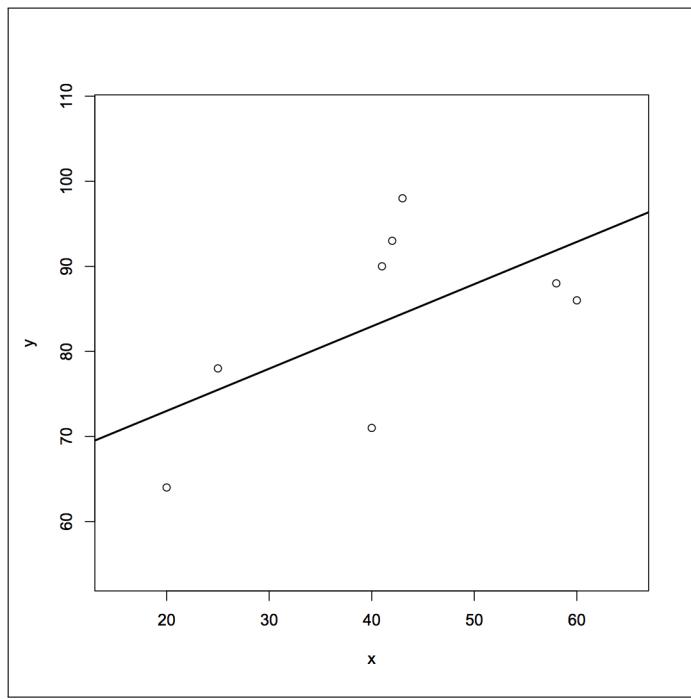


Figure 27.1: Linear Regression by R

each biomarker will have 45,000+ (*key, value*) pairs. Imagine, you want to do a linear regression for 40,000 biosets, this will lead us to 1.8 billion data points, which is not possible to do the linear regression computation in a single server. Here our interest is to do a linear regression per GENE-ID. Explicitly, our objective is to find the following per GENE-ID:

- Intercept (the intercept of the estimated regression line)
- Slope (the slope of the estimated regression line).
- Significance (pvalue or significance level of the slope correlation).

## 27.5 Input Data

Input data will be comprised of a set of biosets/biomarkers (each bioset is identified by a unique BIOSET-ID). Each bioset is comprised of 45,000+ records and each record will contain the following (here, we consider `gene\_value` as variable *x*)

Format: <gene\_id><,><reference><;><bioset\_id><,><gene\_value>  
where <gene\_id> = a gene represented by an ID  
<reference> will be one the following:  
    r1 = reference to normal patient  
    r2 = reference to disease  
    r3 = reference to paired  
    r4 = none

Example-1: 1234,r2;3344550,0.43

Example-2: 122765,r1;3344550,1.78

Also, we will have another input vector (to represent variable *y*) such as "survival time" for patients (which is an array of "double" data type and each item corresponds with a bioset). For example, to analyze 5000 biosets, we do need to pass an array of "survival time" for size of 5000.

## 27.6 Expected Output

After linear regression job is completed, we should have our analysis result in the following format (one record per GENE-ID):

Format: <gene\_id><,><reference><,><slope><,><intercept><,><pvalue>

Example: 1234,r2,1.002,3.12007,0.000098

## 27.7 MapReduce Solution using Apache Commons SimpleRegression

MapReduce solution for linear regression is comprised of a mapper and a reducer. The main task of map() function is to identify variable  $x$  and pass it to a reducer (reducer key will be the GENE-ID-and-REFERENCE and its associated value will be the GENE-VALUE (variable  $x$  in linear regression). Variable  $y$  (here in our example is the "survival time") is passed from driver class to MapReduce/Hadoop Configuration object, which will be retrieved by the reducer (for performing linear regression).

Each reducer will handle a (*key, value*) received. The *key* is the GENE-ID-and-REFERENCE and *value* is a list of GENE-VALUE. The reduce() function will get the  $y$  values from MapReduce/Hadoop Configuration object (in Hadoop this will be done by *setup()* method). Once we have  $x$ 's and  $y$ 's, then we will use SimpleRegression<sup>5</sup> class to perform linear regression and then write the result (*slope*, *intercept*, and *pvalue*) back to HDFS. Using SimpleRegression class is very simple:

```
// 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
// we may add any number of sr.addData(x, y)
// now all statistics are defined.
double pvalue = sr.getSignificance();
double intercept = sr.getIntercept();
double slope = sr.getSlope();
```

Note that the GENE-VALUE represents  $x$ 's and we will pass bioset IDs and "survival time" (represent's  $y$ 's). Why do we need to pass bioset IDs (

---

<sup>5</sup>org.apache.commons.math3.stat.regression.SimpleRegression (Apache Commons Math)

through Job's Configuration object)? Because bioset\_id[i] corresponds with survival\_time[i], we have to make sure that we are using proper  $x$ 's and  $y$ 's for linear regression.

Listing 27.1: map() for Linear Regression

```
/**  
 * @param key is generated by Hadoop (ignored here)  
 * @param value's format: <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, ";");  
  
    //String <gene_id><,><reference> = tokens[0];  
    //String <bioset_id><,><gene_value> = tokens[1];  
    if (tokens.length == 2) {  
        // prepare (key, value) for reducer  
        emit(tokens[0], tokens[1]);  
    }  
}
```

The reduce class, LinearRegressionReducer, is presented below.

Listing 27.2: LinearRegressionReducer for Linear Regression

```
public class LinearRegressionReducer ... {  
  
    // instance variables  
    private Configuration conf = null;  
    private List<Double> time = null;  
    // biosetIDs as Strings : NOTE order of biosets are VERY IMPORTANT  
    private List<String> biosets = null;  
  
    // will be run only once  
    public void setup(Context context)  
        this.conf = context.getConfiguration();
```

```

// get parameters from Hadoop's configuration
String biosetsAsString = conf.get("biosets");
this.biosets = DataStructuresUtil.splitOnToString(
    biosetsAsString, ",");

String timeAsCommaSeparatedString = conf.get("time");
this.time = DataStructuresUtil.splitOnToString(
    timeAsCommaSeparatedString, ",");
}

// key = <gene_id><,><reference>
// values = { bioset_id,value }
// biosets are: B1, B2, B3, ..., Bn
// time are : T1, T2, T3, ..., Tn
public void reduce(Text key, Iterable<Text> values) {
    // see next section ...
}
}

```

---

Listing 27.3: reduce() for Linear Regression

```

public class LinearRegressionReducer ... {
    ...

    // key = <gene_id><,><reference>
    // values = { bioset_id,value }
    // biosets are: B1, B2, B3, ..., Bn
    // time are : T1, T2, T3, ..., Tn
    public void reduce(Text key, Iterable<Text> values) {
        int number0fValues = 0;
        SimpleRegression sr = new SimpleRegression();
        Iterator<Text> iter = values.iterator();
        while (iter.hasNext()) {
            Text pairAsText = iter.next();
            if (pairAsText == null) {
                continue;
            }
            String pairAsString = pairAsText.toString();
            String[] tokens = StringUtils.split(pairAsString, ",");
            // biosetId = tokens[0]
            // value = tokens[1]
            if (tokens.length != 2) {
                // then ignore that (value, time) from regression

```

```

        continue;
    }

    int index = biosets.indexOf(tokens[0]);
    if (index == -1) {
        // biosetId not found
        continue;
    }

    // biosetId found at index
    // sr.addData(xPos.get(i), yPos.get(i));
    double dvalue = Double.parseDouble(tokens[1]);
    sr.addData(dvalue, time.get(index));
    numberofValues++;
}

if (numberofValues > 0) {
    StringBuilder builder = new StringBuilder();
    builder.append(key.toString()); // gene_id_and_reference: 1234,r2
    builder.append(",");
    builder.append(sr.getSignificance()); // pvalue
    builder.append(",");
    builder.append(sr.getIntercept()); // intercept
    builder.append(",");
    builder.append(sr.getSlope()); // slope

    // prepare reducer for output
    Text reducerValue = new Text(builder.toString()); // pvalue,
        intercept,slope
    emit(null, reducerValue);
}
}
}

```

---

## 27.8 Hadoop Implementation using Apache Commons SimpleRegression

Our Mapreduce/Hadoop Implementation is comprised of the following classes:

<i>Class Name</i>	<i>Class Description</i>
LinearRegressionDriver.java	The driver class, defines input/output and registers plug-in classes.
LinearRegressionMapper.java	Defines the map() function
LinearRegressionReducer.java	Defines the reduce() function
LinearRegressionAnalyzer.java	Defines how output data will be read from HDFS
LinearRegressionClient.java	Client class to submit job

## 27.9 MapReduce Solution using R's Linear Model

MapReduce solution for linear regression using R's Linear Model is comprised of two MapReduce jobs. The first job aggregates required data and generates input files, to be read/used in the second MapReduce job. The second MapReduce job reads input (generated by reducers in phase 1) and then applies the R's Linear Model (*lm()*) function (for details, see <http://data.princeton.edu/R/linearModels.html>). R's Linear Model is more accurate and comprehensive than Apache's SimpleRegression class. The reason for having second Mapreduce job is to avoid calling R's Linear Model (*lm()*) function for evry single GENE-ID. We basically collect as many as possible in a text file and then call R's Linear Model (*lm()*) function.

Before I describe 2-phase MapReduce solution, let me show you how R's Linear Model (*lm()* function) can be used:

```

1 # R
2 R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
3 Copyright (C) 2012 The R Foundation for Statistical Computing
4 Platform: x86_64-apple-darwin9.8.0/x86_64 (64-bit)
5
6 > x <- c(-1.0, 1.1, 2.2, 3.3, 4.4, 5.5, -1.5)
7 > y <- c(-1.1, 1.88, 2.88, 3.44, 4.44, 5.99, -1.8)
8 > fit = lm(x ~ y)
9 > intercept = fit$coef[1]
10 > intercept
11 (Intercept)
12 -0.07142071

```

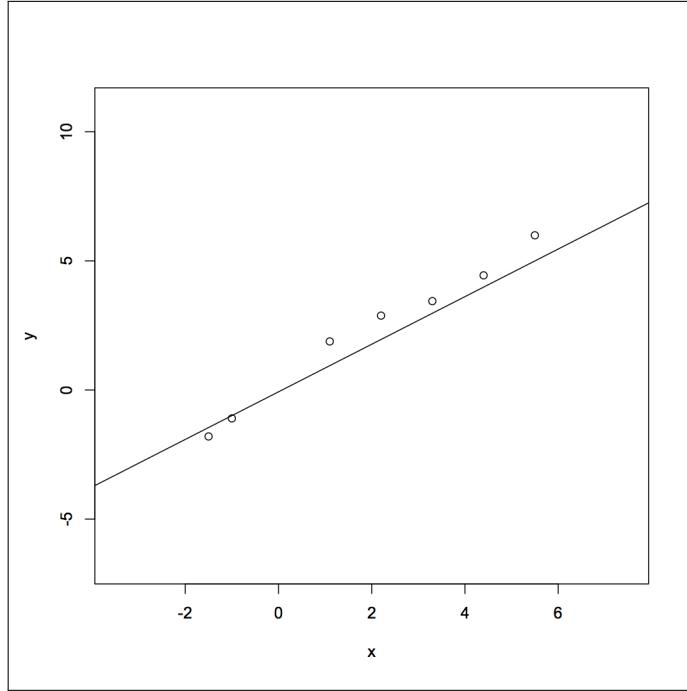


Figure 27.2: Linear Regression Model

```

13 > slope = fit$coef[2]
14 > slope
15      y
16 0.921802
17 > pvalue = anova(fit)$'Pr(>F)'[1]
18 > pvalue
19 [1] 1.279226e-05
20 > rsquare = summary(fit)$r.squared
21 > rsquare
22 [1] 0.9830423
23 > plot(x, y, xlim=c(min(x)-2, max(x)+2), ylim=c(min(y)-5, max(y)+5))
24 > abline(fit, lwd=1)

```

Plotting by R's *plot* and *abline* functions generates the following linear model (image: [27.2](#)).

### 27.9.1 MapReduce Solution using R's Linear Model: Phase 1

The main task of map() function for Phase-1 is to identify variable  $x$  and pass it to a reducer (reducer key will be the GENE-ID-and-REFERENCE and its associated value will be the GENE-VALUE (variable  $x$  in linear regression). Variable  $y$  (here in our example is the "survival time") is passed from driver class to MapReduce/Hadoop Configuration object, which will be retrieved by the Phase-1 reducer function (for generating proper input data to be consumed by the Phase-2 map() function). So, we pass BIOSET-ID(s) and SURVIVAL-TIME(s) by MapReduce/Hadoop Configuration object, which will be retrieved by the Phase-1 reducer function. Let BIOSET-ID(s) be  $\{B_1, B_2, \dots, B_n\}$  and SURVIVAL-TIME(s) be  $\{T_1, T_2, \dots, T_n\}$  (note that the size of BIOSET-ID(s) and SURVIVAL-TIME(s) must be the same).

Each reducer will handle a  $(key, value)$  received. The *key* is the GENE-ID-and-REFERENCE and *value* is a list of GENE-VALUE. The reduce() function will get the  $y$  values from MapReduce/Hadoop Configuration object (in Hadoop this will be done by *setup()* method). Once we have  $x$ 's and  $y$ 's, then we write them to a text file, which will be saved in HDFS (to be consumed by the Phase-2 map() function). For example, reducers will generate the following text files:

```
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, ..., T1mn
```

Note that the GENE-VALUE represents  $x$ 's and we will pass bioset IDs and "survival time" (represent's  $y$ 's). Why do we need to pass bioset IDs (through Job's Configuration object)? Because bioset\_id[i] corresponds with survival\_time[i], we have to make sure that we are using proper  $x$ 's and  $y$ 's for linear regression.

Let's say that we have 200 mappers and 30 reducers for Phase-1. With this configuration, our MapReduce job for Phase-1 will create 30 output files (in HDFS):

```
/<hdfs-output-directory>/part-00000  
/<hdfs-output-directory>/part-00001  
...  
/<hdfs-output-directory>/part-00029
```

The input to map() function of Phase-2 will be theses HDFS files (generated by reducers of Phase-1 job).

Listing 27.4: map() for Linear Regression for Phase-1

```
/*
 * @param key is generated by Hadoop (ignored here)
 * @param value's format: <gene_id_and_ref><;><biiset_id><;><gene_value>
 */
map(key, value) {
    String line = value.toString().trim();
    if ((line == null) || (line.length() == 0)) {
        return;
    }

    String[] tokens = StringUtils.split(line, ";");
    String gene_id_and_ref = tokens[0];
    String biiset_id_and_value = tokens[1];
    if (tokens.length == 2) {
        // prepare (key, value) for reducer
        emit(gene_id_and_ref, biiset_id_and_value);
    }
}
```

Listing 27.5: reduce() for Linear Regression for Phase-1

```
public class LinearModelReducer extends Reducer<Text, Text, NullWritable,
    Text> {

    // instance variables
    private Configuration conf = null;
    private String type = null;
    // biosetIDs as Strings : NOTE oreder of biosets are VERY IMPORTANT
    public List<String> biosets = null;
    // NOTE: oreder of time items are VERY IMPORTANT
    public List<Double> time = null;

    // will be run only once
    public void setup(Context context) {
        this.conf = context.getConfiguration();
        this.type = conf.get("type");
```

```

// get parameters from Hadoop's configuration
String biosetsAsString = conf.get("biosets");
this.biosets = DataStructuresUtil.splitOnToString(
    biosetsAsString, ",");
}

String timeAsCommaSeparatedString = conf.get("time");
this.time = DataStructuresUtil.toListOfDouble(
    timeAsCommaSeparatedString);
}

// key = geneid_and_ref
// values = { biosetId,value }
public void reduce(Text key, Iterable<Text> values, Context context) {
    //
    // we will generate the following per key
    // (( key=<geneID><r{1,2,3,4}> [example of key: 1234r2] ))
    // <geneID><;><V1, V2, ..., Vn><;><T1,T2, ..., Tn>
    // where V's are gene values and T's are time values
    //
    // T1 is for V1
    // T2 is for V2
    // ...
    // Tn is for Vn
    //
    int numberofValues = 0;
    Iterator<Text> iter = values.iterator();
    StringBuilder valuebuilder = new StringBuilder(); // generates
        geneID; V1, V2, ..., Vn
    StringBuilder timebuilder = new StringBuilder(); // generates T1, T2
        , ..., Tn

    valuebuilder.append(key.toString());
    valuebuilder.append(";" );

    while (iter.hasNext()) {
        Text pairAsText = iter.next();
        if (pairAsText == null) {
            continue;
        }
        String pairAsString = pairAsText.toString();
        String[] tokens = StringUtils.split(pairAsString, ",");
        // biosetId = tokens[0]
        // value = tokens[1]
        if (tokens.length != 2) {

```

```

    // we are done here, all values must be present for linear
    // model
    // no need to write anything to hadoop
    continue;
}

int index = biosets.indexOf(tokens[0]);
if (index == -1) {
    // biosetID not found
    continue;
}

// biosetID found at index
// biosets.get(index) = biosetID;
double valueAsDouble = Double.parseDouble(tokens[1]);

// update value
valuebuilder.append(valueAsDouble);
valuebuilder.append(",");

// update time
timebuilder.append(time.get(index));
timebuilder.append(",");

numberOfValues++;
}

if (numberOfValues > 2) {
    // prepare reducer for output
    String geneAndValues = valuebuilder.toString();
    // chop off the last ","
    String geneAndValuesFinal = geneAndValues.substring(0,
        geneAndValues.length()-1);
    String timesAsString = timebuilder.toString();
    // chop off the last ","
    String timeFinal = timesAsString.substring(0, timesAsString.length
        ()-1);

    // reducerValue = <geneID><;><V1, V2, ..., Vn><;><T1,T2, ..., Tn>
    Text reducerValue = new Text(geneAndValuesFinal + ";" + timeFinal)
        ;
    context.write(null, reducerValue);
}

```

```
}
```

---

### 27.9.2 MapReduce Solution using R's Linear Model: Phase 2

This phase has a mapper, but no reducers are required. The map() function will read text files (output of Phase-1 reduce() function) and then call R's *lm()* (linear model function). The input for map() function will be the following HDFS files generated by reducers of Phase-1 (if we had 30 reducers in Phase-1, then we will have 30 input files for mappers of Phase-2):

```
/<hdfs-output-directory>/part-00000
/<hdfs-output-directory>/part-00001
...
/<hdfs-output-directory>/part-00029
```

The input to map() function of Phase-2 will be these HDFS files (generated by reducers of Phase-1 job). The map() of Phase-2 will pass each file to a shell script, which will call R's *lm()* function. R processing becomes very efficient when we bundle lots of input in text files (this way we can launch only one R process and call *lm()* once per record read from input files).

The following R script (as a template , we use FreeMarker to generate the actual Linux shell script) shows how linear model (*lm()*) is invoked:

```
$ cat linear_model.template.r

1 #!/usr/local/bin/Rscript
2 # input_file is part-nnnnn file in local file system
3 input_file = "${input_file}"
4 cat("input_file=", input_file, "\n")
5 output_file = "${output_file}"
6 cat("output_file=", output_file, "\n")

7 #
8 # each record for file will have the following format
9 # <geneID><;><V1, V2, ..., Vn><;><T1, T2, ..., Tn>
10 #
11 #
```

```

12  # > a = "g1;1,2,3; 4,5,6"
13  # > items = unlist(strsplit(a, ";"))
14  # > items
15  # [1] "g1"      "1,2,3"   "4,5,6"
16  # > geneID = items[[1]]
17  # > geneID
18  # [1] "g1"
19  # > value = as.double(unlist(strsplit(items[[2]], ",")))
20  # > value
21  # [1] 1 2 3
22  # > time = as.double(unlist(strsplit(items[[3]], ",")))
23  # > time
24  # [1] 4 5 6
25  #
26
27 linear_model_fun <- function(line){
28   items = unlist(strsplit(line, ";"))
29   geneID = items[[1]]
30   #cat(geneID, "\n")
31   value = as.double(unlist(strsplit(items[[2]], ",")))
32   #cat(value, "\n")
33   time = as.double(unlist(strsplit(items[[3]], ",")))
34   #cat(time, "\n")
35   fit = lm(value ~ time)
36   intercept = fit$coef[1]
37   slope = fit$coef[2]
38   pvalue = anova(fit)$'Pr(>F)'[1]
39   rsquare = summary(fit)$r.squared
40   cat(geneID, intercept, slope, pvalue, rsquare, "\n", file=output_file, append=T)
41 }
42
43 conn <- file(input_file, open="r")
44 while(length(line <- readLines(conn, 1)) > 0) {
45   try.output <- try( linear_model_fun(line) )
46 }
47 close.connection(conn)

```

### 27.9.3 Hadoop Implementation using R's Linear Model

Our Hadoop Implementation is comprised of the following classes (note that there are no reducers for phase 2):

<i>Class Name</i>	<i>Class Description</i>
LinearRegressionClientPhase2.java	Client class to submit job
LinearRegressionDriverPhase2.java	The driver class, defines input/output and registers plug-in classes.
LinearRegressionMapperPhase2.java	Defines the map() function
LinearRegressionAnalyzerPhase2.java	Defines how output data will be read from HDFS

# Chapter 28

## MapReduce and Monoids

### 28.1 Introduction

This chapter is based on Jimmy Lin's paper[13] titled "Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms." Jimmy Lin clearly defines and relates monoids as a design principle for efficient MapReduce algorithms. But what is a monoid, what properties defines it, and how does it aid the MapReduce paradigm. Following abstract algebra, "a monoid is an algebraic structure with a single associative binary operation and an identity element." (source: Wikipedia<sup>1</sup>). Jimmy Lin shows that when your MapReduce operations are not monoids, then it is very hard to use "combiners" efficiently. Next we will briefly review MapReduce's combiners and abstract algebra's monoids and see how they are related to each other.

Also, David Saile[21] states that "recall that a monoid is an algebraic structure with a single associative binary operation and an identity element. For example, the natural numbers<sup>2</sup>  $\mathbb{N}$  form a monoid under addition with identity element zero. In classic MapReduce, the mapper is not constrained, but the reducer is required to be (the iterated application of) an associative operation. Recent research argued that reduction is in fact monoidal in known applications of MapReduce. That is, reduction is indeed the iterated application of an associative operation " $\bullet$ " with a unit  $u$ . In the case of the

<sup>1</sup> <http://en.wikipedia.org/wiki/Monoid>

<sup>2</sup>The term "natural numbers" refers either to the set of positive integers  $\{1, 2, 3, \dots\}$  or to the set of non-negative integers  $\{0, 1, 2, 3, \dots\}$

word-occurrence count example, reduction iterates addition "+" with "0" as unit. The parallel execution schedule may be more flexible if commutativity is required in addition to associativity."

A detailed analysis of common MapReduce computations on the basis of monoids can be found in [4].

In MapReduce framework, the combiner (as an optional plug-in component) is a "local-reduce" process which operates only on data generated by one server. Successful use of combiners reduces the amount of intermediate data generated by the mappers on a given single server (that is why it is so-called a local reducer). Combiners can be used as a MapReduce optimization to reduce network traffic (by decreasing size of the transient data) between mappers and reducers. Typically, combiners have the same interface as reducers. The combiner must have the following characteristics:

- Combiners receives as input all the data emitted by the mapper instances on a given server (this is called a local aggregation)
- Combiners output is sent to the reducers — some programmers call this as a local server reduction!
- The combiner must be side-effect free; combiners may run an indeterminate number of times.
- The combiner must have the same input and output key types (see example below)
- The combiner must have the same input and output value types (see example below)
- The combiner runs in memory after the map phase

Therefore, a combiner skeleton should be defined as:

#### **Listing 28.1:** Combiner Template

```
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 }

```

---

This template indicates that (key, value) pairs generated by a combiner has to match the (key, value) pairs received (as an input) by reducers. For example, if mapper outputs  $(T_1, T_2)$  pairs (key is type of  $T_1$  and value is type  $T_2$ ) then a combiner has to emit  $(T_1, T_2)$  pairs as well.

The MapReduce/Hadoop does not have a `combine()` function, we just use `reduce()` in hadoop to implement combiners, but we use a plugin of `Job.setCombinerClass()` to define a combiner class.

Furthermore, Jimmy Lin[13] concludes that "one principle for designing efficient MapReduce algorithms can be precisely articulated as follows: create a monoid out of the intermediate value emitted by the mapper. Once we `monoidify` the object, proper use of combiners and the in-mapper combining techniques becomes straightforward."

The Haskell programming language has a direct support for monoids<sup>3</sup>. In Haskell<sup>4</sup>, "a monoid is a type with a rule for how two elements of that type can be combined to make another element of the same type."

## 28.2 Definition of Monoid

Monoid is a triplet  $(S, f, e)$ , where  $S$  is a set (called underlying set of monoid) and  $f : S \times S \rightarrow S$  is a mapping (called binary operation of monoid), and  $e \in S$  is the identity element of monoid. A monoid with binary operation  $\bullet$  satisfies the following three axioms (note that  $f(a, b) = a \bullet b$ ):

- ✓ Closure: for all  $a, b$  in  $S$ , the result of the operation  $a \bullet b$  is also in  $S$ .
- ✓ Associativity: for all  $a, b$  and  $c$  in  $S$ , the following equation holds:  

$$(a \bullet b) \bullet c = a \bullet (b \bullet c)$$

---

<sup>3</sup> <http://www.haskell.org/haskellwiki/Monoid>

<sup>4</sup> <http://blog.sigfpe.com/2009/01/haskell-monoids-and-their-uses.html>

- ✓ Identity element: there exists an element  $e$  in  $S$ , such that for all elements  $a$  in  $S$ , the following two equations hold:

$$\begin{aligned} e \bullet a &= a \\ a \bullet e &= a \end{aligned}$$

And in mathematical notation we can write these as

- ✓ Closure:  $\forall a, b \in S : a \bullet b \in S$
- ✓ Associativity:  $\forall a, b, c \in S : (a \bullet b) \bullet c = a \bullet (b \bullet c)$
- ✓ Identity element:  $\exists e \in S : \forall a \in S : e \bullet a = a \bullet e = a$

A monoid might have other properties: The monoid operator might (but isn't required to) obey other properties like:

- ✓ idempotency:  $\forall a \in S : a \bullet a = a$
- ✓ commutativity:  $\forall a, b \in S : a \bullet b = b \bullet a$

### 28.2.1 How to form a Monoid?

To form a monoid first we need a type  $S$ , which can define a set of values such as integers:  $\{0, -1, +1, -2, +2, \dots\}$ . The second component is a binary function

$$\bullet : S \times S \rightarrow S$$

Then we need to make sure that for any two values  $x \in S$  and  $y \in S$  we get a result object, the combination of  $x$  and  $y$ :

$$x \bullet y : S$$

For example, If  $S$  is a set of integers, then the binary opearation  $\bullet$  may be addition ( $+$ ), multiplication ( $\times$ ) or division ( $\div$ ). Finally, as the third and most important ingredient we need  $\bullet$  to follow a set of laws. If it does, then  $S$  together with  $\bullet$  is called a monoid. We say:  $(S, \bullet, e)$  is a monoid, where  $e \in S$  is the identity element (such as 0 for addition and 1 for multiplication). Also note that the binary division operator ( $\div$ ) over a set of real numbers is not a monoid:

$$\begin{aligned} ((12 \div 4) \div 2) &\neq (12 \div (4 \div 2)) \\ ((12 \div 4) \div 2) &= (3 \div 2) = 1.5 \\ (12 \div (4 \div 2)) &= (12 \div 2) = 6.0 \end{aligned}$$

It seems that "monoids capture the notion of combining arbitrarily many things into a single thing together with a notion of an empty thing called the identity. One example is addition on natural numbers. The addition function  $+$  allows us to combine arbitrarily many natural numbers into a single natural number, the sum. The identity is the empty sum, zero. Another example is string concatenation. The concatenation operator allows us to combine arbitrarily many strings into a single string. The identity is the empty concatenation, the empty string." (source [monoids](#))

## 28.3 Monoidic and Non-Monoidic Examples

Several examples are listed below to understand the concept of monoid.

### 28.3.1 Subtraction over Set of Integers

The set  $S = \{0, 1, 2, \dots\}$  is a commutative monoid for the MAX (maximum) operation, whose identity element is 0

$$\begin{aligned} \text{MAX}(a, \text{MAX}(b, c)) &= \text{MAX}(\text{MAX}(a, b), c) \\ \text{MAX}(a, 0) &= \text{MAX}(0, a) = a \\ \text{MAX}(a, b) &\in S \end{aligned}$$

### 28.3.2 Subtraction over Set of Integers

Subtraction operator  $(-)$  over a set of integers does not define a monoid: this operation is not associative:

$$\begin{aligned} (1 - 2) - 3 &= -4 \\ 1 - (2 - 3) &= 2 \end{aligned}$$

### 28.3.3 Addition over Set of Integers

Addition operator  $(+)$  over a set of integers defines a monoid: this operation is commutative and associative and the identity element is 0:

$$\begin{aligned} (1 + 2) + 3 &= 6 \\ 1 + (2 + 3) &= 6 \\ n + 0 &= n \\ 0 + n &= n \end{aligned}$$

we can formalize this monoid as (below  $e(+)$  defines an identity element):

$$\begin{aligned} S &= \{0, -1, +1, -2, +2, -3, +3, \dots\} \\ e(+) &= \text{Identity element is } 0 \\ m(a, b) &= m(b, a) = a + b \end{aligned}$$

### 28.3.4 Multiplication over Set of Integers

The natural numbers,  $N = \{0, 1, 2, 3, \dots\}$ , form a commutative monoid under multiplication (identity element one).

### 28.3.5 Mean over Set of Integers

On the other hand, the natural numbers,  $N = \{0, 1, 2, 3, \dots\}$ , does not form a monoid under the mean (average) function (the following example shows that the mean of means of arbitrary subsets of a set of values is not the same as the mean of the set of values):

$$\begin{aligned} \text{MEAN}(1, 2, 3, 4, 5) &\neq \text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5)) \\ \text{MEAN}(1, 2, 3, 4, 5) &= \frac{(1 + 2 + 3 + 4 + 5)}{5} = \frac{15}{5} = 3 \\ \text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5)) &= \text{MEAN}(2, 4.5) = \frac{(2 + 4.5)}{2} = 3.25 \end{aligned}$$

### 28.3.6 Non-Commutative Example

For a non-commutative example, consider the collection of all binary strings: an element is a finite ordered sequence of 0's and 1's. The binary operation is just concatenation: e.g.  $\text{concat}(1011, 001001) = 1011001001$ . The identity element is the "empty string". Therefore concatenation of binary strings is a monoid.

### 28.3.7 Median over Set of Integers

The natural numbers does not form a monoid under the MEDIAN function:

$$\text{MEDIAN}(1, 2, 3, 5, 6, 7, 8, 9) \neq \text{MEDIAN}(\text{MEDIAN}(1, 2, 3), \text{MEDIAN}(5, 6, 7, 8, 9))$$

$$\text{MEDIAN}(1, 2, 3, 5, 6, 7, 8, 9) = \frac{(5 + 6)}{2} = 5.5$$

$$\text{MEDIAN}(\text{MEDIAN}(1, 2, 3), \text{MEDIAN}(5, 6, 7, 8, 9)) = \text{MEDIAN}(2, 7) = \frac{(2 + 7)}{2} = 4.5$$

### 28.3.8 Concatenation over Lists

A good example is lists. Lists with concatenation (+) and the empty list (as []) are a monoid: for any list, we can write:

$$l + [] == l, [] + l = l$$

and concatenation is associative. Given two lists, say [1,2,3] and [7,8], you can join them together using + to get [1,2,3,7,8]. There's also the empty list []. Using + to combine [] with any list gives you back the same list, for example []+[1,2,3] = [1,2,3] and [1,2,3] + [] = [1,2,3].

### 28.3.9 Union/Intersection over Integers

Sets under union or intersection over a set of integers forms a monoid.

### 28.3.10 Functional Example

This example is from Mike Stay's Monoids blog<sup>5</sup>. Functions from a set T to itself under composition

$$\begin{aligned} S &= \{ \text{ all functions of the form } a : T \rightarrow T \} \\ e(\bullet) &= \text{the identity function} \\ m(a, b) &= b \circ a, \text{ where } \circ \text{ is composition of functions:} \\ (b \circ a)(x) &= b(a(x)) \end{aligned}$$

For example, given the set  $T = \{0, 1\}$ , we have

- ✓ S contains four possible functions from T back to itself:  $S = \{k_0, 1_T, NOT, k_1\}$

---

<sup>5</sup> <http://reperiendi.wordpress.com/2007/09/12/monoids/>

(1) The constant function mapping everything to zero:

$$\begin{aligned} k_0 : T &\rightarrow T \\ k_0(t) &= 0 \end{aligned}$$

(2) The identity function:

$$\begin{aligned} 1_T : T &\rightarrow T \\ 1_T(t) &= t \end{aligned}$$

(3) The function that toggles the input

$$\begin{aligned} \text{NOT} : T &\rightarrow T \\ \text{NOT}(t) &= 1 - t \end{aligned}$$

(4) The constant function mapping everything to one:

$$\begin{aligned} k_1 : T &\rightarrow T \\ k_1(t) &= 1 \end{aligned}$$

✓  $e(\bullet) = 1_T$  (this is the identity function)

✓  $m(a, b) = b \circ a$

Note that the identity function is the unit for composition:

$$\begin{aligned} (a \circ 1_T)(t) &= a(1_T(t)) = a(t) \\ (1_T \circ a)(t) &= 1_T(a(t)) = a(t) \end{aligned}$$

Exercise: verify that composition is associative.

### 28.3.11 Matrix Example

This matrix example is from John Perry[20]. Let  $N = \{1, 2, 3, \dots\}$ . Let  $m, n \in N$ . Then the set of  $m \times n$  matrices with integer entries, written  $\mathbb{Z}^{m \times n}$ , satisfies properties that make it a monoid under addition:

- closure is guaranteed by the definition;
- the associative property is guaranteed by the associative property of its elements; and
- the additive identity is 0, the zero matrix

## 28.4 MapReduce Example: Not a Monoid

We follow the example provided by Jimmy Lin[13]. Given a large number of key–value pairs where the keys are strings and the values are integers, the goal is to find the average of all the values by key. In SQL, this is accomplished as (assuming that our `mytable` has `key` and `value` columns):

- Select All Data

SELECT key, value FROM mytable;	
key	value
key1	10
key1	20
key1	30
key2	40
key2	60
key3	20
key3	30

- Select All Data and GROUP BY key

SELECT key, AVG(value) FROM mytable GROUP BY key;	
key	value
key1	20
key2	50
key3	25

Here is the first version of MapReduce algorithm, where the mapper is not generating monoid outputs for the mean/average function.:

**Listing 28.2:** Mapper: not generating monoids

```
1 /**
2  * @param key a string object
3  * @param value an long associated with key
4 */
5 map(String key, Long value) {
6     emit(key, value);
7 }
```

**Listing 28.3:** Reducer: not receiving monoids

```
1 /**
2  * @param key a string object
3  * @param values as list of longs: [i1, i2, ...]
4 */
5 reduce(String key, List<Long> list) {
6     Long sum = 0;
7     Integer count = 0;
8     for (Long i : list) {
9         sum = sum + i;
10        count++;
11    }
12    double average = sum/count;
13    emit(key, average);
14 }
```

Two observations from the first version of MapReduce algorithm:

- the algorithm is not very efficient: since there will be too much work by shuffle() and sort() functions of MapReduce framework
- We can not use the reducer as a combiner: since we know that the mean of means of arbitrary subsets of a set of values is not the same as the mean of the set of values.

Note that using combiners make MapReduce algorithms efficient by reducing network traffic (you need to ensure that the combiner provides sufficient aggregation) and reducing the load of shuffle() and sort() functions of MapReduce framework. Now the question is how can we use our reducer to work as a combiner? The answer is to make output of the mapper to be a monoid: we change the output of a mapper. Once your mapper outputs monoids, then combiners and reducers will behave correctly.

## 28.5 MapReduce Example: Monoid

We are revising the mapper to generate key-value pairs where key is the string and value is a pair (sum, count), which has a monoid property.

**Listing 28.4:** Mapper: generating monoids

```
1  /**
2   * @param key a string object
3   * @param value a Pair(long : sum, int: count) associated with key
4   */
5  map(String key, Long value) {
6      emit(key, Pair(value, 1));
7 }
```

As you can see, the key is the same as before, but the value is a pair of (sum, count). Now, the output of mapper is a monoid where the identity element is (0, 0). The element-wise sum operation can be performed as:

$$(a, b) \oplus (c, d) = (a + c, b + d)$$

Now the mean function will be calculated correctly since mappers output monoids:

$$\begin{aligned} \text{MEAN}(1, 2, 3, 4, 5) &= \text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5)) \\ \text{MEAN}(1, 2, 3, 4, 5) &= \frac{(1 + 2 + 3 + 4 + 5)}{5} = \frac{15}{5} = 3 \\ \text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5)) &= \text{MEAN}(\text{MEAN}(6, 3), \text{MEAN}(9, 2)) = \text{MEAN}(15, 5) = 3 \end{aligned}$$

The revised algorithm, where mappers (defined above) outputs are monoids is presented below:

**Listing 28.5:** New Combiner: receiving monoid values

```
1 /**
2  * @param key a string object
3  * @param value is a list = [(v1, c1), (v2, c2), ...]
4  */
5 combine(String key, List<Pair<Long, Integer>> list) {
6     Long sum = 0;
7     Integer count = 0;
```

```

8   for (Pair<Long, Integer> pair : list) {
9     sum += pair.v;
10    count += pair.c
11  }
12  emit(key, new Pair(sum, count));
13 }

```

---

**Listing 28.6:** New Reducer: receiving monoid values

```

1 /**
2  * @param key a string object
3  * @param value is a list = [(v1, c1), (v2, c2), ...]
4 */
5 reduce(String key, List<Pair<Long, Integer>> list) {
6   Long sum = 0;
7   Integer count = 0;
8   for (Pair<Long, Integer> pair : list) {
9     sum += pair.v;
10    count += pair.c
11  }
12  double average = sum/count;
13  emit(key, average);
14 }

```

---

## 28.6 Hadoop Implementation of Monodized MapReduce

The Java classes used in Monodized MapReduce solution are listed below. In our solution, we utilized the `PairOfLongInt`<sup>6</sup> class to represent a Hadoop's `WritableComparable` representing a pair consisting of a `long` and an `int`.

---

<sup>6</sup> <http://lintool.github.io/Cloud9/docs/api/edu/umd/cloud9/io/pair/PairOfLongInt.html>  
(Cloud<sup>9</sup>, developed by Jimmy Lin, is a collection of Hadoop tools that tries to make working with big data a bit easier.)

<i>Class name</i>	<i>Description</i>
MeanDriver.java	A driver program to submit Hadoop jobs
MeanMonodizedMapper.java	Defines map()
MeanMonodizedCombiner.java	Defines combiner
MeanMonodizedReducer.java	Defines reduce()
SequenceFileWriterDemo.java	Creates a sample SequenceFile
HadoopUtil.java	Defines some utility functions
edu.umd.cloud9.io.pair.PairOfLongInt	WritableComparable representing a pair consisting of a long and an int.

## 28.7 Sample Run of Monodized Hadoop/MapReduce

### 28.7.1 Create Input File (as a SequenceFile)

We used SequenceFileWriterDemo class to generate sample input as a SequenceFile<sup>7</sup>. The advantage of SequenceFile(s) over text file(s) is that in the map() function, we do not need to parse input to identify key and value fields. SequenceFile(s) can also be compressed, either per record or per block. To create a SequenceFile, we use SequenceFile.createWriter() method which returns a SequenceFile.Writer instance. We then write key-value pairs using the SequenceFile.Writer.append(key, value) method. After we are done, we finally call the close() method.

```
$ java SequenceFileWriterDemo test.seq
key1 1
key1 2
key1 3
key1 4
key1 5
key2 2
key2 4
key2 6
key2 8
```

---

<sup>7</sup><http://wiki.apache.org/hadoop/SequenceFile>

```
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
```

### 28.7.2 Create HDFS Input and Output Directories

```
$ hadoop fs -mkdir /monoid
$ hadoop fs -mkdir /monoid/input
$ hadoop fs -mkdir /monoid/output
```

### 28.7.3 Copy Input File to HDFS and Verify

```
$ 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
```

#### 28.7.4 Prepare a shell script to run your MapReduce job

```
$ cat run.sh
#!/bin/bash

export HADOOP_HOME=/Users/mahmoud/zmp/zs/hadoop
export HADOOP_HOME_WARN_SUPPRESS=true

export JAVA_HOME='/usr/libexec/java_home'
echo "JAVA_HOME=$JAVA_HOME"

PATH=.::/usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
export PATH=$PATH:$HADOOP_HOME/bin:$JAVA_HOME/bin

CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-ant-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-core-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-examples-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-test-1.2.1.jar
CLASSPATH=$CLASSPATH:$HADOOP_HOME/hadoop-tools-1.2.1.jar
```

```

LIB_DIR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/lib
JAR_FILES='find $LIB_DIR -name \'*.jar\''
for jarfile in $JAR_FILES ; do
CLASSPATH=$CLASSPATH:$jarfile
done

echo "CLASSPATH=$CLASSPATH"

export JAR=/Users/mahmoud/zmp/map_reduce_book/hadooptests/monoid/mean_as_monoid.jar
export CLASSPATH=$CLASSPATH:$JAR
export HADOOP_CLASSPATH=$CLASSPATH

javac *.java
jar cvf $JAR *.class
$HADOOP_HOME/bin/hadoop fs -rm /lib/mean_as_monoid.jar
$HADOOP_HOME/bin/hadoop fs -put $JAR /lib/
$HADOOP_HOME/bin/hadoop fs -rmr /monoid/output
$HADOOP_HOME/bin/hadoop jar $JAR MeanDriver /monoid/input /monoid/output

```

### 28.7.5 Run MapReduce Job

```

$ ./run.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
...
added manifest
adding: HadoopUtil.class(in = 1797) (out= 840)(deflated 53%)
adding: MeanDriver.class(in = 2220) (out= 1109)(deflated 50%)
adding: MeanMonodizedCombiner.class(in = 1702) (out= 736)(deflated 56%)
adding: MeanMonodizedMapper.class(in = 1548) (out= 616)(deflated 60%)
adding: MeanMonodizedReducer.class(in = 1741) (out= 761)(deflated 56%)
adding: SequenceFileWriterDemo.class(in = 2148) (out= 1070)(deflated 50%)
Deleted hdfs://localhost:9000/lib/mean_as_monoid.jar
Deleted hdfs://localhost:9000/monoid/output
...
13/10/07 14:45:14 INFO mapred.JobClient: Running job: job_201310071402_0006
13/10/07 14:45:15 INFO mapred.JobClient: map 0% reduce 0%
13/10/07 14:45:19 INFO mapred.JobClient: map 100% reduce 0%

```

```

13/10/07 14:45:28 INFO mapred.JobClient: map 100% reduce 3%
...
13/10/07 14:46:05 INFO mapred.JobClient: map 100% reduce 100%
13/10/07 14:46:05 INFO mapred.JobClient: Job complete: job_201310071402_0006
...
13/10/07 14:46:05 INFO mapred.JobClient: Reduce input records=5
13/10/07 14:46:05 INFO mapred.JobClient: Reduce input groups=5
13/10/07 14:46:05 INFO mapred.JobClient: Combine output records=5
13/10/07 14:46:05 INFO mapred.JobClient: Reduce output records=5
13/10/07 14:46:05 INFO mapred.JobClient: Map output records=25

```

### 28.7.6 View Hadoop Output

```

$ hadoop fs -text /monoid/output/part*
key2      6.0
key3      9.0
key4     12.0
key5     15.0
key1      3.0

```

## 28.8 Conclusion on Using Monoids

We observed that in MapReduce, if your mapper generates monoids then you can utilize combiners for optimization and efficiency purposes (using combiners reduces network traffic and make MapReduce's sort() and shuffle() functions more efficient by processing less data). Also, we showed that how to monodify MapReduce algorithms. The challenge to us is to monodify our MapReduce algorithms. In general, combiners can be used when the function you want to apply is both commutative and associative (properties of a monoid). For example, classic word count is a monoid over a set of integers with the + operation (here you can use a combiner). But the mean function (which is not associative as shown in the counter example above) over a set of integers does not form a monoid. Therefore, if combiners used properly, then it will significantly cuts down the amount of data shuffled from the maps to the reducers.

Monoids have other applications and usefulness in functional programming as well. For example, MarkCC<sup>8</sup> states that

”Why should we care if data structures like are monoids? Because we can write very general code in terms of the algebraic construction, and then use it over all of the different operations. Monoids provide the tools you need to build fold operations. Every kind of fold — that is, operations that collapse a sequence of other operations into a single value — can be defined in terms of monoids. So you can write a fold operation that works on lists, strings, numbers, optional values, maps, and god-only-knows what else. Any data structure which is a monoid is a data structure with a meaningful fold operation: monoids encapsulate the requirements of foldability.”

## 28.9 Functors and Monoids

Now that we have seen the definition and use of monoids in MapReduce framework, we can even apply higher-order functions (like ”functors”) to monoids. A functor is an object that is a function (or we can say that it is a function and object at the same time). The Java<sup>9</sup> programming language (JDK6 and JDK7) does not have direct concept of functors, because functions are not first-class objects in Java; it means that you can not pass a function name as an argument to another function. There is a simple way to simulate functors in Java by defining an interface and a method (this is a very simplistic simulation):

```
public interface FunctorSimulation<T1, T2> {
    T2 apply(T1 input);
}
```

---

<sup>8</sup> <http://scientopia.org/blogs/goodmath/2012/05/13/introducing-algebraic-data-structures-via-category-theory-monoids/>

<sup>9</sup> JDK8 has a direct support for functors and it is named lambdas (for details see <http://openjdk.java.net/projects/lambda/>)

For details on implementation/simulation of functors in Java, you may refer to Guava's<sup>10</sup> `Function` interface and Apache's Commons<sup>11</sup> `Functor` interface. The Haskell<sup>12</sup> programming language has a direct support for monoids and functors. Bruno P. Kinoshita has a good example of using Apache Commons Functor functional interfaces with Java 8 lambdas.

First, we present a "functor" on a monoid by a simple example: Let `MONOID = (t, e, f)` be a monoid, where `t` is a type (set of values), `e` is the identity element, and `f` is the "+" binary plus function:

```
MONOID = {
    type t
    val e : t
    val plus : t x t -> t
}
```

Then we define a functor `Prod` as

```
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))
}
```

Then we can define other functors such as `Square` as:

```
functor Square (M : MONOID) : MONOID = Prod M M
```

Also, a functor between two monoids can be defined as: Let  $(M_1, f_1, e_1)$  and  $(M_2, f_2, e_2)$  be monoids. A functor

$$F : (M_1, f_1, e_1) \rightarrow (M_2, f_2, e_2)$$

is specified by a an object map (monoids are categories with a single object) and an arrow map:  $F : M_1 \rightarrow M_2$  and the following conditions will hold:

$$\begin{aligned} \forall a, b \in M_1, F(f_1(a, b)) &= f_2(F(a), F(b)) \\ F(e_1) &= e_2 \end{aligned}$$

---

<sup>10</sup> <https://code.google.com/p/guava-libraries/>

<sup>11</sup> <http://commons.apache.org/proper/commons-functor/>

<sup>12</sup> <http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

A functor between two monoids is just a "monoid homomorphism." For example, for String data type, function Length() that counts the number of letters in a word is a monoid homomorphism.

- $\text{Length}("") = 0$  (length of an empty string is 0)
- If  $\text{Length}(x) = m$  and  $\text{Length}(y) = n$ , then concatenation  $x + y$  of strings has  $m+n$  letters, for example  
 $\text{Length}("String" + "ology") = \text{Length}("Stringology") = 11 = 6 + 5 = \text{Length}("String") + \text{Length}("ology").$

# Chapter 29

## The Small Files Problem

### 29.1 Introduction

This chapter provides an efficient solution to the "small files" problem. In general, Hadoop handles big files very well, but when the files are small, it just passes each small file to a `map()` function, which is not very efficient because the number of mappers will be big. Having too many small files can be problematic in Hadoop (a specific implementation of a MapReduce paradigm). What does this mean? Hadoop's distributed file system (HDFS) default block size is 64MB (or 67,108,864 bytes). The block size is defined by a parameter called `dfs.block.size`. If you have an application which deals with huge files (such as DNA-Sequencing), then you may even set this to higher size like 128MB.

What is a small file in MapReduce/Hadoop environment? A "small file" is file which is significantly smaller than the HDFS's default block size. Typically, if you're using and storing small files, then you probably have lots of them. For example, to represent a bioset for a gene-expression data type, a file size can be from 2 to 3 MB. So, to process 1000 biosets, you need 1000 mappers (which will be very inefficient – each file will be sent to a mapper). To solve this problem, we should merge many of these small files into one and then process them. In the case of biosets, we might merge every 20 to 25 files into one file (where the size will be closer to 64MB). By merging these files, we might need only 40 to 50 mappers. For details on "small file" problem, refer to [The Small Files Problem](#). Also, note that Hadoop is mainly designed

for batch processing of large volume of data (rather than processing many small files).

## 29.2 Solution to The Small Files Problem

Let's assume that we have to process 20,000 small files (assuming that each file size is much smaller than 64MB) and we want to process them efficiently in MapReduce/Hadoop environment. If you just send these files as input by `FileInputFormat.addInputPath(Job, Path)` as input, then each input file will be sent to a mapper and you will end up with 20,000 mappers, which is very inefficient. Let `dfs.block.size` be 64MB. Further assume that the size of these files are between 2 to 3 MB (so we assume that on average each small file size is 2.5 MB). Further, assume that we have  $M$  (such as 100, 200, 300, ...) mappers available to us. The following multi-threaded algorithm (which is a POJO and non-MapReduce) will solve the small files problem (since our small files are in the range of 2 to 3 MB, we assume that on average they occupy 2.5 MB — therefore we can put 25 [ $25 \times 2.5 \approx 64MB$ ] small files as a one HDFS block, which we call a bucket). Now that we can put 25 small files into a single bucket (which its size will be less than 64MB — ideal size for HDFS), therefore we just need 80 ( $2000 \div 25 = 80$ ) mappers, which will be very efficient compared to 2000 mappers. Our algorithm puts  $N$  files (in our example it is 25) into each bucket and then concurrently merges these small files into one file, which the size is closer to the `dfs.block.size`.

Before submitting small files to MapReduce/Hadoop, we merge these small files into big ones and then submit them into MapReduce driver program. The following code fragment (taken from the driver program which submits MapReduce/Hadoop jobs) shows how to merge small files into a file, where the merged file size is closer to `dfs.block.size`.

**Listing 29.1:** Merging Small Files into a Large File

```
1 // prepare input
2 int NUMBER_OF_MAP_SLOTS_AVAILABLE = <M>;
3 Job job = <define-a-job>;
4 List<Path> smallFiles = <HDFS small input files: file1, file2, ...>;
5 int numberofSmallFiles = smallFiles.size();
6 if ( NUMBER_OF_MAP_SLOTS_AVAILABLE >= numberofSmallFiles ) {
7     // we have enough mappers and there is no need
8     // to merge or consolidate small files; each
```

```

9     // small file will be sent as a block to a mapper
10    for (Path path : smallFiles) {
11        FileInputFormat.addInputPath(job, path);
12    }
13 }
14 else {
15     // the number of mappers are less than the number of small files
16     // create and fill buckets with merged small files
17
18     // Step-1: create empty buckets (each bucket may hold a set of small files)
19     BucketThread[] buckets = SmallFilesConsolidator.createBuckets(
20         smallFiles,
21         NUMBER_OF_MAP_SLOTS_AVAILABLE);
22
23     // Step-2: fill buckets with small files
24     SmallFilesConsolidator.fillBuckets(buckets, smallFiles, job);
25
26     // Step-3: Merge small files per bucket.
27     // each bucket is a thread (implements Runnable interface )
28     // merging is done concurrently for each bucket
29     SmallFilesConsolidator.mergeEachBucket(buckets, job);
30 }

```

---

The `SmallFilesConsolidator` class accepts a set of small Hadoop files and then merges these small files together into larger (size to be closer to `dfs.block.size`) Hadoop files. The optimal solution is to create merged files, when their size is less than or equal to HDFS block size. We generate these large files (as a GUID) under the `"/tmp/"` directory in HDFS (of course you can make `"/tmp/"` as a configurable HDFS directory):

```

// this directory is configurable
private static String MERGED_HDFS_ROOT_DIR = "/tmp/";
...
private static String getParentDir() {
    String guid = UUID.randomUUID().toString();
    return MERGED_HDFS_ROOT_DIR + guid + "/";
}

```

The `BucketThread` class enable us to concatenate small files into one big file, which will be less than the HDFS block size. This way, we will submit less mappers with big input files. The `BucketThread` class implements `Runnable` interface and provide the `copyMerge()` method, which merges

small files into a larger file. Since each `BucketThread` object implements `Runnable` interface, therefore it will be able to run in its own thread. This way all `BucketThread` objects can merge their small files concurrently. The `BucketThread.copyMerge()` is the core method and merges all small files in one bucket into another temporary HDFS file. For example, if a bucket holds the following small files: {File1, File2, File3, File4}, then the merged file will look like (note that the MergedFile is the concatenation of all four files):

MergedFile	File1 File2 File3 File4
------------	----------------------------------

Implementation of `BucketThread.copyMerge()` method is given below:

### Listing 29.2: The copyMerge() Method

```

1 /**
2 * Copy all files in several directories to one output file (mergedFile).
3 *
4 * parentDir will be "/tmp/<guid>/"
5 * targetDir will be "/tmp/<guid>/id/"
6 * targetFile will be "/tmp/<guid>/id/id"
7 *
8 * merge all paths in bucket and return a new directory
9 * (targetDir), which holds merged paths
10 */
11 public void copyMerge() throws IOException {
12
13     // if there is only one path/dir in the bucket,
14     // then there is no need to merge it
15     if ( size() < 2 ) {
16         return;
17     }
18
19     // here bucket.size() >= 2
20     Path hdfsTargetFile = new Path(targetFile);
21     OutputStream mergedFile = fs.create(hdfsTargetFile);
22     try {
23         for (int i = 0; i < bucket.size(); i++) {
24             FileStatus contents[] = fs.listStatus(bucket.get(i));
25             for (int k = 0; k < contents.length; k++) {
26                 if (!contents[k].isDir()) {
27                     InputStream smallFile = fs.open(contents[k].getPath());
28                     try {
29                         IOUtils.copyBytes(smallFile, mergedFile, conf, false);
30                     }
31                     finally {

```

```

32             HadoopUtil.close(smallFile);
33         }
34     }
35 } //for k
36 } // for i
37 }
38 finally {
39     HadoopUtil.close(mergedFile);
40 }
41 }

```

---

The `SmallFilesConsolidator` class provides 3 piece of functionality:

1. create required empty buckets: each bucket will hold a set of small files.  
This will be done by `SmallFilesConsolidator.createBuckets()`.
2. fill buckets: we will place enough small files in a bucket so that the total size of all small files will be about `dfs.block.size`. This behavior is implemented by `SmallFilesConsolidator.fillBuckets()`
3. merge each bucket: here we will merge all small files in the bucket to create a single large file, where its size will be about `dfs.block.size`.  
This is accomplished by `SmallFilesConsolidator.mergeEachBucket()`

To demonstrate the small files problem, we will run the classic `word count` with and without `SmallFilesConsolidator` class. For input, for each case we will use 30 small files. As clearly we see that using `SmallFilesConsolidator` outperforms the original solution. For example, using 30 small files, the "Solution With SmallFilesConsolidator" finished in 58,235 milliseconds while the original word count with small files finished in 80,435 milliseconds.

### 29.2.1 Input Data

We used the following input data (30 small files) for both solutions.

```

# hadoop fs -ls /small_input_files/input/
Found 30 items
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-1
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-10
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-11
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-12
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-13
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-14
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-15
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-16

```

```

-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-17
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-18
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-19
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-2
-rw-r--r-- 1 mahmoud staff 51 2013-11-05 10:16 /small_input_files/input/Document-20
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-21
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-22
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-23
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-24
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-25
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-26
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-27
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-28
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-29
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-3
-rw-r--r-- 1 mahmoud staff 33 2013-11-05 10:16 /small_input_files/input/Document-30
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-4
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-5
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-6
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-7
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-8
-rw-r--r-- 1 mahmoud staff 31 2013-11-05 10:16 /small_input_files/input/Document-9

```

## 29.3 Solution With SmallFilesConsolidator

In this solution we will use `SmallFilesConsolidator` class to merge the small files into a larger file.

### 29.3.1 Java Source Files

Class name	Class Description
BucketThread	Used to merge small files to larger files
HadoopUtil	Defines some basic Hadoop utilities
SmallFilesConsolidator	Manages consolidation of small files into a larger file
WordCountDriverWithConsolidator	Wrod Count Driver with Consolidator
WordCountMapper	Defines map()
WordCountReducer	Defines reduce() and combine()

### SmallFilesConsolidator Class

The `SmallFilesConsolidator` class is a driver class to consolidate the small ones into files, where the size is closer to the HDFS's block size. The main methods are:

- `getNumberOfBuckets()`: determines the number of buckets needed for merging all files into bigger files.

```
public static int getNumberOfBuckets(int totalFiles,
                                    int numberOfMapSlotsAvailable,
                                    int maxFilesPerBucket)
```

- `createBuckets()`: create required buckets

```
public static BucketThread[] createBuckets(
    int totalFiles,
    int numberOfMapSlotsAvailable,
    int maxFilesPerBucket)
```

- `fillBuckets()`: this method fills each bucket with small files

```
public static void fillBuckets(
    BucketThread[] buckets,
    List<String> smallFiles, // list of small files
    Job job,
    int maxFilesPerBucket)
```

- `mergeEachBucket()`: this method merges small files to create a larger file

```
public static void mergeEachBucket(BucketThread[] buckets,
                                   Job job)
```

### 29.3.2 Sample Run

```
# ./run_with_consolidator.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
added manifest
adding: BucketThread.class(in = 4068) (out= 1974)(deflated 51%)
adding: HadoopUtil.class(in = 2981) (out= 1317)(deflated 55%)
adding: SmallFilesConsolidator.class(in = 3091) (out= 1664)(deflated 46%)
adding: WordCountDriverWithConsolidator.class(in = 5277) (out= 2557)(deflated 51%)
adding: WordCountDriverWithoutConsolidator.class(in = 3280) (out= 1602)(deflated 51%)
adding: WordCountMapper.class(in = 2705) (out= 1114)(deflated 58%)
adding: WordCountReducer.class(in = 1598) (out= 665)(deflated 58%)
Deleted hdfs://localhost:9000/small_input_files/output
13/11/05 10:54:04 INFO WordCountDriverWithConsolidator: inputDir=/small_input_files/input
13/11/05 10:54:04 INFO WordCountDriverWithConsolidator: outputDir=/small_input_files/output
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/0
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/1
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/2
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/3
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/4
```

```

13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/5
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/6
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/7
...
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 10:54:12 INFO mapred.JobClient: map 25% reduce 0%
...
13/11/05 10:55:00 INFO mapred.JobClient: map 100% reduce 83%
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: Counters: 26
13/11/05 10:55:02 INFO mapred.JobClient: Job Counters
13/11/05 10:55:02 INFO mapred.JobClient: Launched reduce tasks=10
13/11/05 10:55:02 INFO mapred.JobClient: SLOTS_MILLIS_MAPS=34127
13/11/05 10:55:02 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving slots (ms)=0
13/11/05 10:55:02 INFO mapred.JobClient: Total time spent by all maps waiting for reserving slots (ms)=0
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: SLOTS_MILLIS_REDUCES=96622
13/11/05 10:55:02 INFO mapred.JobClient: File Output Format Counters
13/11/05 10:55:02 INFO mapred.JobClient: Bytes Written=55
13/11/05 10:55:02 INFO mapred.JobClient: FileSystemCounters
13/11/05 10:55:02 INFO mapred.JobClient: FILE_BYTES_READ=579
13/11/05 10:55:02 INFO mapred.JobClient: HDFS_BYTES_READ=2140
13/11/05 10:55:02 INFO mapred.JobClient: FILE_BYTES_WRITTEN=1215442
13/11/05 10:55:02 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=55
13/11/05 10:55:02 INFO mapred.JobClient: File Input Format Counters
13/11/05 10:55:02 INFO mapred.JobClient: Bytes Read=1092
13/11/05 10:55:02 INFO mapred.JobClient: Map-Reduce Framework
13/11/05 10:55:02 INFO mapred.JobClient: Map output materialized bytes=999
13/11/05 10:55:02 INFO mapred.JobClient: Map input records=48
13/11/05 10:55:02 INFO mapred.JobClient: Reduce shuffle bytes=999
13/11/05 10:55:02 INFO mapred.JobClient: Spilled Records=96
13/11/05 10:55:02 INFO mapred.JobClient: Map output bytes=1704
13/11/05 10:55:02 INFO mapred.JobClient: Total committed heap usage (bytes)=2334351360
13/11/05 10:55:02 INFO mapred.JobClient: Combine input records=201
13/11/05 10:55:02 INFO mapred.JobClient: SPLIT_RAW_BYTES=1048
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: Combine output records=48
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: run(): status=true
13/11/05 10:55:02 INFO WordCountDriverWithConsolidator: submitJob(): status=0
13/11/05 10:55:02 INFO WordCountDriverWithConsolidator: returnStatus=0
13/11/05 10:55:02 INFO WordCountDriverWithConsolidator: Finished in milliseconds: 58235

```

As you can observe from the log of sample run, we have consolidated 30 HDFS small files into 8 large HDFS files:

```

13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/0
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/1
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/2
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/3
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/4
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/5
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/6
13/11/05 10:54:05 INFO SmallFilesConsolidator: added path: /tmp/906e6c30-c411-4a70-b68f-114ba7511e63/7

```

## 29.4 Solution Without SmallFilesConsolidator

This solution is just a basic word count without using the `SmallFilesConsolidator` class. As we notice from the sample run (below), the total number of input paths to process is 30, which is exactly the number of the small files we want to process. This solution is not an optimal solution at all since every small file will be sent to a mapper. The ideal case is to send input files, which their size is greater than or equal to HDFS block size (Hadoop is designed for handling large files).

```
...
13/11/05 10:29:13 INFO input.FileInputFormat: Total input paths to process : 30
...
```

### 29.4.1 Java Source Files

Class name	Class Description
HadoopUtil	Defines some basic Hadoop utilities
WordCountDriverWithoutConsolidator	Word Count Driver without Consolidator
WordCountMapper	Defines map()
WordCountReducer	Defines reduce() and combine()

### 29.4.2 Sample Run

```
# ./run_without_consolidator.sh
JAVA_HOME=/System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
added manifest
adding: BucketThread.class(in = 4068) (out= 1974)(deflated 51%)
adding: HadoopUtil.class(in = 2981) (out= 1317)(deflated 55%)
adding: SmallFilesConsolidator.class(in = 3091) (out= 1664)(deflated 46%)
adding: WordCountDriverWithConsolidator.class(in = 5277) (out= 2557)(deflated 51%)
adding: WordCountDriverWithoutConsolidator.class(in = 3280) (out= 1602)(deflated 51%)
adding: WordCountMapper.class(in = 2705) (out= 1114)(deflated 58%)
adding: WordCountReducer.class(in = 1598) (out= 665)(deflated 58%)
Deleted hdfs://localhost:9000/small_input_files/output
13/11/05 10:29:12 INFO WordCountDriverWithoutConsolidator: inputDir=/small_input_files/input
13/11/05 10:29:12 INFO WordCountDriverWithoutConsolidator: 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 10:29:22 INFO mapred.JobClient: map 16% reduce 0%
...
13/11/05 10:30:31 INFO mapred.JobClient: map 100% reduce 93%
```

```

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: Counters: 26
13/11/05 10:30:33 INFO mapred.JobClient: Job Counters
13/11/05 10:30:33 INFO mapred.JobClient: Launched reduce tasks=10
13/11/05 10:30:33 INFO mapred.JobClient: SLOTS_MILLIS_MAPS=144653
13/11/05 10:30:33 INFO mapred.JobClient: Total time spent by all reduces waiting after reserving slots (ms)=0
13/11/05 10:30:33 INFO mapred.JobClient: Total time spent by all maps waiting after reserving slots (ms)=0
13/11/05 10:30:33 INFO mapred.JobClient: Launched map tasks=30
13/11/05 10:30:33 INFO mapred.JobClient: Data-local map tasks=30
13/11/05 10:30:33 INFO mapred.JobClient: SLOTS_MILLIS_REDUCES=137261
13/11/05 10:30:33 INFO mapred.JobClient: File Output Format Counters
13/11/05 10:30:33 INFO mapred.JobClient: Bytes Written=55
13/11/05 10:30:33 INFO mapred.JobClient: FileSystemCounters
13/11/05 10:30:33 INFO mapred.JobClient: FILE_BYTES_READ=1686
13/11/05 10:30:33 INFO mapred.JobClient: HDFS_BYTES_READ=4743
13/11/05 10:30:33 INFO mapred.JobClient: FILE_BYTES_WRITTEN=2687892
13/11/05 10:30:33 INFO mapred.JobClient: HDFS_BYTES_WRITTEN=55
13/11/05 10:30:33 INFO mapred.JobClient: File Input Format Counters
13/11/05 10:30:33 INFO mapred.JobClient: Bytes Read=1092
13/11/05 10:30:33 INFO mapred.JobClient: Map-Reduce Framework
13/11/05 10:30:33 INFO mapred.JobClient: Map output materialized bytes=3426
13/11/05 10:30:33 INFO mapred.JobClient: Map input records=48
13/11/05 10:30:33 INFO mapred.JobClient: Reduce shuffle bytes=3426
13/11/05 10:30:33 INFO mapred.JobClient: Spilled Records=306
13/11/05 10:30:33 INFO mapred.JobClient: Map output bytes=1704
13/11/05 10:30:33 INFO mapred.JobClient: Total committed heap usage (bytes)=6399668224
13/11/05 10:30:33 INFO mapred.JobClient: Combine input records=201
13/11/05 10:30:33 INFO mapred.JobClient: SPLIT_RAW_BYTES=3651
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: returnStatus=0
13/11/05 10:30:33 INFO WordCountDriverWithoutConsolidator: Finished in milliseconds: 80435

```

# Chapter 30

## Huge Cache for MapReduce

### 30.1 Introduction

The purpose of this chapter is to show how to use and read a huge cache (comprised of billions of (key, value) pairs, which can not fit in commodity server's memory) in MapReduce algorithms. The algorithms presented in this chapter are generic enough and can be used in any MapReduce paradigms (such as MapReduce/Hadoop and Spark/Hadoop).

There exist MapReduce algorithms, which might require access to some huge (in billions of records) static reference relational tables. Typically, these reference relational tables do not change for a long period of time, but they are needed either in `map()` or `reduce()` phase of MapReduce programs. One example of such a table is a "position feature" table, which is used for germline<sup>1</sup> datatype ingestion and variant classification. The "position feature" table might have the following attributes (a composite key is (`chromosome_id`, `position`)):

---

<sup>1</sup>Germline refers to the sequence of cells in the line of direct descent from zygote to gametes, as opposed to somatic cells (all other body cells). Mutations in germline cells are transmitted to offspring; those in somatic cells are not. source: <http://medical-dictionary.thefreedictionary.com/germline>

Table: position_feature	
Column Name	Characteristics
chromosome_id	PK-1
position	PK-2
feature_id	basic attribute
mrna_feature_id	basic attribute
sequence_data_type_id	basic attribute
mapping	basic attribute

In expressing your solution in MapReduce paradigm, either in `map()` or `reduce()`, given a `key=(chromosome_id, position)`, you want to return a `List<String>` where each element of list is comprised of remaining attributes `{feature_id, mrna_feature_id, sequence_data_type_id, mapping}`. For a germline data type, a "position feature" table can have up to 12 billion records (which might take about one TB of disk space in MySQL or Oracle database system). Now imagine that your mapper or reducer wants to access the "position feature" table for a given `key=(chromosome_id, position)`. Since you will be firing many requests (several millions per second) of this type, it will bring your database server down to its knees and will not scale (this is a fact). One possible solution is to cache all static data into a hash table and then use the hash table instead of using a relational table (as we will see shortly in the next section, this is not a proper solution at all and it will not scale – since the size of such a Hash Table will be about one TB and will not fit in memory).

## 30.2 Implementation Options

What are the optimal and pragmatic options for caching 12 billion records in a MapReduce environment? In this section, I will present several options and discuss the feasibility of their implementations.

- **OPTION-1:** Use a relational database (such as MySQL or Oracle). This option is not a viable solution and will not scale at all. A mapper or reducer will constantly hit the database and the database server will not be able to handle thousands or millions of requests per second. If even we use a set of replicated relational databases, still this option will

not scale out (since the number of database connections are limited for a large set of Hadoop cluster).

- **OPTION-2:** Use a memcached server. Memcached<sup>2</sup> is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering. If for every few slave nodes you have a dedicated memcached cluster (which will be very costly in a large cluster environment), then this is a viable and proper option. This solution will not scale out due to high cost of having so many expensive memcached servers.
- **OPTION-3:** Use a Redis server. Redis<sup>3</sup> is an open source, BSD licensed, advanced key-value store. It is often referred to as a data structure server since keys can contain strings, hashes, lists, sets and sorted sets. Again, if for every few slave nodes you have a dedicated redis server (which will be very costly in a large cluster environment), then this a viable and proper option. This solution will not scale out due to high cost of having so many expensive redis servers (for one TB of (key, value) pairs, Redis requires at least 6 TB of RAM).
- **OPTION-4:** Use a MapReduce join between your input and 12 billion static records. The idea is to flatten 12 billion records and then use a MapReduce join between this 12 billion records and your input. Perform the join on the `key=(chromosome_id, position)`. This is a viable option if your input is huge too (I mean in billions rather than in millions). For example, for a germline ingestion process, the VCF<sup>4</sup> file will not have more than 6 million records. It is not proper to join a table of 6 million records against 12 billion records (wast of lots of time).
- **OPTION-5:** Partition 12 billion static records into small chunks (each chuck will be 64MB – easily you can load into RAM and when not needed you can evict it) and use Map-LRU (least recently used) Hash Table. This simple elegant idea works very well: scales out and you

---

<sup>2</sup><http://memcached.org/>

<sup>3</sup><http://redis.io/>

<sup>4</sup>VCF – Variant Call Format – is a text file format. It contains meta-information lines, a header line, and then data lines each containing information about a position in the genome. For more detail, see <http://samtools.github.io/hts-specs/VCFv4.2.pdf>

do not need dedicated cache servers. This solution is detailed out in the following sections. No matter what, in a distributed environment, the mapper or reducer does not have an access to an unlimited amount of memory/RAM. This solution will work out in any MapReduce environment and does not need extra RAM for caching purposes.

### 30.3 Fromalizing the Cache Problem

In a nutshell, given a set of 12 billion records, the goal is to find an associated value for a given composite key `key=(chromosome_id, position)`. Let's say that you are going to ingest a VCF file (this is our input file – it is about 5 million records for a germline data type – your MapReduce input per job is a VCF file) into your genome system: for every record of a given VCF, you need to find out:

- mutation class
- list of genes
- list of feature
- ...

To find out detailed information per VCF record, you do need a set of static tables such as a "position feature" table (details were outlined in the introduction section). As we discussed before, a "position feature" table has at least 12 billion records. The first step of a germline ingestion process is to find records from a "position feature" table for a given `key=(chromosome_id, position)` (every VCF record will have `chromosome_id` and `position` fields).

### 30.4 Elegant Scalable Solution

The solution presented here is a local cache solution. This means that every worker/slave node of a MapReduce/Hadoop cluster will have its own local cache (on hard disk – hard disk is cheap enough) and there will not be a network traffic for accessing cache components and data. The local cache will reside in a hard disk and brought to memory on a needed basis by using LRU-Map cache technique. This is how it works: we assume that we have

a relational table called `position_feature` (as defined in the Introduction section of this chapter), which has over 12 billion records.

- First, we partition 12 billion records by `chromosome_id` (1, 2, 3, ..., 24, 25). This will give us 25 files (let's name these files are labeled as `chr1.txt`, `chr2.txt`, ..., `chr25.txt`), where each record (in these text files per chromosome) will have the following format (note that `key=(chromosome_id, position)` returns multiple records from the relational table):

```
<position><;><Record1><;><Record2><;>...<;><RecordN>
```

and each `record<i>` is comprised of the following

```
<feature_id><,;><mRNA_feature_id><,;><sequence_data_type_id><,;><mapping>
```

Therefore, each raw cache data file (`chr1.txt`, `chr2.txt`, ..., `chr25.txt`) corresponds to the following SQL query (you may repeat this script per `chromosome_id` (the following query is for `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;
```

- Next we sort each of these files (`chr1.txt`, `chr2.txt`, ..., `chr25.txt`) by position and generate `chr1.txt.sorted`, `chr2.txt.sorted`, ..., `chr25.txt.sorted`.
- Since memory is limited (say 1 to 4 GB at most) per Mapper/Reducer of a MapReduce job, then we partition each sorted file into chunks of 64MB (without breaking any lines). For example, to partition sorted files into 64MB chunks we execute the following command:

```
#!/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
```

For example, for `chromosome_id=1` we will have:

```
# 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
```

Note that each of these partitioned files have a range of `position` values. We will use these ranges in our cache implementation. Therefore, given a `chromosome_id=1` and a `position`, we exactly know, which

partition holds the result of a query. Let's look at the content of one of these sorted partitioned files:

```
# 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:...
```

You can see that all positions are sorted within each partition. To support metadata for all partitions in using "LRU Map", we do need an additional data structures to keep track of (begin, end) positions. For each partitioned file we will keep (`partition_name`, `begin`, `end`) information. For example for `chromosome_id=1` and `chromosome_id=2` we will have:

```
# 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
```

The partition data structure is illustrated below.

# Partition Data Structures

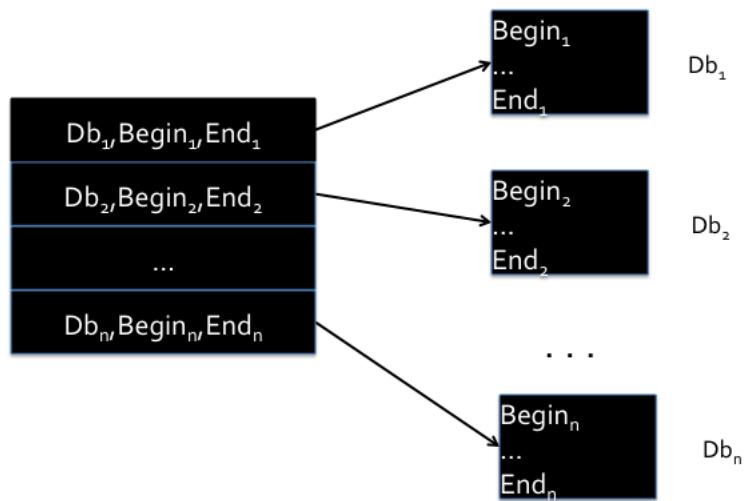


Figure 30.1: Hige Cache Partition Data Structure

- Next, we will convert each partition (of 64MB) into a Hash Table, implemented by MapDB<sup>5</sup>.
- Since memory is limited per Mapper/Reducer, we use `LRUMap<K, V>(N)` to hold at most  $N$  MapDB data structures (each MapDB persistent Map corresponds to one sorted partition (of 64MB)). The idea of `LRUMap<K, V>` is to hold at most  $N$  partitions such that  $N \times 64\text{MB}$  will be smaller than the memory available per mapper/reducer. When you want to insert the  $N+1$ 'st entry into `LRUMap<K, V>(N)`, then the oldest entry will be evicted (then you can properly close the MapDB object – releases all memory and closes all file handles). For LRU Map implementation, we use `org.apache.commons.collections4.map.LRUMap<K, V>`.
- The final thing we do is to sort our input by `key=(chromosome_id, position)`. This sorting will minimize the eviction of MapDB entries from `LRUMap<K, V>(N)`. The sorting of an input file is a huge design/implementation criteria and it will have a big impact on the performance of our MapReduce job (germline ingestion). Without sorting our input file, the eviction rate might be very high. But sorting the input file will minimize the eviction rate. Note that sorting input file is very fast and should not take more than few seconds (this can be done by a Linux `sort` command or you may use MapReduce/Hadoop to sort the input data).

## 30.5 Implementation of Elegant Scalable Solution

### 30.5.1 Use of LRU Map

For implementation of LRU (least recently used) Map, I selected the `org.apache.commons.collect` which is a Map implementation with a fixed maximum size which removes the least recently used entry if an entry is added when full. The least recently used algorithm works on the get and put operations only. Iteration of any kind, including setting the value by iteration, does not change the order (for

---

<sup>5</sup>MapDB is implemented by Jan Kotek and its source code is hosted at <https://github.com/jankotek/MapDB>. MapDB provides concurrent Maps, Sets and Queues backed by disk storage or off-heap-memory. It is a fast and easy to use embedded Java database engine (<http://www.mapdb.org>)

details, you may consult with Apache Commons Collections<sup>6</sup>. For our cache implemetation, we extend the LRUMap class (the MapDBEntry class is a simple class, which represents a sorted partition of 64MB as a Map data structure implemented in MapDB.

#### Listing 30.1: Custom LRU Map

```
1 import org.apache.commons.collections4.map.LRUMap;
2
3 public class CustomLRUMap<K, V> extends LRUMap<K, V> {
4
5     private K key = null;
6     private V value = null;
7     private LinkEntry<K, V> entry = null;
8
9     public CustomLRUMap(final int size) {
10         super(size);
11     }
12
13     @Override
14     protected boolean removeLRU(final LinkEntry<K, V> entry) {
15         System.out.println("begin remove LRU entry ...");
16         this.entry = entry;
17         this.key = entry.getKey();
18         this.value = entry.getValue();
19
20         if (key instanceof String) {
21             String keyAsString = (String) key;
22             System.out.println("evicting key=" + keyAsString);
23         }
24
25         if (value instanceof MapDBEntry) {
26             // release resources held by MapDBEntry
27             MapDBEntry mapdbEntry = (MapDBEntry) value;
28             mapdbEntry.close();
29         }
30
31         return true; // actually delete entry
32     }
33 }
```

### 30.5.2 Test LRU Map

The following test program shows how `CustomLRUMap<K, V>` (`CustomLRUMap<K, V>` extends the `LRUMap<K, V>` class and redefines the eviction policy by the `removeLRU()` method) works. In this example, we only keep 3 map entries at any time. No matter how many etries you add to the `LRUMap` object, the `LRUMap.size()`

<sup>6</sup><http://commons.apache.org/proper/commons-collections/>

can not exceed 3 (the maximum size is given at the time of creation of `CustomLRUMap<K, V>` object).

### Listing 30.2: Test Custom LRU Map

```
1 # cat CustomLRUMapTest.java
2
3 import org.apache.commons.collections4.map.LRUMap;
4
5 public class CustomLRUMapTest {
6     public static void main(String[] args) throws Exception {
7         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 }
22
23 # javac CustomLRUMapTest.java
24 # java CustomLRUMapTest
25 map={k1=v1, k2=v2, k3=v3}
26 begin removeLRU...
27 evicting key=k1
28 v=v2
29 map={k3=v3, k4=v4, k2=v2}
30 begin removeLRU...
31 evicting key=k3
32 map={k4=v4, k2=v2, k5=v5}
33 begin removeLRU...
34 evicting key=k4
35 map={k2=v2, k5=v5, k6=v6}
```

### Listing 30.3: MapDBEntry Class

```
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;
18         }
19         return map.get(key);
20     }
21
22     // eviction policy
23     public void close() {
24         closeDB();
25         closeMap();
26     }
27
28     private void closeDB() {
29         if (db != null) {
30             db.close();
31         }
32     }
33
34     private void closeMap() {
35         if (map != null) {
36             map = null;
37         }
38     }
39
40     public static MapDBEntry create(String dbName) {
41         DB db = DBMaker.newFileDB(new File(dbName))
42             .closeOnJvmShutdown()
43             .readOnly()
44             .make();
45         Map<String, String> map = db.getTreeMap("collectionName");
46         MapDBEntry entry = new MapDBEntry(db, map);
47         return entry;
48     }
49 }
```

---

### 30.5.3 Use of MapDB

How do we create persistent  $\text{Map}\langle K, V \rangle$  using MapDB? The following program shows how to create a MapDB for a sorted partition file:

**Listing 30.4:** MapDBEntry Class

```

1 import org.mapdb.DB;
2 import org.mapdb.DBMaker;
3 import java.io.BufferedReader;
4 import java.io.FileReader;
```

```

5 import java.io.File;
6 import java.util.concurrent.ConcurrentNavigableMap;
7
8 public class GenerateMapDB {
9
10    public static void main(String[] args) throws Exception {
11        String inputFileName = args[0];
12        String mapdbName = args[1];
13        create(inputFileName, mapdbName);
14    }
15
16    public static void create(String inputFileName, String mapdbName)
17        throws Exception {
18        //Configure and open database using builder pattern.
19        //All options are available with code auto-completion.
20        DB db = DBMaker.newFileDB(new File(mapdbName))
21            .closeOnJvmShutdown()
22            .make();
23
24        //open an collection, TreeMap has better performance then HashMap
25        ConcurrentNavigableMap<String, String> map = db.getTreeMap("collectionName");
26
27        //
28        // line = <position><;><v><:><v>:...:<v>
29        //   where <v> has the following format:
30        //   <feature_id><,><mrna_feature_id><,><sequence_data_type_id><,><mapping>
31        //
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                } else {
42                    System.out.println("error line="+line);
43                }
44            }
45        }
46    }
47    finally {
48        reader.close(); // close input file
49        db.commit(); // persist changes into disk
50        db.close(); // close database resources
51    }
52 }
53 }

```

---

To query MapDB database, we can write a very simple class:

### Listing 30.5: MapDBEntry Class

```
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 /**
8  * This class defines basic query on MapDB.
9  *
10 */
11 * @author Mahmoud Parsian
12 *
13 */
14 public class QueryMapDB {
15
16
17     public static void main(String[] args) throws Exception {
18         long beginTime = System.currentTimeMillis();
19         String mapdbName = args[0];
20         String position = args[1];
21         THE_LOGGER.info("mapdbName="+mapdbName);
22         THE_LOGGER.info("position="+position);
23         String value = query(mapdbName, position);
24         THE_LOGGER.info("value="+value);
25         long elapsedTime = System.currentTimeMillis() -beginTime;
26         THE_LOGGER.info("elapsedTime (in millis) =" +elapsedTime);
27         System.exit(0);
28     }
29
30     public static String query(String mapdbName, String key) throws Exception {
31         String value = null;
32         DB db = null;
33         try {
34             db = DBMaker.newFileDB(new File(mapdbName))
35                 .closeOnJvmShutdown()
36                 .readOnly()
37                 .make();
38             Map<String, String> map = map = db.getTreeMap("collectionName");
39             value = map.get(key);
40         }
41         finally {
42             if (db != null) {
43                 db.close();
44             }
45         }
46         return value;
47     }
48 }
```

### 30.5.4 Test of MapDB: put() and get()

#### 30.5.4.1 Test of MapDB: put()

```
# 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
```

As you can observe from the generated output, MapDB generates 2 files (`mapdbtest` and `mapdbtest.p`) for each persistent hash table.

#### 30.5.4.2 Test of MapDB: get()

```
# java QueryMapDB mapdbtest 19105201
Feb 11 2014 16:41:16 [main] [INFO ] [QueryMapDB] - mapdbName=mapdbtest
Feb 11 2014 16:41:16 [main] [INFO ] [QueryMapDB] - position=19105201
Feb 11 2014 16:41:16 [main] [INFO ] [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
Feb 11 2014 16:41:16 [main] [INFO ] [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 19105202
Feb 11 2014 16:41:21 [main] [INFO ] [QueryMapDB] - mapdbName=mapdbtest
Feb 11 2014 16:41:21 [main] [INFO ] [QueryMapDB] - position=19105202
Feb 11 2014 16:41:21 [main] [INFO ] [QueryMapDB] - value=35302633,35292056,2,1:20773813,35399339,2,1
Feb 11 2014 16:41:21 [main] [INFO ] [QueryMapDB] - elapsedTime (in millis) = 2

# java QueryMapDB mapdbtest 191052023333
```

```

Feb 11 2014 16:41:55 [main] [INFO ] [QueryMapDB] - mapdbName=mapdbtest
Feb 11 2014 16:41:55 [main] [INFO ] [QueryMapDB] - position=191052023333
Feb 11 2014 16:41:55 [main] [INFO ] [QueryMapDB] - value=null
Feb 11 2014 16:41:55 [main] [INFO ] [QueryMapDB] - elapsedTime (in millis) = 1

```

## 30.6 MapReduce Using LRU-Map-Cache

Now that we have an efficient LRU-Map-Cache, we may use it either in `map()` or `reduce()` functions. To use LRU-Map-Cache in `map()` or `reduce()`, you need to do the following (this can be applied in mapper or reducer class):

- FIRST: Initialize and set up LRU-Map-Cache. This step can be accomplished in the `setup()` method. This will done once.
- SECOND: Use the LRU-Map-Cache in `map()` or `reduce()`. This will be done many times.
- FINALLY: Close the LRU-Map-Cache objects (to release all non-needed resources). This step can be accomplished in the `cleanup()` method. This will done once.

To make these steps easier, we define a service class called `CacheManager`, which can be used as:

**Listing 30.6:** CacheManager Usage

```

1   try {
2       //
3       // initialize cache
4       //
5       CacheManager.init();
6
7       //
8       // use cache
9       //
10      String chrID = ...;
11      String position = ...;
12      List<String> valueAsList = CacheManager.get(chrID, position);
13  }
14  finally {
15      //
16      // close cache
17      //

```

```
18     CacheManager.close();
19 }
```

## 30.7 CacheManager Definition

CacheManager is a service class, which provides 3 basic services: initialization, service, and closing cache. Since this is a service class, all methods are defined as **static**. The BeginEndPosition object implements the partition data structures such that you can get the Db name (a 64MB hash table) for a given key of (chrID, position).

**Listing 30.7:** CacheManager Definition

```
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>> theCustomLRUMap = null;
7     private static BeginEndPosition beginend = null;
8     private static String mapdbRootDirName = "/data/mapdb/pf";
9     private static String mapdbBeginEndDirName = "/data/mapdb/pf/begin_end_position";
10    private static boolean initialized = false;
11
12    public static void setLRUMapSize(int size) {
13        theLRUMapSize = size;
14    }
15
16    public static int getLRUMapSize() {
17        return theLRUMapSize;
18    }
19
20    // initialize the LRUMap
21    public static void init() throws Exception {...}
22
23    // initialize the LRUMap
24    public static void init(int size) throws Exception {...}
25
26    // close the cache database in the LRUMap
27    public static void close() throws Exception {...}
28
29    // get value from the cache database for a given (chrID, position).
30    public static String get(int chrID, int position) throws Exception {...}
31
32    // get value from the cache database for a given (chrID, position).
33    public static String get(String chrID, String position) throws Exception {...}
34
35    // close the cache database in the LRU-Map-Cache
36    public static void close() throws Exception {...}
```

```
37  
38 }
```

### 30.7.1 CacheManager Initialization

**Listing 30.8:** CacheManager Initialization

```
1  public static void setLRUMapSize(int size) {  
2      theLRUMapSize = size;  
3  }  
4  
5  public static int getLRUMapSize() {  
6      return theLRUMapSize;  
7  }  
8  
9  // initialize the LRUMap  
10 public static void init() throws Exception {  
11     if (initialized) {  
12         return;  
13     }  
14     theCustomLRUMap = new CustomLRUMap<String, MapDBEntry<String, String>>(theLRUMapSize);  
15     beginend = new BeginEndPosition(mapdbBeginEndDirName);  
16     beginend.build(mapdbRootDirName);  
17     initialized = true;  
18 }  
19  
20 // initialize the LRUMap  
21 public static void init(int size) throws Exception {  
22     if (initialized) {  
23         return;  
24     }  
25     setLRUMapSize(size);  
26     init();  
27 }
```

### 30.7.2 CacheManager Closing

**Listing 30.9:** CacheManager Closing

```
1  // close the cache database in the LRUMap  
2  public static void close() throws Exception {  
3      if (theCustomLRUMap != null) {  
4          for (Map.Entry<String, MapDBEntry<String, String>> entry : theCustomLRUMap.entrySet()) {  
5              entry.getValue().close();  
6          }  
7      }  
8  }
```

```
7     }
8 }
```

---

### 30.7.3 CacheManager Usage

**Listing 30.10:** CacheManager Usage

```
1 /**
2  * Get value from the LRU-Map-Cache for a given (chrID, position).
3  * @param chrID chrID
4  * @param position position
5  */
6 public static String get(int chrID, int position) throws Exception {
7     return get(String.valueOf(chrID), String.valueOf(position));
8 }
9
10 /**
11  * Get value from the cache database (value is the snpID) for a given (chrID, position).
12 * @param chrID chrID
13 * @param position position
14 */
15 public static String get(String chrID, String position) throws Exception {
16     String dbName = getdbName(chrID, position);
17     if (dbName == null) {
18         return null;
19     }
20     // now return the cache vale
21     MapDBEntry<String, String> entry = theCustomLRUMap.get(dbName);
22     if (entry == null) {
23         entry = MapDBEntryFactory.create(dbName);
24         theCustomLRUMap.put(dbName, entry);
25     }
26     return entry.getValue(position);
27 }
28
29 private static String getdbName(String chrID, String position){
30     // query parameters are: chrID and position
31     List<Interval> results = beginend.query(chrID, position);
32     if ((results == null) || (results.isEmpty()) || (results.size() == 0)) {
33         return null;
34     }
35     else {
36         return results.get(0).db();
37     }
38 }
39 }
```

---

# Chapter 31

## Bloom Filter

### 31.1 Introduction

The simple goal of this chapter is introduce the concept of "Bloom filter" and use it in a "reduce-side join" which engages a Bloom filter on the map phase of a MapReduce job. What is a Bloom filter? How it can be used in a MapReduce environment? How can it speed up a join operation between two big relations/tables? According to Wikipedia: "A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not; i.e. a query returns either **possibly in set** or **definitely not in set**. Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter). The more elements that are added to the set, the larger the probability of false positives" (source: [http://en.wikipedia.org/wiki/Bloom\\_filter](http://en.wikipedia.org/wiki/Bloom_filter)). The Bloom filter data structure may return true for elements that are not actually members of the set (this is called false-positives errors), but it will never return false for elements that are in the set; for each element in the set, the Bloom filter must return true. There is a very nice tutorial on the Bloom filter by [Bill Mill](#). This is another good introduction to Bloom filter data structure by [Jacob Honorooff](#).

Therefore, in a nutshell, we can summarize Bloom filter properties as:

- Given a big set  $S = \{x_1, x_2, \dots, x_n\}$ , Bloom filter is a probabilistic,

fast, and space efficient cache builder for a big data set; this is basically approximating set membership operation:

Is  $x \in S$ ?

- It tries to answer lookup questions: `does item "x" exist in a set S?`
- It allows **false positive errors**, as they only cost us an extra data set access. This means that for some  $x$ , which is not in the set, Bloom filter might indicate that  $x$  is in the set.
- It does not allow **false negative errors**, because they result in wrong answers. This means that if  $x$  is not in the set, then for sure it will indicate that  $x$  is not in the set. Thus, the Bloom filter does not allow false negatives, but can allow false positives.

Let's focus on a simple join example between two relations/tables: let's say that we want to join  $R(a, b)$  and  $S(a, c)$  on a common field  $a$ . Further assume that

$$\begin{aligned}size(R) &= 1000,000,000 (\text{larger data set}) \\size(S) &= 10,000,000 (\text{smaller data set})\end{aligned}$$

To do basic join, we need to check  $10,000,000,000,000,000$  records, which is very huge and time consuming process. One idea to reduce time and complexity of join operation between  $R$  and  $S$  is to use a Bloom filter on relation  $S$  (smaller data set) and then use the built Bloom filter data structure on relation  $R$ . This can eliminate the non-needed records from  $R$  (it might reduce it to 20,000,000) and hence the join becomes fast and efficient.

Next we semi-formalize Bloom filter data structure: what is involved in Bloom filter probabilistic data structures? How do we construct one? what is the probability of **false positive errors**, and how we can decrease the probability of **false positive errors**. This is how it works:

- Given a big set  $S = \{x_1, x_2, \dots, x_n\}$

- Let B be an m ( $m > 1$ ) bit array, initialized with 0s. B's elements are  $B[0], B[1], B[2], \dots, B[m-1]$ . The memory required for array B is only a fraction of the one needed for storing the whole set S. By selecting the bigger bit vector (array B), the probability of false positive rate will decrease.
- Let  $\{H_1, H_2, \dots, H_k\}$  be a set of  $k$  hash functions, If  $H_i(x_j) = a$  then set  $B[a] = 1$ . You may use SHA1, MD5, and Murmer as hash functions. For example, you may use the following as hash functions:

$$\begin{aligned} - H_i(x) &= MD5(x + i) \\ - H_i(x) &= MD5(x|i) \end{aligned}$$

- To check if  $x \in S$ , check B at  $H_i(x)$ . All  $k$  values must be 1.
- It is possible to have a false positive; all  $k$  values are 1, but  $x$  is not in S.
- What is the probability of false positives? the probability of false positives is (for details, see [http://en.wikipedia.org/wiki/Bloom\\_filter](http://en.wikipedia.org/wiki/Bloom_filter)):

$$\left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k$$

- What are the optimal hash functions? What is the optimal number of hash functions? For a given  $m$  (number of bits selected for Bloom filter) and  $n$  (size of big data set), the value of  $k$  (the number of hash functions) that minimizes the probability of false positives is ( $\ln$  stands for "natural logarithm"<sup>1</sup>)

$$\begin{aligned} k &= \frac{m}{n} \ln(2) \\ m &= -\frac{n \ln(p)}{(\ln(2))^2} \end{aligned}$$

---

<sup>1</sup>Natural logarithm is the logarithm to the base e of a number, where e is the Euler's number: 2.71828183. Let  $x = e^y$ , then  $\ln(x) = \log_e(x) = y$ .

- Therefore, the probability that a specific bit has been flipped to 1 is:

$$1 - (1 - \frac{1}{m})^{kn} \approx 1 - e^{-\frac{kn}{m}}$$

- Following Wikipedia: "unlike a standard hash table, a Bloom filter of a fixed size can represent a set with an arbitrary large number of elements; adding an element never fails due to the data structure "filling up." However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result."

## 31.2 A Simple Bloom Filter Example

In this example, we show how to insert and query on a bloom filter of size 10 ( $m = 10$ ) with three hash functions  $H = \{H_1, H_2, H_3\}$  and let  $H(x)$  denote the result of the three hash functions ( $H(x) = \{H_1(x), H_2(x), H_3(x)\}$ ). We start with a 10-bit long array B initialized to 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

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)

```

### 31.3 Bloom Filter in Guava Library

The Guava<sup>2</sup> library provides an implementation of a Bloom filter. This is how it works: you define your basic type (`Person`, which will be used in the Bloom filter (this example is from the <http://code.google.com/p/guava-libraries/wiki/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;
    }
}

```

Then you define a `Funnel` of the basic type, which you want to use Bloom filter: a `Funnel` describes how to decompose a particular object type into primitive field values. our `Funnel` might look like

```
Funnel<Person> personFunnel = new Funnel<Person>() {
```

---

<sup>2</sup><http://code.google.com/p/guava-libraries/>

```

@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);
}
};

```

Once `Funnel<T>` is defined, then we can define and use a Bloom filter:

```

List<Person> friendsList = {Person1, Person2, ...};
BloomFilter<Person> friends = BloomFilter.create(personFunnel, 500, 0.01);
for(Person friend : friendsList) {
    friends.put(friend);
}

// much later, use the Bloom filter
Person dude = new Person(100, "Alex", "Smith", 1967);
if (friends.mightContain(dude)) {
    // the probability that dude reached this place
    // if he isn't a friend is 1% (0.01), we might,
    // for example, start asynchronously loading things
    // for dude while we do a more expensive exact check
}

```

The Guava library provide the following hash functions for building a Bloom filter data structure:

- `md5()`
- `murmur3_128()`
- `murmur3_32()`
- `sha1()`

- sha256()
- sha512()
- goodFastHash(int bits)

## 31.4 Using Bloom Filter in MapReduce

Bloom filter is a small, compact, and fast data structure for set membership. It can be used in the join of two relations/tables such as R(a, b) and S(a, c) where one of the relations has huge number of records (for example, R to have 1000,000,000 records) and the other relation has small number of records (for example, S to have 10,000,000 records). To do a join on field "a" between R and S, it will take a long time and it is inefficient. We can use Bloom filter data structure in the following way: build a Bloom filter out of relation S(a, c), and then want to test values R(a, b) for membership using the built Bloom filter. Note that, for reduce-side join optimization, we use of a Bloom filter on the map tasks, which will force an I/O cost reduction for the MapReduce job. How do we use the Bloom filter concept? The following is one way to have an optimized reduce-side join with the use of a Bloom filter: we implement it in two steps:

- STEP-1: construction of the bloom filter; this a Mapreduce job which uses the smaller of the two relations/tables for constructing the bloom filter data structure. For this step only mappers are needed to construct the Bloom filter data structure (more than one Bloom filters will be built — one per mapper).
- STEP-2. use of the the Bloom filter data structure (built in STEP-1) in the reduce-side join

Listing 31.1: Join R and S

```
joinResult = {};
for all tuples r in R {
    for all tuples s in S {
        if (joinConditionSatisfied(r, s)) {
            // join condition can be like r.a == s.b
        }
    }
}
```

```
        add (r;s) to the joinResult;
    }
}
}
```

---

# Appendices

# Appendix A

## Bioset

### A.1 Introduction

Biosets (also called "gene signatures"<sup>1</sup>) and "assays"<sup>2</sup>) encompass data in the form of experimental sample comparisons (for transcriptomic, epigenetic, and copy number variation data), as well as genotype signatures (for GWAS and mutational data).

A bioset has an associated data type (a data type can be "gene-expression", "protein-expression", "methylation", "copy-number-variation", "miRNA", or "somatic mutation"). Also, each bioset entry/record has an associated reference type (a reference type can be **r1=normal**, **r2=disease**, **r3=paired**, or **r4=unknown**). Note that a reference type does not apply to "somatic mutation" data type.

The number of entries/records per bioset does depend on the data type of a bioset (see the following table):

---

<sup>1</sup>A gene signature is the group of genes in a type of cell whose combined expression pattern is uniquely characteristic of a medical or other condition. Ideally, the gene signature can be used to select a group of patients at a specific state of a disease with accuracy that facilitates selection of treatments. (source: [http://en.wikipedia.org/wiki/Gene\\_signature](http://en.wikipedia.org/wiki/Gene_signature))

<sup>2</sup>An assay is an investigative (analytic) procedure in laboratory medicine, pharmacology, environmental biology, and molecular biology for qualitatively assessing or quantitatively measuring the presence or amount or the functional activity of a target entity (the analyte), which can be a drug or biochemical substance or a cell in an organism or organic sample. (source: <http://en.wikipedia.org/wiki/Assay>)

<i>Bioset Data Type</i>	<i>Number of Entries/Records</i>
Somatic Mutation	3,000
Methylation	30,000
Gene Expression	50,000
Copy Number Variation	40,000
Germline	4,300,000
Protein Expression	30,000
miRNA	30,000

Appendix **B**

## Spark RDDs

### B.1 Introduction

Sparks main abstraction is a distributed collection of items (such as collection of log records, FASTQ sequences, employee records) called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop InputFormats (such as HDFS files) or by transforming other RDDs. The purpose of this appendix chapter is to introduce Spark RDDs by simple Java examples. The purpose is not to dive into architectural[33] details of RDDs, but merely how to use and utilize them in MapReduce programs. Usage of RDDs are presented by simple data examples. As a user, you can consider an RDD as a handle for a collection of items of type T, which are the result of some computation (type T can be any standard Java data types – such as String, Integer, Map, List, ... – or any custom objects such as Employee, Mutation, ...). RDD has actions (such as `reduce()`, `collect()`, `count()`, `saveAsTextFile()`, ...) and transformations (such as `map()`, `filter()`, `union()`, `groupByKey()`, ...) can be used for more complex computations. All the examples provided here are based on Spark 1.1.0. Also in Spark, there is a heavy use of tuples (such as `scala.Tuple2` (tuple of 2 elements) and `scala.Tuple3` (tuple of 3 elements)).

## B.2 What is a TupleN?

`scala.Tuple<N>` represents a composite data type of  $N$  elements. Spark API use `scala.Tuple2` and `scala.Tuple3` for composite data types and are described below:

- `scala.Tuple2`: represents a tuple of 2 elements; some examples are provided below:

**Listing B.1:** Example of Tuple2

```
1 import scala.Tuple2;
2 import java.util.List;
3 import java.util.Arrays;
4 ...
5 Tuple2<String, Integer> t21 = new Tuple2<String, Integer>("abc", 234);
6 String str21 = t21._1; // str21 holds "abc"
7 Integer int21 = t21._2; // int21 holds 234
8
9 List<Integer> listofint = Arrays.asList(1, 2, 3);
10 Tuple2<String, List<Integer>> t22 =
11     new Tuple2<String, List<Integer>>("z1z2", listofint);
12 String str22 = t22._1; // str22 holds "z1z2"
13 List<Integer> list22 = t22._2; // int21 points to List(1, 2, 3)
```

- `scala.Tuple3`: represents a tuple of 3 elements; a simple example is provided below:

**Listing B.2:** Example of 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 holds "a1a2"
9 Integer int3 = t3._2; // int3 holds 567
10 List<String> list3 = t3._3; // list3 points to List("a1", "a2", "a3")
```

- Note that `scala.Tuple4` through `scala.Tuple22` are defined as part of Java Spark API.

## B.3 What is an RDD

An RDD stands for a Resilient Distributed Dataset, which is the basic data abstraction in Spark. As `class` and `interface` are the fundamental units of abstractions in Java, `JavaRDD` and `JavaPairRDD` are fundamental units of data abstractions in Spark. It means that to perform `map()`, `reduce()`, or `groupByKey()` operations, you use an RDD as input and output. By definition, an RDD represents an immutable<sup>1</sup>, partitioned collection of elements that can be operated on in parallel. This means that if I have `JavaRDD<String>`, which has 100M elements, then it can be partitioned into 10, 100, or 1000 segments and each segments can be operated independently (this partitioning is important and does depend on your cluster size, number of cores available per server, and the total amount of RAM available in your cluster – there is no silver bullet for finding the proper number of partition; but this can be properly set by some trial and experience). For details of RDD properties, read <http://www.cs.berkeley.edu/~pendell/strataconf/api/core/spark/RDD.html>.

`JavaRDD` and `JavaPairRDD` are members of the `org.apache.spark.api.java` package and defined as:

**Listing B.3:** `JavaRDD` and `JavaPairRDD`

```
1  public class JavaRDD<T> ...
2  // represents elements of type T
3  // Examples of JavaRDD<T> are:
4  //   JavaRDD<String>
5  //   JavaRDD<Integer>
6  //   JavaRDD<Map<String, Long>>
7  //   JavaRDD<Employee>
8  //   ...
9
10 public class JavaPairRDD<K,V> ...
11 // represents pairs of (key,value) of type K and V
12 // Examples of JavaPairRDD<K,V> are:
13 //   JavaPairRDD<String, Integer>
14 //   JavaPairRDD<String, Tuple2<Integer, Integer>>
15 //   JavaPairRDD<Integer, Map<String, Long>>
16 //   ...
```

---

<sup>1</sup>RDDs are read-only and can not be modified or updated.

## B.4 How to Create RDDs

For most of the MapReduce applications, RDDs are created by `JavaContextObject`, `JavaRDD` and `JavaPairRDD`. Initial/first RDDs are created by `JavaContextObject` and subsequent RDDs are created by many other classes including `JavaRDD` and `JavaPairRDD`. For most of the applications, this is the order to create and manipulate RDDs:

- First, create a `JavaContextObject`: this can be accomplished in many ways: you may create this object by using "Spark master URL" or by using "YARN's resource manager host name". You may use `SparkUtil`<sup>2</sup> class to create `JavaContextObject`. Also, you may create `JavaContextObject` by the following code:

---

```
1 import org.apache.spark.SparkConf;
2 import org.apache.spark.api.java.JavaSparkContext;
3 ...
4 // create JavaSparkContext
5 SparkConf sparkConf = new SparkConf().setAppName("myapp");
6 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
```

---

- Second, once you have an instance of `JavaContextObject`, then you may create RDDs. RDDs can be created from many different sources: from Java collections objects, from text files, from HDFS files (text or binary).
- Third, once you have an instance of `JavaRDD` or `JavaPairRDD`, then creation of new RDDs are accomplished easily by using `map()`, `reduceByKey()`, `filter()`, `groupByKey()`, ... methods. You may create RDDs: by using Java collection objects or by reading files (text files or text/binary HDFS files).

## B.5 Create RDDs by Collection Objects

You may create RDDs by Java collection objects. Te following simple example, create an RDD (called `rdd1`) from a `java.util.List` object:

---

<sup>2</sup><https://github.com/mahmoudparsian/data-algorithms-book/blob/master/src/main/java/org/dataalgorithms/util/SparkUtil.java>

### Listing B.4: JavaRDD Creation

```
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 // create JavaSparkContext
7 SparkConf sparkConf = new SparkConf().setAppName("myapp");
8 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
9
10 // create your desired collection object
11 final List<String> list1 = Lists.newArrayList(
12     "url1,2",
13     "url1,3",
14     "url2,7",
15     "url2,6",
16     "url3,4",
17     "url3,5",
18     "url3,6"
19 );
20
21 // create an RDD from Java collection object
22 JavaRDD<String> rdd1 = ctx.parallelize(list1);
```

Now `rdd1` is a `JavaRDD<String>` with 7 elements (each element has the format of `<url><,><count>`) in it. Once RDD is created, you may start applying functions and transformations (such as `map()`, `collect()`, `reduceByKey()`, and many others).

## B.6 Collect Elements of an RDD

The generated `rdd1` can be collected as `List<String>`:

---

```
1 // create an RDD from Java collection object
2 JavaRDD<String> rdd1 = ctx.parallelize(list1);
3
4 // collect RDD's elements:
5 // java.util.List<T> collect()
6 // Return an array that contains all of the elements in this RDD.
7 java.util.List<String> collected = rdd1.collect();
```

---

## B.7 Transform RDD into New RDD

Let's take `rdd1` and transform it into (K,V) pairs (as `JavaPairRDD` object): for transformation, we use `JavaRDD.mapToPair()` method.

**Listing B.5:** JavaPairRDD Creation

```
1 // create an RDD from Java collection object
2 JavaRDD<String> rdd1 = ctx.parallelize(list1);
3
4 // create (K,V) pairs where K is a URL and V is a count
5 JavaPairRDD<String, Integer> kv1 = rdd1.mapToPair(
6     new PairFunction<String, // T as input
7         String, // K as output
8         Integer // V as output
9     >() {
10     @Override
11     public Tuple2<String, Integer> call(String element) {
12         String[] tokens = element.split(",");
13         // tokens[0] = URL
14         // tokens[1] = count
15         return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
16     }
17 });
```

## B.8 Create RDDs by Reading Files

For most of the MapReduce applications, you will read data from HDFS and then create RDDs, and possibly save the results as in HDFS. Let's assume that we have two HDFS files:

**Listing B.6:** Listing HDFS Files

```
1 $ hadoop fs -ls /myinput/logs/
2 ... /myinput/logs/file1.txt
3 ... /myinput/logs/file2.txt
4
5 $ hadoop fs -cat /myinput/logs/file1.txt
6 url5,2
7 url5,3
8 url6,7
9 url6,8
10 $ hadoop fs -cat /myinput/logs/file2.txt
11 url7,1
12 url7,2
13 url8,5
14 url8,6
```

```
15 url9,7  
16 url9,8  
17 url5,9  
18 $
```

---

Next, we create an RDD from these two files

```
1 // create JavaSparkContext  
2 SparkConf sparkConf = new SparkConf().setAppName("myapp");  
3 JavaSparkContext ctx = new JavaSparkContext(sparkConf);  
4  
5 // read input file from HDFS  
6 // input record format: <string-key><,><integer-value>  
7 String hdfsPath = "/myinput/logs";  
8 // public JavaRDD<String> textFile(String path)  
9 // Read a text file from HDFS, a local file system  
10 // (available on all nodes), or any Hadoop-supported  
11 // file system URI, and return it as an RDD of Strings.  
12 JavaRDD<String> rdd2 = ctx.textFile(hdfsPath);
```

---

At this point, `rdd2` has 11 `String` elements (4 from `file1.txt` and 7 from `file2.txt`). Now, you may use the same technique (presented in the preceding sections) to create `JavaPairRDD<String, Integer>` where key is a URL and value is the URL's associated count.

## B.9 Grouping By Key

Here we will group elements by key for `rdd2`:

**Listing B.7:** JavaPairRDD Creation

```
1 // rdd2 is already created (see previous section)  
2 // create (K,V) pairs where K is a URL and V is a count  
3 JavaPairRDD<String, Integer> kv2 = rdd2.mapToPair(  
4     new PairFunction<String, // T as input  
5             String, // K as output  
6             Integer // V as output  
7     >() {  
8         @Override  
9         public Tuple2<String, Integer> call(String element) {  
10             String[] tokens = element.split(",");  
11             // tokens[0] = URL  
12             // tokens[1] = count  
13             return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
```

```

14     }
15 });
16
17 // next group RDD's elements by key
18 JavaPairRDD<String, Iterable<Integer>> grouped2 = kv2.groupByKey();
```

---

The resulting RDD (`group2`) will have the following content:

(K,V) Generated by groupByKey() Method	
Key	Value
url5	[2, 3, 9]
url6	[7, 8]
url7	[1, 2]
url8	[5, 6]
url9	[7, 8]

## B.10 Map Values

If you have a `JavaPairRDD`, which you want to alter values and create new set of values (without changing the keys), then you may use `JavaPairRDD.mapValues()` method. For example, we will take `grouped2` created in the preceding section and return the maximum of URL counts.

**Listing B.8:** `JavaPairRDD.mapValues()`

```

1 // public <U> JavaPairRDD<K,U> mapValues(Function<V,U> f)
2 // Pass each value in the key-value pair RDD through a map
3 // function without changing the keys; this also retains
4 // the original RDD's partitioning.
5 JavaPairRDD<String, Integer> mapped2 = grouped2.mapValues(
6     new Function<Iterable<Integer>, // input
7         Integer // output
8     >() {
9     public Integer call(Iterable<Integer> list) {
10         Integer max = null;
11         for (Integer element : list) {
12             if (max == null) {
13                 max = element;
14             }
15             else {
16                 if (element > max) {
17                     max = element;
18                 }
19             }
20         }
21     }
22 }
```

```

20         }
21     return max;
22   }
23 });

```

---

The resulting RDD (`mapped2`) will have the following content:

(K,V) Generated by mapValues() Method	
Key	Value
url5	9
url6	8
url7	2
url8	6
url9	8

## B.11 Reducing By Key

Here we will reduce elements by key for `rdd2`: the goal is to add the counts for the same URL (create a final count per URL). We use `JavaPairRDD.reduceByKey()` method.

**Listing B.9:** `JavaPairRDD.mapToPair()`

```

1 // rdd2 is already created (see previous section)
2 // create (K,V) pairs where K is a URL and V is a count
3 JavaPairRDD<String, Integer> kv2 = rdd2.mapToPair(
4     new PairFunction<String,           // T as input
5         String,                      // K as output
6         Integer // V as output
7     >() {
8     @Override
9     public Tuple2<String, Integer> call(String element) {
10        String[] tokens = element.split(",");
11        // tokens[0] = URL
12        // tokens[1] = count
13        return new Tuple2<String, Integer>(tokens[0], new Integer(tokens[1]));
14    }
15 });
16
17 // next reduce JavaPairRDD's elements by key
18 // public JavaPairRDD<K,V> reduceByKey(Function2<V,V,V> func)
19 // Merge the values for each key using an associative reduce
20 // function. This will also perform the merging locally on each
21 // mapper before sending results to a reducer, similarly to a
22 // "combiner" in MapReduce. Output will be hash-partitioned with

```

```

23 // the existing partitioner/ parallelism level.
24 JavaPairRDD<String, Integer> counts = kv2.reduceByKey(
25     new Function2<
26         Integer, // input T1
27         Integer, // input T2
28         Integer // output T
29     >() {
30     @Override
31     public Integer call(Integer count1, Integer count2) {
32         return count1 + count2;
33     }
34 });

```

---

The resulting RDD (`counts`) will have the following content:

(K,V) Generated by reduceByKey() Method	
Key	Value
url5	14
url6	15
url7	3
url8	11
url9	15

## B.12 Filtering an RDD

Spark provides a very simple and powerful API for filtering RDDs. To filter RDD elements, we just need to implement a `filter()` function: return `true` for the elements you want to keep and return `false` for the elements you want to toss out. Consider the following RDD labeled as `logRDD`. We write a `filter()` function, which keeps the elements, which has the "normal" keyword and tosses out the remaining elements.

**Listing B.10:** Sample JavaRDD

```

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 // create JavaSparkContext
7 SparkConf sparkConf = new SparkConf().setAppName("logs");
8 JavaSparkContext ctx = new JavaSparkContext(sparkConf);
9

```

```

10 // create your desired collection object
11 final List<String> logRecords = Lists.newArrayList(
12     "record 1: normal",
13     "record 2: error",
14     "record 3: normal",
15     "record 4: error",
16     "record 5: normal"
17 );
18
19 // create an RDD from Java collection object
20 JavaRDD<String> logRDD = ctx.parallelize(logRecords);
21
22 // To filter elements, we need to implement a filter function:
23 // JavaRDD<T> filter(Function<T,Boolean> f)
24 // Return a new RDD containing only the elements that satisfy a predicate.
25 JavaRDD<String> filteredRDD = logRDD.filter(new Function<String,Boolean>() {
26     public Boolean call(String record) {
27         if (record.contains("normal")) {
28             return true; // do return these records
29         }
30         else {
31             return false; // do not retrun these records
32         }
33     }
34 });
35
36 // At this point, filteredRDD will have the following 3 elements:
37 //     "record 1: normal"
38 //     "record 3: normal"
39 //     "record 5: normal"

```

---

## B.13 Saving RDD as HDFS Text File

You may save your RDDs as Hadoop files (text or sequence file). Let `counts` be an RDD created in the preceding section, then you may save it in HDFS as a text file(s):

---

```

1 // save RDD as an HDFS text file
2 // void saveAsTextFile(String path)
3 // Save this RDD as a text file, using string representations of elements.
4 // Make sure that your output path -- "/my/output" -- does not exist
5 counts.saveAsTextFile("/my/output");

```

---

After it is saved, you may view the saved data as:

```
$ hadoop fs -cat /my/output/part*
```

## B.14 Saving RDD as HDFS Sequence File

You may save your RDDs as Hadoop files (text or sequence file). Let `counts` be an RDD created in the preceding section, then you may save it in HDFS as a sequence file (sequence files are binary files in the form of (K, V) pairs). To save an RDD as a sequence file, you have to convert its elements into Hadoop's `Writable` objects. Since `counts` is an RDD of type `JavaPairRDD<String, Integer>`, we can not directly write it to HDFS as a sequence file; first we need create a new RDD which has a type of `JavaPairRDD<Text, IntWritable>`, where we convert `String` into `Text` and `Integer` into `IntWritable`. This is how we write `counts` into HDFS as a sequence file:

**Listing B.11:** Creating a SequenceFile

```
1 import scala.Tuple2;
2 import org.apache.hadoop.io.Text;
3 import org.apache.hadoop.io.IntWritable;
4 import org.apache.hadoop.mapred.SequenceFileOutputFormat;
5 ...
6
7 // First create a Writable RDD:
8 JavaPairRDD<Text, IntWritable> countsWritable =
9     counts.mapToPair(new PairFunction<
10                     Tuple2<String, Integer>, // T
11                     Text,                  // K
12                     IntWritable           // V
13                 >() {
14             @Override
15             public Tuple2<Text, IntWritable> call(Tuple2<String, Integer> t) {
16                 return new Tuple2<Text, IntWritable>(
17                     new Text(t._1),
18                     new IntWritable(t._2)
19                 );
20         }
21     });
22
23 // Next, write it as a sequence file
24 // make sure that your output path -- "/my/output2" -- does not exist
25 countsWritable.saveAsHadoopFile(
26     "/my/output2",           // name of path
27     Text.class,              // key class
28     IntWritable.class,       // value class
29     SequenceFileOutputFormat.class // output format class
30 );
```

After it is saved, you may view the saved data as

```
$ hadoop fs -text /my/output2/part*
```

## B.15 Reading RDD from HDFS Sequence File

You may create a new RDDs from reading Hadoop's sequence file. The sequence file we created in the preceding section will be read and a new RDD will be created:

**Listing B.12:** Reading a SequenceFile

```
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 // Get an RDD for a Hadoop file with an arbitrary InputFormat
12 // Note: Because Hadoop's RecordReader class re-uses the
13 // same Writable object for each record, directly caching the
14 // returned RDD will create many references to the same object.
15 // If you plan to directly cache Hadoop writable objects, you
16 // should first copy them using a map function.
17 JavaPairRDD<Text, IntWritable> seqRDD = ctx.hadoopFile(
18     "/my/output2", // HDFS path
19     SequenceFileInputFormat.class, // input format class
20     Text.class, // key class
21     IntWritable.class // value class
22 );
```

## B.16 Counting RDD

The `count()` method returns the number of items stored in an RDD.

**Listing B.13:** Counting RDDs

```
1 // count JavaPairRDD
2 JavaPairRDD<Text, IntWritable> pairRDD = ...;
3 int count1 = pairRDD.count();
4
5 // count JavaRDD
6 JavaRDD<String> strRDD = ...;
7 int count2 = strRDD.count();
```

## B.17 Spark RDD Examples in Scala

If you are interested in usage of RDD in Scala, then you may refer to [http://  
homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html](http://homepage.cs.latrobe.edu.au/zhe/ZhenHeSparkRDDAPIExamples.html).

glossary

# Bibliography

- [1] Satnam Alag. *Collective Intelligence in Action*. Manning Publications Co., 2009.
- [2] Carolyn J. Anderson. Exact tests for 2way tables. 2013.
- [3] Sarah Boslaugh. *Statistics in a Nutshell, Second Edition*. O'Reilly Media, Inc., 2013.
- [4] A. Brandt. Algebraic analysis of mapreduce samples. 2010.
- [5] Eirinaios Michelakis Ioannis Koutis Christos Faloutsos Charalampos E. Tsurakakis, Petros Drineas. Spectral counting of triangles in power-law networks via element-wise sparsification.
- [6] Edwin Chen. <http://blog.echen.me/2012/02/09/movie-recommendations-and-more-via-mapreduce-and-scalding/>. Edwin Chen, 2012.
- [7] Anderson de Rezende Rocha. Naive bayes classifier. <http://www.ic.unicamp.br/~rocha/teaching/2011s2/mc906/aulas/naive-bayes-classifier.pdf>.
- [8] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. Proc of the 6th Symposium on Operating Systems Design and Implementation, Reading, MA, 2004.
- [9] Alexander Felfernig Dietmar Jannach, Markus Zanker and Gerhard Friedrich. *Recommender Systems, An Introduction*. Cambridge University Press, 2011.

- [10] G. David Garson. *Cox Regression*. Asheboro, NC: Statistical Associates Publishers, 2012.
- [11] Peter Harrington. *Machine Learning in Action*. Manning Publications Co., 2012.
- [12] Jimmy Lin. Mapreduce algorithm design. 2013.
- [13] Jimmy Lin. Monoidify! monoids as a design principle for efficient mapreduce algorithms. 29 April 2013.
- [14] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. University of Maryland, College Park, MD, 2010.
- [15] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan & Claypool Publishers, January 27, 2013.
- [16] Donald Miner and Adam Shook. *MapReduce Design Patterns*. O'Reilly Media, Inc., 2013.
- [17] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
- [18] Netflix. <http://www.netflixprize.com/>. Netflix, 2009.
- [19] Bo Pang and Lillian Lee. A sentimental education: Sentiment analysis using subjectivity summarization based on minimum cuts.
- [20] John Perry. *Foundations of Nonlinear Algebra*. John Perry: <http://www.math.usm.edu/perry/mat423fa13/main.pdf>, 2012.
- [21] David Saile. Mapreduce with deltas. August 2011.
- [22] Dedy Hartama Ramliana S Elvi Wani Sajadin Sembiring, M. Zarlis. Prediction of student academic performance by an application of data mining techniques. *2011 International Conference on Management and Artificial Intelligence IPEDR vol.6*, 2011.
- [23] Manish Sarkar 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, Lower Kent Ridge Road, Singapore: 119260*, 2000.

- [24] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. [http://i11www.iti.uni-karlsruhe.de/extra/publications/sw-fclt-05\\_t.pdf](http://i11www.iti.uni-karlsruhe.de/extra/publications/sw-fclt-05_t.pdf).
- [25] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. 2005.
- [26] Toby Segaran. *Programming Collective Intelligence: Building Smart Web 2.0 Applications*. O'Reilly Media, Inc., 2007.
- [27] Michael G. Walker. Survival analysis. <http://walkerbioscience.com/pdfs/Survival%20analysis.pdf>.
- [28] Tom White. *Hadoop: The Definitive Guide, Third Edition*. O'Reilly Media, Inc., 2012.
- [29] wikipedia. [http://en.wikipedia.org/wiki/Correlation\\_and\\_dependence](http://en.wikipedia.org/wiki/Correlation_and_dependence). wikipedia.org, 2012.
- [30] Jongwook Woo and Yuhang Xu. *Market Basket Analysis Algorithm with Map/Reduce of Cloud Computing*. Computer Information Systems Department California State University, Los Angeles, CA.
- [31] Therese Sorlie Bjorn Naume Anita Langerod Arnoldo Frigessi Vessela N. Kristensen Anne-Lise Borresen-Dale Ole Christian Lingjaerde Xi Zhao, Einar Andreas Rodland. Combining gene signatures improves prediction of breast cancer survival. <http://www.plosone.org/article/info:doi/10.1371/journal.pone.0017845>.
- [32] Changyu Yang. Triangle counting in large networks, master's thesis. 2012.
- [33] Matei Zaharia. An architecture for fast and general data processing on large clusters. 2014.