

O'REILLY®



Free Sampler

Data Algorithms

RECIPES FOR SCALING UP WITH HADOOP AND SPARK

Mahmoud Parsian

Data Algorithms

If you are ready to dive into the MapReduce framework for processing large datasets, this practical book takes you step by step through the algorithms and tools you need to build distributed MapReduce applications with Apache Hadoop or Apache Spark. Each chapter provides a recipe for solving a massive computational problem, such as building a recommendation system. You'll learn how to implement the appropriate MapReduce solution with code that you can use in your projects.

Dr. Mahmoud Parsian covers basic design patterns, optimization techniques, and data mining and machine learning solutions for problems in bioinformatics, genomics, statistics, and social network analysis. This book also includes an overview of MapReduce, Hadoop, and Spark.

Topics include:

- Market basket analysis for a large set of transactions
- Data mining algorithms (K-means, KNN, and Naive Bayes)
- Using huge genomic data to sequence DNA and RNA
- Naive Bayes theorem and Markov chains for data and market prediction
- Recommendation algorithms and pairwise document similarity
- Linear regression, Cox regression, and Pearson correlation
- Allelic frequency and mining DNA
- Social network analysis (recommendation systems, counting triangles, sentiment analysis)

Mahmoud Parsian, PhD in Computer Science, is a practicing software professional with 30 years of experience as a developer, designer, architect, and author. Currently the leader of Illumina's Big Data team, he's spent the past 15 years working with Java (server-side), databases, MapReduce, and distributed computing. Mahmoud is the author of *JDBC Recipes* and *JDBC Metadata, MySQL, and Oracle Recipes* (both Apress).

DATA/MATH

US \$69.99

CAN \$80.99

ISBN: 978-1-491-90618-7



5 6 9 9 9



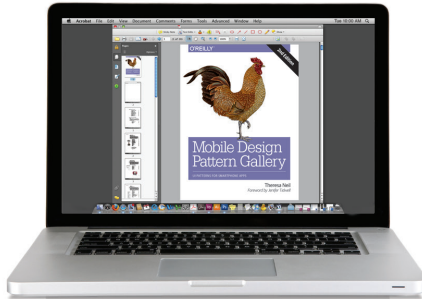
Twitter: @oreillymedia
facebook.com/oreilly

O'Reilly ebooks.

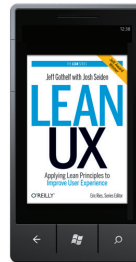
Your bookshelf on your devices.



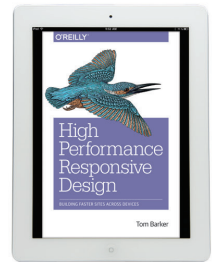
PDF



Mobi



ePub



DAISY

When you buy an ebook through oreilly.com you get lifetime access to the book, and whenever possible we provide it to you in four DRM-free file formats—PDF, .epub, Kindle-compatible .mobi, and DAISY—that you can use on the devices of your choice. Our ebook files are fully searchable, and you can cut-and-paste and print them. We also alert you when we've updated the files with corrections and additions.

Learn more at ebooks.oreilly.com

You can also purchase O'Reilly ebooks through the iBookstore, the [Android Marketplace](http://AndroidMarketplace), and Amazon.com.

O'REILLY®

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055

Data Algorithms

by Mahmoud Parsian

Copyright © 2015 Mahmoud Parsian. All rights reserved.

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editors: Ann Spencer and Marie Beaugureau

Production Editor: Matthew Hacker

Copyeditor: Rachel Monaghan

Proofreader: Rachel Head

Indexer: Judith McConville

Interior Designer: David Futato

Cover Designer: Ellie Volckhausen

Illustrator: Rebecca Demarest

July 2015: First Edition

Revision History for the First Edition

2015-07-10: First Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491906187> for release details.

While the publisher and the author have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the author disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-491-90618-7

[LSI]

Table of Contents

Foreword	xix
Preface	xxi
1. Secondary Sort: Introduction	1
Solutions to the Secondary Sort Problem	3
Implementation Details	3
Data Flow Using Plug-in Classes	6
MapReduce/Hadoop Solution to Secondary Sort	7
Input	7
Expected Output	7
map() Function	8
reduce() Function	8
Hadoop Implementation Classes	9
Sample Run of Hadoop Implementation	10
How to Sort in Ascending or Descending Order	12
Spark Solution to Secondary Sort	12
Time Series as Input	12
Expected Output	13
Option 1: Secondary Sorting in Memory	13
Spark Sample Run	20
Option #2: Secondary Sorting Using the Spark Framework	24
Further Reading on Secondary Sorting	25
2. Secondary Sort: A Detailed Example	27
Secondary Sorting Technique	28
Complete Example of Secondary Sorting	32
Input Format	32

Output Format	33
Composite Key	33
Sample Run—Old Hadoop API	36
Input	36
Running the MapReduce Job	37
Output	37
Sample Run—New Hadoop API	37
Input	38
Running the MapReduce Job	38
Output	39
3. Top 10 List.....	41
Top N, Formalized	42
MapReduce/Hadoop Implementation: Unique Keys	43
Implementation Classes in MapReduce/Hadoop	47
Top 10 Sample Run	47
Finding the Top 5	49
Finding the Bottom 10	49
Spark Implementation: Unique Keys	50
RDD Refresher	50
Spark's Function Classes	51
Review of the Top N Pattern for Spark	52
Complete Spark Top 10 Solution	53
Sample Run: Finding the Top 10	58
Parameterizing Top N	59
Finding the Bottom N	61
Spark Implementation: Nonunique Keys	62
Complete Spark Top 10 Solution	64
Sample Run	72
Spark Top 10 Solution Using takeOrdered()	73
Complete Spark Implementation	74
Finding the Bottom N	79
Alternative to Using takeOrdered()	80
MapReduce/Hadoop Top 10 Solution: Nonunique Keys	81
Sample Run	82
4. Left Outer Join.....	85
Left Outer Join Example	85
Example Queries	87
Implementation of Left Outer Join in MapReduce	88
MapReduce Phase 1: Finding Product Locations	88
MapReduce Phase 2: Counting Unique Locations	92

Implementation Classes in Hadoop	93
Sample Run	93
Spark Implementation of Left Outer Join	95
Spark Program	97
Running the Spark Solution	104
Running Spark on YARN	106
Spark Implementation with leftOuterJoin()	107
Spark Program	109
Sample Run on YARN	116
5. Order Inversion.....	119
Example of the Order Inversion Pattern	120
MapReduce/Hadoop Implementation of the Order Inversion Pattern	122
Custom Partitioner	123
Relative Frequency Mapper	124
Relative Frequency Reducer	126
Implementation Classes in Hadoop	127
Sample Run	127
Input	127
Running the MapReduce Job	127
Generated Output	128
6. Moving Average.....	131
Example 1: Time Series Data (Stock Prices)	131
Example 2: Time Series Data (URL Visits)	132
Formal Definition	133
POJO Moving Average Solutions	134
Solution 1: Using a Queue	134
Solution 2: Using an Array	135
Testing the Moving Average	136
Sample Run	136
MapReduce/Hadoop Moving Average Solution	137
Input	137
Output	137
Option #1: Sorting in Memory	138
Sample Run	141
Option #2: Sorting Using the MapReduce Framework	143
Sample Run	147
7. Market Basket Analysis.....	151
MBA Goals	151
Application Areas for MBA	153

Market Basket Analysis Using MapReduce	153
Input	154
Expected Output for Tuple2 (Order of 2)	155
Expected Output for Tuple3 (Order of 3)	155
Informal Mapper	155
Formal Mapper	156
Reducer	157
MapReduce/Hadoop Implementation Classes	158
Sample Run	162
Spark Solution	163
MapReduce Algorithm Workflow	165
Input	166
Spark Implementation	166
YARN Script for Spark	178
Creating Item Sets from Transactions	178
8. Common Friends.....	181
Input	182
POJO Common Friends Solution	182
MapReduce Algorithm	183
The MapReduce Algorithm in Action	184
Solution 1: Hadoop Implementation Using Text	187
Sample Run for Solution 1	187
Solution 2: Hadoop Implementation Using ArrayListOfLongsWritable	189
Sample Run for Solution 2	189
Spark Solution	190
Spark Program	191
Sample Run of Spark Program	197
9. Recommendation Engines Using MapReduce.....	201
Customers Who Bought This Item Also Bought	202
Input	202
Expected Output	202
MapReduce Solution	203
Frequently Bought Together	206
Input and Expected Output	207
MapReduce Solution	208
Recommend Connection	211
Input	213
Output	214
MapReduce Solution	214
Spark Implementation	216

Sample Run of Spark Program	222
10. Content-Based Recommendation: Movies.....	227
Input	228
MapReduce Phase 1	229
MapReduce Phases 2 and 3	229
MapReduce Phase 2: Mapper	230
MapReduce Phase 2: Reducer	231
MapReduce Phase 3: Mapper	233
MapReduce Phase 3: Reducer	234
Similarity Measures	236
Movie Recommendation Implementation in Spark	236
High-Level Solution in Spark	237
Sample Run of Spark Program	250
11. Smarter Email Marketing with the Markov Model.....	257
Markov Chains in a Nutshell	258
Markov Model Using MapReduce	261
Generating Time-Ordered Transactions with MapReduce	262
Hadoop Solution 1: Time-Ordered Transactions	263
Hadoop Solution 2: Time-Ordered Transactions	264
Generating State Sequences	268
Generating a Markov State Transition Matrix with MapReduce	271
Using the Markov Model to Predict the Next Smart Email Marketing Date	274
Spark Solution	275
Input Format	275
High-Level Steps	276
Spark Program	277
Script to Run the Spark Program	286
Sample Run	287
12. K-Means Clustering.....	289
What Is K-Means Clustering?	292
Application Areas for Clustering	292
Informal K-Means Clustering Method: Partitioning Approach	293
K-Means Distance Function	294
K-Means Clustering Formalized	295
MapReduce Solution for K-Means Clustering	295
MapReduce Solution: map()	297
MapReduce Solution: combine()	298
MapReduce Solution: reduce()	299
K-Means Implementation by Spark	300

Sample Run of Spark K-Means Implementation	302
--------------------------------------------	-----

13. k-Nearest Neighbors	305
kNN Classification	306
Distance Functions	307
kNN Example	308
An Informal kNN Algorithm	308
Formal kNN Algorithm	309
Java-like Non-MapReduce Solution for kNN	309
kNN Implementation in Spark	311
Formalizing kNN for the Spark Implementation	312
Input Data Set Formats	313
Spark Implementation	313
YARN shell script	325
14. Naive Bayes	327
Training and Learning Examples	328
Numeric Training Data	328
Symbolic Training Data	329
Conditional Probability	331
The Naive Bayes Classifier in Depth	331
Naive Bayes Classifier Example	332
The Naive Bayes Classifier: MapReduce Solution for Symbolic Data	334
Stage 1: Building a Classifier Using Symbolic Training Data	335
Stage 2: Using the Classifier to Classify New Symbolic Data	341
The Naive Bayes Classifier: MapReduce Solution for Numeric Data	343
Naive Bayes Classifier Implementation in Spark	345
Stage 1: Building a Classifier Using Training Data	346
Stage 2: Using the Classifier to Classify New Data	355
Using Spark and Mahout	361
Apache Spark	361
Apache Mahout	362
15. Sentiment Analysis	363
Sentiment Examples	364
Sentiment Scores: Positive or Negative	364
A Simple MapReduce Sentiment Analysis Example	365
map() Function for Sentiment Analysis	366
reduce() Function for Sentiment Analysis	367
Sentiment Analysis in the Real World	367

16. Finding, Counting, and Listing All Triangles in Large Graphs.....	369
Basic Graph Concepts	370
Importance of Counting Triangles	372
MapReduce/Hadoop Solution	372
Step 1: MapReduce in Action	373
Step 2: Identify Triangles	375
Step 3: Remove Duplicate Triangles	376
Hadoop Implementation Classes	377
Sample Run	377
Spark Solution	380
High-Level Steps	380
Sample Run	387
17. K-mer Counting.....	391
Input Data for K-mer Counting	392
Sample Data for K-mer Counting	392
Applications of K-mer Counting	392
K-mer Counting Solution in MapReduce/Hadoop	393
The map() Function	393
The reduce() Function	394
Hadoop Implementation Classes	394
K-mer Counting Solution in Spark	395
Spark Solution	396
Sample Run	405
18. DNA Sequencing.....	407
Input Data for DNA Sequencing	409
Input Data Validation	410
DNA Sequence Alignment	411
MapReduce Algorithms for DNA Sequencing	412
Step 1: Alignment	415
Step 2: Recalibration	423
Step 3: Variant Detection	428
19. Cox Regression.....	433
The Cox Model in a Nutshell	434
Cox Regression Basic Terminology	435
Cox Regression Using R	436
Expression Data	436
Cox Regression Application	437
Cox Regression POJO Solution	437
Input for MapReduce	439

Input Format	440
Cox Regression Using MapReduce	440
Cox Regression Phase 1: map()	440
Cox Regression Phase 1: reduce()	441
Cox Regression Phase 2: map()	442
Sample Output Generated by Phase 1 reduce() Function	444
Sample Output Generated by the Phase 2 map() Function	445
Cox Regression Script for MapReduce	445
20. Cochran-Armitage Test for Trend.	447
Cochran-Armitage Algorithm	448
Application of Cochran-Armitage	453
MapReduce Solution	456
Input	456
Expected Output	457
Mapper	458
Reducer	459
MapReduce/Hadoop Implementation Classes	463
Sample Run	463
21. Allelic Frequency.	465
Basic Definitions	466
Chromosome	466
Bioset	466
Allele and Allelic Frequency	467
Source of Data for Allelic Frequency	467
Allelic Frequency Analysis Using Fisher's Exact Test	469
Fisher's Exact Test	469
Formal Problem Statement	471
MapReduce Solution for Allelic Frequency	471
MapReduce Solution, Phase 1	472
Input	472
Output/Result	473
Phase 1 Mapper	474
Phase 1 Reducer	475
Sample Run of Phase 1 MapReduce/Hadoop Implementation	479
Sample Plot of P-Values	480
MapReduce Solution, Phase 2	481
Phase 2 Mapper for Bottom 100 P-Values	482
Phase 2 Reducer for Bottom 100 P-Values	484
Is Our Bottom 100 List a Monoid?	485
Hadoop Implementation Classes for Bottom 100 List	486

MapReduce Solution, Phase 3	486
Phase 3 Mapper for Bottom 100 P-Values	487
Phase 3 Reducer for Bottom 100 P-Values	489
Hadoop Implementation Classes for Bottom 100 List for Each Chromosome	490
Special Handling of Chromosomes X and Y	490
22. The T-Test.....	491
Performing the T-Test on Biosets	492
MapReduce Problem Statement	495
Input	496
Expected Output	496
MapReduce Solution	496
Hadoop Implementation Classes	499
Spark Implementation	499
High-Level Steps	500
T-Test Algorithm	507
Sample Run	509
23. Pearson Correlation.....	513
Pearson Correlation Formula	514
Pearson Correlation Example	516
Data Set for Pearson Correlation	517
POJO Solution for Pearson Correlation	517
POJO Solution Test Drive	518
MapReduce Solution for Pearson Correlation	519
map() Function for Pearson Correlation	519
reduce() Function for Pearson Correlation	520
Hadoop Implementation Classes	521
Spark Solution for Pearson Correlation	522
Input	523
Output	523
Spark Solution	524
High-Level Steps	525
Step 1: Import required classes and interfaces	527
smaller() method	528
MutableDouble class	529
toMap() method	530
toListOfString() method	530
readBiosets() method	531
Step 2: Handle input parameters	532
Step 3: Create a Spark context object	533

Step 4: Create list of input files/biomarkers	534
Step 5: Broadcast reference as global shared object	534
Step 6: Read all biomarkers from HDFS and create the first RDD	534
Step 7: Filter biomarkers by reference	535
Step 8: Create (Gene-ID, (Patient-ID, Gene-Value)) pairs	536
Step 9: Group by gene	537
Step 10: Create Cartesian product of all genes	538
Step 11: Filter redundant pairs of genes	538
Step 12: Calculate Pearson correlation and p-value	539
Pearson Correlation Wrapper Class	542
Testing the Pearson Class	543
Pearson Correlation Using R	543
YARN Script to Run Spark Program	544
Spearman Correlation Using Spark	544
Spearman Correlation Wrapper Class	544
Testing the Spearman Correlation Wrapper Class	545
24. DNA Base Count.....	547
FASTA Format	548
FASTA Format Example	549
FASTQ Format	549
FASTQ Format Example	549
MapReduce Solution: FASTA Format	550
Reading FASTA Files	550
MapReduce FASTA Solution: map()	550
MapReduce FASTA Solution: reduce()	551
Sample Run	552
Log of sample run	552
Generated output	552
Custom Sorting	553
Custom Partitioning	554
MapReduce Solution: FASTQ Format	556
Reading FASTQ Files	557
MapReduce FASTQ Solution: map()	558
MapReduce FASTQ Solution: reduce()	559
Hadoop Implementation Classes: FASTQ Format	560
Sample Run	560
Spark Solution: FASTA Format	561
High-Level Steps	561
Sample Run	564
Spark Solution: FASTQ Format	566
High-Level Steps	566

Step 1: Import required classes and interfaces	567
Step 2: Handle input parameters	567
Step 3: Create a JavaPairRDD from FASTQ input	568
Step 4: Map partitions	568
Step 5: Collect all DNA base counts	569
Step 6: Emit Final Counts	570
Sample Run	570
25. RNA Sequencing.....	573
Data Size and Format	574
MapReduce Workflow	574
Input Data Validation	574
RNA Sequencing Analysis Overview	575
MapReduce Algorithms for RNA Sequencing	578
Step 1: MapReduce TopHat Mapping	579
Step 2: MapReduce Calling Cuffdiff	582
26. Gene Aggregation.....	585
Input	586
Output	586
MapReduce Solutions (Filter by Individual and by Average)	587
Mapper: Filter by Individual	588
Reducer: Filter by Individual	590
Mapper: Filter by Average	590
Reducer: Filter by Average	592
Computing Gene Aggregation	592
Hadoop Implementation Classes	594
Analysis of Output	597
Gene Aggregation in Spark	600
Spark Solution: Filter by Individual	601
Sharing Data Between Cluster Nodes	601
High-Level Steps	602
Utility Functions	607
Sample Run	609
Spark Solution: Filter by Average	610
High-Level Steps	611
Utility Functions	616
Sample Run	619
27. Linear Regression.....	621
Basic Definitions	622
Simple Example	622

Problem Statement	624
Input Data	625
Expected Output	625
MapReduce Solution Using SimpleRegression	626
Hadoop Implementation Classes	628
MapReduce Solution Using R's Linear Model	629
Phase 1	630
Phase 2	633
Hadoop Implementation Using Classes	635
28. MapReduce and Monoids.....	637
Introduction	637
Definition of Monoid	639
How to Form a Monoid	640
Monoidic and Non-Monoidic Examples	640
Maximum over a Set of Integers	641
Subtraction over a Set of Integers	641
Addition over a Set of Integers	641
Multiplication over a Set of Integers	641
Mean over a Set of Integers	642
Non-Commutative Example	642
Median over a Set of Integers	642
Concatenation over Lists	642
Union/Intersection over Integers	643
Functional Example	643
Matrix Example	644
MapReduce Example: Not a Monoid	644
MapReduce Example: Monoid	646
Hadoop Implementation Classes	647
Sample Run	648
View Hadoop output	650
Spark Example Using Monoids	650
High-Level Steps	652
Sample Run	656
Conclusion on Using Monoids	657
Functors and Monoids	658
29. The Small Files Problem.....	661
Solution 1: Merging Small Files Client-Side	662
Input Data	665
Solution with SmallFilesConsolidator	665
Solution Without SmallFilesConsolidator	667

Solution 2: Solving the Small Files Problem with CombineFileInputFormat	668
Custom CombineFileInputFormat	672
Sample Run Using CustomCFIF	672
Alternative Solutions	674
30. Huge Cache for MapReduce.....	675
Implementation Options	676
Formalizing the Cache Problem	677
An Elegant, Scalable Solution	678
Implementing the LRUMap Cache	681
Extending the LRUMap Class	681
Testing the Custom Class	682
The MapDBEntry Class	683
Using MapDB	684
Testing MapDB: put()	686
Testing MapDB: get()	687
MapReduce Using the LRUMap Cache	687
CacheManager Definition	688
Initializing the Cache	689
Using the Cache	690
Closing the Cache	691
31. The Bloom Filter.....	693
Bloom Filter Properties	693
A Simple Bloom Filter Example	696
Bloom Filters in Guava Library	696
Using Bloom Filters in MapReduce	698
A. Bioset.....	699
B. Spark RDDs.....	701
Bibliography.....	721
Index.....	725

Secondary Sort: Introduction

A *secondary sort problem* relates to sorting values associated with a key in the reduce phase. Sometimes, it 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. I will provide concrete examples of how to achieve secondary sorting in ascending or descending order.

The goal of this chapter is to implement the Secondary Sort design pattern in MapReduce/Hadoop and Spark. In software design and programming, a *design pattern* is a reusable algorithm that is used to solve a commonly occurring problem. Typically, a design pattern is not presented in a specific programming language but instead can be implemented by many programming languages.

The 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; they can be in any order. What if you also want to sort a reducer's values? MapReduce/Hadoop and Spark do not sort values for a reducer. So, for those applications (such as time series data) in which you want to sort your reducer data, the Secondary Sort design pattern enables you to do so.

First we'll focus on the MapReduce/Hadoop solution. Let's look at the MapReduce paradigm and then unpack 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, $(key_1, value_1)$. Then it outputs any number of key-value pairs, $(key_2, value_2)$. Next, the `reduce()` function

receives as input another key-value pair, $(key_2, list(value_2))$, and outputs any number of $(key_3, value_3)$ pairs.

Now consider the following key-value pair, $(key_2, list(value_2))$, as an input for a reducer:

$$list(value_2) = (V_1, V_2, \dots, V_n)$$

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

The goal of the Secondary Sort pattern is to give *some ordering* to the values received by a reducer. So, once we apply the pattern to our MapReduce paradigm, then we will have:

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

$$list(value_2) = (S_1, S_2, \dots, S_n)$$

where:

- $S_1 < S_2 < \dots < S_n$ (ascending order), or
- $S_1 > S_2 > \dots > S_n$ (descending order)

Here is an example of a secondary sorting problem: consider the temperature data from a scientific experiment. A dump of the temperature data might look something like the following (columns are year, month, day, and daily temperature, respectively):

```
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 the values sorted in ascending order. Essentially, we want the reducer 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, ...

Solutions to the Secondary Sort Problem

There are at least two possible approaches for sorting the reducer values. These solutions may be applied to both the MapReduce/Hadoop and Spark frameworks:

- 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 doing 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 might cause the reducer to run out of memory (`java.lang.OutOfMemoryError`). On the other hand, this approach can work well if the number of values is small enough that it will not cause an out-of-memory error.
- The second approach involves using the MapReduce framework for sorting the reducer values (this does not require in-reducer sorting of values passed to the reducer). This approach consists of “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 [Java Code Geeks](#). This option is scalable and will not generate out-of-memory errors. Here, we basically offload the sorting to the MapReduce framework (sorting is a paramount feature of the MapReduce/Hadoop framework).

This is a summary of the second approach:

1. Use the *Value-to-Key Conversion* design pattern: form a composite intermediate key, (K, V_1) , where V_1 is the secondary key. Here, K is called a *natural key*. To inject a value (i.e., V_1) into a reducer key, simply create a composite key (for details, see the `DateTemperaturePair` class). In our example, V_1 is the temperature data.
2. Let the MapReduce execution framework do the sorting (rather than sorting in memory, let the framework sort by using the cluster nodes).
3. Preserve state across multiple key-value pairs to handle processing; you can achieve this by having proper mapper output partitioners (for example, we partition the mapper’s output by the natural key).

Implementation Details

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

- How to sort reducer keys
- How to partition keys passed to reducers (custom partitioner)
- How to group data that has arrived at each reducer

Sort order of intermediate keys

To accomplish secondary sorting, we need to take control of the sort order of intermediate keys and the control order in which reducers process keys. First, we inject a value (temperature data) into the composite key, and then we take control of the sort order of intermediate keys. The relationships between the natural key, composite key, and key-value pairs are depicted in [Figure 1-1](#).

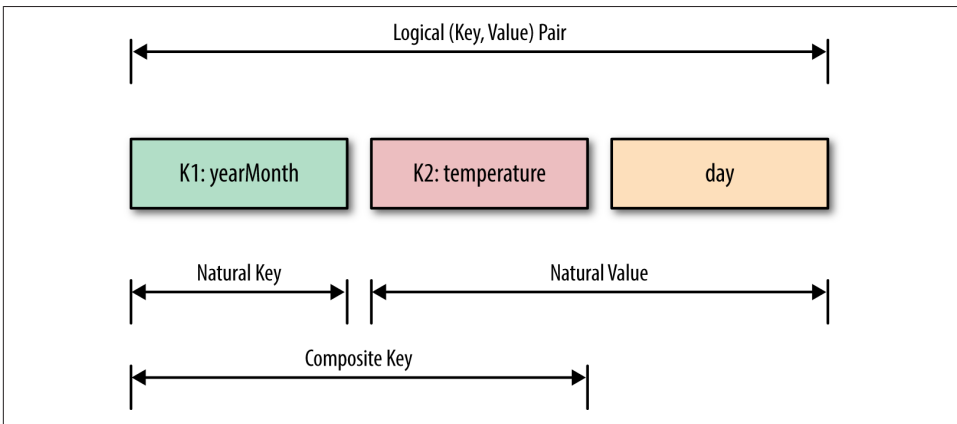


Figure 1-1. Secondary sorting keys

The main question is what value we should add to the natural key to accomplish the secondary sort. The answer is the temperature data field (because we want the reducers' values to be sorted by temperature). So, we have to indicate how `DateTemperaturePair` objects should be sorted using the `compareTo()` method. We need to define a proper data structure for holding our key and value, while also providing the sort order of intermediate keys. In Hadoop, for custom data types (such as `DateTemperaturePair`) to be persisted, 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 `WritableComparable` (see [Example 1-1](#)).

Example 1-1. `DateTemperaturePair` class

```

1 import org.apache.hadoop.io.Writable;
2 import org.apache.hadoop.io.WritableComparable;
3 ...
4 public class DateTemperaturePair

```

```

5 implements Writable, WritableComparable<DateTemperaturePair> {
6
7     private Text yearMonth = new Text();           // natural key
8     private Text day = new Text();
9     private IntWritable temperature = new IntWritable(); // secondary key
10
11     ...
12
13     @Override
14     /**
15      * This comparator controls the sort order of the keys.
16      */
17     public int compareTo(DateTemperaturePair pair) {
18         int compareValue = this.yearMonth.compareTo(pair.getYearMonth());
19         if (compareValue == 0) {
20             compareValue = temperature.compareTo(pair.getTemperature());
21         }
22         //return compareValue; // sort ascending
23         return -1*compareValue; // sort descending
24     }
25     ...
26 }

```

Custom partitioner

In a nutshell, the partitioner decides which mapper's output goes to which reducer based on the mapper's output key. For this, we need two plug-in classes: a custom partitioner to control which reducer processes which keys, and a custom `Comparator` to sort reducer values. The custom partitioner ensures that all data with the same key (the natural key, not including the composite key with the temperature value) is sent to the same reducer. The custom `Comparator` does sorting so that the natural key (year-month) groups the data once it arrives at the reducer.

Example 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 }

```

Hadoop provides a plug-in architecture for injecting the custom partitioner code into the framework. This is how we do so inside the driver class (which submits the Map-Reduce job to Hadoop):

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

Grouping comparator

In [Example 1-3](#), we define the comparator (`DateTemperatureGroupingComparator` class) that controls which keys are grouped together for a single call to the `Reducer.reduce()` function.

Example 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 }
```

Hadoop provides a plug-in architecture for injecting the grouping comparator code into the framework. This is how we do so inside the driver class (which submits the MapReduce job to Hadoop):

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

Data Flow Using Plug-in Classes

To help you understand the `map()` and `reduce()` functions and custom plug-in classes, [Figure 1-2](#) illustrates the data flow for a portion of input.

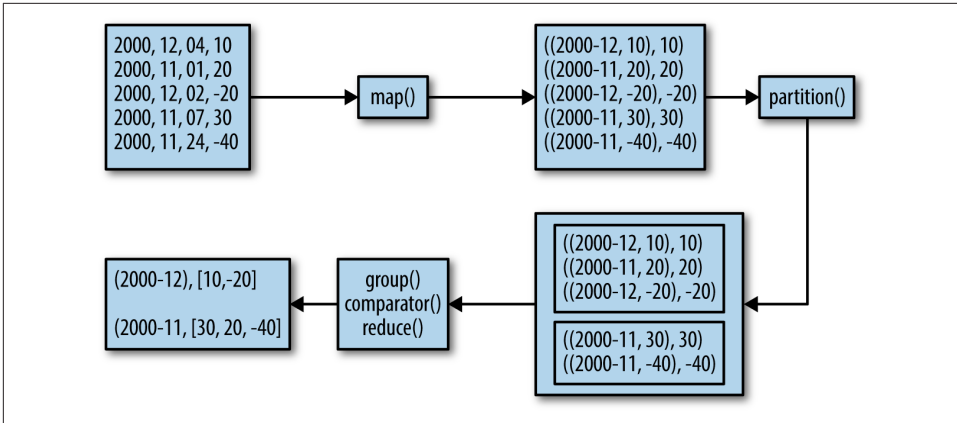


Figure 1-2. Secondary sorting data flow

The mappers create (K,V) pairs, where K is a composite key of (year ,month , tempera ture) and V is temperature. The (year ,month) part of the composite key is the natural key. The partitioner plug-in class enables us to send all natural keys to the same reducer and the grouping comparator plug-in class enables temperatures to arrive sorted at reducers. The Secondary Sort design pattern uses MapReduce’s framework for sorting the reducers’ values rather than collecting them all and then sorting them in memory. The Secondary Sort design pattern enables us to “scale out” no matter how many reducer values we want to sort.

MapReduce/Hadoop Solution to Secondary Sort

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

Input

The input will be a set of files, where each record (line) will have the following format:

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

Example:
 2012, 01, 01, 35
 2011, 12, 23, -4

Expected Output

The expected output will have the following format:

Format:

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

Example:

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

map() Function

The `map()` function parses and tokenizes the input and then injects the value (temperature) into the reducer key, as shown in [Example 1-4](#).

Example 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 }
```

reduce() Function

The reducer's primary function is to concatenate the values (which are already sorted through the Secondary Sort design pattern) and emit them as output. The `reduce()` function is given in [Example 1-5](#).

Example 1-5. reduce() for secondary sorting

```
1 /**  
2  * @param key is a DateTemperaturePair object  
3  * @param value is a list of temperatures  
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 }

```

Hadoop Implementation Classes

The classes shown in [Table 1-1](#) are used to solve the problem.

Table 1-1. Classes used in MapReduce/Hadoop solution

Class name	Class description
SecondarySortDriver	The driver class; defines input/output and registers plug-in classes
SecondarySortMapper	Defines the map() function
SecondarySortReducer	Defines the reduce() function
DateTemperatureGroupingComparator	Defines how keys will be grouped together
DateTemperaturePair	Defines paired date and temperature as a Java object
DateTemperaturePartitioner	Defines custom partitioner

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

The `SecondarySortDriver` is the driver class, which registers the custom plug-in classes (`DateTemperaturePartitioner` and `DateTemperatureGroupingComparator`) with the MapReduce/Hadoop framework. This driver class is presented in [Example 1-6](#).

Example 1-6. SecondarySortDriver class

```

1 public class SecondarySortDriver extends Configured implements Tool {
2     public int run(String[] args) throws Exception {
3         Configuration conf = getConf();
4         Job job = new Job(conf);
5         job.setJarByClass(SecondarySortDriver.class);
6         job.setJobName("SecondarySortDriver");
7
8         Path inputPath = new Path(args[0]);
9         Path outputPath = new Path(args[1]);

```

```

10     FileInputFormat.setInputPaths(job, inputPath);
11     FileOutputFormat.setOutputPath(job, outputPath);
12
13     job.setOutputKeyClass(TemperaturePair.class);
14     job.setOutputValueClass(NullWritable.class);
15
16     job.setMapperClass(SecondarySortingTemperatureMapper.class);
17     job.setReducerClass(SecondarySortingTemperatureReducer.class);
18     job.setPartitionerClass(TemperaturePartitioner.class);
19     job.setGroupingComparatorClass(YearMonthGroupingComparator.class);
20
21     boolean status = job.waitForCompletion(true);
22     theLogger.info("run(): status="+status);
23     return status ? 0 : 1;
24 }
25
26 /**
27  * The main driver for the secondary sort MapReduce program.
28  * Invoke this method to submit the MapReduce job.
29  * @throws Exception when there are communication
30  * 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" +
36             " <input-path> <output-path>");
37     }
38
39     //String inputPath = args[0];
40     //String outputPath = args[1];
41     int returnStatus = ToolRunner.run(new SecondarySortDriver(), args);
42     System.exit(returnStatus);
43 }
44
45 }

```

Sample Run of Hadoop Implementation

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
```

HDFS input

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

The script

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

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
```

Inspecting the 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
```

How to Sort in Ascending or Descending Order

You can easily control the sorting order of the values (ascending or descending) by using the `DateTemperaturePair.compareTo()` method as follows:

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

Spark Solution to Secondary Sort

To solve a secondary sorting problem in Spark, we have at least two options:

Option #1

Read and buffer all of the values for a given *key* in an `Array` or `List` data structure and then do an in-reducer sort on the values. This solution works if you have a small set of values (which will fit in memory) per reducer key.

Option #2

Use the Spark framework for sorting the reducer values (this option 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 the memory of a commodity server).

Time Series as Input

To demonstrate secondary sorting, let’s use time series data:

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

```

z  2  8
z  3  7
z  4  0
p  2  6
p  4  7
p  1  9
p  6  0
p  7  3

```

Expected Output

Our expected output is as follows. Note that 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]

```

Option 1: Secondary Sorting in Memory

Since Spark has a very powerful and high-level API, I will present the entire solution in a single Java class. The Spark API is built upon the basic abstraction concept of the RDD (resilient distributed data set). To fully utilize Spark's API, we have to understand RDDs. An `RDD<T>` (i.e., an RDD of type `T`) *object* represents an immutable, partitioned collection of elements (of 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 the `JavaPairRDD<K,V>` class contains 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](#) and [Appendix B](#) of this book.) 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>` (where each tuple represents a key-value pair).

The Spark-based algorithm is listed next. Although there are 10 steps, most of them are trivial and some are provided for debugging purposes only:

1. We import the 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`
2. We pass input data as arguments and validate.
 3. We connect to the Spark master by creating a `JavaSparkContext` object, which is used to create new RDDs.
 4. Using the context object (created in step 3), we create an RDD for the input file; the resulting RDD will be a `JavaRDD<String>`. Each element of this RDD will be a record of time series data: `<name><,><time><,><value>`.
 5. Next we want to create key-value pairs from a `JavaRDD<String>`, where the key is the name and the value is a pair of (time, value). The resulting RDD will be a `JavaPairRDD<String, Tuple2<Integer, Integer>>`.
 6. To validate step 5, we collect all values from the `JavaPairRDD<>` and print them.
 7. 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. In general, Spark's `reduceByKey()` is preferred over `groupByKey()` for performance reasons, but here we have no other option than `groupByKey()` (since `reduceByKey()` does not allow us to sort the values in place for a given key).

8. To validate step 7, we collect all values from the `JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>>` and print them.
9. We sort the reducer's values to get the final output. We accomplish this by writing a custom `mapValues()` method. We just sort the values (the key remains the same).
10. To validate the final result, we collect all values from the sorted `JavaPairRDD<>` and print them.

A solution for option #1 is implemented by a single driver class: `SecondarySorting` (see [Example 1-7](#)). All steps, 1–10, are listed inside the class definition, which will be presented in the following sections. Typically, a Spark application 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 through the extensive use of RDDs. For further details on RDDs, see [Appendix B](#).

Example 1-7. SecondarySort class overall structure

```
1 // Step 1: import required Java/Spark classes
2 public class SecondarySort {
3     public static void main(String[] args) throws Exception {
4         // Step 2: read input parameters and validate them
5         // Step 3: connect to the Spark master by creating a JavaSparkContext
6         // object (ctx)
7         // Step 4: use ctx to create JavaRDD<String>
8         // Step 5: create key-value pairs from JavaRDD<String>, where
9         // key is the {name} and value is a pair of (time, value)
10        // Step 6: validate step 5-collect all values from JavaPairRDD<>
11        // and print them
12        // Step 7: group JavaPairRDD<> elements by the key ({name})
13        // Step 8: validate step 7-collect all values from JavaPairRDD<>
14        // and print them
15        // Step 9: sort the reducer's values; this will give us the final output
16        // Step 10: validate step 9-collect all values from JavaPairRDD<>
17        // and print them
18        // done
19        ctx.close();
20        System.exit(0);
21    }
22 }
```

Step 1: Import required classes

As shown in [Example 1-8](#), the main Spark package for the Java API is `org.apache.spark.api.java`, which includes the `JavaRDD`, `JavaPairRDD`, and `JavaSparkContext` classes. `JavaSparkContext` is a factory class for creating new RDDs (such as `JavaRDD` and `JavaPairRDD` objects).

Example 1-8. Step 1: Import required classes

```
1 // Step 1: 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;
```


Step 2: Read input parameters

This step, demonstrated in [Example 1-9](#), reads the HDFS input file (Spark may read data from HDFS and other persistent stores, such as a Linux filesystem), which might look like `/dir1/dir2/myfile.txt`.

Example 1-9. Step 2: Read input parameters

```
1 // Step 2: read input parameters and validate them
2 if (args.length < 1) {
3     System.err.println("Usage: SecondarySort <file>");
4     System.exit(1);
5 }
6 String inputPath = args[0];
7 System.out.println("args[0]: <file>="+args[0]);
```

Step 3: Connect to the Spark master

To work with RDDs, first you need to create a `JavaSparkContext` object (as shown in [Example 1-10](#)), which is a factory for creating `JavaRDD` and `JavaPairRDD` objects. It is also possible to create a `JavaSparkContext` object by injecting a `SparkConf` object into the `JavaSparkContext`'s class constructor. This approach is useful when you read your cluster configurations from an XML file. In a nutshell, the `JavaSparkContext` object has the following responsibilities:

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

Example 1-10. Step 3: Connect to the Spark master

```
1 // Step 3: connect to the Spark master by creating a JavaSparkContext object
2 final JavaSparkContext ctx = new JavaSparkContext();
```

Step 4: Use the JavaSparkContext to create a JavaRDD

This step, illustrated in [Example 1-11](#), reads an HDFS file and creates a `JavaRDD<String>` (which represents a set of records where each record is a `String` object). By definition, Spark's RDDs are *immutable* (i.e., they cannot be altered or modified). Note that Spark's RDDs are the basic abstraction for parallel execution. Note also that you may use `textFile()` to read HDFS or non-HDFS files.

Example 1-11. Step 4: Create JavaRDD

```
1 // Step 4: use ctx to create JavaRDD<String>
2 // input record format: <name><,><time><,><value>
3 JavaRDD<String> lines = ctx.textFile(inputPath, 1);
```

Step 5: Create key-value pairs from the JavaRDD

This step, shown in [Example 1-12](#), implements a mapper. Each record (from the `JavaRDD<String>` and consisting of `<name><,><time><,><value>`) is converted to a key-value pair, where the key is a name and the value is a `Tuple2(time, value)`.

Example 1-12. Step 5: Create key-value pairs from JavaRDD

```
1 // Step 5: 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 a 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     lines.mapToPair(new PairFunction<
11         String, // T
12         String, // K
13         Tuple2<Integer, Integer> // V
14     >() {
15     public Tuple2<String, Tuple2<Integer, Integer>> call(String s) {
16         String[] tokens = s.split(","); // x,2,5
17         System.out.println(tokens[0] + "," + tokens[1] + "," + tokens[2]);
18         Integer time = new Integer(tokens[1]);
19         Integer value = new Integer(tokens[2]);
20         Tuple2<Integer, Integer> timevalue =
21         new Tuple2<Integer, Integer>(time, value);
22         return new Tuple2<String, Tuple2<Integer, Integer>>(tokens[0], timevalue);
23     }
24 });
```

Step 6: Validate step 5

To debug and validate your steps in Spark (as shown in [Example 1-13](#)), you may use `JavaRDD.collect()` and `JavaPairRDD.collect()`. Note that `collect()` is used for debugging and educational purposes (but avoid using `collect()` for debugging purposes in production clusters; doing so will impact performance). Also, you may use `JavaRDD.saveAsTextFile()` for debugging as well as creating your desired outputs.

Example 1-13. Step 6: Validate step 5

```
1 // Step 6: validate step 5-collect all values from JavaPairRDD<>
2 // and print them
3 List<Tuple2<String, Tuple2<Integer, Integer>>> output = pairs.collect();
4 for (Tuple2 t : output) {
5     Tuple2<Integer, Integer> timevalue = (Tuple2<Integer, Integer>) t._2;
6     System.out.println(t._1 + "," + timevalue._1 + "," + timevalue._1);
7 }
```

Step 7: Group JavaPairRDD elements by the key (name)

We implement the reducer operation using `groupByKey()`. As you can see in [Example 1-14](#), it is much easier to implement the reducer through Spark than Map-Reduce/Hadoop. Note that in Spark, in general, `reduceByKey()` is more efficient than `groupByKey()`. Here, however, we cannot use `reduceByKey()`.

Example 1-14. Step 7: Group JavaPairRDD elements

```
1 // Step 7: group JavaPairRDD<> elements by the key ({name})
2 JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> groups =
3     pairs.groupByKey();
```

Step 8: Validate step 7

This step, shown in [Example 1-15](#), validates the previous step by using the `collect()` function, which gets all values from the groups RDD.

Example 1-15. Step 8: Validate step 7

```
1 // Step 8: validate step 7-we collect all values from JavaPairRDD<>
2 // and print them
2 System.out.println("===DEBUG1===");
3 List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output2 =
4     groups.collect();
5 for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output2) {
6     Iterable<Tuple2<Integer, Integer>> list = t._2;
7     System.out.println(t._1);
8     for (Tuple2<Integer, Integer> t2 : list) {
9         System.out.println(t2._1 + "," + t2._2);
10    }
11    System.out.println("=====");
12 }
```

The following shows the output of this step. As you can see, 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
=====

```

Step 9: Sort the reducer's values in memory

This step, shown in [Example 1-16](#), uses another powerful Spark method, `mapValues()`, to just sort the values generated by reducers. The `mapValues()` method enables us to convert (K, V_1) into (K, V_2) , where V_2 is a sorted V_1 . One important note about Spark's RDD is that it is immutable and cannot be altered/updated by any means. For example, in this step, to sort our values, we have to copy them into another list first. Immutability applies to the RDD itself and its elements.

Example 1-16. Step 9: sort the reducer's values in memory

```

1 // Step 9: sort the reducer's values; 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(
9         new Function<Iterable<Tuple2<Integer, Integer>>, // input
10             Iterable<Tuple2<Integer, Integer>> // output
11             >() {
12     public Iterable<Tuple2<Integer, Integer>> call(Iterable<Tuple2<Integer,
13                                                     Integer>> s) {
14         List<Tuple2<Integer, Integer>> newList = new ArrayList<Tuple2<Integer,
15                                                     Integer>>(s);
16         Collections.sort(newList, new TupleComparator());
17         return newList;

```

```
18     }
19   });
```

Step 10: output final result

The `collect()` method collects all of the RDD's elements into a `java.util.List` object. Then we iterate through the `List` to get all the final elements (see [Example 1-17](#)).

Example 1-17. Step 10: Output final result

```
1 // Step 10: validate step 9-collect all values from JavaPairRDD<>
2 // and print them
3 System.out.println("===DEBUG===");
4 List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output3 =
5     sorted.collect();
6 for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output3) {
7     Iterable<Tuple2<Integer, Integer>> list = t._2;
8     System.out.println(t._1);
9     for (Tuple2<Integer, Integer> t2 : list) {
10        System.out.println(t2._1 + "," + t2._2);
11    }
12    System.out.println("=====");
13 }
```

Spark Sample Run

As far as Spark/Hadoop is concerned, you can run a Spark application in three different modes:¹

Standalone mode

This is the default setup. You start the Spark master on a master node and a “worker” on every slave node, and submit your Spark application to the Spark master.

YARN client mode

In this mode, you do not start a Spark master or worker nodes. Instead, 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. Instead, you submit the Spark application to YARN, which runs the Spark driver in the ApplicationMaster in YARN.

¹ For details, see the [Spark documentation](#).

Next, we will cover how to submit the secondary sort application in the standalone and YARN cluster modes.

Running Spark in standalone mode

The following subsections provide the input, script, and log output of a sample run of our secondary sort application in Spark's standalone mode.

HDFS input.

```
# 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
```

The script.

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

Log of the run.

```
# ./run_secondarysorting.sh
args[0]: <file>=/mp/timeseries.txt
...
===  DEBUG STEP 5  ===
```

```

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

```

```

...
14/06/04 08:42:55 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0,
  whose tasks have all completed, from pool
14/06/04 08:42:55 INFO spark.SparkContext: Job finished: collect at
  SecondarySort.java:158, took 0.074271723 s
z
1,4
2,8
3,7
4,0
=====
p
1,9
2,6
4,7
6,0
7,3
=====
x
1,3
2,9
3,6
=====
y
1,7
2,5
3,1
=====

```

Typically, you save the final result to HDFS. You can accomplish this by adding the following line of code after creating your “sorted” RDD:

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

Then you may view the output as follows:

```

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

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

```

Running Spark in YARN cluster mode. The script to submit our Spark application in YARN cluster mode is as follows:

```

# cat run_secondarysorting_yarn.sh
#!/bin/bash
export JAVA_HOME=/usr/java/jdk7

```



```

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

```

Option #2: Secondary Sorting Using the Spark Framework

In the solution for option #1, we sorted reducer values in memory (using Java's `Collections.sort()` method), which might not scale if the reducer values will not fit in a commodity server's memory. Next we will implement option #2 for the MapReduce/Hadoop framework. We cannot achieve this in the current Spark (Spark-1.1.0) framework, because currently Spark's shuffle is based on a hash, which is different from MapReduce's sort-based shuffle. So, you should implement sorting explicitly using an RDD operator. If we had a partitioner by a natural key (name) that preserved the order of the RDD, that would be a viable solution—for example, if we sorted 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 the original RDD elements. Therefore, option #2 cannot be implemented by the current version of Spark (1.1.0).

Further Reading on Secondary Sorting

To support secondary sorting in Spark, you may extend the `JavaPairRDD` class and add additional methods such as `groupByKeyAndSortValues()`. For further work on this topic, you may refer to the following:

- Support sorting of values in addition to keys (i.e., secondary sort)
- <https://github.com/tresata/spark-sorted>

Chapter 2 provides a detailed implementation of the Secondary Sort design pattern using the MapReduce and Spark frameworks.

Want to read more?

You can [buy this book](#) at oreilly.com in print and ebook format.

Buy 2 books, get the 3rd FREE!

Use discount code OPC10

All orders over \$29.95 qualify for **free shipping** within the US.

It's also available at your favorite book retailer, including the iBookstore, the [Android Marketplace](#), and [Amazon.com](#).



O'REILLY[®]

©2015 O'Reilly Media, Inc. The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. 15055