O'REILLY®


# Data Algorithms

RECIPES FOR SCALING UP WITH HADOOP AND SPARK

Mahmoud Parsian

O'REILLY®

# 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


Twitter: @oreillymedia
facebook.com/oreilly

**Data Algorithms**
by Mahmoud Parsian



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.



978-1-491-90618-7

[LSI]

# Table of Contents

CHAPTER 1

# 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:

$$\texttt{map}(\texttt{key}_1, \texttt{value}_1) \rightarrow \texttt{list}(\texttt{key}_2, \texttt{value}_2)$$

$$\texttt{reduce}(\texttt{key}_2, \texttt{list}(\texttt{value}_2)) \rightarrow \texttt{list}(\texttt{key}_3, \texttt{value}_3)$$

First, the `map()` function receives a key-value pair input, ($\texttt{key}_1$, $\texttt{value}_1$). Then it outputs any number of key-value pairs, ($\texttt{key}_2$, $\texttt{value}_2$). 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:

$$list(value_2) = (V_1, V_2, ..., V_n)$$

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

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:

$$SORT(V_1, V_2, ..., V_n) = (S_1, S_2, ..., S_n)$$
$$list(value_2) = (S_1, S_2, ..., S_n)$$

where:

- $S_1 < S_2 < ... < S_n$ (ascending order), or
- $S_1 > S_2 > ... > S_n$ (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.

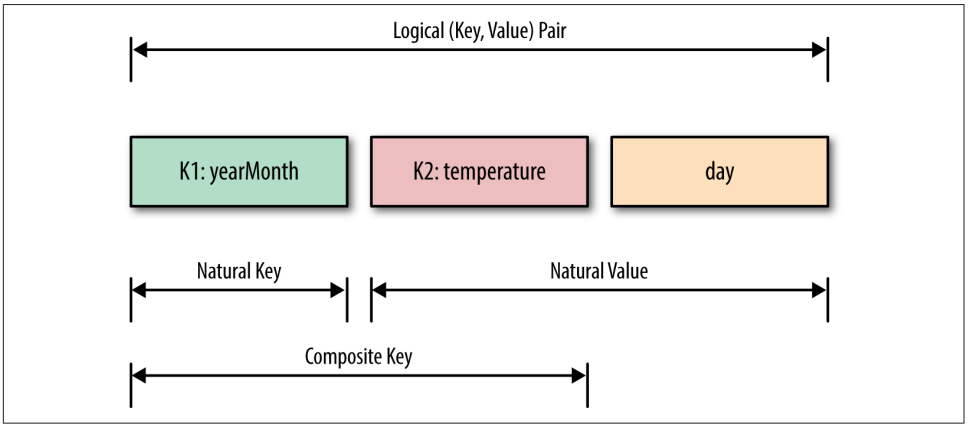


*Figure 1-1. Secondary sorting keys*

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
```

```
   implements Writable, WritableComparable<DateTemperaturePair> {

    private Text yearMonth = new Text();                // natural key
    private Text day = new Text();
    private IntWritable temperature = new IntWritable(); // secondary key

    ...

    @Override
    /**
     * This comparator controls the sort order of the keys.
     */
    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
    }
    ...
}
```

### 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*

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Partitioner;

public class DateTemperaturePartitioner
   extends Partitioner<DateTemperaturePair, Text> {

    @Override
    public int getPartition(DateTemperaturePair pair,
                            Text text,
                            int numberOfPartitions) {
        // make sure that partitions are non-negative
        return Math.abs(pair.getYearMonth().hashCode() % numberOfPartitions);
     }
}
```

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 MapReduce 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*

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

public class DateTemperatureGroupingComparator
   extends WritableComparator {

    public DateTemperatureGroupingComparator() {
        super(DateTemperaturePair.class, true);
    }

    @Override
    /**
     * This comparator controls which keys are grouped
     * together into a single call to the reduce() method
     */
    public int compare(WritableComparable wc1, WritableComparable wc2) {
        DateTemperaturePair pair = (DateTemperaturePair) wc1;
        DateTemperaturePair pair2 = (DateTemperaturePair) wc2;
        return pair.getYearMonth().compareTo(pair2.getYearMonth());
    }
}
```

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.

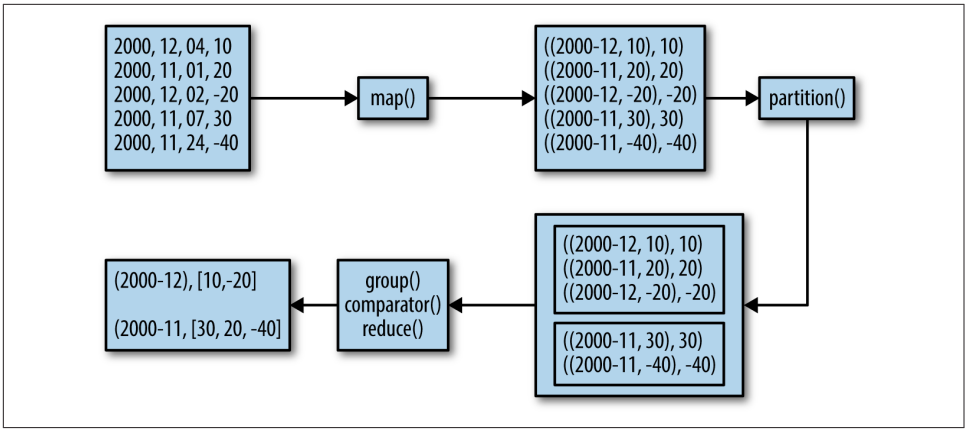


*Figure 1-2. Secondary sorting data flow*

The mappers create (*K*,*V*) pairs, where *K* is a composite key of `(year,month,temperature)` 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*

```
/**
 * @param key is generated by Hadoop (ignored here)
 * @param value has this format: "YYYY,MM,DD,temperature"
 */
map(key, value) {
   String[] tokens = value.split(",");
   // YYYY = tokens[0]
   // MM = tokens[1]
   // DD = tokens[2]
   // temperature = tokens[3]
   String yearMonth = tokens[0] + tokens[1];
   String day = tokens[2];
   int temperature = Integer.parseInt(tokens[3]);
   // prepare reducer key
   DateTemperaturePair reducerKey = new DateTemperaturePair();
   reducerKey.setYearMonth(yearMonth);
   reducerKey.setDay(day);
   reducerKey.setTemperature(temperature); // inject value into key
   // send it to reducer
   emit(reducerKey, temperature);
}
```

## 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*

```
/**
 * @param key is a DateTemperaturePair object
 * @param value is a list of temperatures
 */
```

```
reduce(key, value) {
   StringBuilder sortedTemperatureList = new StringBuilder();
   for (Integer temperature : value) {
      sortedTemperatureList.append(temperature);
      sortedTemperatureList.append(",");
   }
   emit(key, sortedTemperatureList);
}
```

## 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*

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

        Path inputPath = new Path(args[0]);
        Path outputPath = new Path(args[1]);
```

```
        FileInputFormat.setInputPaths(job, inputPath);
        FileOutputFormat.setOutputPath(job, outputPath);

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

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

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

    /**
    * The main driver for the secondary sort MapReduce program.
    * Invoke this method to submit the MapReduce job.
    * @throws Exception when there are communication
    * problems with the job tracker.
    */
    public static void main(String[] args) throws Exception {
        // Make sure there are exactly 2 parameters
        if (args.length != 2) {
            throw new IllegalArgumentException("Usage: SecondarySortDriver" +
                                               " <input-path> <output-path>");
        }

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

}
```

# 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*

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

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

### 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*

```
// Step 1: import required Java/Spark classes
import scala.Tuple2;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaPairRDD;
import org.apache.spark.api.java.JavaSparkContext;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.Function2;
import org.apache.spark.api.java.function.PairFunction;

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

### 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*

```
// Step 4: use ctx to create JavaRDD<String>
// input record format: <name><,><time><,><value>
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*

```
// Step 5: 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 a JavaPairRDD<String, Tuple2<Integer, Integer>>.
// Convert each record into Tuple2(name, time, value).
// PairFunction<T, K, V>
//     T => Tuple2(K, V) where T is input (as String),
//     K=String
//     V=Tuple2<Integer, Integer>
JavaPairRDD<String, Tuple2<Integer, Integer>> pairs =
      lines.mapToPair(new PairFunction<
                                       String,                  // T
                                       String,                  // K
                                       Tuple2<Integer, Integer> // V
                                      >() {
  public Tuple2<String, Tuple2<Integer, Integer>> call(String s) {
    String[] tokens = s.split(","); // x,2,5
    System.out.println(tokens[0] + "," + tokens[1] + "," + tokens[2]);
    Integer time = new Integer(tokens[1]);
    Integer value = new Integer(tokens[2]);
    Tuple2<Integer, Integer> timevalue =
    new Tuple2<Integer, Integer>(time, value);
    return new Tuple2<String, Tuple2<Integer, Integer>>(tokens[0], timevalue);
  }
});
```

### 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*

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

### 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 MapReduce/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*

```
// Step 7: group JavaPairRDD<> elements by the key ({name})
JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> groups =
      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*

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

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*

```
// Step 9: sort the reducer's values; this will give us the final output.
// Option #1: worked
// mapValues[U](f: (V) => U): JavaPairRDD[K, U]
// 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.
JavaPairRDD<String, Iterable<Tuple2<Integer, Integer>>> sorted =
      groups.mapValues(
        new Function<Iterable<Tuple2<Integer, Integer>>,  // input
                     Iterable<Tuple2<Integer, Integer>>   // output
                    >() {
  public Iterable<Tuple2<Integer, Integer>> call(Iterable<Tuple2<Integer,
                                                                 Integer>> s) {
    List<Tuple2<Integer, Integer>> newList = new ArrayList<Tuple2<Integer,
                                                                  Integer>>(s);
    Collections.sort(newList, new TupleComparator());
    return newList;
```

```
    }
  });
```

### 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*

```
// Step 10: validate step 9-collect all values from JavaPairRDD<>
// and print them
System.out.println("===DEBUG2=");
List<Tuple2<String, Iterable<Tuple2<Integer, Integer>>>> output3 =
      sorted.collect();
for (Tuple2<String, Iterable<Tuple2<Integer, Integer>>> t : output3) {
   Iterable<Tuple2<Integer, Integer>> list = t._2;
   System.out.println(t._1);
   for (Tuple2<Integer, Integer> t2 : list) {
      System.out.println(t2._1 + "," + t2._2);
   }
   System.out.println("=====");
}
```

## Spark 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.