

O'REILLY®

Data Algorithms with Spark

Recipes and Design Patterns for Scaling Up
Using PySpark



Early
Release
RAW &
UNEDITED

Mahmoud Parsian

Data Algorithms with Spark

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Recipes and Design Patterns for Scaling Up using
PySpark

Mahmoud Parsian

Data Algorithms with Spark

by Mahmoud Parsian

Copyright © 2021 Mahomoud 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://oreilly.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editors: Melissa Potter and Jessica Haberman

Production Editor: Katherine Tozer

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2021: First Edition

Revision History for the Early Release

- 2020-08-26: First Release
- 2021-01-20: Second Release
- 2021-05-24: Third Release
- 2021-09-10: Fourth Release

See <http://oreilly.com/catalog/errata.csp?isbn=9781492082385> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Data Algorithms with Spark*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the author, and do not represent the publisher's views. 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.

978-1-492-08231-6

Chapter 1. Introduction to Data Algorithms

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

Spark is a powerful analytics engine for large-scale data processing. Spark aims at light speed, ease of use, extensibility in big data. Spark has introduced high-level APIs in Java, Scala, Python, and R. In this book, I use PySpark (Python API for Spark) for the simplicity.

The purpose of this chapter is to introduce PySpark as a main component of Spark echo system and show that it can effectively be used to solve big data problems such as ETL, indexing billions of documents, ingesting millions of genomes, machine learning algorithms, graph data analysis, DNA data analysis, and much more. To understand PySpark , we will review Spark and PySpark architectures and provide examples to show the expressive power of PySpark. The reason I chose Spark is very simple: Spark is a proven and well-adopted technology at many high tech industries, who handle big data every day. Spark is an analytics engine designed for large-scale distributed data processing, in the cloud or private data centers.

This chapter covers

- Why Spark for data analytics?
- What is PySpark API?
- What problems does PySpark solve?
- PySpark with examples
- Interactive PySpark Shell
- Running a PySpark application

This book provides a practical introduction to data algorithms (practical and pragmatic big data transformations) using PySpark (Python API for Spark).

To get the most out of this book, you should be comfortable reading and writing very basic Python code, and have some familiarity with fundamentals of algorithms (how to solve computational problems in simple steps). By data algorithms, I mean solving big data problems by using a set of scalable and efficient data transformations (such as mappers, reducers, filters, ...).

In this book, I explore and teach the following basic concepts:

- How to design and develop data algorithms by using PySpark
- How to solve big data problems by using Spark's data abstractions (RDDs, DataFrames, and GraphFrames)
- How to partition big data in such a way to achieve maximum performance of ETL and data transformations such as summary data patterns
- How to integrate fundamental data design patterns (such as Top-10, MinMax, InMapper Combiners, ...) with a useful and practical Spark transformations by using PySpark

In this book, I will provide simple working examples in PySpark so that you can adapt (cut-and-paste-and-modify) them to solve your big data problems.

All of the source code in this book is hosted at the GitHub [Data Algorithms with Spark](#).

My focus in this book is on PySpark rather than Spark. This means that the provided examples will focus on solving big data problems using PySpark and on getting effective by using PySpark and not an exhaustive reference on Spark's features.

This chapter will present an introduction to Spark, PySpark, and data analytics with PySpark. We will present the core architecture of Spark, PySpark, Spark Application, and Spark's Structured APIs using RDDs (Resilient Distributed Dataset), DataFrames, and SQL. We will touch on Spark's core functions (transformations and actions) and concepts so that you are empowered to start using Spark and PySpark right away. Spark's data abstractions are RDDs, DataFrames, and DataSets. This means that you can represent your data (stored as Hadoop files, Amazon S3 objects, Linux files, collection data structures, relational database tables, ...) in any combinations of RDDs and DataFrames.

Once your data is represented in a Spark data abstraction (such as an RDD or a DataFrame), then you may apply transformations on them and create new data abstractions until you come up with final desired data. Spark's transformations (such as `map()` and `reduceByKey()`) can be used to convert your data from one form into another one until you get your desired result. I will explain these data abstractions shortly.

SOURCE CODE

NOTE

Complete programs for this chapter are presented in [GitHub Chapter 1](#).

Why Spark for Data Analytics

Why should we use Spark? Spark is a powerful analytics engine for large scale data processing. The most important reasons for using Spark are listed:

- Spark is simple, powerful, and fast (uses RAM rather than disk — Spark runs workloads 100x faster.)
- Spark is open-source, free, and can solve any big data problem
- Spark runs everywhere (Hadoop, Mesos, Kubernetes, standalone, or in the cloud).
- Spark can read/write data from/to many data sources
- Spark can read/write data in row-based and column-based (such as Parquet and ORC) formats

Spark's high level architecture is illustrated by Figure 1.1.

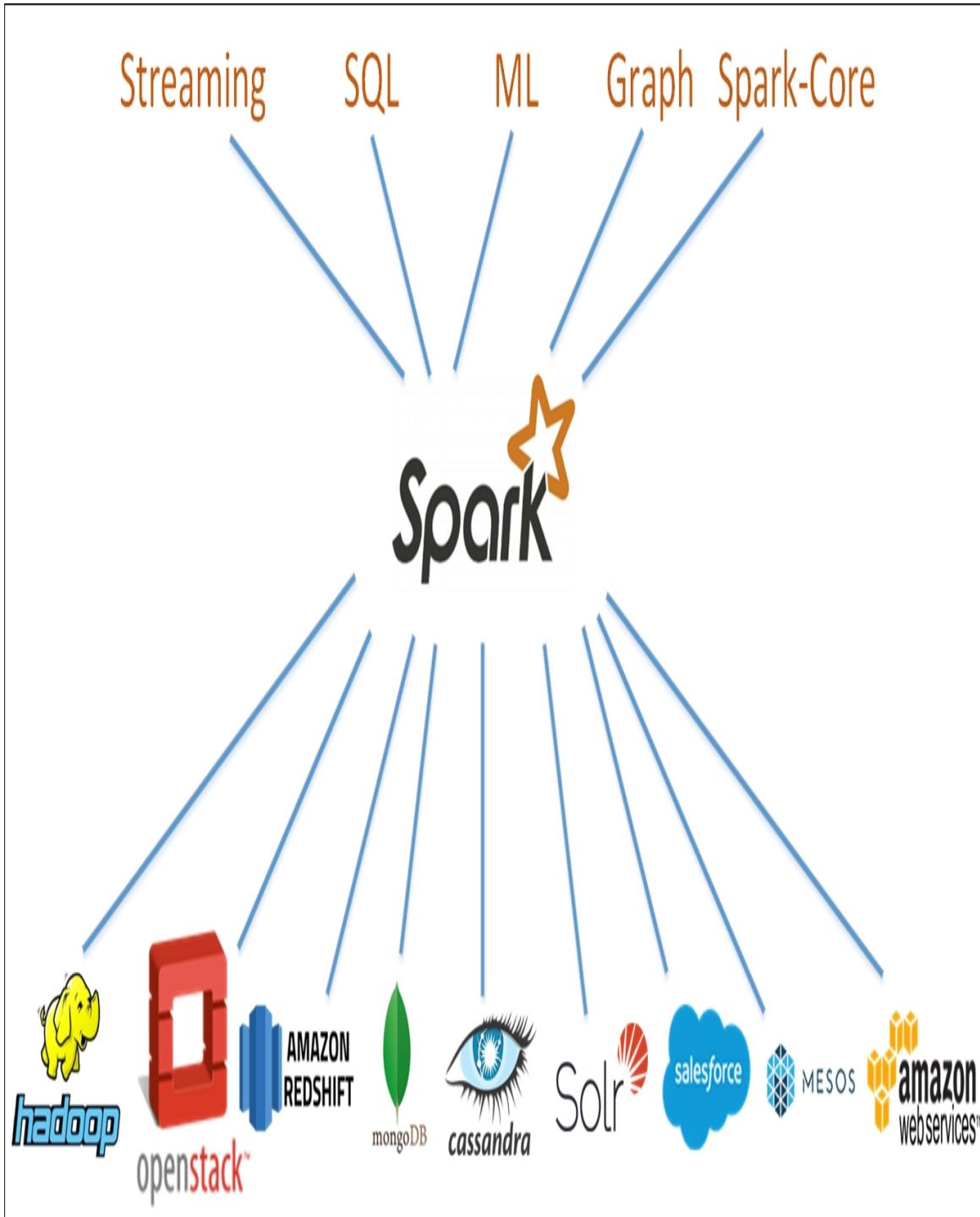


Figure 1-1. Spark's high level architecture

Spark architecture (Figure 1.1) shows that it can read/write data from any data source (Amazon S3, Hadoop HDFS, relational databases, ...) and can be integrated with almost any data applications. Spark has a rich but simple set of API in any kind of ETL, streaming, graph data analysis, machine

learning and SQL. In the past 5 years Spark has progressed in such a way to solve any big data problem. This is proven by the fact that all big data companies (FaceBook, Illumina, IBM, Google, ...) use Spark every day in production systems.

In a nutshell, Spark unlocks the power of data by handling big data with power, ease of use, and speed. Spark is one of the best choices for large-scale data processing and for solving MapReduce problems and beyond. Spark unlocks the power of data by handling big data with powerful APIs and speed. Using MapReduce/Hadoop to solve big data problems is complex and you have to write ton of low level code to solve primitive problems — this is where the power and simplicity of Spark comes in to solve complex big data problems. Apache **Spark** is much faster than Apache **Hadoop** because it uses in-memory caching and optimized execution for fast performance, and it supports general batch processing, streaming analytics, machine learning, graph algorithms, and SQL queries.

Spark’s “native” language is Scala, but you can use language APIs to run Spark code from other programming languages (for example, Java, R, and Python). In this book, I teach you how to use PySpark to solve big data problems in Spark.

In this book, you will learn how to solve your big data problems in Spark by expressing your solution in PySpark. You will learn how to read your data and represent it as an RDD and DataFrame. RDD is a fundamental data abstraction of Spark. DataFrame (a distributed table of rows with named columns) in Spark allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. Once your data is represented as an RDD or a DataFrame, then you may apply transformation functions (such as mappers, filters, reducers) on them to transform your data to your desired form. I have presented many Spark transformations, which can be used to solve your ETL, analysis, and data intensive computations.

Some simple RDD transformations are represented by the following Figure 1.1.

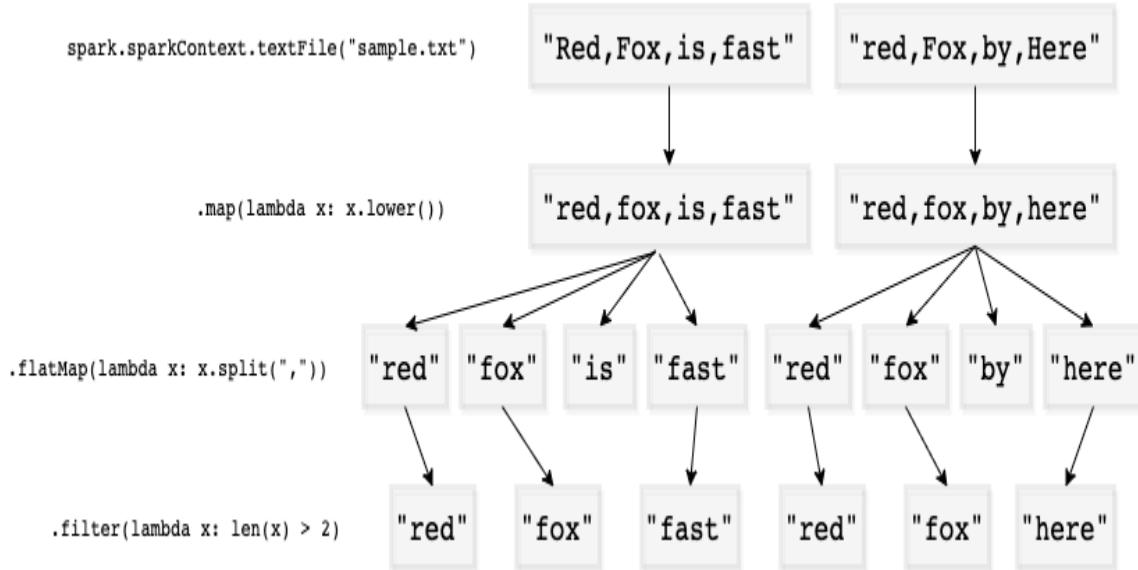


Figure 1-2. Simple RDD Transformation

The following transformations were performed (see Figure 1.1):

- First we read our input data (represented as a text file of `sample.txt` — here, I only show the first two rows/records of input data) by an instance of `SparkSession`, which is the entry point to programming Spark. Here, `SparkSession` instance is represented as a `spark` object. Reading input creates a new RDD as `RDD[String]`: each input record is converted to an RDD element of a `String` object (if your input path has `N` records, then the number of RDD elements is `N`). This is accomplished by

```

# spark : an instance of SparkSession
# creat an RDD[String], which represents all input
# records; each record becomes an RDD element
spark.sparkContext.textFile("sample.txt")

```

- Next, we convert all characters to lower case letters: this is accomplished by the `map()` transformation, which is a one-to-one transformation:

```

# convert each element of RDD to a lowercase
.map(lambda x: x.lower())

```

- Next, we use a `flatMap()` transformation (which is a one-to-many transformation) to convert each element (representing a single record) into a sequence of target elements (representing a word). The `flatMap()` transformation returns a new RDD by first applying a function (here the `split(",")`) to all elements of the source RDD, and then flattening the results. This is done as:

```
# split each record into a list of words
.flatMap(lambda x: x.split(","))
```

- Finally, we drop non-needed word elements, where the length of elements are less than or equal 2. The following `filter()` keep words if and only if the length of word is greater than 2.

```
# keep words if its length is greater than 2
.filter(lambda x: len(x) > 2)
```

As you can observe: Spark transformations are distributed, parallel, high-level, powerful, and simple.

Spark's ecosystem

Spark's eco-system is presented in Figure 1.4 which has three main components:

- **Environments:**
can run anywhere and integrate well with other environments
- **Applications:**
it integrates well with big data platforms and applications
- **Data Sources:**
can read/write data from/to many data sources

Open Source Ecosystem

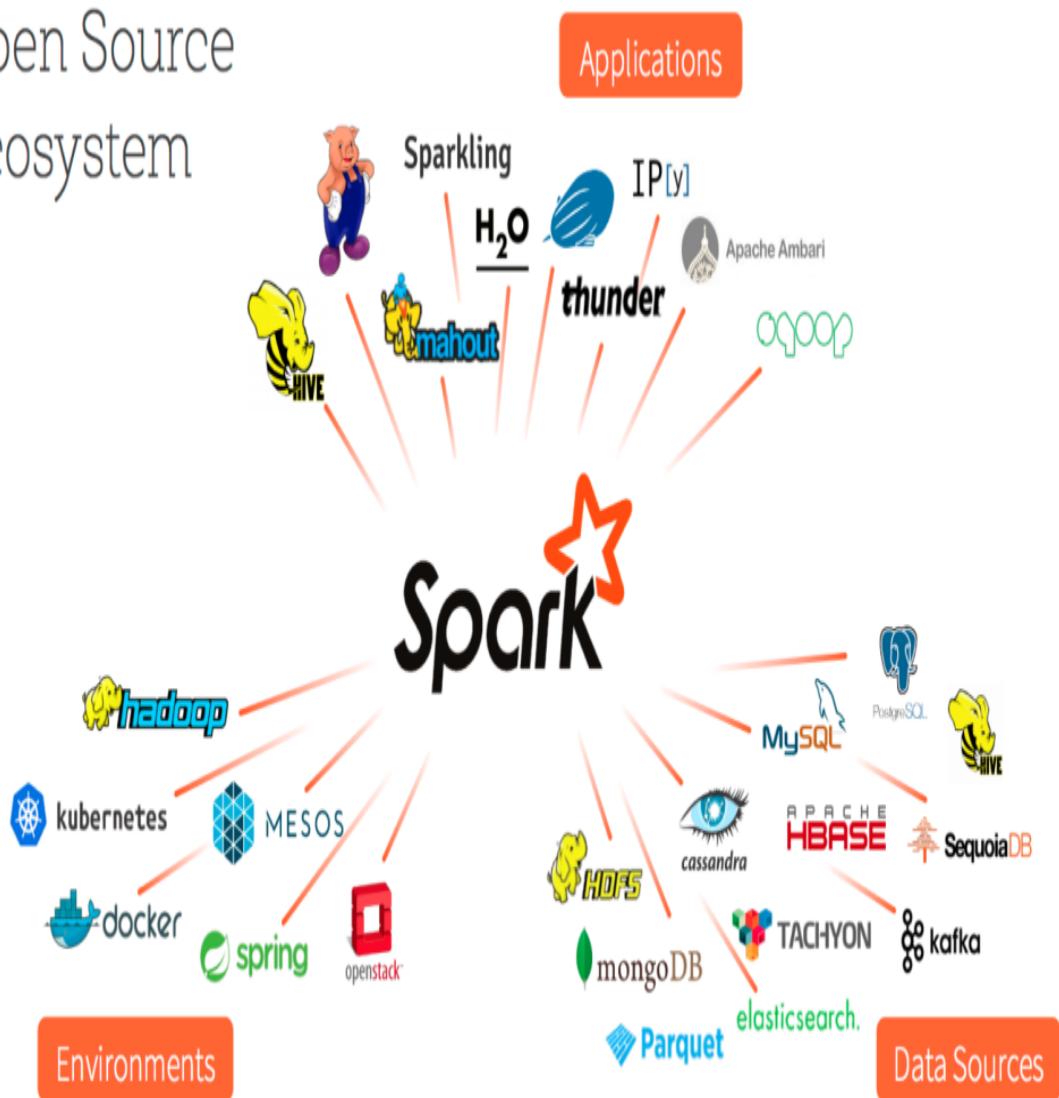


Figure 1-3. Spark Eco-System (source Databricks)

What can I do with PySpark? Spark's expansive echo system makes PySpark as a great tool for ETL, data analysis, and much more. With PySpark, you can read a ton of data from many different data sources (Linux file system, Amazon S3, Hadoop distributed file system, relational tables, MongoDB, ElasticSearch, Parquet files, ...) and represent your data as an Spark data abstraction. Once your data is in Spark data abstraction (such as RDDs and DataFrames — to be discussed shortly) then you can use a series of simple and powerful Spark transformations to transform your data into your desired content and format. For example, you may use the `filter()` transformation

to drop unwanted records, use `groupByKey()` to group your data by your desired key, and finally use the `mapValues()` transformation to perform final aggregation (such as finding average, median, and standard deviation of numbers) on the grouped data. All of these transformations are very possible by using the simple but powerful PySpark API.

Spark Architecture

Spark is an open-source cluster computing tool for data-intensive computations, managing and coordinating the execution of tasks on data across a cluster of computers. When you have small data, it is possible to analyze it with a single computer in a reasonable amount of time. When data is huge, using a single computer to store and analyze that data might take a long time and might be even impossible to analyze and process it. For big data, we may use Spark. Spark is a tool for analyzing, processing, managing and coordinating the execution of tasks on big data across a cluster of computers.

A high-level Spark architecture is presented by the Figure 1.7. Informally, a Spark cluster is comprised of a master node (denoted by a “cluster manager”) and a set of “worker” nodes, which are responsible for executing tasks submitted by the Spark application (your application which you want to run on the Spark cluster) and the cluster manager (which is responsible in managing your application).

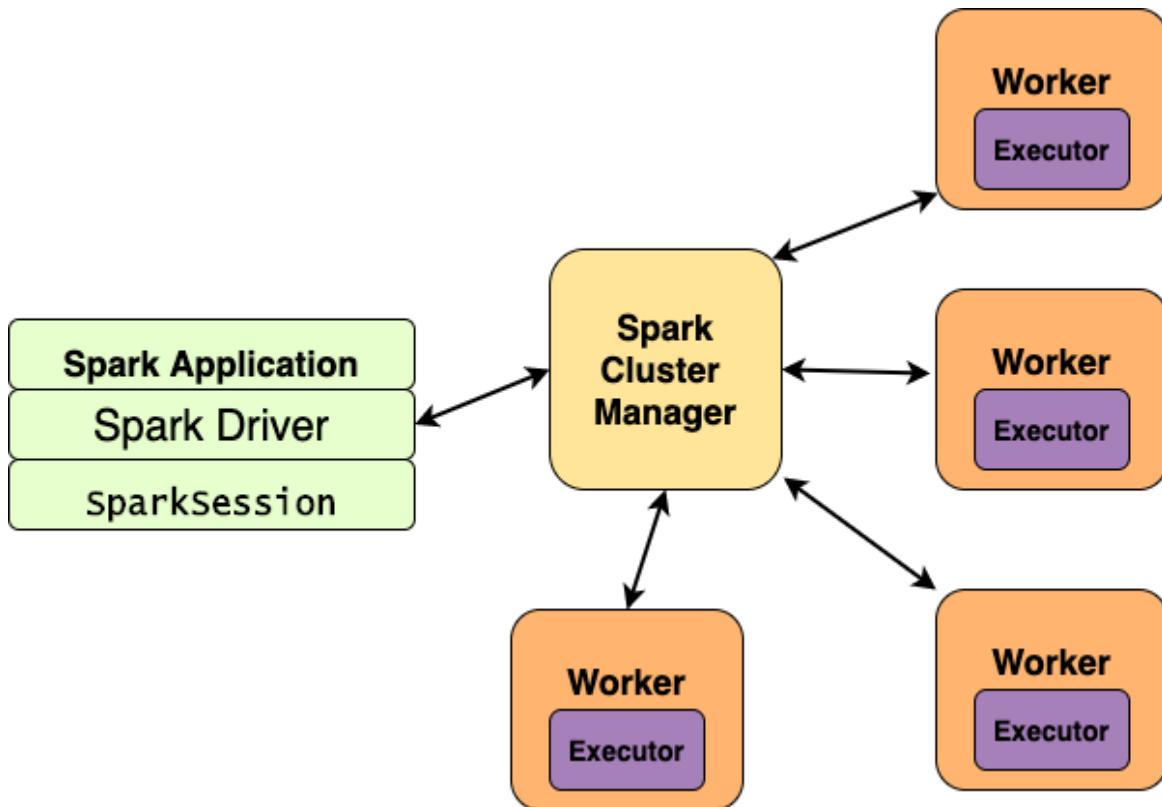


Figure 1-4. Spark Architecture

Spark uses the cluster of servers (comprised of a master node and a set of worker nodes) to analyze your big data. The cluster of servers that Spark will use to execute tasks will be managed by a cluster manager like Spark's Standalone cluster manager, Hadoop YARN, or Mesos. When Spark cluster is ready and running, then we can submit Spark applications to these cluster managers which will grant resources to our application so that we can complete our data analysis.

Spark has a core library and a set of built-in libraries (SQL, GraphX, Streaming, MLlib). This relationship is illustrated by the Figure 1.8. As you can observe, Spark can interact (using Data Access API) with many data sources such as Hadoop, HBase, Amazon S3, ElasticSearch, and MySQL to mention a few.

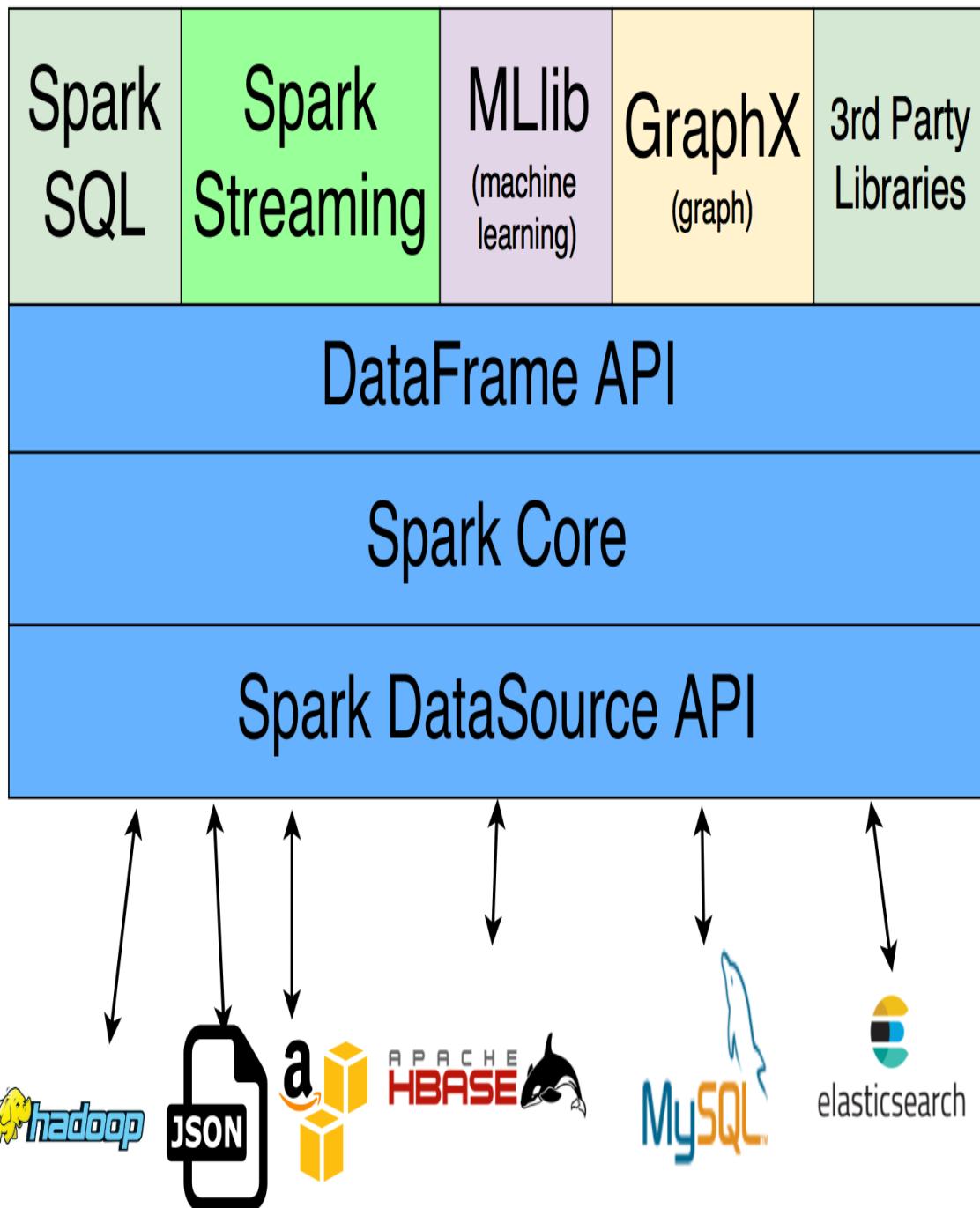


Figure 1-5. Spark Libraries

Figure 1.8 shows the real power of Spark: You may use several languages (Python, Scala, Java, R) to write your Spark applications and then you may use rich libraries (such as Spark SQL, Streaming, machine library, GraphX, GraphFrames, ...) for solving any big data problems and meanwhile you may read/write data from any data sources.

In this book, you'll interface Spark with Python through PySpark, the Spark Python API that exposes the Spark programming model to Python. **PySpark** has a comprehensive API (comprised of packages, modules, classes, and methods) to access Spark API. It is important to note that, for this book, all Spark API, packages, modules, classes, and methods are PySpark specific. For example, when I refer to the `SparkContext` class, I am referring to the `pyspark.SparkContext` (Python class, defined in the `pyspark` package) and when referring to the `SparkSession`, I am referring to the `pyspark.sql.SparkSession` (Python class, in `pyspark.sql` module).

To understand PySpark better, we do need to understand Spark, since PySpark is a Python API for Spark. The Spark architecture is based on a master/worker architecture. Typically, a Spark cluster has a “master” (controller and manager) node and a set of “worker” (executor) nodes. The number of worker nodes can be from 1 to tens, hundreds, and thousands (based on the need of your business and project requirements). You may also, run Spark on a standalone server (such as MacBook, Linux, or Windows — typically, for production environment, Spark is run on cluster of Linux servers). To run a Spark program, you need to have an access to a Spark cluster (comprised of one or many nodes) and have a “driver program”, which talks to a single coordinator called master (also called a “cluster manager”) that manages workers in which executors run. For this book, all driver programs will be in PySpark.

To understand Spark architecture, you'll need to understand the following two classes: `SparkSession` and `SparkContext`. In order to do something useful with the Spark cluster, the first thing we have to do is to create an instance of `SparkSession` and from the `SparkSession` we can access the

`SparkContext` object as an attribute. Once you create an instance of a ``SparkSession``, then `SparkContext` becomes available inside your `SparkSession`, in other words, `SparkSession` contains an instance of a `SparkContext` as `SparkSession.sparkContext`.

PySpark defines `SparkSession` as:

```
pyspark.sql.SparkSession (Python class, in pyspark.sql module)
class pyspark.sql.SparkSession(sparkContext, jsparkSession=None)
```

`SparkSession`: the entry point to programming Spark
with the RDD and DataFrame API.

When you start a PySpark shell, you automatically get an instance of a `SparkSession` (denoted by the `spark` variable). If you write a self-contained PySpark application (a Python driver, which uses PySpark API), then you have to explicitly create an instance of a `SparkSession` by using the “builder pattern” (see Listing 1.1). `SparkSession` can be used to

- Create RDDs (a main data abstraction in Spark)
- Create DataFrames (a high-level data abstraction in Spark, which will be explained shortly)
- Register DataFrames as tables
- Execute SQL over tables, cache tables
- Read text, CSV, JSON, and parquet files

To create a `SparkSession` in Python, use the following builder pattern (Listing 1.1). Note that when we use PySpark shell — an interactive tool for running PySpark — an instance of `SparkSession` is automatically created and provided as the `spark` variable — no need to create the `SparkSession` inside the PySpark shell, it will be already created on your behalf.

```
# import required Spark class
from pyspark.sql import SparkSession ①
```

```

# create an instance of SparkSession as spark
spark = SparkSession.builder \ ②
    .master("local") \
    .appName("my-application-name") \
    .config("spark.some.config.option", "some-value") \ ③
    .getOrCreate() ④

# to debug SparkSession
print(spark) ⑤

# create a reference to SparkContext as sc
# SparkContext is used to create new RDDs
sc = spark.sparkContext ⑥

# to debug SparkContext
print(sc)

```

- ❶ Import the `SparkSession` class from the `pyspark.sql` module
- ❷ A class attribute having a Builder to construct `SparkSession` instances
- ❸ Sets a `config` option. Options set using this method are automatically propagated to both `SparkConf` and `SparkSession`'s own configuration. When creating a `SparkSession` object, you can define any number of `config(key, value)` options.
- ❹ Gets an existing `SparkSession` or, if there is no existing one, creates a new one based on the options set in this builder
- ❺ For debugging purposes only
- ❻ `SparkContext` can be referenced from an instance of a `SparkSession`

PySpark defines `SparkContext` (a class defined in the `pyspark` module) as:

```
class pyspark.SparkContext(master=None, appName=None, ...)
```

`SparkContext`: the main entry point for Spark functionality.
 A `SparkContext` represents the connection to a Spark cluster,
 and can be used to create `RDD` (the main data abstraction for

Spark) and broadcast variables (such as collections and data structures) on that cluster.

SparkContext is the main entry point for Spark functionality. A shell (such as PySpark shell) or PySpark driver program cannot create more than one instance of SparkContext. A SparkContext represents the connection to a Spark cluster, and can be used to create new RDDs and broadcast variables (shared data structures and collections — kind of a read-only global variables) on that cluster. Spark's RDDs and DataFrames are used to represent your big data and transform them to your desired state. The Figure 1.9 shows SparkContext usage for creation of a new RDD from an input text file (labeled as records_rdd) and then transforms it into another RDD by using the flatMap() transformation (labeled as words_rdd).

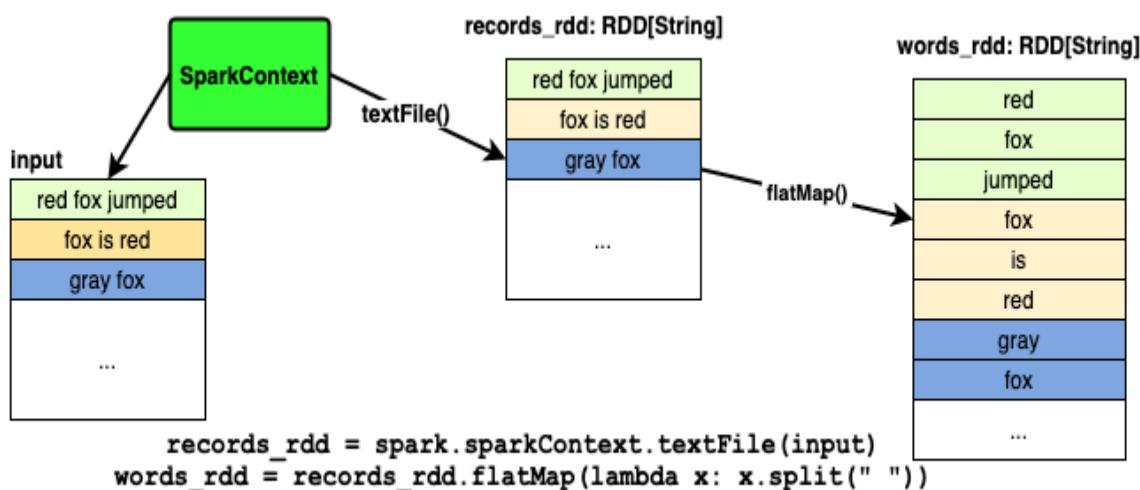


Figure 1-6. Creation of RDDs by SparkContext

To create a SparkContext object, use the following pattern:

```
# create an instance of SparkSession
spark_session = SparkSession.builder...

# use SparkSession to access SparkContext
spark_context = spark_session.sparkContext
```

Spark Architecture is illustrated below.

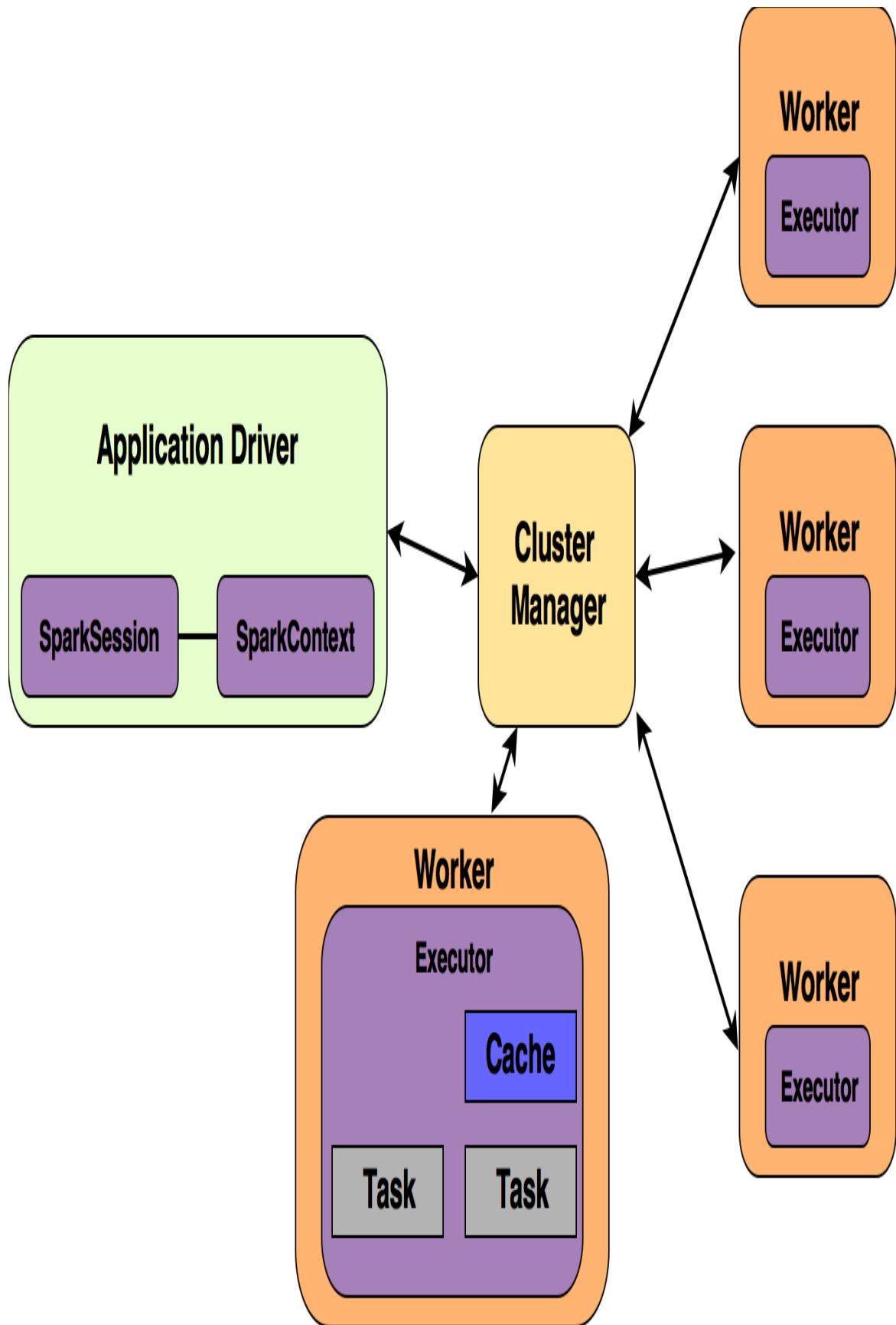


Figure 1-7. Spark Architecture

The following key terms used in Spark architecture:

- **SparkSession:**

It is a class defined in the `pyspark.sql` package. This class is the entry point to programming Spark with the `DataSet` and `DataFrame` API. From this class, you may access to an instance of `SparkContext`. A `SparkSession` can be used create `DataFrame`, register `DataFrame` as tables, execute SQL over tables, cache tables, and read parquet files.

- **SparkContext:**

It is a class defined in the `pyspark` package and represents the connection to a Spark cluster. It is the main entry point for Spark functionality. It holds a connection with Spark “cluster manager” (also known as a master node) and can be used to create `RDD` and broadcast variables on that cluster. All Spark applications (including PySpark shell and standalone Python programs) run as an independent set of processes. These processes are coordinated by a `SparkContext` in a driver program.

- **Driver:**

To submit a standalone Python program to Spark, you need to write a driver program with PySpark API (or Java or Scala). A driver program is in charge of the process of running the `main()` function of an application and creating the `SparkContext`. Also, the driver program is in charge of creating Spark’s `RDDs` and `DataFrames`.

- **Worker:**

In Spark cluster environment, there are two type of nodes: one (or two — for high availability) master and a set of workers. A worker, is any node that can run program in the cluster. If a process is launched for an application, then this application acquires executors at worker nodes, which carry out executing Spark’s application tasks.

- **Cluster Manager:**

The “master” node is known as the “cluster manager”. The main function of a Cluster Manager is to manage the cluster environment. The cluster of servers that Spark will leverage to execute tasks will be managed by a cluster manager. Cluster manager allocates resources to each application in driver program. Spark runs everywhere: there are 5 types of cluster managers supported by Spark:

1. Standalone: this is Spark’s own built-in clustered environment
2. Mesos: a distributed systems kernel
3. Hadoop YARN
4. Kubernetes
5. Amazon EC2

Spark Usage

Is Spark used by many notable companies? Here are some examples of how big companies use Spark.

- FaceBook processes 60 TB of data on a daily basis. Spark and MapReduce are at the heart of their algorithms and processing production data
- Viacom, with its 170 cable, broadcast, and online networks in around 160 countries, is transforming itself into a data-driven enterprise — collecting and analyzing petabytes of network data to increase viewer loyalty and revenue.
- Illumina¹ ingests thousands of genomes (big data — by big data we mean that data can not fit and processed in one server) using Spark, PySpark, MapReduce, and distributed algorithms.

- IBM and many social network and search engine companies (such as Google, Twitter and Facebook) use Spark, MapReduce, and distributed algorithms on a daily basis to scale out their computations and operations.

Spark in a Nutshell

There are four main reasons (as noted on Spark's website) to choose Spark:

1. Speed

Spark programs can run up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing (using memory is much faster than using Disk for intensive calculations and I/O). DAG (Directed Acyclic Graph) in Spark is a set of Nodes and Edges, where Nodes represent the RDDs and the edges represent the Operation (transformation or action) to be applied on RDD. On the calling of an Action, the created DAG submits to DAG Scheduler which further splits the graph into the stages of the task. Spark's DAG Visualization is presented as Figure 1.2.



DAG Visualisation

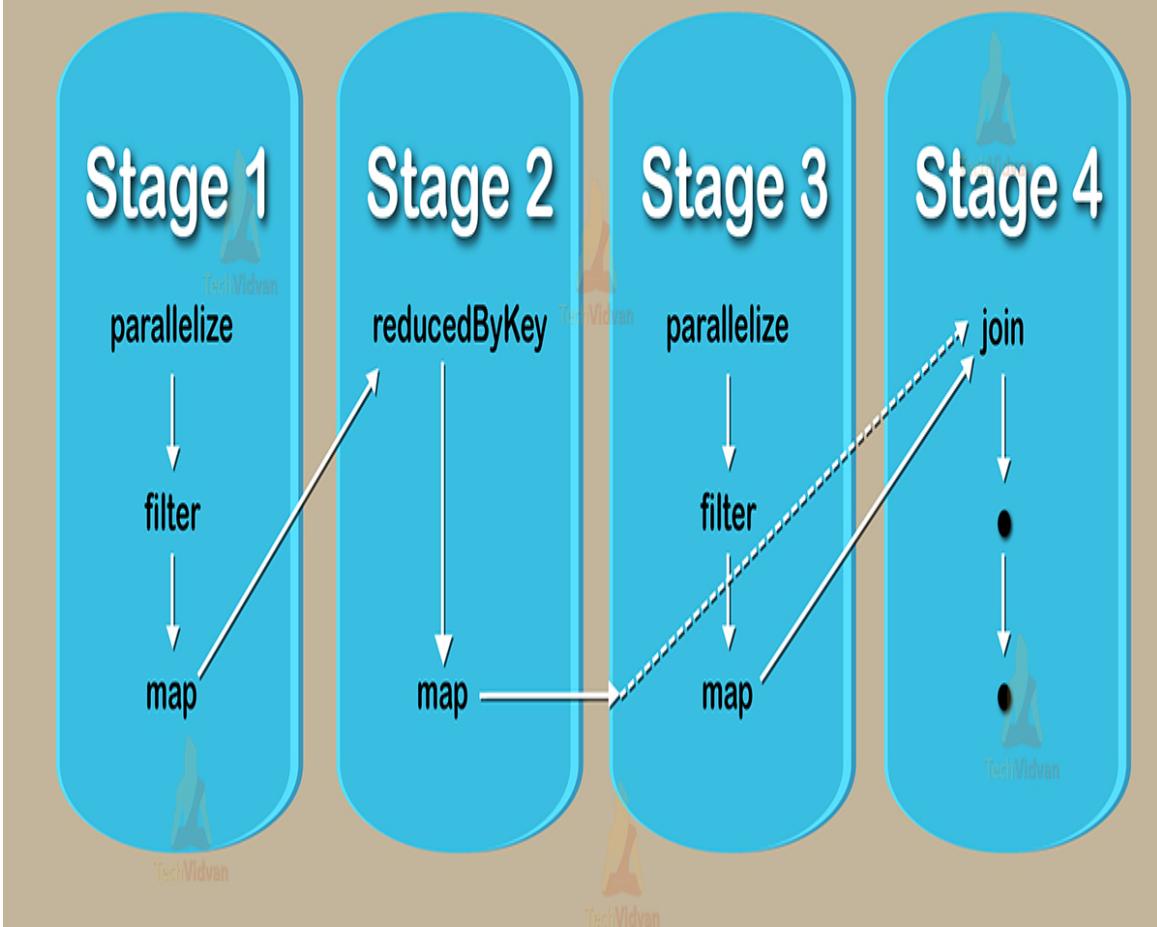


Figure 1-8. Spark DAG Visualization

1. Ease of Use

You can write Spark applications quickly in Java, Scala, Python, R, and SQL.

2. Generality

Spark is a general compute engine and it can be used for solving any type of problems. Spark provides combination of SQL, streaming, and complex analytics. Spark powers a set of libraries including

SQL and DataFrames, MLlib for machine learning, GraphX, and Spark Streaming. You can combine all of these libraries seamlessly in the same application. Also, there are a lot of external libraries for Spark, which can make data and graph processing (such as [GraphFrames](#)) like a breeze.

3. Runs Everywhere

- Spark runs on Hadoop, Mesos, standalone, or in the cloud (such as Amazon Glue and Google Cloud). Spark stack is presented by [Figure 1-9](#)
- It can access diverse data sources including Text files, relational databases, HDFS, Cassandra, HBase, and S3.

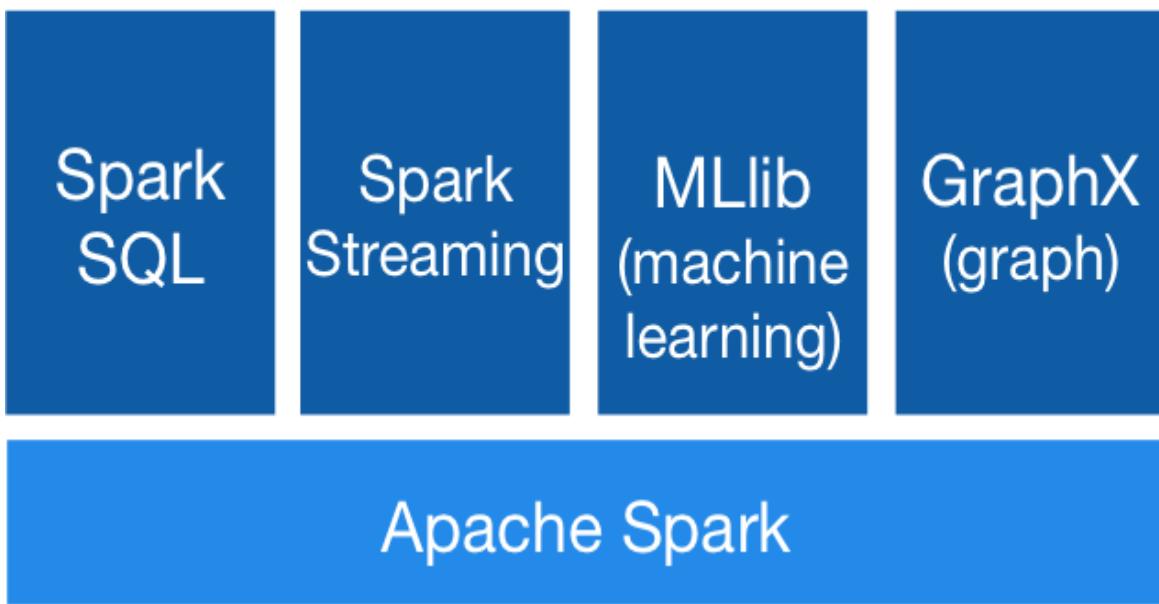


Figure 1-9. Spark Stack

What is PySpark?

In a nutshell, PySpark is the collaboration of **Apache Spark** and **Python Programming Language**.



Figure 1-10. What is PySpark

Python is a popular programming language that you can use to work quickly and integrate systems more effectively. PySpark is a Python API for Spark. PySpark usage has increased in the past few years (see Figure 1.4). Since PySpark applications and analytics run on Spark clusters, we need to understand the basics of Spark echo system.

SQL (Structured Query Language) is the most widely used query language used to communicate with relational databases. Spark enables us to represent structured data as a DataFrame (a table with named columns) and then query data by using SQL—SQL queries are converted into a set of `map()`, `filter()`, `groupByKey()` ... transformations, and then executed by the Spark engine.

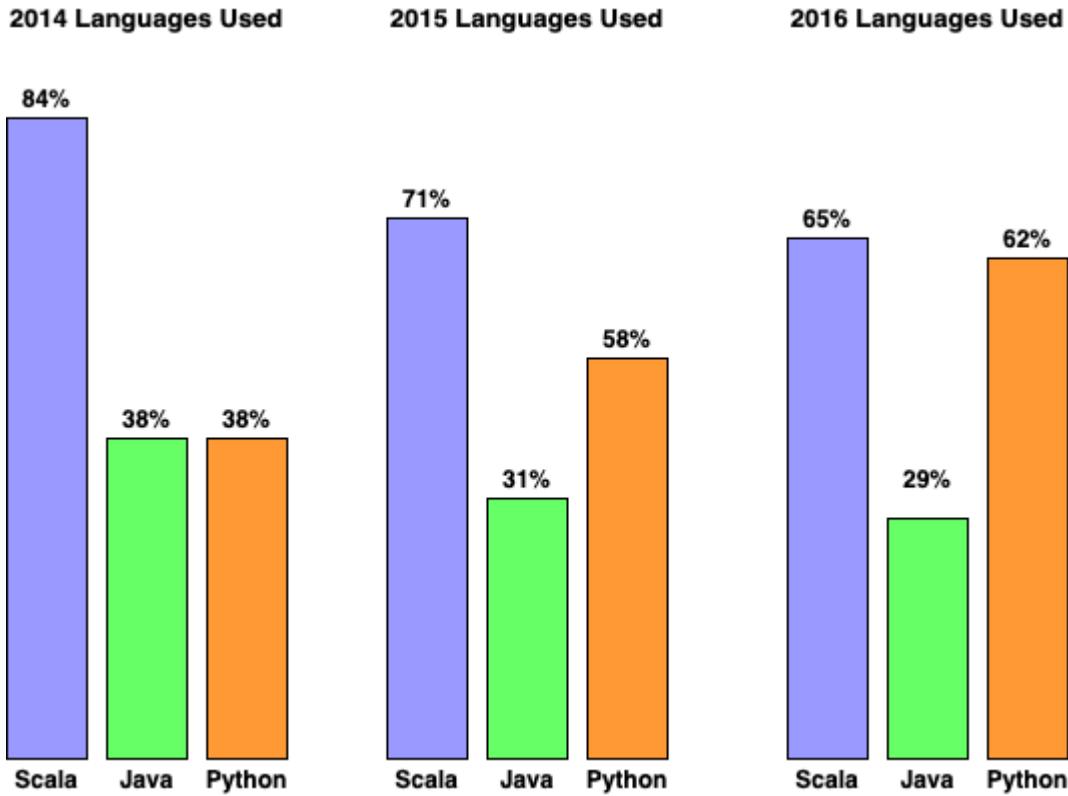


Figure 1-11. Languages Used for Developing Spark Applications

Python for PySpark

Python is simple and powerful programming language and integrates very well with Spark analytics engine. Overall, no general programming language alone can handle big data processing efficiently. There is always a need for a distributed computing framework like Spark, which is the defacto standard for big data analysis. PySpark is the Python programming language bindings (Python API) for Spark. Spark is an open-source distributed processing system commonly used for big data workloads and intensive computations. Since most data scientists already know Python, PySpark eases their ability to write short but concise code for distributed computing using Spark. I am assuming that you have access to a Spark cluster or you have already installed it on your laptop. In a nutshell, PySpark is an all-in-one ecosystem which can handle the complex data requirements with its RDDs, DataFrames, GraphFrames, MLlib, SQL, and Structured data processing API.

Power of PySpark

I'll show the amazing power of PySpark with a simple example. Let's say that we have lots of data records measuring the URL visits by users (collected by a search engine from many web servers) in the following format:

```
<URL-address><,><frequency>
```

Sample records are given here:

```
http://mapreduce4hackers.com,19779  
http://mapreduce4hackers.com,31230  
http://mapreduce4hackers.com,15708  
...  
https://www.illumina.com,87000  
https://www.illumina.com,58086  
...
```

Let's say we want to find out average, median, and the standard deviation of visiting numbers per key (as a URL-address). Assume that another requirement is that to drop records, if the length of any record is less than 5 (may be as a malformed URL). It is easy to express an elegant solution for this in PySpark. This workflow can be expressed by Figure 1.6.

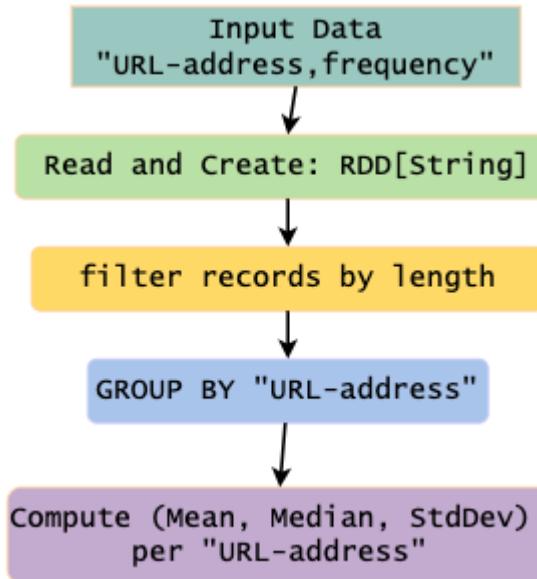


Figure 1-12. Simple Workflow to Compute Mean, Median, Standard Deviation

First, let's create some basic Python functions, which will help us in solving our simple problem. since we are going to work with (key, value) pairs, the first function, `create_pair()` accepts a single record of <URL-address><,><frequency> and returns a (key, value) as a pair (which will enable us to do “group by” on the key field later on), where key is a URL-address and value is the associated frequency.

The `create_pair()` function is illustrated below:

```

# create a pair of (URL-address, frequency)
# where URL-address is a key and frequency is a value
# record: <url-address><,><frequency>
def create_pair(record): ①
    tokens = record.split(',') ②
    url_address = tokens[0]
    frequency = tokens[1]
    return (url_address, frequency) ③
#end-def

```

- ① Accept a record of the form “URL-address,frequency”
- ② Tokenize input record, `tokens[0]`: URL-address as a key, `tokens[1]`: frequency as a value

- ③ Return a pair of (URL-address, frequency)

The next function accepts a list of frequencies (as numbers) and computes three values: average, median, and standard deviation:

```
# Compute average, median, and standard
# deviation for a given set of numbers
import statistics ①
# frequencies = [number1, number2, ...]
def compute_stats(frequencies): ②
    average = statistics.mean(frequencies) ③
    median = statistics.median(frequencies) ④
    standard_deviation = statistics.stdev(frequencies) ⑤
    return (average, median, standard_deviation) ⑥
#end-def
```

- ① This module provides functions for calculating mathematical statistics of numeric data
- ② Accept a list of frequencies
- ③ Compute average of frequencies
- ④ Compute median of frequencies
- ⑤ Compute standard deviation of frequencies
- ⑥ Return the result as a triplet

Next, we show the amazing power of PySpark in just few lines of code using Spark transformations and our custom Python defined functions:

```
# input_path = "s3://<bucket>/key"
input_path = "/tmp/myinput.txt"
results = spark ①
    .sparkContext ②
    .textFile(input_path) ③
    .filter(lambda record: len(record) > 5) ④
    .map(create_pair) ⑤
```

```
.groupByKey() ⑥  
.mapValues(compute_stats) ⑦
```

- ❶ `spark` denotes an instance of a `SparkSession``, the entry point to programming Spark
- ❷ `sparkContext` (is an attribute of `SparkSession`), main entry point for Spark functionality
- ❸ Read data as distributed set of string records (creates an `RDD[String]`)
- ❹ Drop out records, where record size is less than or equal 5 (keep records if their length is greater than 5)
- ❺ Create (URL-address, frequency) pairs from a given input record
- ❻ Group your data by keys (each key — as a URL-address — will be associated with a list of frequencies)
- ❼ Apply the `compute_stats()` function to list of frequencies

The results will represent a set of (key, value) pairs as:

```
(URL-address, (average, median, standard_deviation))
```

where `URL-address` is a key and `(average, median, standard_deviation)` is the value.

The most important thing about Spark is that it maximizes concurrency of functions and operations by means of partitioning data. For example if your input data has 600 billion rows and you are using a cluster of 10 nodes, your input data will be partitioned in to N (> 1) chunks and then chunks are processed independently and in parallel. For example, if $N = 20,000$ (the number of chunks or partitions), then each chunk will have about 30 million records/elements where:

$$600,000,000,000 = 20,000 \times 30,000,000$$

For example, if you have a big cluster then all of your 20,000 chunks (called a partition) might be processed in one shot. But, If you have a smaller cluster, then maybe every 100 chunks can be processed independently and in parallel. This process will continue until all of your 20,000 partitions are processed. Spark is an analytic and compute engine for parallel processing of data on a cluster. Parallelism in Spark allows developers to perform tasks on hundreds of computer servers in a cluster in parallel and independently. Spark's data abstractions (RDD and DataFrame) operate on partitioned data. Therefore, we can say that partition is the main unit of parallelism in Spark.

PySpark Architecture

The PySpark API is defined here: [PySpark Documentation](#). Spark has APIs for Java, Python (PySpark), Scala, and R. Spark is built for cluster computing: having a “master” server and a set of “worker” servers working together to enable big data computing. Typically, a Spark-based distributed algorithm will run on a cluster of connected computers. Spark, MapReduce paradigm, and distributed computing enables high performance computing by using a set of commodity servers.

PySpark is built on top of Spark’s Java API. Data is processed in Python and cached / shuffled in the Java Virtual Machine (JVM). I will cover “shuffling” concept in chapter 2. PySpark’s high-level architecture is presented by the Figure 1.11.

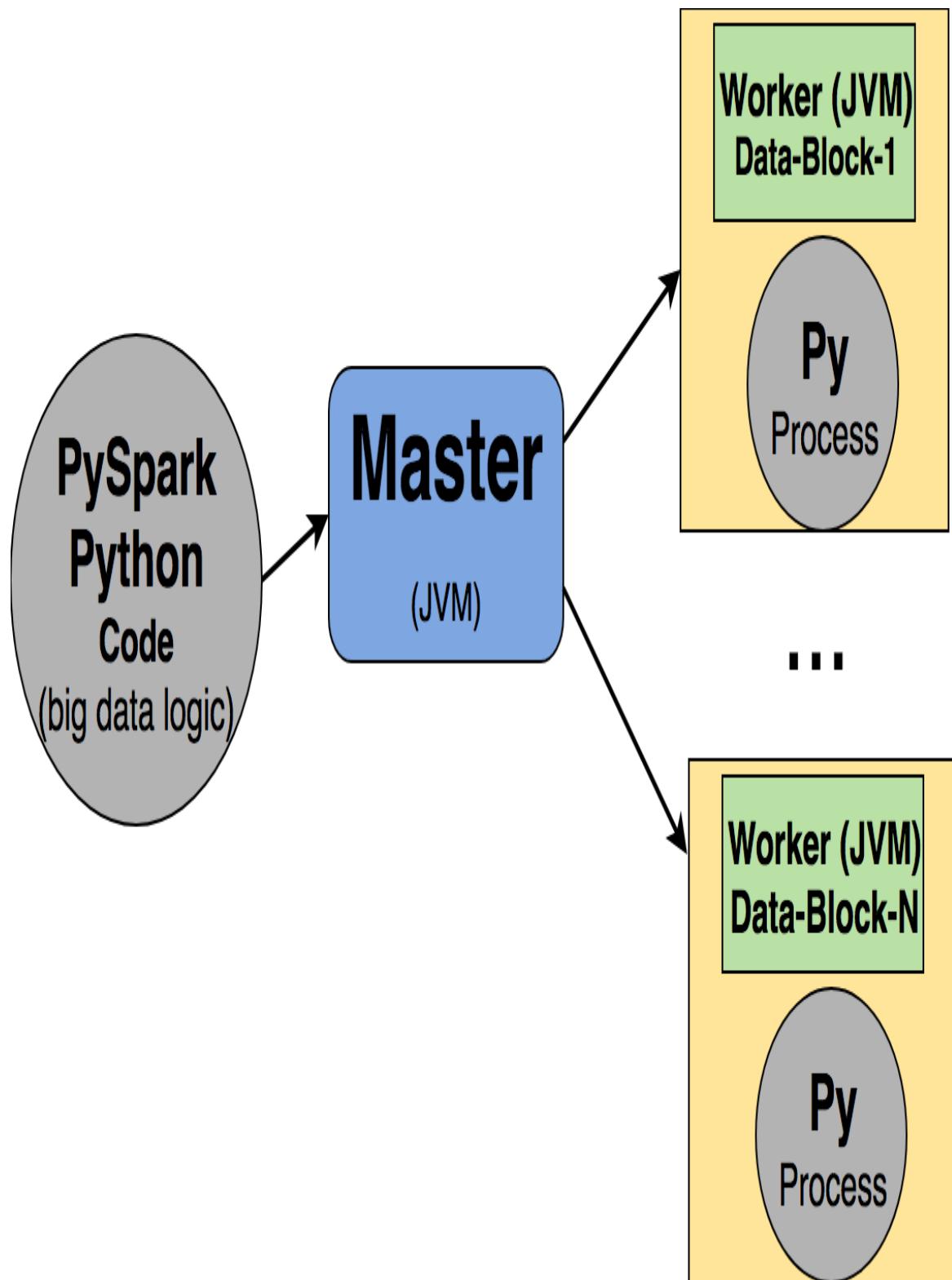


Figure 1-13. PySpark High-Level Architecture

PySpark's High-Level Architecture and data flow is illustrated by the Figure 1.12.

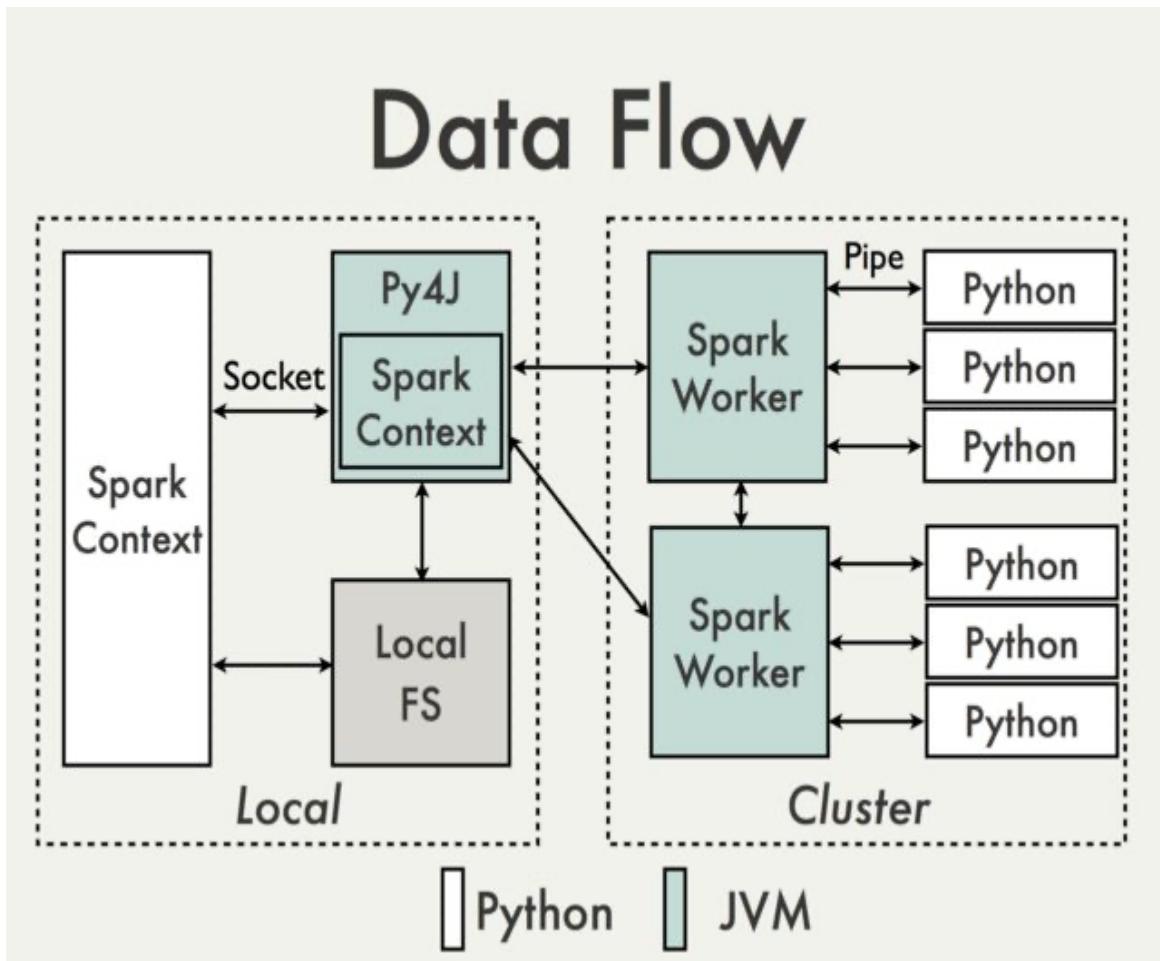


Figure 1-14. PySpark High-Level Architecture and Data Flow

According to the Spark documentation: In the Python driver program (which is your Spark application in Python), `SparkContext` uses **Py4J software — A Bridge between Python and Java** to launch a JVM and create a `JavaSparkContext`. Py4J is only used on the driver for local communication between the Python and Java `SparkContext` objects; large data transfers are performed through a different mechanism. RDD transformations in Python are mapped to transformations on `PythonRDD` objects in Java. On remote worker machines, `PythonRDD` objects launch Python subprocesses and communicate with them using pipes, sending the user's code and the data to be processed.

SparkContext uses **Py4J** software, which enables Python programs running in a Python interpreter to dynamically access Java objects in a JVM. Methods are called as if the Java objects resided in the Python interpreter and Java collections can be accessed through standard Python collection methods. Py4J also enables Java programs to call back Python objects.

Spark Data Abstractions

To manipulate data in Python programming language, you use integers, strings, lists, and dictionaries. To manipulate and analyze data in Spark, you have to represent it as a Spark “data set”. Spark supports 3 types of data set abstractions:

- **RDD** (Resilient Distributed Dataset):
 - Low level API
 - Denoted by `RDD[T]` (each element has type `T`)
- **DataFrame** (similar to relational tables):
 - High level API
 - Denoted by `DataFrame(column_name_1, column_name_2, ...)`
- **Dataset** (similar to relational tables):
 - High level API (not available in PySpark)

Dataset data abstraction is used in a very strongly typed languages such as Java and not supported in PySpark. RDDs and DataFrames will be discussed in detail in the following chapters.

RDD Example

Basically, RDD represents your data as a collection of elements. In a nutshell, RDD is an immutable set of distributed elements of type `T`, denoted

as $\text{RDD}[\text{T}]$.

Table 1.1 denotes 3 simple RDDs:

- $\text{RDD}[\text{Integer}]$: each element is an Integer
- $\text{RDD}[\text{String}]$: each element is a String
- $\text{RDD}[(\text{String}, \text{Integer})]$: each element is a pair of (String , Integer)

T
a
b
l
e

I
-
I
.
S
i
m
p
l
e

R
D
D
s

RDD[Integer]	RDD[String]	RDD[(String, Integer)]
2	"abc"	('A', 4)
-730	"fox is red"	('B', 7)
320	"Python is cool"	('ZZ', 9)
...

Table 1.2 denotes an RDD, where each element is a pair of (key, value), where key is a String and value is a triplet of (Integer, Integer, Double).

T
a
b
l
e

l
-

2

.

R
D
D
s
o
f
D
i
f
f
e
r
e
n
t
E
l
e
m
e
n
t
T
y

```
p  
e  
s  
(  
C  
o  
m  
p  
l  
e  
x
```

```
R  
D  
D  
)
```

RDD[(String, (Integer, Integer, Double))]

```
("cat", (20, 40, 1.8))
```

```
("cat", (30, 10, 3.9))
```

```
("lion king", (27, 32, 4.5))
```

```
("python is fun", (2, 3, 0.6))
```

```
...
```

DataFrame Example

Similar to an RDD, Spark's DataFrame is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Spark's DataFrame is designed to make large data sets processing even easier by using named columns. DataFrame

allows programmers to impose a structure onto a distributed collection of data, allowing higher-level abstraction. Spark's DataFrame enables to process CSV and JSON files much easier than RDDs.

A DataFrame example is provided below: this DataFrame has 3 columns:

- `DataFrame[name, age, salary]`
 - `name: String`
 - `age: Integer`
 - `salary: Integer`

name	age	salary
bob	33	45000
jeff	44	78000
mary	40	67000
...

A DataFrame can be created from many different sources such as Hive tables, Structured Data files, external databases, or existing RDDs. The DataFrames API was designed for modern Big Data and data science applications taking inspiration from DataFrame in R Programming and Pandas in Python. As we will see in other chapters, we can use SQL queries against DataFrames.

Spark SQL comes with a set of DataFrame operations. The list of operations is powerful and quite extensive and includes:

- Aggregate functions (min, max, sum, average, ...)
- Collection functions
- Math functions
- Sorting functions

- String functions
- User defined functions (UDF)

For example, you can easily read a CSV files(s) and create a DataFrame:

```
# define input path
virus_input_path = "s3://mybucket/projects/cases/case.csv"

# read CSV file and create a DataFrame
cases_dataframe = spark.read.load(virus_input_path,format="csv",
    sep=",", inferSchema="true", header="true")

# show the first 3 rows of created dataframe
cases_dataframe.show(3)
+-----+-----+-----+-----+
|case_id|country|      city|infection_case|confirmed|
+-----+-----+-----+-----+
| C0001|    USA| New York|       contact|     175|
+-----+-----+-----+-----+
| C0008|    USA| New Jersey|      unknown|      25|
+-----+-----+-----+-----+
| C0009|    USA| Cupertino|       contact|     100|
+-----+-----+-----+-----+
```

To see the most cases at the top (use descending Sort), we use the `sort()` function:

```
# We can do this using the F.desc function:
from pyspark.sql import functions as F
cases_dataframe.sort(F.desc("confirmed")).show()
+-----+-----+-----+-----+
|case_id|country|      city|infection_case|confirmed|
+-----+-----+-----+-----+
| C0001|    USA| New York|       contact|     175|
+-----+-----+-----+-----+
| C0009|    USA| Cupertino|       contact|     100|
+-----+-----+-----+-----+
| C0008|    USA| New Jersey|      unknown|      25|
+-----+-----+-----+-----+
```

You can easily filter rows:

```

cases_dataframe.filter((cases_dataframe.confirmed > 100) &
                      (cases_dataframe.country == 'USA')).show()

+-----+-----+-----+-----+
|case_id|country|      city|infection_case|confirmed|
+-----+-----+-----+-----+
| C0001|    USA| New York|       contact|      175|
+-----+-----+-----+-----+
...

```

The power of Spark's DataFrame is expressed by an example: we will create a DataFrame and find average and sum of hours per dept of employees:

```

# Import required libraries
from pyspark.sql import SparkSession
from pyspark.sql.functions import avg, sum

# Create a DataFrame using SparkSession
spark = SparkSession.builder.appName("demo").getOrCreate()
dept_emps = [("Sales", "Barb", 40), ("Sales", "Dan", 20),
             ("IT", "Alex", 22), ("IT", "Jane", 24),
             ("HR", "Alex", 20), ("HR", "Mary", 30)]
df = spark.createDataFrame(dept_emps, ["dept", "name", "hours"])

# Group the same depts together, aggregate their hours, and compute an average
averages = df.groupBy("dept")
    .agg(avg("hours").alias('average'),
         sum("hours").alias('total'))

# Show the results of the final execution
averages.show()
+-----+-----+
| dept| average| total|
+-----+-----+
| Sales|    30.0|  60.0|
|   IT|    23.0|  46.0|
|   HR|    25.0|  50.0|
+-----+-----+

```

As you can observe, Spark's DataFrame is very powerful to manipulate billions of rows with simple but powerful functions.

Spark's Operations

Spark provides two types of operations on RDDs:

- **Transformations** (transforms source RDD to target RDD(s))
- **Actions** (transforms source RDD to non-RDD object)

Transformations

Chapter 3 is dedicated for understanding Spark's transformations by working examples. A transformation is a function which takes an existing RDD (source RDD), applies the transformation and creates a new RDD (target RDD). Spark RDDs are immutable. Once created, it cannot be edited, added or changed. Each Spark transformation creates a new RDD. RDDs are not evaluated until an action is performed on them: this means that transformations are lazily evaluated. If an RDD fails during a transformation, the data lineage of transformations rebuilds the RDD. Examples of Spark's transformations are: `map()`, `flatMap()`, `groupByKey()`, `reduceByKey()`, `filter()`, ...

Informally, we can express a transformation as:

transformation: `source_RDD[V] --> target_RDD[T]` ①

- ① Transform `source_RDD` of type `V` into `target_RDD` of type `T`

Therefore, a transformation accepts an RDD and may create one or more new RDDs. Most of the Spark's transformations creates a single target RDD. The relationship between RDDs, transformations, and actions are presented by the figure 1.14.

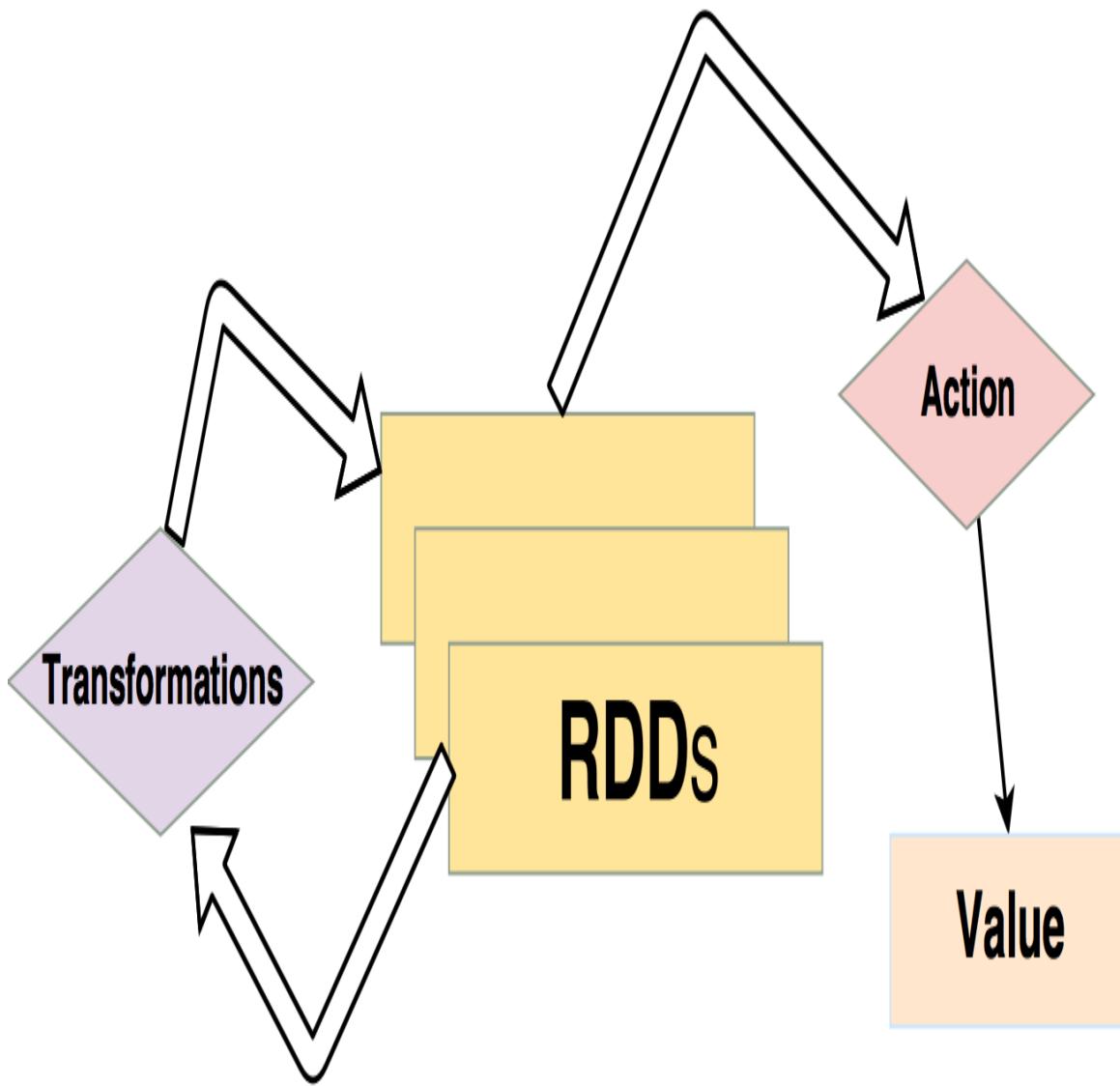


Figure 1-15. RDDs, Transformations, and Actions

I present a sequence of transformations below:

```

tuples = [('A', 7), ('A', 8), ('A', -4),
          ('B', 3), ('B', 9), ('B', -1),
          ('C', 1), ('C', 5)]
rdd = spark.sparkContext.parallelize(tuples)

# drop negative values
positives = rdd.filter(lambda x: x[1] > 0)
positives.collect()
[('A', 7), ('A', 8), ('B', 3), ('B', 9), ('C', 1), ('C', 5)]

# find sum and average per key using groupByKey()

```

```

sum_and_avg = positives.groupByKey()
    .mapValues(lambda v: (sum(v), float(sum(v))/len(v)))

# find sum and average per key using reduceByKey()
# 1. create (sum, count) per key
sum_count = positives.mapValues(lambda v: (v, 1))
# 2. aggregate (sum, count) per key
sum_count_agg = sum_count.reduceByKey(lambda x, y:
    (x[0]+y[0], x[1]+y[1]))
# 3. finalize sum and average per key
sum_and_avg = sum_count_agg.mapValues(
    lambda v: (v[0], float(v[0])/v[1]))

```

TIP

The `groupByKey()` which acts as a SQL's GROUP BY statement, groups the values for each key in the RDD into a single sequence. The `groupByKey()` transformation can cause out of disk problems as data is sent over the network of Spark servers and collected on the reduce workers. When the number values per key are in the thousands or millions, there might be an OOM error.

On the other hand using the `reduceByKey()` transformation, data are combined at each partition, only one output for one key at each partition to send over the network of Spark servers. The `reduceByKey()` merges the values for each key using an associative and commutative reduce function. The `reduceByKey()` transformation is required combining all values (per key) into another value with the exact same data type (this is a limitation, in which can be overcome by using `combineByKey()` transformation). Overall, the `reduceByKey()` is more scalable than the `groupByKey()`.

Actions

Spark actions are RDD operations or functions that produce non-RDD values. Actions may trigger a previously constructed, lazy RDDs to be evaluated. Informally, we can express an action as:

```
action: RDD => non-RDD value
```

Action output can be a tangible value: such as a saved file, a value such as integer, count of elements, list of values, dictionary, and so on.

The following are examples of actions:

- `reduce()`: apply a function to deliver a single value such as adding values for a given `RDD[Integer]`
- `collect()`: convert an `RDD[T]` into a list of type `T`
- `count()`: find the number of elements for a given `RDD`
- `saveAsTextFile()`: save `RDD` elements to a disk
- `saveAsMap()`: save `RDD[K,V]` elements to a disk as a `dict[K, V]`

Don't collect() on Large RDDs

For test and example programs in this book, I have often used the `RDD.collect()` action for testing, debugging, educational, and demonstration purposes. You should avoid using it on the production servers. Therefore, for production programs, avoid using `collect()` unless you really have a requirement for it. When a `collect()` operation is called on a `RDD`, the entire `RDD`'s dataset is copied to the driver program. If `RDD`'s dataset is too large, then a memory exception will be thrown (as it might not fit in memory). If the `RDD`'s dataset is too large to fit in memory then you should use `take()` or `takeSample()` actions instead of `collect()`. For example `RDD.take(N)` returns the first `N` elements of the `RDD` and `DataFrame.take(N)` returns the first `N` rows as a list of `Row` objects.

To summarize, `RDD.collect()` returns a list that contains all of the elements in this `RDD`. According to the Spark documentation: **this method (`collect()`) should only be used if the resulting array is expected to be small, as all the data is loaded into the driver's memory**. Using `collect()` on a large `RDD` might cause OOM exception.

TIP

If you have a requirement to manipulate all of your RDD elements, then rather than using the `collect()`, you may use some proper transformations such as `map()`, `filter()`, `flatMap()`, or `foreach(func)` to mention a few. To view some elements of your RDD, you may use `RDD.take(N)`, which returns N elements of RDD.

Using PySpark Shell Programming

There are at least two ways you can use PySpark:

- Use PySpark's shell (for testing and interactive programming)
- Use PySpark in a self-contained application: write a Python driver program (call it `my_pyspark_program.py`) using PySpark API and then run it by using the `spark-submit` command:

```
export SUBMIT=$SPARK_HOME/bin/spark-submit  
$SUBMIT my_pyspark_program.py <parameters>
```

where `<parameters>` is a list of parameters consumed by your PySpark (`my_pyspark_program.py`) program

NOTE

For details on using the `spark-submit` command, refer to the [Submitting Spark Applications](#).

To do something useful with your data, you can do it interactively by PySpark shell (Spark also provides a Scala shell as well) or wrap your Spark transformations and actions inside a Python program and submit a batch job using the “spark-submit” command. Spark provides an interactive shell for Python users called `$SPARK_HOME/bin/pyspark` – a powerful tool to analyze data interactively and see the results immediately. Spark’s primary data abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD). RDDs can be created from Hadoop Input

Formats (such as HDFS files), Linux file system, Amazon’s S3, collection data structures or by transforming one RDD into another one.

To open Spark Interactive Shell, let SPARK_HOME denotes the installation directory for the Spark. Then, the following command is used to open PySpark shell.

```
export SPARK_HOME=<spark-installed-directory>
$SPARK_HOME/bin/pyspark
```

Once PySpark shell launches successfully, PySpark displays some useful information. It displays information about the Python version it is using as well as the PySpark version. The >>> symbol is used as a PySpark shell prompt. PySpark shell prompt indicates that we can now write our Python or PySpark commands and view the results. The PySpark shell can work on both a single-machine installation and a cluster installation of Spark (pyspark output is trimmed to fit the page).

```
export SPARK_HOME="/home/spark" ❶
$SPARK_HOME/bin/pyspark ❷
Python 3.7.2 (default, Jul 7 2019, 00:08:15)

Welcome to Spark version 3.0.0
Using Python version 3.7.2 (default, Nov 7 2020 00:08:15)
SparkSession available as 'spark'.
>>>
```

- ❶ Define Spark’s installed directory
- ❷ Invoke PySpark shell

The PySpark shell prompts are denoted by “>>>” and Spark shell gives you an instance of a `SparkSession` as a “`spark`” object, which you may use to create new `DataFrames` and `RDDs` by using Spark’s transformations. Below, we execute PySpark commands to show you the power of Spark: Create simple `RDDs`: let us create a simple `RDD` from a data structure (note that for most of the real spark applications, you will read data from the text files

rather than collection data structures). The following are executed in PySpark shell.

Using PySpark Shell

The following steps show how to use PySpark in an interactive shell environment.

Step-1: Enter into PySpark Shell

To enter into a PySpark shell, we execute `pyspark` as:

- ① Executing `pyspark` will create a new shell, output is trimmed to fit the page
 - ② Verify that `SparkSession` is created as `spark`
 - ③ Create a `SparkContext` as `sc`
 - ④ Examine `SparkContext` as `sc`

Once you enter into PySpark shell, an instance of `SparkSession` is given as “spark” variable. The `SparkSession` is the entry point to programming Spark with the Dataset and DataFrame API. A `SparkSession` can be used to create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files. If you want to use PySpark in a self-contained application, then you have to explicitly create a `SparkSession`, with using the builder pattern provided in the preceding sections.

Once you have an access to an instance of a `SparkSession`, (say “spark”) then you can reference an instance of `SparkContext` as:

```
# spark : an instance of SparkSession
# sc : an instance of SparkContext
sc = spark.sparkContext
```

Step-2: Create RDD from Collection

Spark enable us to create new RDDs from files and collections (data structures such as lists). Below, we use `SparkContext.parallelize()` to create a new RDD from a collection (represented as data):

```
>>> data = [ ❶
    ("fox", 6), ("dog", 5), ("fox", 3), ("dog", 8),
    ("cat", 1), ("cat", 2), ("cat", 3), ("cat", 4)
]
>>>
>>> # create SparkContext as sc
>>> sc = spark.sparkContext

>>> # create an RDD as rdd
>>> rdd = sc.parallelize(data) ❷
>>> rdd.collect() ❸
[
    ('fox', 6), ('dog', 5), ('fox', 3), ('dog', 8),
    ('cat', 1), ('cat', 2), ('cat', 3), ('cat', 4)
]
>>> rdd.count() ❹
8
```

- ❶ Define your Collection

- ② Create a new RDD from a Collection
- ③ Display the content of new RDD
- ④ Count the number of new RDD elements

Step-3: Aggregate and Merge Values of Keys

The `reduceByKey()` transformation is used to merge and aggregate values. In the following transformation x and y refers to the values of the same key.

```
>>> sum = rdd.reduceByKey(lambda x, y : x+y) ❶
>>> sum.collect() ❷
[
  ('fox', 9),
  ('dog', 13),
  ('cat', 10)
]
>>>
```

- ❶ merge and aggregate values of the same key
- ❷ Collect the elements of RDD

According to the Spark documentation, the `reduceByKey()` transformation (in simplest form) is defined as:

```
pyspark.RDD.reduceByKey (Python method, in pyspark package)
reduceByKey(func)
```

The source RDD for this transformation has to be (key, value) pairs. The `reduceByKey()` merges the values for each key using an associative and commutative reduce function. This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a “combiner” in MapReduce. Output will be partitioned with `numPartitions` partitions, or the default parallelism level if `numPartitions` is not specified. Default partitioner is hash-partition.

Let T be the type of value for (key , value) pairs, then the `reduceByKey()`'s func() can be defined as:

```
# source_rdd : RDD[(K, T)]
# target_rdd : RDD[(K, T)]
target_rdd = source_rdd.reduceByKey(lambda x, y: func(x, y))
```

where

```
func(T, T) -> T
```

Then you may define `'func()'` in Python as:

```
# x: type of T
# y: type of T
def func(x, y):
    result = <aggregation of x and y: return a result of type T>
    return result
#end-def
```

This means that

- There are two input arguments (of the same type T) for the reducer `func()`
- The return type of `func()` must be the same as the input type T (this limitation can be avoided if you use the `combineByKey()` transformation)
- The reducer `func()` has to be associative: informally, a binary operation f() on a set T is called associative if it satisfies the associative law: the associative law can also be expressed in functional notation thus:

TIP

Associative law:
 $f(f(x, y), z) = f(x, f(y, z))$

Note that the Commutative Law works on addition (+) and multiplication (*), but does not work for either subtraction (-) or division (/).

- The reducer `func()` has to be commutative: informally, a function $f()$ for which $f(x, y) = f(y, x)$ for all values of x and y . For example, addition (+) and multiplication (*) are both commutative. On the other hand, subtraction (-), and division (/) functions are not. For example, $5 + 3 = 3 + 5$ but $5 - 3 \neq 3 - 5$. The commutative law can also be expressed in functional notation as:

TIP

Commutative law:
 $f(x, y) = f(y, x)$

Note that the Commutative Law works on addition (+) and multiplication (*), but does not work for either subtraction (-) or division (/).

Therefore, you may not use subtraction or division operations in `reduceByKey()` transformation.

Step-4: Filter RDD's Elements

Next, we use the `filter()` transformation to return a new RDD containing only the elements that satisfy a predicate.

```

>>> sum_filtered = sum.filter(lambda (k,v) : v > 9) ①
>>> sum_filtered.collect() ②
[
  ('cat', 10),
  ('dog', 13)
]
>>>

```

- ① keep the (key, value) pairs if value is greater than 9
- ② Collect the elements of RDD

The specified predicate keeps the (key, value) pairs, if value is > 9 . The filter() transformation is defined as:

```

pyspark.RDD.filter (Python method, in pyspark package)
filter(f)
Description: Return a new RDD containing only
the elements that satisfy a predicate.

```

Step-5: Group Similar Keys

Finally, we use the groupByKey() transformation, which group the values for each key in the RDD into a single sequence.

```

>>> grouped = rdd.groupByKey() ①
>>> grouped.collect() ②
[
  ('fox', <ResultIterable object at 0x10f45c790>), ③
  ('dog', <ResultIterable object at 0x10f45c810>),
  ('cat', <ResultIterable object at 0x10f45cd90>)
]
>>>
>>># list(v) converts v as a ResultIterable into a list
>>> grouped.map(lambda (k,v) : (k, list(v))).collect() ④
[
  ('fox', [6, 3]),
  ('dog', [5, 8]),
  ('cat', [1, 2, 3, 4])
]
>>>

```

- ① Group elements of the same key into a sequence of elements
- ② View the result
- ③ The full name of `ResultIterable` is
`pyspark.resultiterable.ResultIterable`
- ④ First apply `map()`, and then `collect()`, which returns a list that contains all of the elements in result RDD. The `list()` function converts `ResultIterable` into a list of objects.

In the simplest form, the `groupByKey()` transformation is defined as:

```
pyspark.RDD.groupByKey (Python method, in pyspark package)
groupByKey()
```

The source RDD for this transformation has to be (key, value) pairs. The `groupByKey()` groups the values for each key in the RDD into a single sequence. Hash-partitions the resulting RDD with `numPartitions` partitions. Note that if you are grouping (using the `groupByKey()` transformation) in order to perform an aggregation (such as a sum or average) over each key, using `reduceByKey()` or `aggregateByKey()` will provide much better performance.

Step-6: Aggregate Values for Similar Keys

To aggregate and sum up the values for each key, we may use the `groupValues()` transformation and the `sum()` function:

```
aggregated = grouped.mapValues(lambda values : sum(values)) ①
aggregated.collect() ②
[
  ('fox', 9),
  ('dog', 13),
  ('cat', 10)
]
```

- ① The `values` refer to a sequence of values per key. Pass each `value` in the `(key, value)` pair RDD through a map function (add all `values` by `sum(values)`) without changing the keys
- ② For debugging, return a list that contains all of the elements in this RDD

As you can observe, for aggregation and summing up values we have several choices: `reduceByKey()` and `groupByKey()` to mention a few. In general, the `reduceByKey()` transformation is efficient than the `groupByKey()` transformation. Details on this are explained on the “Spark Reductions” chapter.

Spark has many other powerful transformations, which can convert an RDD into a new RDD. RDDs are read-only, immutable, and distributed. RDD transformations returns pointer to new RDD and allows you to create dependencies between RDDs. Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD. Another fact about Spark: Spark is lazy, so nothing will be executed unless you call some action (such as `count()` or `collect()`) that will trigger job creation and execution.

Serverless Spark

If you don't have the budget to set up your own dedicated Spark cluster, then you can use either Amazon's AWS Glue or Databrick's Serverless Spark. Both these options remove the operational complexities for both big data and interactive data science.

For example, you can use Amazon's AWS Glue to create your PySpark job and then submit it to a dynamic cluster. The dynamic cluster means that you are submitting your job to a cluster without owning the cluster. You just pay for the compute time of your PySpark job.

Overall, Serverless Spark can reduce operational cost by using predefined pool of clusters provided by cloud services such as Amazon and Databricks.

It is expected that most of the cloud services will provide Serverless Spark as a service.

Data Analysis tools for PySpark

- **Jupyter**

Jupyter is a great tool to test and prototype programs. PySpark can also be used from Jupyter notebooks. Using Jupyter is very practical for explorative data analysis.

- **Apache Zeppelin**

Zeppelin is a web-based notebook that enables data-driven, interactive data analytics and collaborative documents with SQL, Python, Scala and more.

ETL Example

ETL stands for “extract, transform, and load.” In data analysis and computing, extract, transform, load (ETL) is the general procedure of copying data from one or more sources into a destination system which represents the data differently from the source(s) or in a different context than the source(s). Here I will show that how Spark makes ETL possible and an easy process.

For this ETL, I use a census data (census_2010.json) as a JSON format.

```
wc -l census_2010.json
101 census_2010.json

head -5 census_2010.json
{"females": 1994141, "males": 2085528, "age": 0, "year": 2010}
 {"females": 1997991, "males": 2087350, "age": 1, "year": 2010}
 {"females": 2000746, "males": 2088549, "age": 2, "year": 2010}
 {"females": 2002756, "males": 2089465, "age": 3, "year": 2010}
 {"females": 2004366, "males": 2090436, "age": 4, "year": 2010}
```

Let's define our ETL process:

Extraction

first, we create a DataFrame from a given JSON

Transformation

then we filter data and keep the records for seniors (`age > 54`). Next, we add a new column `total`, which is the total of males and females.

Load

Finally, we write the revised DataFrame into MySQL database and verify the load process.

Extraction

To do a proper extraction, we need to create an instance of the `SparkSession` class and then read JSON and create a DataFrame:

```
from pyspark.sql import SparkSession
spark = SparkSession.builder \
    .master("local") \
    .appName("ETL") \
    .getOrCreate()
```

Next, read JSON and create a DataFrame:

```
input_path = "census_2010.json"
census_df = spark.read.json(input_path)
census_df.count()
101
>>> census_df.show(200)
+---+-----+-----+
| age|females|  males|year|
+---+-----+-----+
|  0|1994141|2085528|2010|
|  1|1997991|2087350|2010|
|  2|2000746|2088549|2010|
...
| 54|2221350|2121536|2010|
| 55|2167706|2059204|2010|
| 56|2106460|1989505|2010|
```

```

...
| 98| 35778| 8321|2010|
| 99| 25673| 4612|2010|
+---+-----+-----+
only showing top 100 rows

```

Transformation

Transformation can involve many processes whose purpose is to clean, format, compute the data to suit the needs of your requirements. For example, you can remove missing data, duplicate data, join columns to create new columns, filter out rows or columns. Once your DataFrame is created from the extraction process, then you can perform many useful transformations such as selecting seniors from your created DataFrame.

```

>>> seniors = census_df[census_df['age'] > 54]
seniors.count()
46
seniors.show(200)
+---+-----+-----+
|age|females| males|year|
+---+-----+-----+
| 55|2167706|2059204|2010|
| 56|2106460|1989505|2010|
| 57|2048896|1924113|2010|
...
| 98| 35778| 8321|2010|
| 99| 25673| 4612|2010|
|100| 51007| 9506|2010|
+---+-----+-----+

```

Next we create a new aggregated column called **total**, which adds up the number of males and females:

```

from pyspark.sql.functions import lit
seniors_final = seniors.withColumn('total',
    lit(seniors.males + seniors.females))
>>> seniors_final.show(200)
+---+-----+-----+-----+
|age|females| males|year| total|
+---+-----+-----+-----+
| 55|2167706|2059204|2010|4226910|

```

```

| 56|2106460|1989505|2010|4095965|
| 57|2048896|1924113|2010|3973009|
...
| 98| 35778|    8321|2010|  44099|
| 99| 25673|    4612|2010|  30285|
|100| 51007|    9506|2010|  60513|
+---+-----+-----+-----+

```

Loading

The loading process involves saving or writing the final output of the transformation step: here we will write the `seniors_final` DataFrame into a MySQL table:

```

seniors_final\
  .write\
  .format("jdbc")\
  .option("driver", "com.mysql.jdbc.Driver")\
  .mode("overwrite")\
  .option("url", "jdbc:mysql://localhost/testdb")\
  .option("dbtable", "seniors")\
  .option("user", "root")\
  .option("password", "root_password")\
  .save()

```

The final step of loading is to verify the load process:

```

$ mysql -uroot -p
Enter password:
Your MySQL connection id is 9
Server version: 5.7.30 MySQL Community Server (GPL)

mysql> use testdb;
Database changed
mysql> select * from seniors;
+-----+-----+-----+-----+
| age | females | males | year | total |
+-----+-----+-----+-----+
| 55 | 2167706 | 2059204 | 2010 | 4226910 |
| 56 | 2106460 | 1989505 | 2010 | 4095965 |
| 57 | 2048896 | 1924113 | 2010 | 3973009 |
...
| 98 | 35778 | 8321 | 2010 | 44099 |
| 99 | 25673 | 4612 | 2010 | 30285 |

```

```
| 100 | 51007 | 9506 | 2010 | 60513 |
+-----+-----+-----+-----+
46 rows in set (0.00 sec)
```

Summary

- Spark is a fast and powerful unified analytics engine (up to 100x faster than traditional Hadoop MapReduce) due to in-memory operation and it offers robust, distributed, fault-tolerant data objects (called RDDs and DataFrames). Further, Spark integrates with the world of machine learning and graph analytics through packages **Mlib** (machine Learning library), and **GraphX** (graph library).
- This chapter presented Spark's and PySpark's high-level architectures. Also, we presented the relationship between an application program and the Spark cluster
- To do something useful with Spark, you may use Spark's transformations and actions in four programming languages: Java, Scala, R, and Python. PySpark (Python API for Spark) can be used for solving big data problems; By using Spark's transformations, PySpark can effectively transform your data into your desired result and format
- The big data can be represented as an Spark's data abstraction (RDD, DataFrame, and DataSet—all of these are distributed data sets)
- You can run PySpark from the PySpark shell (`pyspark` command from a command line) for interactive Spark programming. Using PySpark shell, you can create and manipulate RDDs and DataFrames
- We can submit a standalone PySpark application to a Spark cluster by using the `spark-submit` command; self-contained applications using PySpark are deployed to the production environments

- In solving big data problems, there are lot of choices in selecting Spark's transformations (for example `reduceByKey()` versus `groupByKey()`) and actions and their performances are different from each other

¹ Illumina, Inc. is an American company that develops, manufactures and markets integrated systems for the analysis of genetic variation and biological function

Chapter 2. Transformations in Action

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In this chapter, we will learn the most important Spark transformations (mappers and reducers in the context of data summarization design patterns) and then we will examine how to select specific transformations for targeted problems.

This chapter shows that for a given problem (such as a DNA Base Count), there are multiple PySpark solutions (using different Spark transformations), but efficiencies of these transformations differ due to their implementation and shuffle process — when grouping of values per key happens. DNA base count algorithm is very similar to the classic word count (finding frequency of unique words over a set of files/documents) with the difference that in DNA base counting, you find the frequencies of DNA letters (A,T, C, G), while in classic word count you find the frequency of unique words.

What is the reason for selection of DNA base count problem? By solving this problem, we will learn a data summarization design pattern, which

summarizes a lot of DNA data strings/sequences into a small set of useful information (frequency of DNA letters).

This chapter

- Introduces a simple real problem: DNA Base Count
- Provides a complete, end-to-end solutions in PySpark
- Uses different mappers and reductions to solve the DNA Base Count problem and introduces data summarization design patterns
- Provides performances of using different mappers and reductions

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 2](#).

DNA Base Count Example

The purpose of this example is to count DNA Bases for a give set of DNA string/ sequences. Don't worry — you don't need to be an expert in DNA, biology, and genomics to understand this example. I'll cover the basics for you, which should be all you need to get the idea.

Human DNA consists of about 3 billion bases, and more than 99 percent of those bases are the same in all people. What is DNA Base Counting? To understand DNA Base Counting, we need to first understand DNA strings.

DNA strings are constructed from the alphabet {A, C, G, T}, whose symbols represent the bases adenine (A), cytosine (C), guanine (G), and thymine (T). Our DNA is composed of a set of DNA strings. These four letters represent the bases that make up DNA. The question is how many times does a certain base letter occur in the DNA string? For example, if a

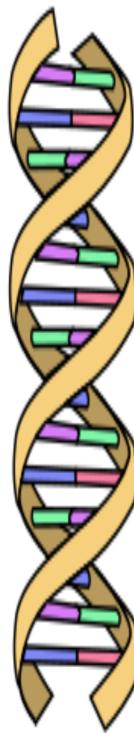
DNA string is "AAATGGCATTA" and we ask how many times the base A occurs in this string, the answer is 5; if we ask how many times the base T occurs in this string, the answer is 3. Therefore, we want to count the number of each base character, ignoring cases. Since DNA machines might create uppercase and lowercase letters, then we will convert all of them to lowercase.

For this problem, I provide three distinct solutions using a set of very powerful and efficient Spark transformations such as `map()`, `flatMap()`, `filter()`, `reduceByKey()`, `groupByKey()`, and `mapPartitions()`. Even though all solutions generate the same results, their performances will be different due to the selection of different Spark transformations.

Figure 2.1 illustrates the process of solving DNA Base Count using Spark. To solve this problem, we

1. Write a driver program in Python, using the PySpark API (which is a series of Spark's transformations and actions)
2. Submit the program to a Spark cluster (explained in the following sections).

All the solutions will read input (FASTA files format— to be defined shortly) and produce a dictionary, where the key is a DNA letter and the value is the associated frequency.



DNA

■ = Adenine
■ = Thymine
■ = Cytosine
■ = Guanine
■ = Phosphate backbone

Spark
Cluster

Spark
Driver Program

Input: FASTA Input Format

```
>seq1
cGTAaccaataaaaaaacaagcttaacctaattc
>seq2
agcttagTTGGatctggccgggg
>seq3
gcggatttactcCCCCCAAataaatggagtctg
gaattcgacca
...
```

Output

```
(a, 127098762)
(t, 456349900)
(c, 780009873)
(g, 568900333)
```

Figure 2-1. DNA Base Count

For DNA Base Count problem, I will provide three different and distinct PySpark solutions (labeled as Versions 1, 2, and 3) using variety of Spark's transformations and actions. These three solutions will show that we have choices in selecting Spark transformations for solving our DNA Base Count problem (and any data problem you are trying to solve) and each transformation has a different performance. The complete PySpark solutions are provided as Table 2.1. By the end of this chapter you will learn that there are many options in selecting Spark's transformations and actions, but only your data and selected transformations will give you an insight to the performance of your data solution. This means that you should test your solution using real production data — **not just toy data** — before deploying your solution to the production environment. In the solution 3, I will introduce a very powerful Spark transformation called `mapPartitions()`, which is an ideal for reducing large volume of data into small amount of desired information (such as frequencies of DNA letters, which is a small hash map table).

T

a

b

l

e

2

-

l

.

S

o

l

u

t

i

o

n

s

f

o

r

D

N

A

B

a

s

e

C

o

u

n

t

Features	Solution 1	Solution 2	Solution 3
Program	<code>dna_bc_ver_1.py</code>	<code>dna_bc_ver_2.py</code>	<code>dna_bc_ver_3.py</code>
Design Pattern: <i>Summarization design pattern</i>	<i>Basic MapReduce</i>	<i>In Mapper Combiner</i>	<i>Mapping Partitions</i>
Transformations	1. <code>textFile()</code> 2. <code>flatMap()</code> 3. <code>reduceByKey()</code>	1. <code>textFile()</code> 2. <code>flatMap()</code> 3. <code>reduceByKey()</code>	1. <code>textFile()</code> 2. <code>mapPartitions()</code> 3. <code>reduceByKey()</code>

Performance of these 3 programs using my MacBook (with 16 GB RAM, 2.3 GHZ Intel Processor, 500 GB hard disk) are very different. Note that for these 3 programs, I just used the default parameters (no optimization is done for any solution). The performances are presented (used default parameters with `$SPARK_HOME/bin/park-submit` command) in Table 2.2.

T

a

b

l

e

2

-

2

.

P

e

r

f

o

r

m

a

n

c

e

f

o

r

D

N

A

B

a

s

e

C

o

u

n

t

Input Data (in bytes)	Version-1	Version-2	Version-3
253,935,557	72 seconds	27 seconds	18 seconds
1,095,573,358	258 seconds	79 seconds	57 seconds

What does this basic performance table tell you? When you write your PySpark application, you have a lot of choices in selecting different transformations and actions. What are the rules for picking the “Right Spark Transformations and Actions”. There are no hard and fast rules for picking the perfect transformations. Only your data and Spark programs can determine this for you. In general, when you try to write a PySpark application, you can usually choose from many arrangements of transformations and actions that will produce the same results. However, not all these arrangements of transformations and actions will result in the same performance: avoiding common pitfalls and picking the right arrangement can make a world of difference in an application’s performance.

We will touch on “reduction transformations” subject in chapter 6: “Reductions in Spark”. For example, for a large set of `(key, value)` pairs, `reduceByKey()` or `combineByKey()` is more efficient than using combination of `groupByKey()` and `mapValues()`, because `reduceByKey()` or `combineByKey()` reduces the shuffling time. For example, if your RDD (represented by variable `rdd`) is an `RDD[(String, Integer)]` (an RDD, where each element is a pair of `(key-as-String, value-as-Integer)`) then the following:

```
# rdd : RDD[(String, Integer)]
rdd.groupByKey().mapValues(lambda values : sum(values))
```

will produce the same results as this:

```
# rdd : RDD[(String, Integer)]  
rdd.reduceByKey(lambda x,y: x+y)
```

However, the `groupByKey()` will transfer the entire dataset across the cluster network (this is a performance penalty), while the `reduceByKey()` will compute local sums for each key (performance improvement) in each partition and combine those local sums into larger sums after shuffling. Therefore, `reduceByKey()` will transfer much less data than the `groupByKey()` across the cluster network. In most of the situations, `reduceByKey()` will out perform combination of `groupByKey()` and `mapValues()` transformations.

DNA Base Count Problem

What is the DNA Base Count Problem? According to the [your genome](#): “DNA’s code is written in only four letters, called A, C, T and G. The meaning of this code lies in the sequence of the letters A, T, C and G in the same way that the meaning of a word lies in the sequence of alphabet letters. Your cells read the DNA sequence to make chemicals that your body needs to survive.”

The goal of this example is to find frequencies (or percentages) of A, T, C, G, and N (the letter N denotes any letter other than A, T, C, or G) in a given set of DNA sequences. Letter N represents the number of error letters generated by DNA machines. As I mentioned, { 'A', 'T', 'C', 'G' } stand to the four nitrogenous bases associated with DNA (Table 2.3).

T

a

b

l

e

2

-

3

.

D

N

A

B

a

s

e

L

e

t

t

e

r

s

DNA Letter	Name
-------------------	-------------

A	Adenine
---	---------

T	Thymine
---	---------

C	Cytosine
---	----------

G	Guanine
---	---------

N	any thing other than {'A', 'T', 'C', 'G'}
---	---

For example, "ACGGGTACGAAT" is a very small DNA sequence/string. DNA sequences can be huge and can have uppercase and lowercase letters. For consistency, we will convert all letters to lowercase. For example, human genome consists of three billion DNA base pairs, while the diploid genome (found in somatic cells) has twice the DNA content. The goal of DNA base counting for our example is to generate the data in Table 2.4 (frequencies for each DNA base). Note that I am using the z key to find the total number of DNA sequences processed.

T

a

b

l

e

2

-

4

.

D

N

A

B

a

s

e

C

o

u

n

t

E

x

a

m

p

l

e

Base	Count
-------------	--------------

a	4
t	2

c	2
g	4
n	0
z	1 (the total number of DNA Sequences)

The next subsection introduces the format of DNA strings/sequences.

FASTA Format

DNA sequences can be represented in many different formats including FASTA and FASTQ, which are popular text-based formats — input is given as a text file. Our solution will only handle FASTA format since it is much easier to read FASTA files. Both FASTA and FASTQ formats store sequence data, and sequence metadata. With some minor modifications to presented solutions (modifications required to reading data portion), you can run the revised solutions for the FASTQ format. A DNA Base Count solution using FASTQ format is given at the end of this chapter, so we'll discuss those modifications in detail there.

A sequence file in **FASTA Format** can contain many DNA sequences. Each sequence begins with a single-line description, followed by one or many lines of sequence data. According to the FASTA format specifications, the description line must begin with a “greater-than” (“>”) symbol in the first column. Note that the description line may be used for counting the number of sequences and does not have any DNA sequence data.

FASTA Example

An example sequence in FASTA format is presented by the `sample.fasta` file. This FASTA file has 4 DNA sequences and case (upper-case or lower-case) of characters are irrelevant: this small FASTA file, which will be used as a test case for our PySpark programs.

```
cat sample.fasta
>seq1
cGTAaccaataaaaaacaagcttaacctaattc
>seq2
agcttagTTTGGatctggccgggg
>seq3
gcggatttactcCCCCCAAAAAGgggagagcccagataaatggagtctgtcgccaca
gaattcgcacca
AATAAAACCTCACCCAT
agagcccagaatttactcCCC
>seq4
gcggatttactcaggggagagcccagGataaatggagtctgtcgccaca
gaattcgcacca
```

FASTA Data Download

To test the DNA Base Count programs, provided in this chapter, you may download FASTA data from the [University of California](#). As of writing this book, GitHub does not permit files greater than 100MB. Some sample FASTA files are provided in this book's [GIT repository](#).

Next, we'll go through the three distinct PySpark solutions for the DNA Base Count problem, using different Spark transformations. Remember that while outcome of all solutions are the same (same results), performance of each solution will differ due to the nature of data and selection of transformations.

DNA Base Count Solution 1

This solution provides a very basic solution for the DNA Base Count problem. The high level workflow for solution 1 is presented by the Figure 2.2.

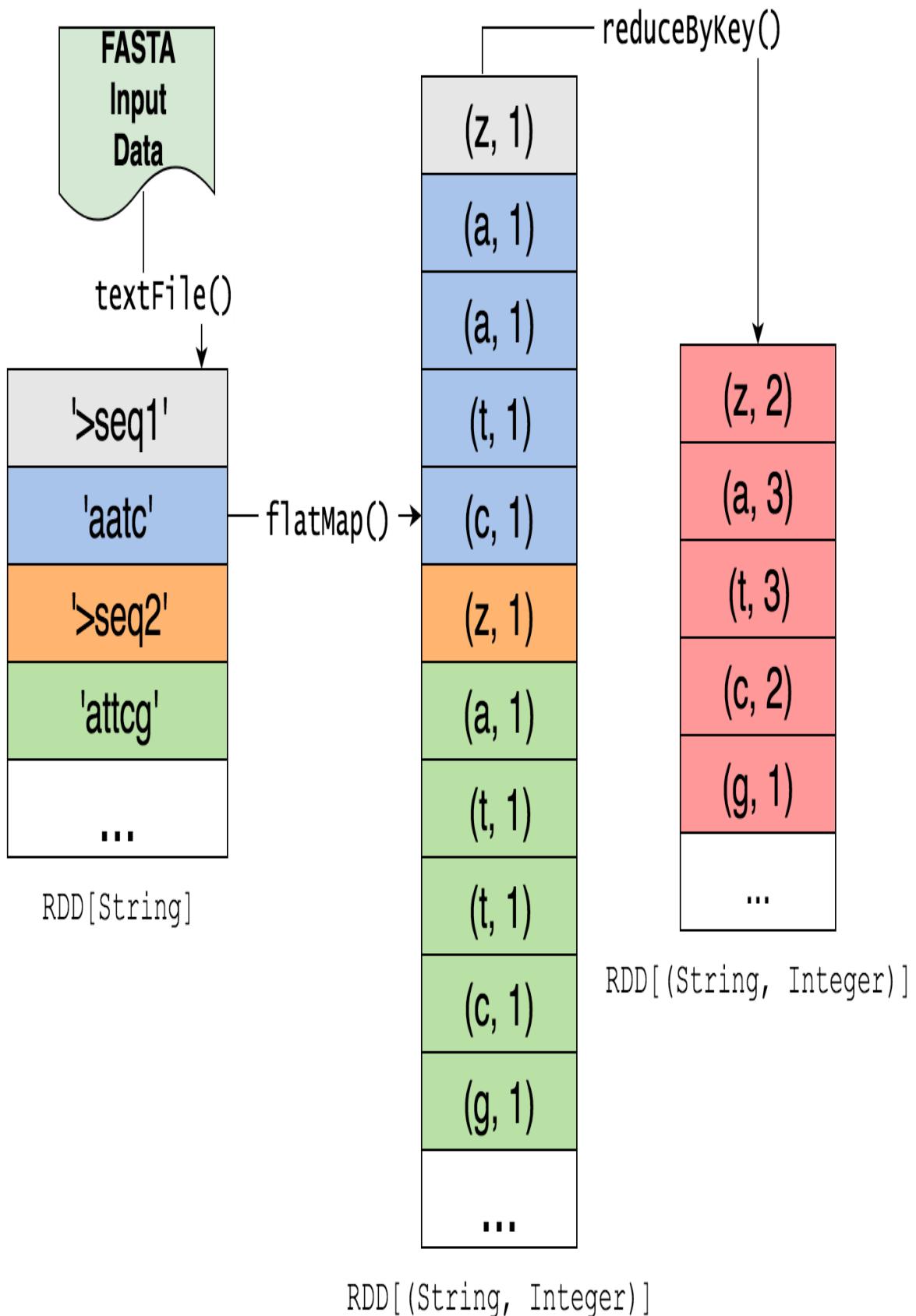


Figure 2-2. DNA Base Count: Solution Version 1

The high-level solution is presented in three simple steps:

1. Read FASTA input data and create an `RDD[String]`, where each RDD element is a FASTA record (it can be either a comment line or an actual DNA sequence)
2. Define a mapper function: for every DNA letter in a FASTA record, emit a pair of (`dna_letter`, 1) where `dna_letter` is in {A, T, C, G} and 1 is a frequency (similar to a word count solution)
3. Sum up frequencies for all DNA letters (this is a reduction step). For each unique `dna_letter`, group and add all frequencies.

For testing this solution, I will use the provided sample FASTA file (`sample.fasta` as an input) for running our PySpark solution.

Step 1: Create an RDD of String from Input

The `SparkContext.textFile()` function is used to create an `RDD[String]` for input in FASTA text-based format. The `textFile()` can be used to read a text file from HDFS, Amazon S3, a local file system (available on all Spark nodes), or any Hadoop-supported file system URI, and return it as an RDD of Strings. Let “spark” be an instance of the `SparkSession` class, then to create FASTA records RDD (as denoted by `records_rdd`), we have at least two options:

Option 1: Use SparkSession

```
>>># spark : instance of SparkSession
>>> input_path = "./code/chap02/sample.fasta" ❶
>>> records_rdd = spark.read
      .text(input_path)
      .rdd.map(lambda r: r[0]) ❷
```

- ❶ Define input path

- ② Use DataFrameReader (`spark.read`) to create a DataFrame and then convert it to an `RDD[String]`

Option 2: Use SparkContext

```
>>> input_path = "./code/chap02/sample.fasta" ❶
>>> # Let 'spark' be an instance of SparkSession
>>> # access SparkContext from a SparkSession
>>> sc = spark.sparkContext ❷
>>> records_rdd = sc.textFile(input_path) ❸
```

- ❶ Define input path
- ❷ Create an instance of a SparkContext (as `sc`)
- ❸ Use SparkContext to read input and create an `RDD[String]`

To read a text file and convert it to an RDD, Option 2 is preferable, because it is easy and efficient. Option 1 works too, but first it creates a DataFrame, and then converts it to an RDD and eventually performs another mapper transformation.

Next we examine the content of the created RDD: each RDD element (as a String) is denoted by `u'<element>'`. The `RDD.collect()` is used to get the content as a list of String objects and display it. For large RDDs, you should not use `collect()`, which might cause OOM (**Out Of Memory**) problem as well as a performance penalty. To just view the first N elements of an RDD, you may use `RDD.take(N)`.

Next we examine the whole content of created RDD (since input file is small enough — for large RDDs, we should avoid the `collect()`) due to a possible OOM errors:

```
>>> records_rdd.collect()
[
    u'>seq1',
    u'cGTAaccaataaaaaacaagcttaacctaattc',
```

```

u'>seq2',
u'agcttagTTTGGatctggccgggg',
u'>seq3',
u'gcggatttactcCCCCCAAAAANaggggagagcccagataaatggagtctgtgcgtccaca',
u'gaattcgcacca',
u'AATAAACCTCACCAT',
u'agagcccagaatttactcCCC',
u'>seq4',
u'gcggatttactcagggagagcccagGataaatggagtctgtgcgtccaca',
u'gaattcgcacca'
]

```

Step 2: Define a Mapper Function

To map RDD elements into a set of pairs (`dna_letter`, 1), we define a Python function, which is passed to the `flatMap()` transformation—`flatMap()` is a “**1-to-many**” transformation—returns a new RDD by first applying a function to all elements of this RDD, and then flattening the results. For, example if your Python function, which is passed to the `flatMap()` transformation, returns a list as [V~1~, V~2~, V~3~], then that will be flattened into 3 target RDD elements as V~1~, V~2~, and V~3~.

To process a source RDD element, we define a function, `process_FASTA_record`, which accepts an RDD element (a single record of FASTA file as a String), and returns a list of pairs as (`dna_letter`, 1). For example, if your input record to the `flatMap()` is “AATTG”, then it will emit the following (key, value) pairs (note that all DNA letters are converted to lower case — input is mixed of uppercase and lowercase characters):

```

('a', 1)
('a', 1)
('t', 1)
('t', 1)
('g', 1)

```

Next, we define the `process_FASTA_record()` function, (denoted by Listing 2.1) which accepts a string (an element of source RDD) and returns a list of (key, value) pairs. If the input is a “description record” (no sequence data, which begins with `>seq`), then we emit (“z”, 1). The key z is used to find the

total number of sequences for a given input (the letter z is not a DNA letter and may be used to count the total number of DNA sequences). This will enable us to find the number of sequences as well. If the input is a DNA sequence, then we tokenize it by characters and then for each DNA letter (denoted by `dna_letter`), we emit (`dna_letter`, 1), and finally return list of these pairs. I have included some `print` statements for debugging purposes. For production environments, these `print` statements should be removed as they will cause performance penalties.

```
# Parameter : fasta_record : String,
#           a single FASTA record
#
# output: a list of (key, value) pairs,
#         where key is a DNA-Letter and
#         value is a frequency
#
def process_FASTA_record(fasta_record):
    key_value_list = [] ①
    #
    if (fasta_record.startswith(">")):
        # z counts the number of FASTA sequences
        key_value_list.append(("z", 1)) ②
    else:
        chars = fasta_record.lower()
        for c in chars:
            key_value_list.append((c, 1)) ③
    #
    print(key_value_list) ④
    return key_value_list ⑤
#end-def
```

- ❶ Create an empty list, we will keep adding (key, value) pairs (this is our output from this function)
- ❷ Append ("z", 1) to list
- ❸ Append (c, 1) to list, where c is a DNA-letter
- ❹ For debugging purposes only

- ⑤ Return a list of (key, value) pairs, which will be flattened by the `flatMap()`

Now, we will use this function to apply the `flatMap()` transformation to the `records_rdd` (`RDD[String]`) created above:

```
>>># rec refers to an element of records_rdd
>>># Lambda is a notation which defines input and output
>>>#   input: "rec" as an records_rdd element ❶
>>>#   output: result of process_FASTA_record(rec)
>>> pairs_rdd = records_rdd.flatMap(lambda rec: process_FASTA_record(rec)) ❷
```

- ❶ Source RDD (`records_rdd`) is an `RDD[String]`
- ❷ We use a lambda expression, where `rec` denotes a single element of `records_rdd`. The target RDD (`pairs_rdd`) is an `RDD[(String, Integer)]`

or we may write its equivalent as (without using lambda expressions):

```
>>> pairs_rdd = records_rdd.flatMap(process_FASTA_record)
```

For example, if an element of `records_rdd` contains DNA sequence as `gaattcg`, then it will be flattened into the following (key, value) pairs:

```
(g, 1)
(a, 1)
(a, 1)
(t, 1)
(t, 1)
(c, 1)
(g, 1)
```

and if an element of `records_rdd` contains `>seq`, then it will be flattened into the following (key, value) pair (we use the key `z` to represent total number of DNA sequences for a given input):

(z, 1)

Step 3: Find Frequencies of DNA Letters

Now, the `pairs_rdd` contains a set of `(key, value)` pairs where `key` is a DNA letter and `value` is a frequency of that letter as 1. Next, we apply the `reduceByKey()` transformation to the `pairs_rdd` to find the aggregated frequencies for all DNA letters.

The `reduceByKey()` transformation merges the values for each unique key using an associative and commutative reduce function (we use addition as our reduction function). Therefore, we can now see that we are simply taking an **accumulated value** for the given key and summing it with the **next value** of that key. In the simplest form, let key K have five pairs in the RDD: (K, 2), (K, 3), (K, 6), (K, 7), and (K, 8), then the `reduceByKey()` transformation will transform these five pairs into a single pair as (K, 26) where $26=2+3+6+7+8$. For example, if we had 2 partitions for these 5 pairs, then each partition will be processed in parallel and independently:

- Processing Partition-1:

```
Partition-1: {
    (K, 2),
    (K, 3)
}
```

$$(K, 2), (K, 3) \Rightarrow (K, 2+3) = (K, 5)$$

Result of Partition-1: (K, 5)

- Processing Partition-2:

```
Partition-2: {
    (K, 6),
    (K, 7),
    (K, 8)
```

```
}
```

```
(K, 6), (K, 7) => (K, 6+7) = (K, 13)
```

```
(K, 8), (K, 13) => (K, 8+13) = (K, 21)
```

```
Result of Partition-2: (K, 21)
```

- Next, the partitions are merged:

```
Merge Partitions:
```

```
=> Partition-1, Partition-2
```

```
=> (K,5), (K, 21)
```

```
=> (K, 5+21) = (K, 26)
```

```
Final result: (K, 26)
```

For viewing the final result, you may use different actions to get the job done. Below, I use `collect()` and `collectAsMap()` to view the final result. If you want to persist your RDD (as a final result) on a disk, then you may use the `RDD.saveAsTextFile(path)` where `path` is your output directory name.

Next, I present the final reduction for DNA base count:

```
# x and y refers to the frequencies of the same key
# source: pairs_rdd : RDD[(String, Integer)]
# target: frequencies_rdd : RDD[(String, Integer)]
frequencies_rdd = pairs_rdd.reduceByKey(lambda x, y: x+y)
```

Note that the source and target data types for `reduceByKey()` are the same. That is if source RDD is `RDD[(K, V)]` then the target RDD will be the same as `RDD[(K, V)]`. Spark's `combineByKey()` transformation does not have the data types restrictions for values imposed by `reduceByKey()`.

There are several ways that you can view the final output. For example, you may use the `RDD.collect()` function to get the final RDD's elements as a list of pairs:

```

frequencies_rdd.collect()
[
    (u'a', 73),
    (u'c', 61),
    (u't', 45),
    (u'g', 53),
    (u'n', 2),
    (u'z', 4)
]

```

Also, we may use the `RDD.collectAsMap()` action to return the result as a hash map:

```

>>> frequencies_rdd.collectAsMap()
{
    u'a': 73,
    u'c': 61,
    u't': 45,
    u'g': 53,
    u'n': 2,
    u'z': 4
}
>>>

```

We may use other Spark transformations to aggregate frequencies of DNA letters. For example, we may group frequencies (using `groupByKey()` transformation) of a DNA letter and then add all frequencies together. This solution is a less efficient than the one presented by the `reduceByKey()` transformation.

```

grouped_rdd = pairs_rdd.groupByKey() ❶
frequencies_rdd = grouped_rdd.mapValues(lambda values : sum(values)) ❷
frequencies_rdd.collect()

```

- ❶ `grouped_rdd : RDD[(String, [Integer])]`, where key is a String and value is a list/iterable of Integers (as frequencies)
- ❷ `frequencies_rdd : RDD[(String, Integer)]`

For example, if `pairs_rdd` contains 4 pairs of ('z', 1), then the `grouped_rdd` will have a single pair of ('z', [1, 1, 1, 1]). That is it groups values for the same key. While both of these transformations (`reduceByKey()` and `groupByKey()`) produce the correct answer, the `reduceByKey()` solution works much better on a large FASTA dataset. That's because Spark knows it can combine output with a common key (DNA Letter) on each partition before shuffling the data. Spark experts recommend that we should **Avoid `groupByKey()`** and use `reduceByKey()` and `combineByKey()` whenever possible. In other words, we can say that `reduceByKey()` and `combineByKey()` scale-out better than `groupByKey()`.

Pros and Cons of Solution Version 1

Pros:

- The provided solution works and is simple: uses minimal amount of code to get the job done and uses the Spark's simple `map()` and `reduceByKey()` transformations.
- There is no scalability issue since we use `reduceByKey()` for reducing all (key, value) pairs, which will automatically perform the `combine()` optimization (local aggregation) on all worker nodes.

Cons:

- This solution emits too many (key, value) pairs, where key is a DNA-letter and value is 1, as frequency. Sometimes, emitting too many (key, value) pairs might cause memory problems. If you get any error due to too many (key, value) pairs, then you might adjust the RDD's `StorageLevel`. By default, Spark uses MEMORY, but you can set the `StorageLevel` to MEMORY and DISK combinations for that RDD.
- Performance is not optimal since emitting too many (key, value) pairs will take network time and prolong the shuffle time. As during the second processing step we defined, too many single frequency

tuples are emitted, network time will prove a bottleneck when scaling this solution.

Next I present Solution 2 for DNA Base Count, which improves on Solution 1.

DNA Base Count Solution 2

Solution 2 is an improved version of solution 1. In solution 1, we emitted pairs of (`dna_letter`, 1) for each DNA letter in given DNA sequences. In solution 2, since some of the FASTA sequences can be very long, instead of emitting (`dna_letter`, 1) per DNA letter, we aggregate/group them into a hash map (dictionary concept in Python) and then flatten the hash map into a list and finally do the frequencies aggregation. For example, if a given FASTA sequence record is "aaatttcggggaa", then 2nd column (Solution 2) will be emitted instead of 1st column (Solution 1).

T
a
b
l
e

2
-
5

.

E
m
i
t
t
e
d

(
k
e
y
,

v

a
l

u

e

)

p

a

i

r

s

Solution 1**Solution 2**

(a, 1) (a, 5)

(a, 1) (t, 3)

(a, 1) (c, 1)

(t, 1) (g, 4)

(t, 1)

(t, 1)

(c, 1)

(g, 1)

(g, 1)

(g, 1)

(a, 1)

(a, 1)

The advantage of solution Version 2 is that we emit much fewer (key, value) pairs which will ease up the cluster network traffic. Meanwhile, we do an “InMapper Combiner” (IMC) optimization. IMC is an approach to possibly improve the speed of a MapReduce job by reducing the number of intermediary (key, value) pairs emitted from mappers to reducers. For a given DNA string, in the mapper step, by aggregating the frequencies for the same DNA letter, we will emit much fewer (key, value) pairs, which can improve the cluster network traffic and hence can improve the overall performance of your program.

Therefore, our solution for Version 2 is presented as:

1. Read FASTA input data and create an `RDD[String]`, where each RDD element is a FASTA record. This step is the same as Version 1.
2. Mapper step: for every FASTA record, create a `HashMap[Key, Value]` (a dictionary or hash table) where Key is the DNA letter and Value is an aggregated frequency for that DNA letter. Then flatten (using Spark's `flatMap()`) the hash map into a list of (Key, Value) pairs. This step is different from Version 1. Compared with Version 1, this step enable us to emit less (key, value) pairs
3. Find frequencies for all DNA letters by aggregating frequencies of the same DNA letter (this is a reduction step). For each `dna_letter`, group and add all frequencies. This step is the same as Version 1.

The high level workflow for solution Version 1 is presented by the Figure 2.3.

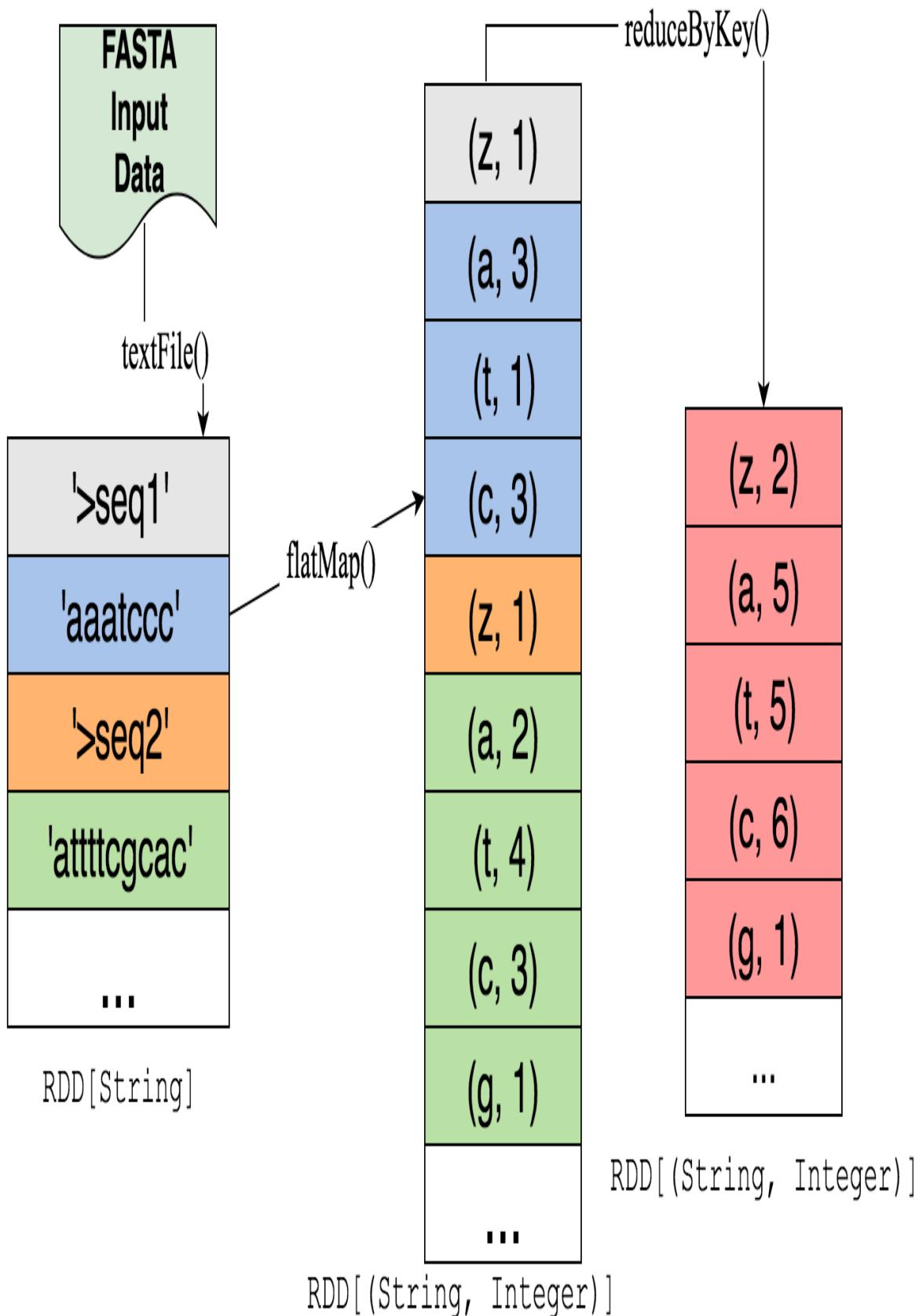


Figure 2-3. DNA Base Count: Solution Version 2

Let's dig into the details of each step.

Step-1: Create an RDD[String] from Input

The `SparkContext.textFile()` function is used to create an RDD for input in FASTA text-based format. Let `spark` be a `SparkSession` object.

```
>>># spark : an instance of SparkSession
>>># then access sparkContext from a SparkSession
>>> input_path = "./code/chap02/sample.fasta"
>>> records_rdd = spark.sparkContext.textFile(input_path) ❶
```

❶ `records_rdd : RDD[String]`

Step 2: Define a Mapper Function

Map each RDD element (which represents a single FASTA record as a String) into a list of (key, value) pairs, where key is a unique DNA letter and value is an aggregated frequency for the entire record. We define a Python function, which is passed to the `flatMap()` transformation to return a new RDD by first applying a function to all elements of this RDD, and then flattening the results.

To process an RDD element, we define a Python function, `process_FASTA_as_hashmap` (Listing 4.2), which accepts an RDD element (as a String), and returns a list of (`dna_letter`, `frequency`). For debugging and teaching purposes, I have included some `print` statements, which should be removed for production environments.

```
# Parameter: fasta_record : String,
#             a single FASTA record
#
# output: a list of (DNA-Letter, frequency)
#
def process_FASTA_as_hashmap(fasta_record):
    if (fasta_record.startswith(">")): ❶
```

```

    return [("z", 1)]

    hashmap = defaultdict(int) ②
    chars = fasta_record.lower()
    for c in chars: ③
        hashmap[c] += 1
    #end-for
    print("hashmap=", hashmap)

    key_value_list = [(k, v) for k, v in hashmap.items()] ④
    print("key_value_list=", key_value_list)
    return key_value_list ⑤
#end-def

```

- ❶ It is a comment line for a DNA sequence
- ❷ Create a dictionary[String, Integer]
- ❸ Aggregate DNA-letters
- ❹ Flatten dictionary into a list of (DNA-Letter, frequency)
- ❺ Return the flattened list of (DNA-Letter, frequency)

Now, we will use this Python function, `process_FASTA_as_hashmap()`, to apply the `flatMap()` transformation to the `records_rdd` (as `RDD[String]`) created above:

```

>>># source: records_rdd (as RDD[String])
>>># target: pairs_rdd (as RDD[(String, Integer)])
>>> pairs_rdd = records_rdd.flatMap(lambda rec: process_FASTA_as_hashmap(rec))

```

or we may write this as:

```

>>># source: records_rdd (as RDD[String])
>>># target: pairs_rdd (as RDD[(String, Integer)])
>>> pairs_rdd = records_rdd.flatMap(process_FASTA_as_hashmap)

```

For example, if `records_rdd` element contains '`gggggaaattcccg`', then it will be flattened into the following (key, value) pairs:

```
(g, 6)  
(a, 3)  
(t, 2)  
(c, 4)
```

To count the total number of DNA Sequences, if `records_rdd` element begins with "`>seq`", then it will be flattened into the following single (key, value) pair:

```
(z, 1)
```

Step 3: Find Frequencies of DNA Letters

Now, `pairs_rdd` contains (key, value) pairs where key is a `dna_letter` and value is a frequency of that letter. Next, we apply the `reduceByKey()` transformation to the `pairs_rdd` to find the aggregated frequencies for all DNA letters. Note that the key '`z`' is used to count the total number of DNA sequences and '`n`' is the key other than DNA bases.

```
# x and y refers to the frequencies of the same key  
frequencies_rdd = pairs_rdd.reduceByKey(lambda x, y: x+y) ❶  
frequencies_rdd.collect() ❷  
[  
    (u'a', 73),  
    (u'c', 61),  
    (u't', 45),  
    (u'g', 53),  
    (u'n', 2),  
    (u'z', 4)  
]
```

- ❶ `pairs_rdd` (as `RDD[(String, Integer)]`)
- ❷ `frequencies_rdd` (as `RDD[(String, Integer)]`)

Also, we may use the `collectAsMap()` action to return the result as a hash map:

```
>>> frequencies_rdd.collectAsMap()
{
    u'a': 73,
    u'c': 61,
    u't': 45,
    u'g': 53,
    u'n': 2,
    u'z': 4
}
>>>
```

Pros and Cons of Solution 2

Pros:

- The provided solution works, simple, and semi-efficient. This solution improves on Version 1, by emitting much less (key, value) pairs, since we create a dictionary per input record and then flatten it into a list of (key, value) pairs, where key is a DNA-letter and value is an associated aggregated frequency of the DNA-letter.
- Network traffic is improved by emitting much fewer (key, value) pairs.
- There is no scalability issue since we use `reduceByKey()` for reducing all (key, value) pairs

Cons:

- For each DNA sequence, this solution emits up to 6 (key, value) pairs, where key is a DNA-letter and value is 1, frequency. This is a much improvement over solution version 1
- Performance is not an optimal since we are still emitting about 6 (key, value) pairs per DNA string

- This solution might be using too much memory due to creation of a dictionary per DNA sequence

DNA Base Count Solution 3

This solution improves on Versions 1 and 2 and is an optimal solution with no scalability problem at all. Here we solve the DNA Base Count problem by a very powerful and efficient Spark transformation called `mapPartitions()`. Before presenting a solution for Version 3, I will deep dive into understanding of the `mapPartitions()` transformation.

Understanding `mapPartitions()` Transformation

Let source RDD be `RDD[T]` and target RDD be `RDD[U]`, then the `mapPartitions()` transformation is defined as:

```
pyspark.RDD.mapPartitions(f, preservesPartitioning=False)
```

`mapPartitions()` is a method in the `pyspark.RDD` class.

Description:

Return a new RDD (called target RDD) by applying a function `f()` to each partition of the source RDD.

Input to `f()` is an iterator (of type `T`), which represents a single partition of the source RDD.
Function `f()` returns an object of type `U`.

`f: Iterator<T> --> U` ①

`mapPartitions : RDD[T] -- f() --> RDD[U]` ②

- ➊ The function `f()` accepts a pointer to a single partition (as an `iterator` of type `T`) and returns an object of type `U`; `T` and `U` can be any data types and they do not have to be the same
- ➋ Transform an `RDD[T]` to `RDD[U]`

To understand the semantics of the `mapPartitions()` transformation, first, we should understand the concept of a “partition” and partitioning in Spark. Informally, using Spark’s terminology, input data (in this case, DNA sequences in FASTA format) is represented as an RDD. To understand partitioning and `mapPartitions()` transformation, as an example say that we have 6,000 million or — about 6 billion — records). Then the Spark partitioner partitions input data into 3,000 chunks/partitions and sends it to a mapper transformations. So, each partition will have roughly 2 million records — therefore each partition is processed by a single `mapPartitions()` transformation. If you divide 6 billion records into 3,000 partitions, then each partition will have about 2 million records. Therefore, a function `f()`, which is used in the `mapPartitions()` transformation, will accept an iterator (as an argument) to handle one partition — 2 million records. To make this transformation efficient, we will use a single dictionary per partition to aggregate DNA letters and its associated frequencies.

In solution 3, we will create one dictionary per partition rather than a dictionary per FASTA record. This is a huge improvement over solution of versions 1 and 2. Creation of 3,000 hash tables in a cluster uses very little memory compared in creating a dictionary per input record. This solution is a scale-out solution, since creating and aggregating 3,000 dictionaries takes minimal amount of memory and it is fast due to concurrent and independent processing of all partitions in the cluster.

So, what is the main difference between `map()` and `mapPartitions()` transformations? In a nutshell, `map()` is a “**1-to-1**” transformation: maps each element of the source RDD into a single element of target RDD. While `mapPartitions()` can be considered as a “**Many-to-1**” transformation: maps each partition (comprised of many elements of the source RDD — each partition may have thousands or millions of elements) of the source RDD into a single element of target RDD.

The `map()` transformation converts each element of the source RDD into a single element of the target RDD by applying a mapper function. Let `func` be

a user provided function, then `mapPartitions(func)` converts each partition (can be comprised of thousands or millions of RDD elements) of the source RDD into multiple elements of the result (possibly none) by applying the function `func` to each partition. Therefore, the `mapPartitions(func)` returns a new RDD by applying a function to each partition of this input RDD. The `mapPartitions()` transformation is a map operation over partitions and not over the elements of the partition (the `func` receives an iterator, which you can iterate over elements of a partition). The `mapPartitions()` transformation is called once for each input RDD partition unlike `map()` and `foreach()` which is called for each element in the RDD. The main advantage being that, we can do initialization on per-partition basis instead of per-element basis (as done by `map()` & `foreach()`).

The `mapPartitions()` transformation semantics for the DNA Base Count is illustrated by the Figure 2.4.

mapPartitions(func) : SourceRDD → TargetRDD

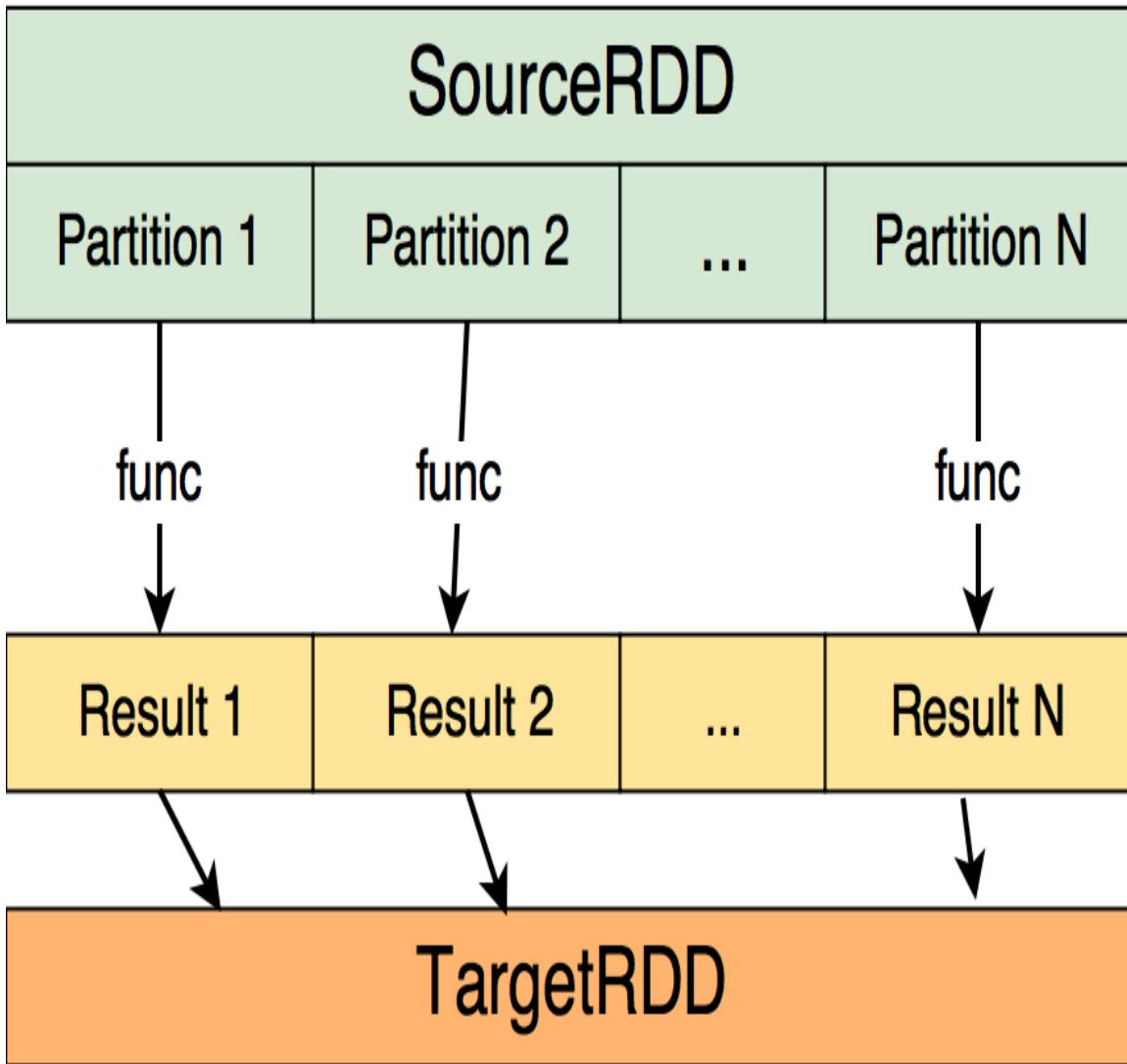


Figure 2-4. The `mapPartitions()` Transformation

Let's walk through Figure 2.4:

- The SourceRDD represents all of our input as `RDD[String]`, since each record of FASTA file is a String object
- The entire input is partitioned into N (where N can be 100, 200, 1000, ... — based on data size and cluster resources) chunks/partitions, where each chunk/partition may have thousands or

million of DNA sequences (each DNA sequence is a record of `String` data type). Partitioning of source RDD into partitions is similar to the Linux's `split` command, which splits a file into pieces.

- Each partition is sent to a `mapPartitions()` mapper/worker/executor to be processed by your provided `func()`. Your `func()` accepts a partition (as an iterator of `String` type) and returns at most 6 (key, value) pairs, where key is a DNA-letter and value is the total frequency of that letter for that partition. Note that partitions are processed in parallel and independently
- Once processing all partitions are completed, the results are merged into the target RDD (depicted as `TargetRDD`), which is an `RDD[(String, Integer)]`, where the key is a DNA-letter and value is the frequency of the DNA-letter

The detailed `mapPartitions()` transformation semantics for our DNA Base Count is illustrated by the Figure 2.5.

`mapPartitions(func) : SourceRDD → TargetRDD`

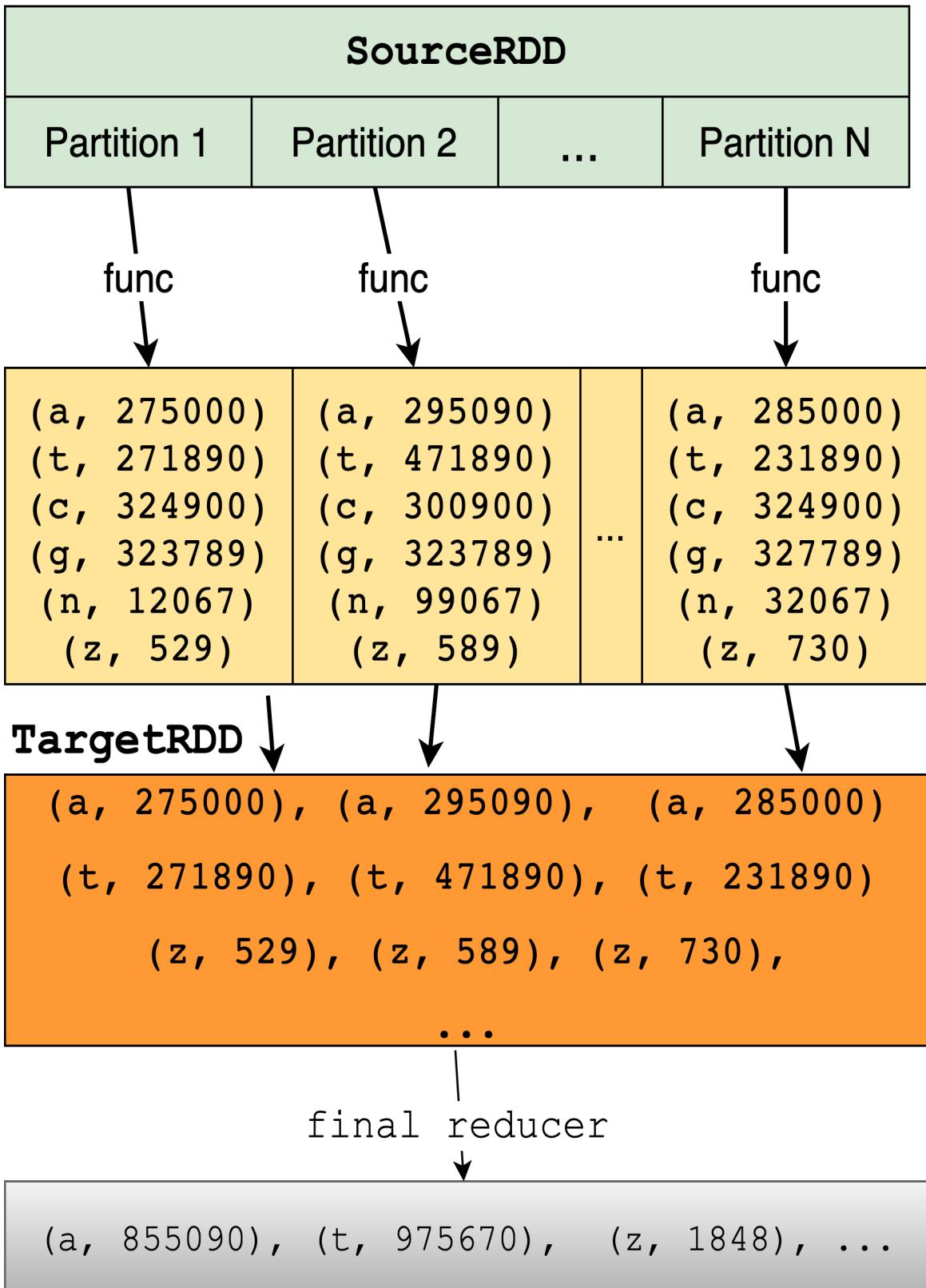


Figure 2-5. Detailed `mapPartitions()` Transformation

How does the `mapPartitions()` work? The Figure 2.5 shows that our input (FASTA format data) has been partitioned into N chunks/partitions where each partition can be handled by a mapper/worker/executor independently and in parallel. For example, if our input has total of 5 billion records and $N = 50,000$, then each partition will have about 100,000 FASTA records ($5 \text{ billion} = 50,000 * 100,000$). Therefore, each `func()` will process (by means of iteration) about 100,000 FASTA records. Each partition will emit at most 6 (key, value) pairs, where keys will be in {"a", "t", "c", "g", "n", "z"} where n as a key for non-DNA letters and z as a key for number of consumed DNA string/sequences.

According to Spark API: the `mapPartitions(func)` transformation is similar to `map()`, but runs separately on each partition (block) of the RDD rather than on elements of an RDD, so `func` must be of `Iterator` type:

```
source: RDD[T] ❶
#
# Parameter p : iterator<T> ❷
func(p): ❸
    u = <create object of type U by iterating all
        elements of a single partition denoted by p >
    return u ❹
#end-def
#
target = source.mapPartitions(func) ❺
#
target: RDD[U] ❻
```

- ❶ Each element of source RDD is type of T
- ❷ Parameter p is an `iterator<T>`, which represent a single partition
- ❸ Each iteration will return an object of type T
- ❹ Define a `func()`, which accepts a single partition as an `iterator<T>` (an iterator of type T — over a single partition of the source `RDD[T]`) and

returns an object of type of U

- ⑤ Apply the transformation
- ⑥ The result is an $\text{RDD}[U]$, where each partition was converted (using `func()`) into a single object type of U

Let's assume that we have source $\text{RDD}[T]$. Therefore, for our example, T represents a String type (which represents a DNA sequence record) and U represents a hash table (a.k.a dictionary in Python) as `HashMap[String, Integer]`, where key is a DNA-letter (as a String object) and value is the associated frequency (as an Integer).

We can define “func” (as a generic template) in Python as denoted by the Listing 2.2.

```
# Parameter: iterator, which represents a single partition
#
# Note that iterator is a parameter from the mapPartitions()
# transformation, through which we can iterate through all
# the elements in a single Partition.
#
# source : RDD[T]
# target : RDD[U]
#
def func(iterator): ❶
    1. make sure that iterator is not empty, if it is empty,
       then handle it properly, you can not ignore empty partitions

    2. initialize your desired "data structures DS"
       (such as dictionaries and lists)

    3. iterate for all records in a given partition
        for record in iterator: ❷
            3.1 process(record)
            3.2 update your "data structure DS"
        end-for

    4. if required, post process your "data structures DS"
       result_for_single_partition = post_process(DS) ❸
```

```
5. return result_for_single_partition  
#end-def
```

- ❶ iterator is a pointer to a single partition, which you can iterate on elements of a partition
- ❷ record is data type of T
- ❸ result_for_single_partition is a data type of U

SUMMARIZATION DESIGN PATTERN

If your data is big and want to summarize view of your data so you can glean insights not available from looking at a localized set of records alone, then Summarization Design Pattern may be used to solve your problem. Summarization design pattern is about grouping similar data together and then performing a Math/Stat operation such as calculating a statistic, building an index, or just simply counting. Spark's `mapPartitions()` can be used to implement Summarization design pattern.

But when should we use the `mapPartitions()`? The `mapPartitions()` transformation should be used when you want to extract some condensed or minimal information or small amount of information (such as finding the minimum and maximum of numbers, top-10 URLs, and such as finding count of DNA bases) from each partition, where each partition is a large set of data. For example, if you want to find the minimum and maximum of all numbers in your input, then using `map()` can be pretty inefficient, since you will be generating tons of intermediate (key, value) pairs, but the bottom line is you just want to find two numbers: the minimum and maximum of all numbers in your input. Therefore, using `mapPartitions()` is the best choice to solve DNA Base Count problem.

The `mapPartitions()` can be used for other MapReduce design patterns. Suppose you want to find top-10 (or bottom-10) for your input, then `mapPartitions()` can work very well: find the top-10 (or bottom-10) per partition, then find the top-10 (or bottom-10) for all partitions: this way you are limiting emitting too many intermediate (key, value) pairs.

For counting DNA bases, the `mapPartitions()` transformation is an ideal solution: from each partition find frequencies of 4 DNA keys: {A, T, C, G}. This solution scales out very well even when your number of partitions is in high thousands. Let's say you partition your input into 100,000 (which is a very high number of partitions — typically the number of partitions will not be this high). Then aggregating 100,000 dictionaries (hash map) is a very trivial and can be accomplished in seconds and there will not be any “out of memory” problems at all and there will not be any scalability problems.

I will mention one more tip about using `mapPartitions()` before presenting a complete DNA Base Count solution using the powerful `mapPartitions()` transformation.

Suppose that you will be accessing a database for some of your data transformations. So, you need a connection to your database. As you know, creating a connection object is expensive and it will take some time (may be a second or two) to create a connection object. If you create a connection object per source RDD element, then your solution will not scale out at all: you will run out of connections and resources and your solution will fail.

Whenever you have heavyweight initialization that (such as creating a database connection object) should be done once for many RDD elements rather than once per RDD element, and if this initialization, such as creation of objects from an external library, cannot be serialized (so that Spark can transmit it across the cluster to the worker nodes), use `mapPartitions()` instead of `map()`. The `mapPartitions()` transformation provides for the **initialization to be done once per worker** task/partition instead of once per RDD data element.

This concept of initialization per partition/worker is presented by the following example:

```
# source_rdd : RDD[T]
# target_rdd : RDD[U]
target_rdd = source_rdd.mapPartitions(func)

def func(partition): ❶
    # create a heavyweight connection object
    connection = <creates a db connection per partition> ❷
```

```

data_structures = <create and initialize your data structure> ③

# iterate all partition elements
for rdd_element in partition: ④
    <use connection and rdd_element to
     make a query to your database>
    <update your data_structures>
#end-for

connection.close() # close db connection here ⑤

u = <prepare object of type U from data_structures> ⑥
return u ⑦
#end-def

```

- ❶ The **partition** parameter is an **iterator<T>**, which represents a single partition of **source_rdd**; this **func()** returns an object of type **U**
- ❷ Create a single **connection** object to be used by all single partition elements
- ❸ The **data_structures** can be a list or dictionary or your desired data structures
- ❹ The **rdd_element** is a single element of type **T**
- ❺ Close the **connection** object (to release allocated resources)
- ❻ Create an object of type **U** from your created **data_structures**
- ❼ Return a single object of type **U** per partition

Now that you understand the basics of Summarization design pattern (to be implemented by Spark's **mapPartitions()**), let's get into the specifics of using it to solve our DNA base problem.

We will use the **.../chap02/sample.fasta** file for testing our PySpark solution 3.

The high level workflow for solution 3 is presented by the Figure 2.6.

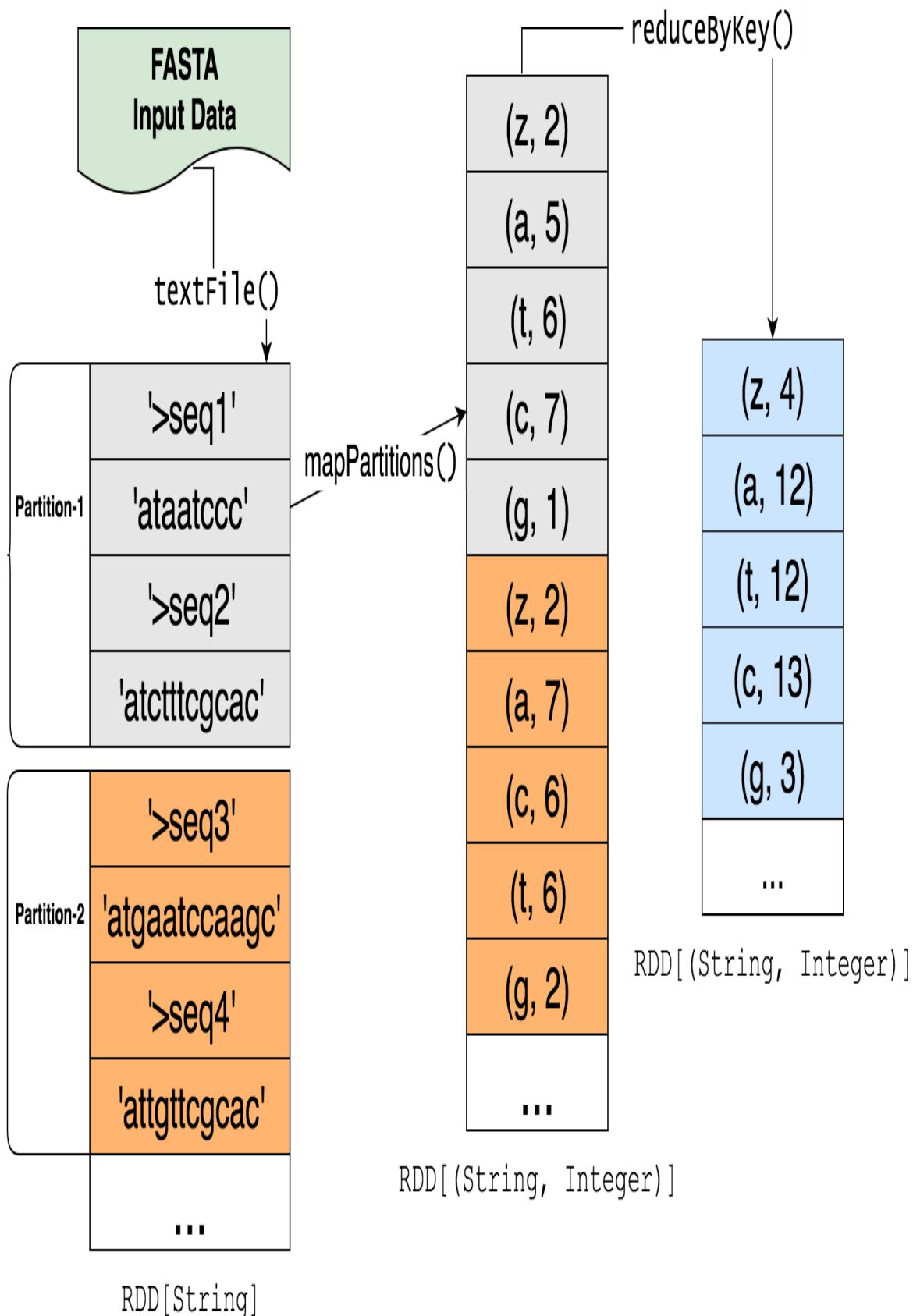


Figure 2-6. DNA Base Count: Solution Version 3

Example 2-1.

Some of key points about Figure 2.6:

- I just show only 4 records (2 FASTA sequences) for a partition, but in reality, each partition may contain thousands or millions of records. If total of all your input is N records and P is the number of partitions, then each partition will have about (N / P) records.
- If we have enough resources in our Spark cluster, then each partition can be processed in parallel and independently
- If you have a lot of data, but your requirement is to extract small amount of information from all data, then `mapPartitions()` might be good choice and will out perform `map()` and `flatMap()` transformations.

The main steps for solution 3 are presented:

Step 1: Create an RDD of String from Input

The `SparkContext.textFile()` function is used to create an RDD for input in FASTA text-based format. This step is identical to the Step 1 of the solution Version 1.

```
input_path = ".../code/chap02/sample.fasta"
>>> records = spark.sparkContext.textFile(input_path) ❶
```

- ❶ Create records as an `RDD[String]`

Step 2: Define a Function to Handle a Partition

Let your RDD be an `RDD[T]` (in our example, T is a `String`). Spark splits your input data into partitions (where each partition is a set of elements of type T) and then executes computations on the partitions independently and in

parallel — this is called divide and conquer model. In using the `mapPartitions()` transformation, the source RDD is partitioned into N partitions (the number of partitions are determined by the size and number of resources available in the Spark cluster) and each partition is passed to a function (this can be a user-defined function). You can control the number of partitions by using `coalesce()` as:

```
RDD.coalesce(numOfPartitions, shuffle=False)
```

which partitions the source RDD into a number of partitions (`numOfPartitions`). If you do not use `RDD.coalesce()` for explicit partitioning, then Spark engine will partition your input based on the cluster configurations and available number of resources in the cluster. For example, this is how, we can partition the RDD: the following example creates an RDD and then partitions it into 3 partitions.

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numOfPartitions = 3
>>> rdd = sc.parallelize(numbers, numOfPartitions) ❶
>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> rdd.getNumPartitions() ❷
3
```

- ❶ Create an RDD and set the number of partitions to 3
- ❷ Check the number of partitions for an RDD

Next, I define a `scan()` function (for testing and debugging purposes) in Python to iterate a given iterator: you may use this function to debug small RDDs and check the partitioning (do not use `scan()` for production environments — this is for teaching purposes only):

```
>>> def scan(iterator): ❶
    elements = []
```

```

...     for x in iterator:
...         elements.append(x)
...     print(elements)
...     print "===="
>>>#end-def
>>> rdd.foreachPartition(scan) ②
1 2 3
===
7 8 9 10
===
4 5 6
===

```

- ❶ Iterate elements of a partition
- ❷ Apply the `scan()` function to a given partition — from output, we can see that there are 3 partitions

To see the behavior of partitions, let's add the values of each partition by defining an `adder()` function in Python:

```

>>> def adder(iterator):
...     yield sum(iterator)
...
>>> rdd.mapPartitions(adder).collect()
[6, 15, 34]

```

For DNA Base counting, to handle (i.e., process all elements of a partition) an RDD partition, we define a function, `process_FASTA_partition` (Listing 2.4), which accepts a single partition (represented as an `iterator`). We then iterate on `iterator` to process all elements in the given RDD's partition. For DNA base counting, each partition produces a dictionary, which then we map it into a list of (`dna_letter`, `frequency`) pairs.

```

#-----
# Parameter: iterator
# we get an iterator: which represents a single
# partition of source RDD, through which we can
# iterate through all of the elements in a Partition.
# In a nutshell, a partition is a set of records.

```

```

#
# This function creates a hash map of DNA
# Letters and then flattens it to (key, value) pairs.
#-----
from collections import defaultdict

def process_FASTA_partition(iterator): ①
    hashmap = defaultdict(int) ②

    for fasta_record in iterator:
        if (fasta_record.startswith(">")): ③
            hashmap["z"] += 1
        else: ④
            chars = fasta_record.lower()
            for c in chars:
                hashmap[c] += 1 ⑤
    #end-for

    print("hashmap=", hashmap)
    key_value_list = [(k, v) for k, v in hashmap.items()] ⑥
    print("key_value_list=", key_value_list)
    return key_value_list ⑦
#
#-----

```

- ① Input parameter `iterator` is a handle/pointer to a single partition
- ② Create a hash table of [String, Integer]
- ③ Handle comments for input data
- ④ Handle a DNA sequence
- ⑤ Populate hash table
- ⑥ Flatten a hash table into a list of (DNA-letter, frequency)
- ⑦ Return list of (DNA-letter, frequency)

In defining the `process_FASTA_partition()` function, we used a `defaultdict(int)`, which works exactly like a normal dictionary (as an

associated-array), but it is initialized with a function (“default factory”) that takes no arguments and provides the default value for a non-existent key. In our function, `process_FASTA_partition()`, a `defaultdict` is used for counting DNA-letters. The default factory is '`int`' (as an Integer data type), which in turn has a default value of zero. For each character in the list, the value is incremented by one where the key is the DNA letter. We do not need to make sure the DNA letter is already a key – it will use the default value of zero.

Step-3: Apply Custom Function to Each Partition

In this step, we apply the `process_FASTA_partition()` function to each partition. I have formatted output (Listing 4.5) and added some comments to show the output per partition (we have 2 partitions).

```
>>> records_rdd.getNumPartitions()
2
>>> pairs_rdd = records_rdd.mapPartitions(process_FASTA_partition)

>>># output for partition 1
hashmap= defaultdict(<type 'int'>,
{
    'a': 38, 'c': 28, 'g': 28,
    'n': 2, 't': 24, 'z': 3
})
key_value_list= [
    ('a', 38), ('c', 28), ('g', 28),
    ('n', 2), ('t', 24), ('z', 3)]

>>># output for partition 2
hashmap= defaultdict(<type 'int'>,
{
    'a': 35, 'c': 33,
    't': 21, 'g': 25, 'z': 1,
})
key_value_list= [
    ('a', 35), ('c', 33),
    ('t', 21), ('g', 25), ('z', 1),
]
```

Note that for solution Version 3, therefore, each partition returns at most 6 (key, value) pairs as:

```
('a', count-of-a)
('t', count-of-t)
('c', count-of-c)
('g', count-of-g)
('n', count-of-non-atcg)
('z', count-of-DNA-sequences)
```

Final collection from all partitions will be:

```
>>> pairs_rdd.collect()
[
    ('a', 38), ('c', 28), ('t', 24), ('z', 3),
    ('g', 28), ('n', 2), ('a', 35), ('c', 33),
    ('t', 21), ('g', 25), ('z', 1)
]
```

Finally, we aggregate and sum up the output (generated from the `mapPartitions()`) for all partitions:

```
>>> frequencies_rdd = pairs_rdd.reduceByKey(lambda a, b: a+b)
>>> frequencies_rdd.collect()
[
    ('a', 73),
    ('c', 61),
    ('g', 53),
    ('t', 45),
    ('n', 2),
    ('z', 4),
]
>>>
```

Pros and Cons of Solution 3

Pros:

- This is the most optimal solution for the DNA Base Count problem. The provided solution works, simple, and efficient. This solution improves on versions 1 and 2 by emitting the least number of (key,

(key, value) pairs, since we create a dictionary per partition (rather than each record) and then flatten it into a list of (key, value) pairs

- There is no scalability issue since we use `mapPartitions()` (for handling each partition) and `reduceByKey()` for reducing all (key, value) pairs emitted by partitions
- At most we will create N dictionaries, where N is the number of partitions for all input data (N can be in hundreds or thousands). This will not be a threat to scalability or memory outrage

Cons:

- None

Handling Empty Partitions

In our solution Version-3, we used the `mapPartitions(func)` transformation, which partitions input data into many partitions and then applies the function `func` (provided by the programmer) to each partition in parallel. But if any of these partitions are empty: it means that there is a partition, but there is no data (RDD elements) to iterate. We do need to write our custom function `func` (a.k.a partition handler) in such a way that to handle an empty partition properly and gracefully. You can not just ignore them.

Empty partitions, for example, may occur if there is an exception (corrupted records after a network failure mid-transfer) for a Spark partitioner in partitioning the data, then some partitions might be empty. Also, empty partitions may happen for many other reasons: one reason can be that the partitioner does not have enough data to put for a given partition. It is rare, but empty partitions can happen during data partitioning. Regardless of how empty partitions are created, we do need to handle all of them proactively and gracefully.

I will show the concept of an “empty partition” by a simple example. First, we define a function, `debug_partition()` (Listing 4.6 — to be used for testing and teaching only — not to be used in production environments), to show the content of each partition: this is the function, which will be invoked per partition.

```
def debug_partition(iterator):
    print("type(iterator)=", type(iterator))
    print("begin partition ===")
    elements = []
    for x in iterator:
        elements.append(x)
    print("elements=", elements)
    print("end partition ===")
#end-def
```

You should note that displaying or debugging content of a partition can be costly and should be avoided by all means in production environments. I have included `print` statements for teaching and debugging purposes only.

Next, we define an RDD and partition it into 4 partitions denoted by Listing 4.7.

```
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> rdd = sc.parallelize(numbers, 4)
>>> rdd.collect()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Next, we examine each partition by using the `debug_partition()` function:

```
>>> rdd.foreachPartition(debug_partition)
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
elements= [8, 9, 10]
end partition ===
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
elements= [5, 6, 7]
end partition ===
begin partition === ①
```

```

('type(iterator)=', <type 'itertools.chain'>)
elements=
end partition === ❷
begin partition ===
('type(iterator)=', <type 'itertools.chain'>)
elements= [1, 2, 3, 4]
end partition ===

```

❶ beginning of an empty partition

❷ end of an empty partition

From this test program we observe the following:

- A partition can be empty (with no RDD elements)
- Your custom function must handle empty partitions proactively and gracefully— must return a proper value; empty partitions can not be just ignored.
- The data type of `iterator` (which represents a single partition and passed as a parameter to the `mapPartitions()`) is the `itertools.chain`. The `itertools.chain` is an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence.

Now the question is how to handle an empty partition PySpark? The following pattern (Listing 2.8) may be used to handle an empty partition gracefully. The basic idea is to use Python's `try-except` combination, where the `try` block lets you test a block of code for errors and the `except` block lets you handle the error.

```

# This is the template function
# to handle a single partition
#
# source RDD: RDD[T]
#
# parameter: iterator

```

```

#
def func(iterator): ❶
    print("type(iterator)=", type(iterator))
    # ('type(iterator)=', <type 'itertools.chain'>)
    #
    try:
        first_element = next(iterator)      ❷
        # if you are here it means that
        # the partition is NOT Empty
        ... process the partition
        ... and return a proper result

    except StopIteration:                ❸
        # if you are here, it means that this
        # partition is Empty; now, you need
        # to handle it and return a proper result

```

- ❶ iterator represent a single partition of elements of type T
- ❷ Try to get the first element (as first_element of type T) for a given partition. If this fails (throws an exception), then control will go to the except (exception happened section).
- ❸ You will be here when a given partition is empty. You can not just ignore empty partitions, you must return a proper value

HANDLING EMPTY PARTITIONS

Typically, for empty partitions, you should return some special values, which can be filtered out easily by the filter() transformation. For example, for DNA Base Count problem, you may return a null value (instead of an actual dictionary) and then filter the null values after the completion of the mapPartitions() transformation.

DNA Base Count by PySpark — FASTQ Version

FASTQ Format

As I mentioned earlier in this chapter, DNA sequences can be represented in many different formats including FASTA and FASTQ popular text-based formats (input is given as a set of text files). In the following sections, DNA Base Count solution is given for the FASTQ format.

A FASTQ file normally uses four (4) lines per DNA sequence:

- Line 1 begins with a @ character and is followed by a sequence identifier and an optional description (like a FASTA title line).
- Line 2 is the raw sequence of DNA letters.
- Line 3 begins with a + character and is optionally followed by the same sequence identifier (and any description) again.
- Line 4 encodes the quality values for the sequence in Line 2, and must contain the same number of symbols as letters in the sequence.

Let's dig into our FASTQ example.

FASTQ Format Example

A FASTQ file containing a single sequence might look like this:

```
@SEQ_ID ①  
GATTTGGGTTCAAACCGAGTATCGATCAAATAGTAAATCCATTGTTCAACTCACAGTTT ②  
+ ③  
!'''*(((((***+))%%%++)(%%%)).1***-+*'')**55CCF>>>>CCCCCCC65 ④
```

- ① Line 1: DNA sequence ID
- ② Line 2: DNA sequence
- ③ Line 3: ignored for DNA Base Count
- ④ Line 4: the quality values for the DNA sequence

FASTQ Data Download

To test the DNA Base Count programs (provided in this chapter), you may download FASTQ data from the [SP1.fq](#). Some sample FASTQ files are provided in this book's [GIT repository](#).

High-Level Solution for FASTQ

Now, let's take a look at our DNA Base count example in PySpark using FASTQ data.

For FASTQ format, I only provide the most optimized solution using the `mapPartitions()` transformation. High-level solution is presented below:

Step-1:

read data and create an `RDD[String]`

Step-2:

drop out the non-needed records (keep only Line 2: DNA sequence)

Step-3:

for every 4 records, just keep Line 2 (the record of the raw sequence of DNA letters).

Step-4:

apply the `mapPartitions()` transformation to perform DNA Base Counting

Since most of the steps for FASTQ solution is similar with FASTA solution, I just provide a complete solution without details in `dna_base_count_FASTQ.py` in this book's [GIT repository](#).

TEST WITH REAL DATA FOR PRODUCTION

In order to deploy your Spark applications to production environment, your program should be tested with real data (the actual size used in production). Also, try different transformations (for example `mapPartitions()` versus `map()`) to find optimized solutions.

Summary

- In solving big data problems, we have a lot of choices in selecting Spark's transformations and actions and their performances are different from each other.
- When selecting and using a transformation to solve a specific data problem, make sure that you test it with “real big data” rather than toy data
- For large volume of (key, value) pairs, overall, the `reduceByKey()` transformation performs better than `groupByKey()` due to different shuffling algorithms
- When you have a big data and you want to just extract and aggregate/derive small amount of information (such as Min-Max, Top-10, DNA Base Count, ...), then the `mapPartitions()` transformation may be used.
- Emitting minimal number of (key, value) pairs improves the performance of your data solutions. This reduces the time for the sort and shuffle phase of your Spark application.

Chapter 3. Mapper Transformations

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

The main purpose of this chapter is to explain the most common Spark mapper transformations by simple working examples. Without understanding these transformations, it is hard to use them in a proper and meaningful way to solve any data problem. We will examine mapper transformations in the context of RDD (Resilient Distributed Dataset — low level Spark data abstraction) data abstractions. In a nutshell, mapper is a function which is used to process all elements of a source RDD and generate the target RDD. For example, a mapper can transform a string record into tuples, (key, value) pairs, or whatever is the desired output. Informally, we can say that a mapper transforms a source $\text{RDD}[V]$ into a target $\text{RDD}[T]$, where V and T are the data type of source and target RDDs respectively.

This chapter covers

- Spark’s mapper transformations with PySpark

- Proper use of mappers using PySpark
- Examples of mappers transformations in PySpark

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 3](#).

Data Abstractions and Mappers

There are so many Spark's transformations and actions, but this chapter is limited to explaining the most common ones, which are used quite often in building Spark applications. Spark's simple and powerful mapper transformations enable us to perform ETL (Extract, Transform and Load) in a simple way.

RDD is an important data abstraction (see Figure 3.1) in Spark, which is suitable for unstructured and semi structured data. An RDD (Resilient Distributed Dataset) is the basic abstraction in Spark, which represents an immutable, partitioned collection of elements that can be operated on in parallel. RDD is a lower level API than a DataFrame. In RDD, each element may have a data type T denoted by $\text{RDD}[T]$.

In every data solution, we use mappers transformations to convert one form of data into another desired from of data (for example, convert a record (as a String) into a (key, value) form). Spark provides 5 important mapper transformations, which are used heavily in RDD transformations.

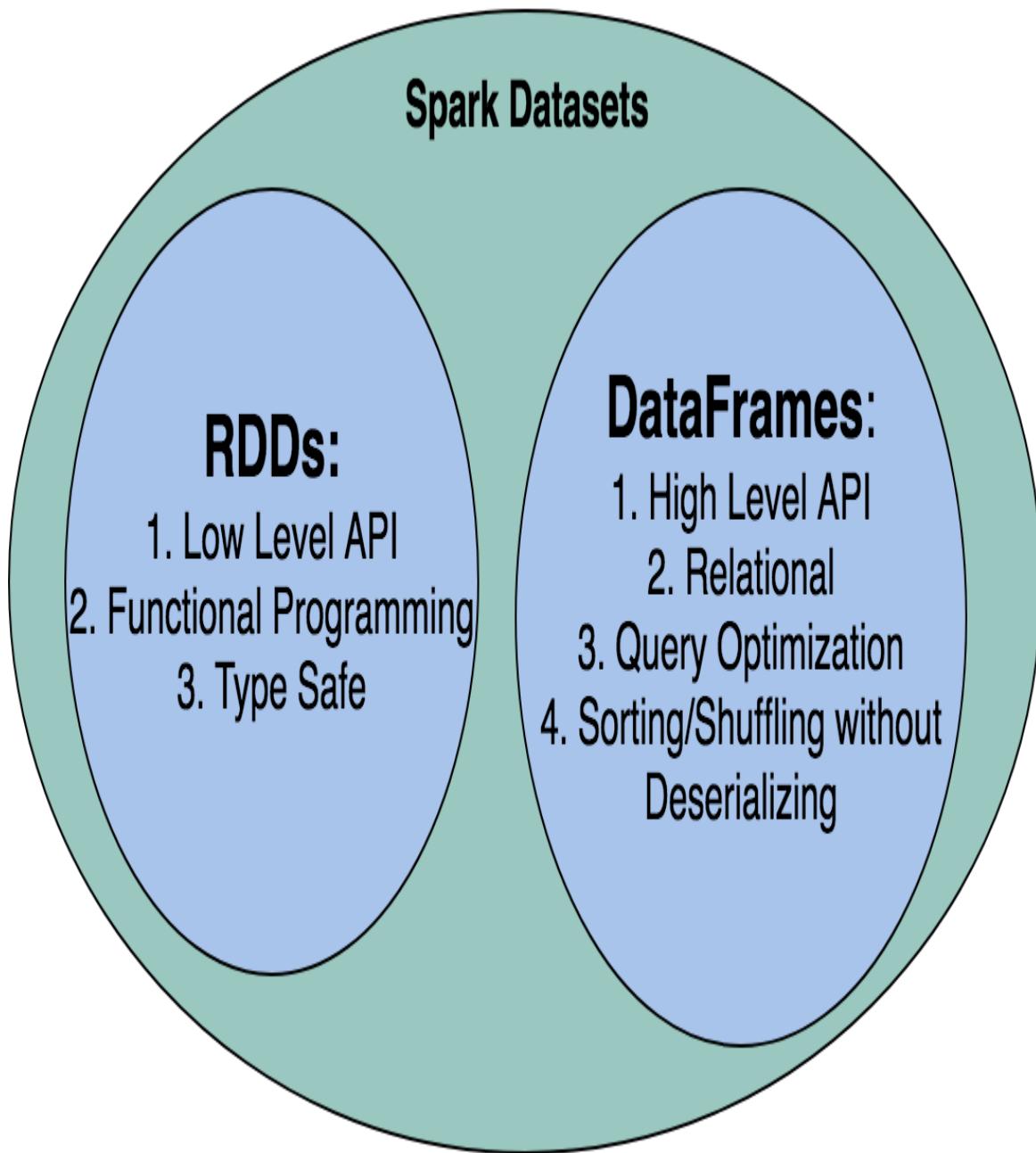


Figure 3-1. Spark's Data Abstractions

In this chapter, I will only discuss the most common and important Spark's mapper transformations. The 5 important mapper transformations are:

T
a
b
l
e

3
-
l
. *M*

a
p
p
e
r
T
r
a
n
s
f
o
r
m
a
t
i
o
n
s

Transformation	Relation Type	Description
-----------------------	----------------------	--------------------

<code>map(f)</code>	1-to-1	Return a new RDD by applying a function (<code>f()</code>) to each element of this RDD. Source and target RDDs will have the same number of elements.
<code>mapValues(f)</code>	1-to-1	Pass each value in the key-value pair RDD through a <code>map(f)</code> function without changing the keys; this also retains the original RDD's partitioning. Source and target RDDs will have the same number of elements.
<code>flatMap(f)</code>	1-to-Many	Return a new RDD by first applying a function (<code>f()</code>) to all elements of this RDD, and then flattening the results. Source and target RDDs might not have the same number of elements.
<code>flatMapValues(f)</code>	1-to-Many	Pass each value in the key-value pair RDD through a <code>flatMap(f)</code> function without changing the keys; this also retains the original RDD's partitioning. Source and target RDDs might not have the same number of elements.
<code>mapPartitions(f)</code>	Many-to-1	Return a new RDD by applying a function (<code>f()</code>) to each partition of this RDD. Source and target RDDs might not have the same number of elements.

What are Transformations?

Before we look into definitions of these mappers transformations, let's understand the meaning of a transformation. What is the meaning of a “*trans·for·ma·tion*”? Transformation is defined as “*a thorough or dramatic change in form or appearance.*” This definition exactly applies to the semantics of Spark transformations, which transforms data from one form to another. For example a `map()` transformation can map take a record of movie information (as a string object of `user_id`, `username`, `movie_name`, `movie_id`, `rating`, `timestamp`, `director`, ...) into a triplet of (`user_id`,

`movie_id, rating`). Another example of a transformation can be: convert a chromosome “chr7:890766:T” into a tuple of (`chr7, 890766, T, 47`), where 47 (as a derived partition number) is a $890766 \% 101$ (modulo of `101`).

Some basic Spark operations (transformations and actions) are illustrated by the Figure 3.2.

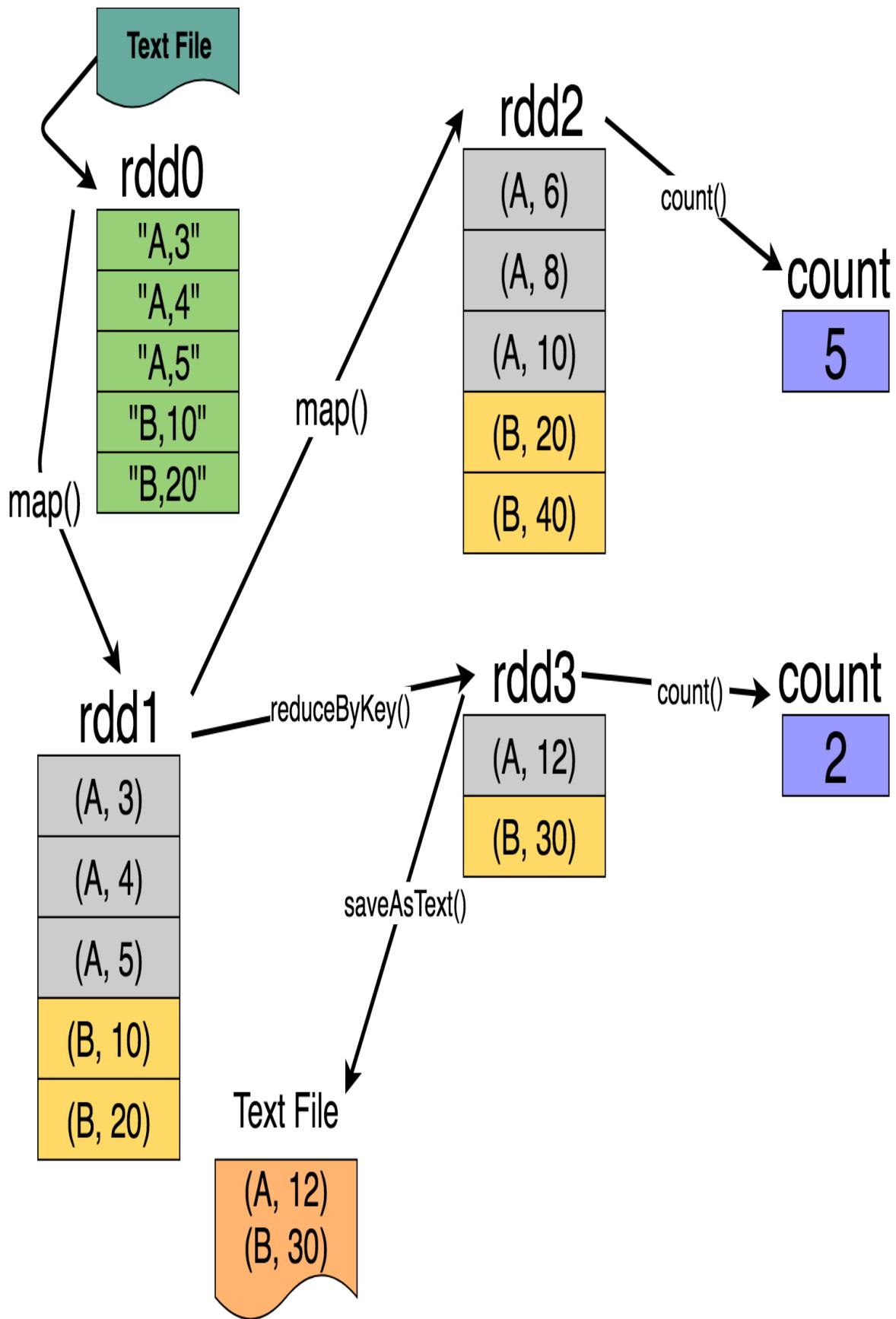


Figure 3-2. Spark operations

Let's walk through what's happening in Figure 3-2.

- **RDDs:** four RDDs are created: `rdd0`, `rdd1`, `rdd2`, and `rdd3`. The `rdd0` (`RDD[String]`) is the first RDD created from a text file:
- **Transformation:** `SparkSession.sparkContext.textFile()` reads our input and creates the first RDD as `rdd0`. The `rdd0` is denoted as `RDD[String]`: means that each element of `rdd0` is an `String` object

```
input_path = "sample_5_records.txt"
rdd0 = spark.sparkContext.textFile(input_path)
```

- **Transformation:** `rdd1` (as `RDD[(String, Integer)]`) is created from `rdd0.map()` transformation: each element of `rdd0` is mapped into (key, value) pairs

```
def create_pair(record):
    tokens = record.split(",")
    return (tokens[0], int(tokens[1]))
#end-def

rdd1 = rdd0.map(create_pair)
```

- **Transformation:** `rdd2` (as `RDD[(String, Integer)]`) is created from `rdd1.map()`, where the mapper doubles the value part of the (key, value) pairs

```
rdd2 = rdd1.map(lambda (k, v): (k, v+v))
-- OR --
rdd2 = rdd1.mapValues(lambda v: v+v)
```

- **Transformation:** `rdd3` (as `RDD[(String, Integer)]`) is created from `rdd1.reduceByKey()`, where the reducer sums up the values of the same keys

```
rdd3 = rdd1.reduceByKey(lambda x, y: x+y)
```

- **Action:** `rdd2.count()` is called to count the number of elements of `rdd2` (result is an integer number, non-RDD)

```
rdd2_count = rdd2.count()
```

- **Action:** `rdd3.count()` is called to count the number of elements of `rdd3` (result is an integer number, non-RDD)

```
rdd3_count = rdd3.count()
```

- **Action:** `rdd3.saveAsText()` is called to persist the content of `rdd3` into a file system (result is a set of output files, non-RDD)

```
rdd3.saveAsText("/tmp/rdd3_output")
```

Informally, Spark's operations (transformations and actions) can be stated as:

```
\begin{align*}
\texttt{action} : \texttt{RDD} \rightarrow \texttt{non-RDD} \\
\texttt{transformation} : \texttt{RDD} \rightarrow \texttt{RDD} \\
\texttt{transformation} : \texttt{RDD} \rightarrow \texttt{Sequence[RDD]}
\end{align*}
```

- Most of the RDD transformations accept a single source RDD and create a single target RDD. A transformation, transforms a source RDD (as `RDD[V]`) to a target RDD (as `RDD[T]`). Note that `V` and `T` can be any different data types.
- Only some of Spark transformations create multiple target RDDs

- Actions create a **non-RDD** element (values such as Integers, String, lists, tuples, dictionaries, and files)

Spark transformation concept is illustrated in Figure 3.3 — where a “Source RDD” is transformed into a “Target RDD”. The Figure 3.3 illustrates the `map(f)` transformation, where each `element` of the “Source RDD” is converted to `f(element)`, where `f()` is a user-defined function.

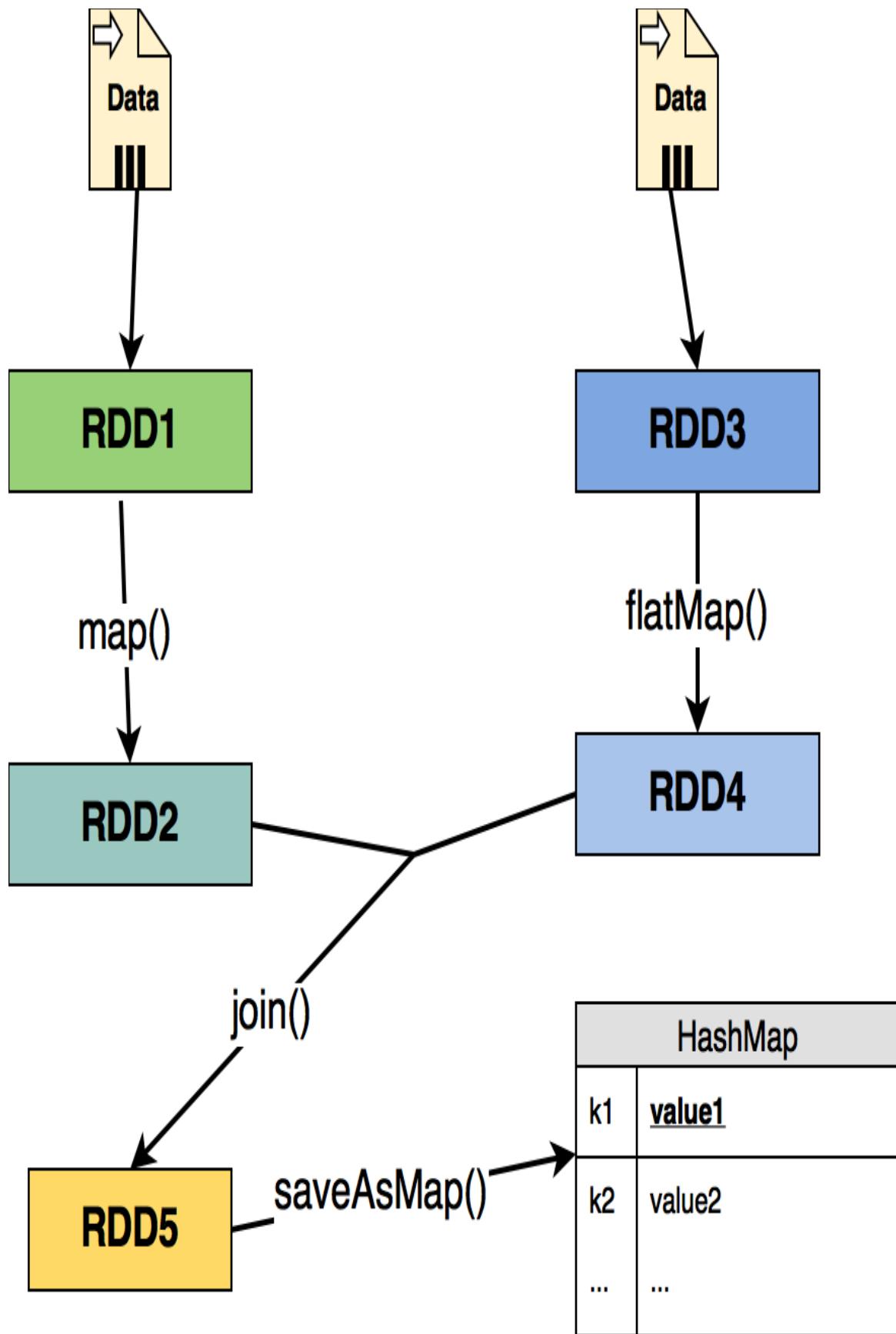


Figure 3-3. Spark Transformation

Let's walk through what's happening in Figure 3-3.

- RDD1 and RDD3 are created from text files
- RDD2 is created by a map() transformation:

```
# source RDD: RDD1[U]
# target RDD: RDD2[V]
RDD2 = RDD1.map(func1)
```

- RDD4 is created by a flatMap() transformation:

```
# source RDD: RDD3[C]
# target RDD: RDD4[D]
RDD4 = RDD3.flatMap(func2)
```

- RDD5 is created by joining two RDDs (RDD2 and RDD4) — return an RDD5 containing all pairs of elements with matching keys in RDD2 and RDD4:

```
# source RDDs: RDD2, RDD4
# target RDD: RDD5
RDD5 = RDD2.join(RDD4)
```

- Finally, the RDD5 elements are saved as a set of text files.

```
# source RDD: RDD5
# action: saveAsMap()
# target: hashmap : dictionary
hashmap = RDD5.saveAsMap()
```

Until the `saveAsMap()` action is executed, no transformation will be evaluated/executed: this is called **lazy evaluation**: which gives a chance to the Spark engine to perform optimizations as much as needed.

A very important note on RDDs and Transformations is the “**lazy evaluation**” concept: transformations are lazily evaluated — they are not executed when

you call the transformation command. RDDs are recomputed when an action is executed. When we say that “transformations are lazy”, we mean that RDDs are computed only when an “action” (such as `count()`, `collect()`, `saveAsTextFile()`, `collectAsMap`, ...) is called/triggered on them. Therefore, lazy evaluation in Spark means that the transformation execution will not start until an action is triggered.

Creating New RDDs

How do you create a brand new RDD? There are at least 3 ways to create a brand new RDD:

1. RDDs can be created from data files: you can use
 - `SparkContext.textFile()` or
 - `SparkSession.spark.read.text()`to read data files, which can be from Amazon S3, Hadoop distributed file system, Linux file system, and many other data sources
2. RDDs can be created from collections (such as a list data structure — for example, a list of numbers, a list of strings, a list of pairs, ...); you can use
 - `SparkContext.parallelize()`
3. Given a source RDD, you can apply a transformation (such as `map()` or `filter()`) to create a brand new RDD; Spark offers many useful transformations, which is the discussion of this chapter.

The relationship between of RDDs and Spark operations (transformations and actions) are illustrated below (Figure 3.4).

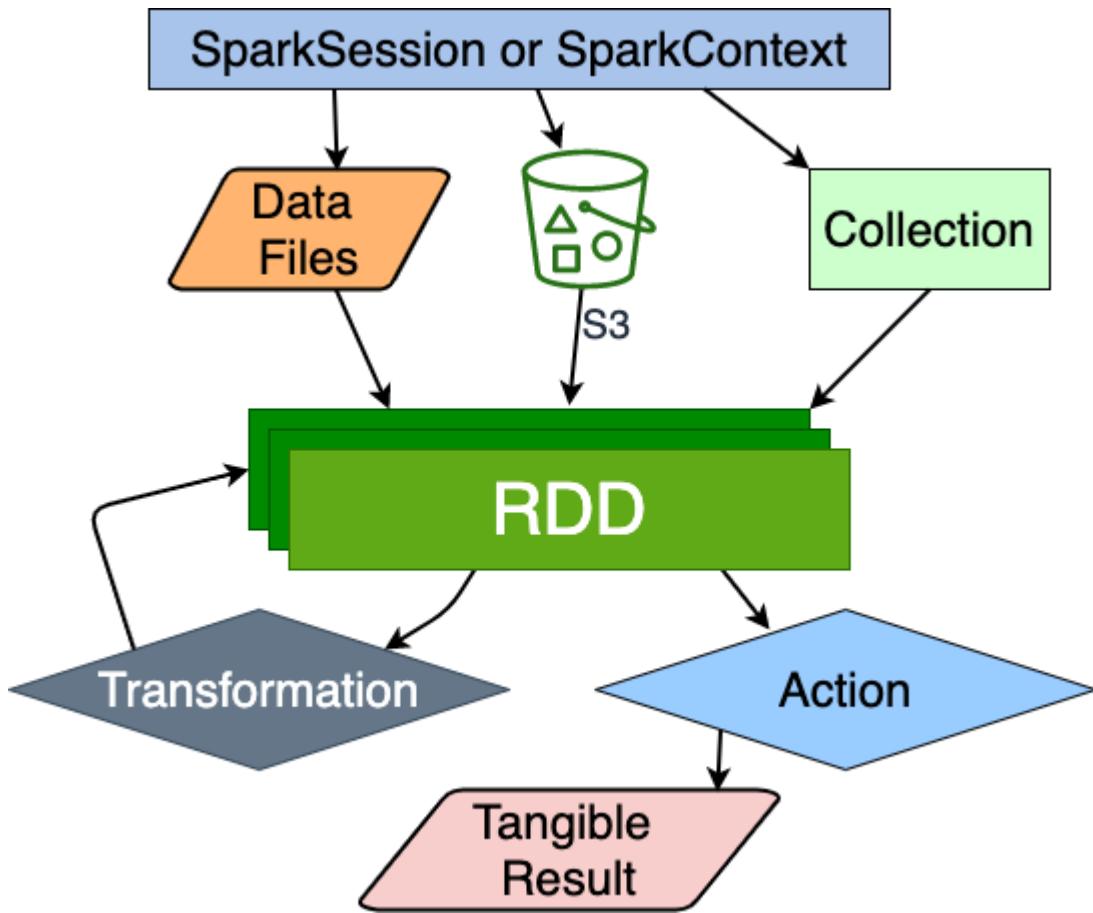


Figure 3-4. Spark Operations and RDDs

The following are some important facts regarding RDDs, Transformations, and Actions:

- A new RDD can be created from a text file or a collection using

```

SparkContext.textFile(files)
SparkContext.parallelize(collection)

```

```

SparkSession.parallelize(collection)
SparkSession.read.text(files).rdd

```

Informally, the `textFile()` and `parallelize()` operations can be stated as:

```
parallelize : collection --> RDD[T]
# where T is the type of collection elements

textFile : file(s) --> RDD[String]
# reading text files always create RDD[String]
```

- A transformation (such as `map()` or `filter()`) on a source RDD (source element type of U) creates a new RDD (target element type of V),

```
transformation : RDD[U] --> RDD[V]
where
U: data type of source RDD elements
V: data type of target RDD elements
```

- An action (such as `collectAsMap()` or `count()`) on a source RDD creates a tangible result (non-RDD) such as integer, string, list, file, or dictionary

```
action : RDD[U] --> non-RDD
```

Let's dig a little deeper into Spark's lazy transformations.

Lazy Transformations

When running a Spark application (in Python, Java or Scala), for transformations, Spark creates a DAG (Directed Acyclic Graph) and adds transformations to a DAG of computation only when driver requests some data (such as saving RDD as a List or Hash Map), does this DAG actually gets executed. Remember that in Spark, transformations are lazily evaluated, which means that the execution will not start until an action (such as `collect()`, `count()`, ...) is triggered. Spark engine can make optimization decisions after it had a chance to look at the DAG in entirety rather than looking at the individual transformations and actions. For example, it is

possible to write a Spark program which creates 10 RDDs, but 3 of them are never used (non-reachable RDDs): Spark engine does not need to compute those 3 non-needed RDDs and hence speeds up the computation, which reduces total execution time.

Spark's DAG (Directed Acyclic Graph) visualization is illustrated below. DAG in Apache Spark is a set of vertices and edges, where vertices represent the RDDs and the edges represent the Operation (transformation or action) to be applied on RDD. In Spark DAG, every edge is directed from earlier to later in the sequence. On calling of an action (such as `saveAsMap()`, `count()`, `collect()`, or `collectAsMap()`) the created DAG is submitted to Spark's DAG Scheduler which further splits the graph into the stages of the task. The Spark's DAGs can be viewed from this URL: [http:<master>:4040](http://<master>:4040), where the port number 4040 is configurable by Spark's configuration files.

▼ DAG Visualization

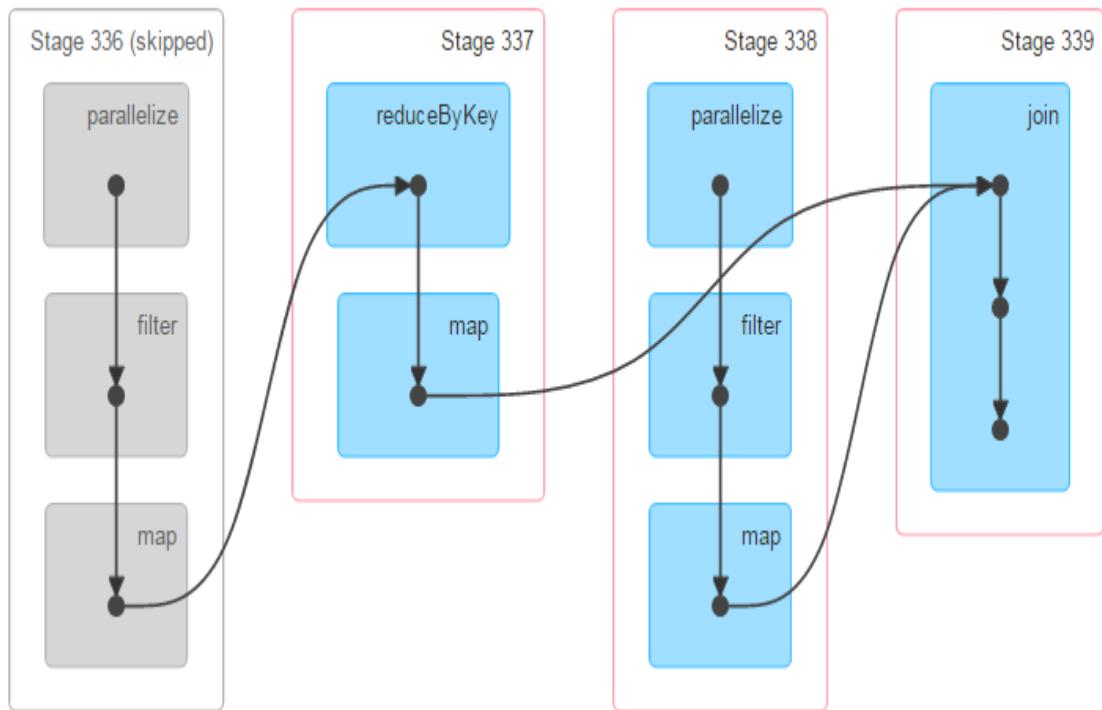


Figure 3-5. Spark's DAG

What are the benefits of the Lazy evaluation in Spark? The following are some benefits of Lazy evaluations in Spark:

- Increases manageability of transformations
- Saves computation and increases speed
- Reduces complexities
- Optimization of transformations

According to [RDD Programming Guide](#): “*all transformations in Spark are lazy*, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through `map()` will be used in a `reduce()` and return only the result of the `reduce()` to the driver, rather than the larger mapped dataset.” In conclusion we can say that: lazy evaluation enhances the power of Spark by reducing the execution time of the RDD operations (transformations and actions). It maintains the lineage graph to remember the operations on RDD. As a result, it optimizes the performance and achieves fault tolerance.

Next, we dig into the meanings of Spark’s most used mapper transformations.

Semantics of Mapper Transformations

I opened this chapter by providing a list of the important mapper transformations that I’d be covering in this chapter. Before we start digging into each one, let’s go over that list again. Brief descriptions of mappers transformations are provided below:

map(f)

Spark’s `map()` function is a **one-to-one** transformation. It transforms each element of the source $\text{RDD}[V]$ into one element of the resulting target

$\text{RDD}[T]$. If source RDD has N elements, then the target RDD will have exactly N elements.

mapValues(f)

Pass each value in the key-value pair $\text{RDD}[(K,V)]$ through a `map()` function without changing the keys; this also retains the original RDD's partitioning. The result will be $\text{RDD}[(K,T)]$ (V is mapped to T by function $f()$).

flatMap()

Spark's `flatMap` function is a **one-to-many** transformation. It transforms each element of source $\text{RDD}[V]$ to 0 or more elements of target $\text{RDD}[T]$.

mapPartitions()

Spark's `mapPartitions()` function is a **one-to-one** transformation. It transforms each **partition** (comprised of many elements, in hundreds, thousands, and millions) of the source $\text{RDD}[V]$ into one element of the resulting target $\text{RDD}[T]$. This transformation implements a Summarization Design Pattern, which summarizes many records/elements into a target data structure such as lists, dictionaries, or tuples.

Next, I discuss the `map()` transformation, which is the most widely used transformation in any Spark application.

The `map()` Transformation

The `map()` transformation is the most common transformation in Spark and MapReduce paradigm. The goal of `map()` is to map/transform/convert every element of the source $\text{RDD}[V]$ into a mapped element of the target $\text{RDD}[T]$ by using a function (the function can be a predefined such as an `add` operator or a custom user-defined function). The `map()` transformation is defined as:

```

pyspark.RDD.map (Python method)
map(f, preservesPartitioning=False)

f: V --> T ❶
map: RDD[V] --> RDD[T] ❷

```

- ❶ Function `f()` accepts a `V` type element and returns an element of type `T` (this function is a 1-to-1)
- ❷ Using function `f()`, the `map()` transformation transforms `RDD[V]` to `RDD[T]`

The `map()` transformation returns a new RDD by applying a function `f()` to each element of the source RDD. If your source RDD has `N` elements, then the resulting/ target RDD will have exactly `N` elements as well. The `map()` transformation is a 1-to-1 mapping from source RDD of type `V` to the resulting/target RDD of type `T`. Bear in mind that the `map()` transformation is not a sequential function. Your source RDD is partitioned into `P` partitions and then processed independently and concurrently. For example, if your source RDD has 40 billion elelemnts and `P` is 20,000, then each partition will have roughly 2 million elements ($40 \text{ billion} = 20,000 \times 2 \text{ million}$). If the number of avaiable mappers are 80 (this number depends on the available resources in your cluster), then 80 partitions can be mapped at the same time independently and concurrently.

Informally, `map()` can be defined as:

```

target_rdd = source_rdd.map(f)

source_rdd:RDD[V] -- map(f) --> target_rdd:RDD[T]
element-1           f(element-1)
element-2           f(element-2)
...
element-N          f(element-N)

```

The function `f()` for the `map()` transformation can be defined as:

```

# v : a data type of V
# return an object of type T
def convert_V_to_T(v):
    t = <convert v to an object of data type T>
    return t
#end-def

# source RDD: source_rdd : RDD[V]
# target RDD: target_rdd : RDD[T]
target_rdd = source_rdd.map(convert_V_to_T)

```

or you may create your target RDD (`rdd_v`) by using Lambda Expression as:

```
target_rdd = source_rdd.map(lambda v : convert_V_to_T(v))
```

The Figure 3.7 demonstrates the semantical nature of the `map()` transformation.

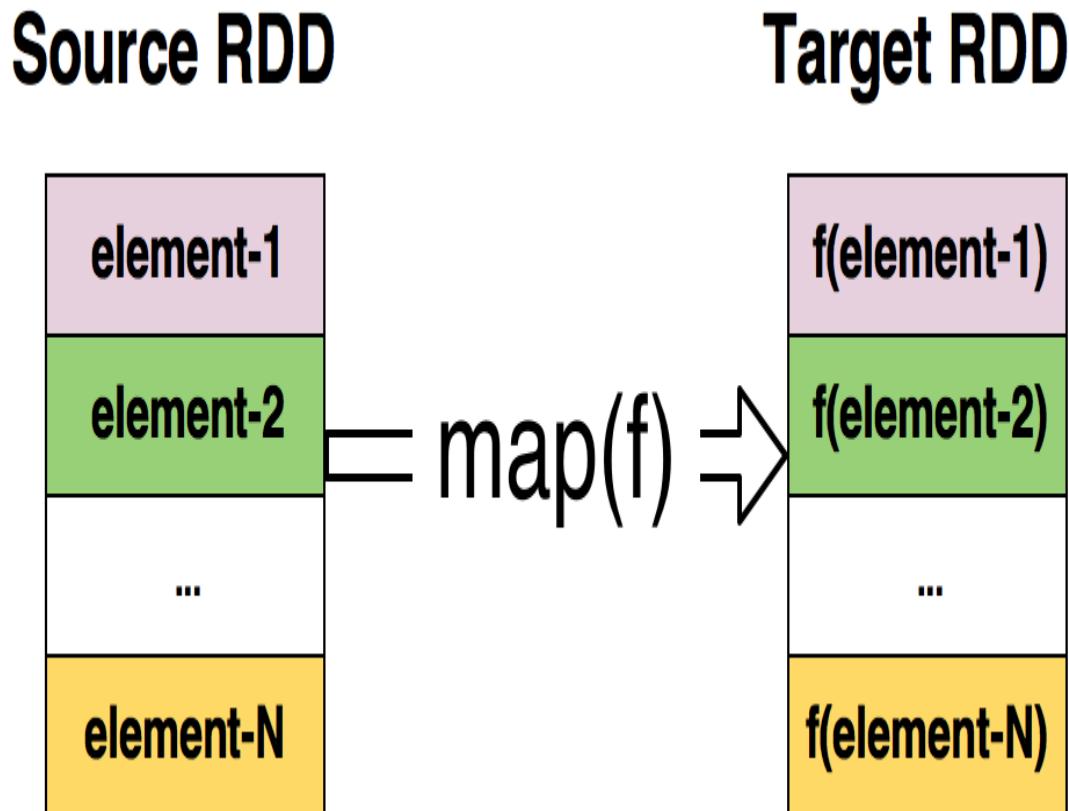


Figure 3-6. The `map()` Transformation

Using PySpark shell, the following example shows how to use the `map()` transformation: this example maps source `RDD[Integer]` to target `RDD[Integer]`. This example transforms an RDD which contains a list of numbers to a new RDD which has every positive element increased by 5 and every 0 and negative element changed to 0.

First, let's define a mapper function as `mapper_func()`:

```
>>># define a simple mapper function
>>> def mapper_func(x):
...     if (x > 0):
...         return x+5
...     else:
...         return 0
>>>#end-def
```

Next, let's apply a `map()` transformation and see how it works:

```
>>># spark : SparkSession
>>> data = [1, -1, -2, 3, 4]
>>> rdd = spark.sparkContext.parallelize(data) ❶
>>> rdd.collect()
[1, -1, -2, 3, 4]
>>># use lambda expression
>>> rdd2 = rdd.map(lambda x : mapper_func(x)) ❷
>>> rdd2.collect()
[6, 0, 0, 8, 9]
>>># do not use lambda expression, use function
>>> rdd3 = rdd.map(mapper_func) ❸
>>> rdd3.collect()
[6, 0, 0, 8, 9]
>>>
>>> rdd4 = rdd.map(lambda x : (x, mapper_func(x))) ❹
>>> rdd4.collect()
[(1, 6), (-1, 0), (-2, 0), (3, 8), (4, 9)]
>>> rdd4.count()
5
```

❶ `rdd` is `RDD[Integer]`

❷ `rdd2` is `RDD[Integer]`

- ③ rdd3 is RDD[Integer]
- ④ rdd4 is RDD[(Integer, Integer)]

Using PySpark shell, the following example shows how to use the `map()` transformation: this example maps RDD[(String, Integer)] to RDD[(String, Integer, String)]. This example transforms element of the form (key, value) pairs into (key, value, value+100) triplets.

```
>>> pairs = [('a', 2), ('b', -1), ('d', -2), ('e', 3)]
>>> rdd = spark.sparkContext.parallelize(pairs) ❶
>>> rdd.collect()
[('a', 2), ('b', -1), ('d', -2), ('e', 3)]
>>> rdd2 = rdd.map(lambda (k, v) : (k, v, v+100)) ❷
>>> rdd2.collect()
[
  ('a', 2, 102),
  ('b', -1, 99),
  ('d', -2, 98),
  ('e', 3, 103)
]
```

- ❶ rdd is RDD[(String, Integer)]
- ❷ rdd2 is RDD[(String, Integer, Integer)]

Also, it straightforward to create (key, value) pairs from string objects:

```
>>> def create_key_value(string):
>>>     tokens = string.split(",")
>>>     return (tokens[0], (tokens[1], tokens[2]))
>>>
>>> strings = ['a,10,11', 'b,8,19', 'c,20,21', 'c,2,8']
>>> rdd = spark.sparkContext.parallelize(strings) ❶
>>> rdd.collect()
['a,10,11', 'b,8,19', 'c,20,21', 'c,2,8']
>>> pairs = rdd.map(create_key_value) ❷
>>> rdd2.collect()
[
  ('a', (10, 11)),
  ('b', (8, 19)),
  ('c', (20, 21)),
  ('c', (2, 8))
]
```

```
('b', (8, 19)),  
('c', (20, 21)),  
('c', (2, 8))  
]  
>>>
```

- ❶ rdd is RDD[String]
- ❷ rdd2 is RDD[(String, Integer, Integer)]

Nex, I discuss custom mapper functions, where a programmer can supply any defined function.

Custom Map Functions

In using Spark's transformations, you may use custom Python functions to parse lines, perform computations, and finally create your desired collections, data structures, and tuples.

Suppose we have this sample data, where each record has the following format:

- Record format:

```
<id>,<name>,<age>,<number-of-friends>
```

- File content:

```
$ cat /tmp/users.txt  
1,Alex,30,124  
2,Bert,32,234  
3,Curt,28,312  
4,Don,32,180  
5,Mary,30,100  
6,Jane,28,212
```

```
7,Joe,28,128  
8,Al,40,600
```

For each age category, we want to get the average number of friends. You can write your own custom mapper function like this:

```
# record=<id>,<name>,<age>,<number-of-friends>  
# parse record and return a pair as (age, number_of_friends)  
def parse_record(record):  
    # split record into a list at comma positions  
    tokens = record.split(",")  
    # extract and typecast relevant fields  
    age = int(tokens[2])  
    number_of_friends = int(tokens[3])  
    return (age, number_of_friends)  
#end-def
```

Next, we read our data and use the custom function:

```
users_path = '/tmp/users.txt'  
users = spark.sparkContext.textFile(users_path) ①  
pairs = users.map(parse_record) ②
```

- ① `users` as `RDD[String]`
- ② `pairs` as `RDD[(Integer, Integer)]`, where each record gets sent through `parse_record()`

For our sample data, `pairs` (as `RDD[(Integer, Integer)]`) will be

```
(30, 124), (32, 234), (28, 312), (32, 180),  
(30, 100), (28, 212), (28, 128), (40, 600)
```

To get the average per age category, we first get the sum total and the number of entries per age.

```
totals_by_age = pairs \ ①  
.mapValues(lambda x: (x, 1)) \ ②  
.reduceByKey(lambda x, y: (x[0] + y[0], x[1] + y[1])) ③ ④
```

- ① pairs as $\text{RDD}[(\text{Integer}, \text{Integer})]$
- ② convert `number_of_friends` field to $(\text{number_of_friends}, 1)$ pair
- ③ perform reduction on age to find $(\text{total-sum-of-friends}, \text{frequency-count-of-friends})$ per age,
- ④ `totals_by_age` is $\text{RDD}[(\text{Integer}, (\text{Integer}, \text{Integer}))]$

For our data, `totals_by_age` (as $\text{RDD}[(\text{Integer}, (\text{Integer}, \text{Integer}))]$) will be

```
(30, (124+100, 1+1))      --> (30, (224, 2))
(32, (234+180, 1+1))      --> (32, (414, 2))
(28, (312+212+128, 1+1+1)) --> (28, (652, 3))
(40, (600, 1))              --> (40, (600, 1))
```

Now, for each age, to compute average number of friends, we do need one more transformation: divide the total sum by the total count to get the average.

```
# x = (sum_of_friends, frequency_count)
# x[0] = sum_of_friends
# x[1] = frequency_count
averages_by_age = totals_by_age.mapValues(lambda x: float(x[0]) / float(x[1]))
averages_by_age.collect()
```

For our data, `averages_by_age` (as $\text{RDD}[(\text{Integer}, \text{Integer})]$) will be

```
(30, (224 / 2)) = (30, 112)
(32, (414 / 2)) = (32, 207)
(28, (652 / 3)) = (28, 217)
(40, (600 / 1)) = (40, 600)
```

The `map()` transformation is the most common transformation in Spark and almost any application uses multiple of it. The main idea of `map()` transformation is to convert $\text{RDD}[U]$ to $\text{RDD}[T]$: the map function converts an

element of source RDD of type U to a target element of of RDD of type T. Note that data types U and T can be the same or completely different.

Next, I discuss the **flatMap()** transformation, which returns a new target RDD by first applying a function to all elements of the source RDD, and then flattening the results. As it can be guessed by its name, is the combination of a **map()** and a **flat()** operation. That means that you first apply a function to your source RDD elements, and then flatten it — flattened elements become the elements of the target RDD. Note that **map()** only applies a function to the source RDD elements without flattening the stream.

The **flatMap()** Transformation

The **flatMap()** transformation takes one-by-one each element of the source RDD as input and processes it according to a custom code or function (specified by the programmer) and returns 0, 1, 2, or more elements at a time and **flattens** them. The **flatMap()** transforms an RDD[U] of length N into target RDD[V] of length M (note that M and N can be different), where U is the data type of source RDD elements and V is the data type of target RDD elements. Using **flatMap()**, you need to make sure that source RDD elements are iterable (such as a list of items).

For example, if an element of source RDD is [10, 20, 30] (which is an iterable — list of 3 numbers), then it will be mapped as 3 elements (namely, 10, 20, and 30) of target RDD; and if an element of source RDD is [] (as an empty list, which is iterable), then it will be dropped and will not be mapped to target RDD at all. Furthermore, if any element of source RDD is not iterable, then an exception will be raised.

Note that the **map()** transforms an RDD of length N into another RDD of length N (the same length). But the **flatMap()** transforms an RDD of length N into a set of N iterable collections, then flattens these into a single RDD of results. Note that the empty collections are dropped from the target result. So the source and target RDDs may have different sizes.

```
pyspark.RDD.flatMap (Python method)
flatMap(f, preservesPartitioning=False)
```

U: iterable collection of V

source RDD: RDD[U]

target RDD: RDD[V]

f: U --> [V] ①

flatMap: RDD[U] --> RDD[V]

- ① Function f() accepts a U type element and converts it to a list of elements of type V (this list may have 0, 1, 2, or more elements), which then flattened; note that empty lists are dropped. The Function f() must create an iterable object.

The flatMap() transformation returns a new RDD by first applying a function to all elements of this RDD, and then **flattening** the results. The flatMap() is a **1-to-many** transformation: means that every element of the source RDD can be mapped into 0, 1, 2, or many elements of the target RDD. The flatMap() transformation example is illustrated below.

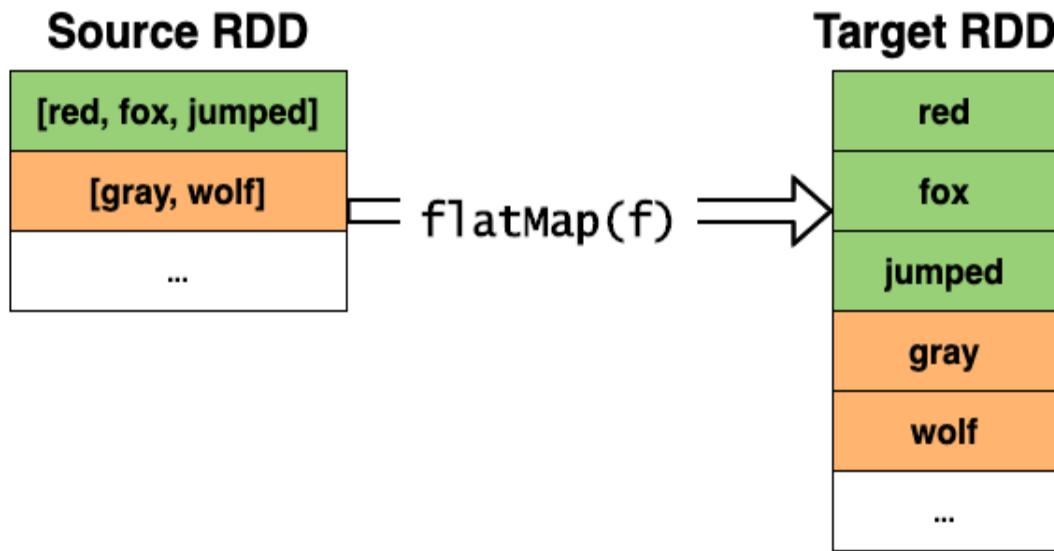


Figure 3-7. The flatMap() Transformation

In Figure 5.10, each element (as a String) of source RDD is tokenized into a list of Strings and then flattened into String object. For example the first

element "red fox jumped" is converted into a list of Strings as ["red", "fox", "jumped"] and then the list is flattened into 3 String objects as "red", "fox", and "jumped". Therefore, the first source element is mapped into 3 target elements.

The following example shows how to use the `flatMap()` transformation:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> rdd = spark.sparkContext.parallelize(numbers)
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> rdd2 = rdd.flatMap(lambda x: range(1, x))
>>> rdd2.collect()
[1, 1, 2, 1, 2, 3, 1, 2, 3, 4]
>>>
>>> rdd3 = rdd.flatMap(lambda x: [(x, x+1), (x+1, x)])
>>> rdd3.collect()
[
  (1, 2), (2, 1),
  (2, 3), (3, 2),
  (3, 4), (4, 3),
  (4, 5), (5, 4),
  (5, 6), (6, 5)
]
>>> rdd3.count()
10
```

Let's examine how `rdd2` is created:

```
1 --> range(1, 1) --> []
--> dropped since empty

2 --> range(1, 2) --> [1]
--> maps into one element as 1

3 --> range(1, 3) --> [1, 2]
--> maps into two elements as 1, 2

4 --> range(1, 4) --> [1, 2, 3]
--> maps into three elements as 1, 2, 3

5 --> range(1, 5) --> [1, 2, 3, 4]
--> maps into four elements as 1, 2, 3, 4
```

You may also use a function instead of using a `lambda` expression:

```
>>> numbers = [1, 2, 3, 4, 5]
>>> rdd = spark.sparkContext.parallelize(numbers) ❶
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> def create_list(x):
...     return [(x, x+1), (x, x+2)]
>>>#end-def
...
>>> rdd4 = rdd.flatMap(create_list) ❷
>>> rdd4.collect()
[
(1, 2), (1, 3),
(2, 3), (2, 4),
(3, 4), (3, 5),
(4, 5), (4, 6),
(5, 6), (5, 7)
]
>>> rdd4.count()
10
```

❶ `rdd` is `RDD[Integer]` and has 5 elements

❷ `rdd4` is `RDD[(Integer, Integer)]` and has 10 elements

The following example shows how to return 0, 1, 2, or many elements by using the `flatMap()`:

```
>>> def my_flatmap_func(x):
...     if x == 0:
...         return [(0, 0)]
...     elif x < 0:
...         return []
...     else:
...         return [(x, 1), (x, 2), (x, 3)]
>>>#end-def
>>> numbers = [0, 0, 100, -200, 300, 400, -500]
>>> rdd = spark.sparkContext.parallelize(numbers) ❶
>>> rdd.collect()
[0, 0, 100, -200, 300, 400, -500]
>>> flattened = rdd.flatMap(my_flatmap_func) ❷
>>> flattened.collect()
```

```

[
(0, 0),
(0, 0),
(100, 1), (100, 2), (100, 3),
(300, 1), (300, 2), (300, 3),
(400, 1), (400, 2), (400, 3)
]
>>>

```

- ❶ `rdd` is `RDD[Integer]` and has 7 elements
- ❷ `flattened` is `RDD[(Integer, Integer)]` and has 11 elements; note that negative numbers are dropped since an empty list is returned for them

For the preceding example, as you can note, when the source RDD element is equal 0, we returned one element, when RDD element is negative then we returned zero (no) elements (an empty list), and finally when the RDD element is positive, we returned 3 elements. The `flatMap()` transformation performs maps and then flattens the results.

The following example clearly shows the difference between `map()` and `flatMap()`: as you can see from the outputs, the `flatMap()` flattens its output, while the `map()` is a 1-to-1 mapping and does not flatten its output.

```

def to_list(x): return [x, x+x, x*x]
#
# rdd1 : RDD[Integer] (element type is Integer)
rdd1 = spark.sparkContext.parallelize([3,4,5]) ❶
    .map(to_list) ❷
rdd1.collect()
# output: notice non-flattened list
[[3, 6, 9], [4, 8, 16], [5, 10, 25]]
rdd1.count()
3

# rdd2 : RDD[[Integer]] (element type is [Integer])
rdd2 = spark.sparkContext.parallelize([3,4,5]) ❸
    .flatMap(to_list) ❹
rdd2.collect()
# output: notice flattened list

```

```
[3, 6, 9, 4, 8, 16, 5, 10, 25]
rdd2.count()
9
```

- ❶ Create an RDD[Integer]
- ❷ Each element of rdd1 is a “list of integer” numbers (as RDD[[Integer]])
- ❸ Create an RDD[Integer]
- ❹ Each element of rdd2 is an integer number (as RDD[Integer])

The `flatMap()` visualization is presented by the Figure 3.10.

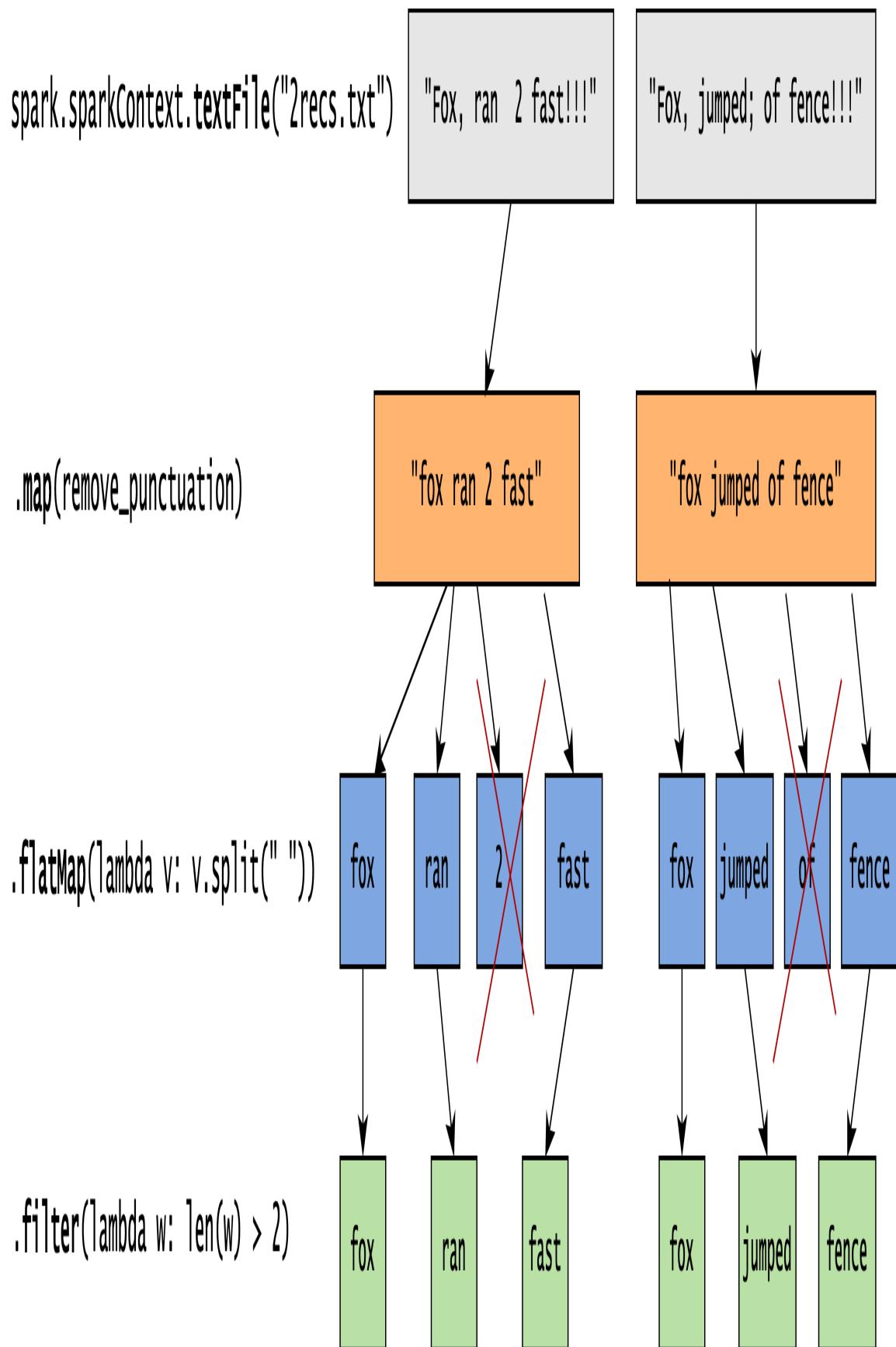


Figure 3-8. The flatMap() Visualization

To understand flatMap() Visualization, transformation steps are presented in detail. First, lets example the content of input file 2recs.txt:

```
$ cat 2recs.txt
Fox, ran 2 fast!!!
Fox, jumped; of fence!!!
```

- Step-1: we create an RDD[String] with only two records/elements.

```
rdd = spark.sparkContext.textFile("2recs.txt")
rdd.collect()
[
    "Fox, ran 2 fast!!",
    "Fox, jumped; of fence!!"
]
```

- Step-2: this step applies a map() transformation to all elements of RDD, which removes all punctuation, reduces multiple spaces into a single step and converts all letters to lowercase. This is accomplished by a simple Python function:

```
import string, re
def no_punctuation(record_str):
    exclude = set(string.punctuation)
    t = ''.join(ch for ch in record_str if ch not in exclude)
    trimmed = re.sub('\s+', ' ', t)
    return trimmed
#end-def

rdd_cleaned = rdd.map(no_punctuation)
rdd_cleaned.collect()
[
    "fox ran 2 fast",
    "fox jumped of fence"
]
```

- Step-2: this step applies a flatMap() transformation to rdd_cleaned. First tokenize elements of rdd_cleaned, and then flatten it; this is accomplished by:

```
flattened = rdd_cleaned.flatMap(lambda v: v.split(" "))
flattened.collect()
['fox', 'ran', '2', 'fast', 'fox', 'jumped', 'of', 'fence']
```

- Step-3: finally, the `filter()` transformation drops elements of the `flattened` RDD, which keeps elements if their length is greater than 2. This is accomplished by:

```
final_rdd = flattened.filter(lambda w: len(w) > 2)
final_rdd.collect()
['fox', 'ran', 'fast', 'fox', 'jumped', 'fence']
```

Note that the filtered out elements are crossed by red marks (Figure 3.10).

map() vs. flatMap()

We have seen examples of `map()` and `flatMap()` transformations. So, what is the main difference between them? As per the Spark documentation, the differences between `map()` and `flatMap()` are:

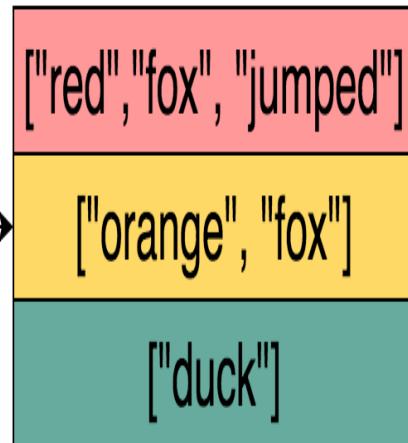
- `map()` : it is a "**1-to-1**" transformation: it returns a new RDD by applying given function to each element of the RDD. Function in `map()` returns only one item.
- `flatMap()`: it is a "**1-to-Many**" transformation: similar to `map()` , it returns a new RDD by applying a function to each element of the RDD, but output is flattened.

The difference between `map()` and `flatMap()` is illustrated by the Figure 3.11.

rdd1: RDD[String]



rdd2: RDD[[String]]



map(lambda x: x.split()) →

rdd3: RDD[String]



flatMap(lambda x: x.split()) →

Figure 3-9. Spark Transformation

The following snippet is an equivalent of the Figure 5.11:

```
>>> str_list = ["red fox jumped", "orange fox", "duck"]
>>> rdd1 = spark.sparkContext.parallelize(str_list) ❶
>>> rdd1.count()
3
>>> rdd1.collect()
['red fox jumped', 'orange fox', 'duck']
>>> rdd2 = rdd1.map(lambda x: x.split()) ❷
>>> rdd2.count() ❸
3
>>> rdd2.collect()
[
  ['red', 'fox', 'jumped'],
  ['orange', 'fox'],
  ['duck']
]
>>> rdd3 = rdd1.flatMap(lambda x: x.split()) ❹
>>> rdd3.count()
6
>>> rdd3.collect()
[
  'red', 'fox', 'jumped',
  'orange', 'fox',
  'duck'
]
```

- ❶ Create and RDD[`String`], which has 3 elements
- ❷ The `map()` result is not flattened
- ❸ `rdd1.count()` and `rdd2.count()` will be the same, since we used a `map()` to transform `rdd1` to `rdd2`
- ❹ `flatMap()` result is flattened

Next, we look the `mapValues()` transformation, which is used mainly for pair RDDs — `RDD[(K, V)]`, where each element is a pair of `(K, V)`.

The `mapValues()` Transformation

The `mapValues()` transformation is only applicable for pair RDDs (`RDD[(K, V)]`), meaning RDDs of (key: K, value: V) pairs. The `mapValues()` operates on the value only (V, the second part of the tuple and keys are untouched/unchanged), while the `map()` transformation operates on the entire RDD element.

Informally we can write:

- Let source RDD be `RDD[(K, V)]`
- Given function $f: V \rightarrow T$

Then we may say that: `rdd.mapValues(f)` is equivalent to the following `map()`:

```
# source rdd : RDD[(K, V)]
# target result : RDD[(K, T)]
result = rdd.map( lambda (k, v) : (k, f(v)) )
```

The `mapValues()` transformation is defined as:

```
pyspark.RDD.mapValues (Python method)
mapValues(f)

f: V --> U ❶
mapValues: RDD[(K, V)] --> RDD[(K, f(V))]
```

- ❶ The function `f()` can transform data type V to any desired data type of T (it is also okay that V and T be the same/different data types)

The `mapValues()` transformation passes each value in the key-value pair RDD through a map function without changing the keys; this also retains the original RDD's partitioning.

The `mapValues()` transformation is applicable only `RDD[(K, V)]` (pair RDDs — an RDD, where each element is a (key, value) pair). This

transformation only operates on the values of the pair RDDs instead of operating on the whole tuple. Function `f()` is only applied to the values and keys are untouched.

The `mapValues()` transformation acts on a pair RDD (each RDD element is a `(key, value)` pair), it runs a map operation on the RDD values. This transformation is really useful if the user is only interested in transforming the values on a pair RDD.

The following is an example of the `mapValues()` transformation:

```
>>> pairs = [
    ("A", []), ("Z", [40]),
    ("C", [10, 20, 30]), ("D", [60, 70])
]
>>> rdd = spark.sparkContext.parallelize(pairs) ❶
>>> rdd.collect()
[
    ('A', []), ('Z', [40]),
    ('C', [10, 20, 30]), ('D', [60, 70])
]
>>>
>>> def f(x): return len(x) ❷
>>>
>>> rdd2 = rdd.mapValues(f) ❸
>>> rdd2.collect()
[
    ('A', 0), ('Z', 1),
    ('C', 3), ('D', 2)
]
```

- ❶ `rdd` is `RDD[(String, [Integer])]`
- ❷ The function `f()` accepts a list of integers and return an integer (length of the given list)
- ❸ `rdd2` is `RDD[(String, Integer)]`

The `mapValues()` transformation example is illustrated by the figure 3.12.

Source: $\text{RDD}[(K, V)]$

Target: $\text{RDD}[(K, T)]$

key-1	element-1
key-2	element-2
...	...
key-N	element-N

$$f: V \rightarrow T$$

`mapValues(f)` →

key-1	$f(\text{element-1})$
key-2	$f(\text{element-2})$
...	...
key-N	$f(\text{element-N})$

Figure 3-10. The `mapValues()` Transformation

The `flatMapValues()` Transformation

The `flatMapValues()` transformation is a combination of `flatMap()` and `mapValues()`. In other words, we can say that this is similar to `mapValues()`, but `flatMapValues()` runs the `flatMap()` function on the values of `RDD[(K, V)]` (RDD of (key, value) pairs).

The `flatMapValues()` transformation is defined as:

```
pyspark.RDD.flatMapValues (Python method)
flatMapValues(f)
```

This transformation passes each value in the key-value pair RDD through a `flatMap` function without changing the keys; this also retains the original RDD's partitioning.

```
>>> rdd = spark.sparkContext.parallelize([
    ('S', []),
    ('Z', [7]),
    ('A', [1, 2, 3]),
    ('B',[4, 5])
]) ❷

>>># function is applied to entire
>>># value, and then result is flattened
>>> rdd2 = rdd.flatMapValues(lambda v: [i*3 for i in v]) ❸
>>> rdd2.collect()
[('Z', 21),
 ('A', 3), ('A', 6), ('A', 9),
 ('B', 12), ('B', 15)]
```

- ❶ this element will be dropped since value is empty
- ❷ `rdd` is `RDD[(String, [Integer])]`
- ❸ `rdd2` is `RDD[(String, Integer)]`; note that the key `S` is dropped since its value was an empty list

For this example, if the values for a key is empty ([]), then no value is generated (keys are dropped as well). Therefore, no element is generated for the keys S and T since their associated values are empty ([]).

Another example for `flatMapValues()`:

```
>>> rdd = spark.sparkContext.parallelize([
    ("A", ["x", "y", "z"]),
    ("B", ["p", "r"]),
    ("C", ["q"]),
    ("D", [])
]) ❶

>>> def f(x): return x
>>> rdd2 = rdd.flatMapValues(f) ❷
>>> rdd2.collect()
[
    ('A', 'x'), ('A', 'y'), ('A', 'z'),
    ('B', 'p'), ('B', 'r'),
    ('C', 'q')
]
```

❶ `rdd` is `RDD[(String, [String])]`

❷ `rdd2` is `RDD[(String, String)]`

Again, for this example, if the values for a key is empty ([]), then no output value is generated (keys are dropped as well). Therefore, no element is generated for the D key.

Next, I discuss the `mapPartitions()` transformation, which is the most important mapper ever created.

The `mapPartitions()` Transformation

`mapPartitions()` is a very powerful distributed mapper transformation, which processes one partition (instead of an element) at a time and implements Summarization Design Pattern — summarize each partition of a

source RDD into a single element of the target RDD. The goal of this transformation is to process one partition at a time (although, many partitions can be processed independently and concurrently), iterate all of the partition elements and summarize the result in a compact data structure such as a dictionary, list of elements, tuples, or list of tuples.

The `mapPartitions()` transformation has the following singnature:

```
mapPartitions(f, preservesPartitioning=False)
# Returns a new RDD by applying a function, f(), to each partition
# of this RDD. If source RDD has N partitions, then your function
# will be called N times, independently and concurrently
```

A typical `map()` transformation maps a single element of source RDD into a single element of the target RDD by applying a function (defined or custom); therefore a `map()` input is a single element of source RDD and the output is a single element of the target RDD.

Let's say that your source RDD has N partitions. Then the `mapPartitions()` transformation maps a single partition of source RDD into your desired data type of T. Therefore the target RDD will be `RDD[T]` of length N. This is an ideal transformation when you want to reduce (or aggregate) each partition (comprised of a set of source RDD elements) into a condensed data structure of type T. The `mapPartitions()` transformation is a summarization data pattern: maps a single partition into single element (data type T) of target RDD.

High-level logic of the `mapPartitions()` transformation is illustrated by the Figure 3.12.

mapPartitions(func) : SourceRDD → TargetRDD

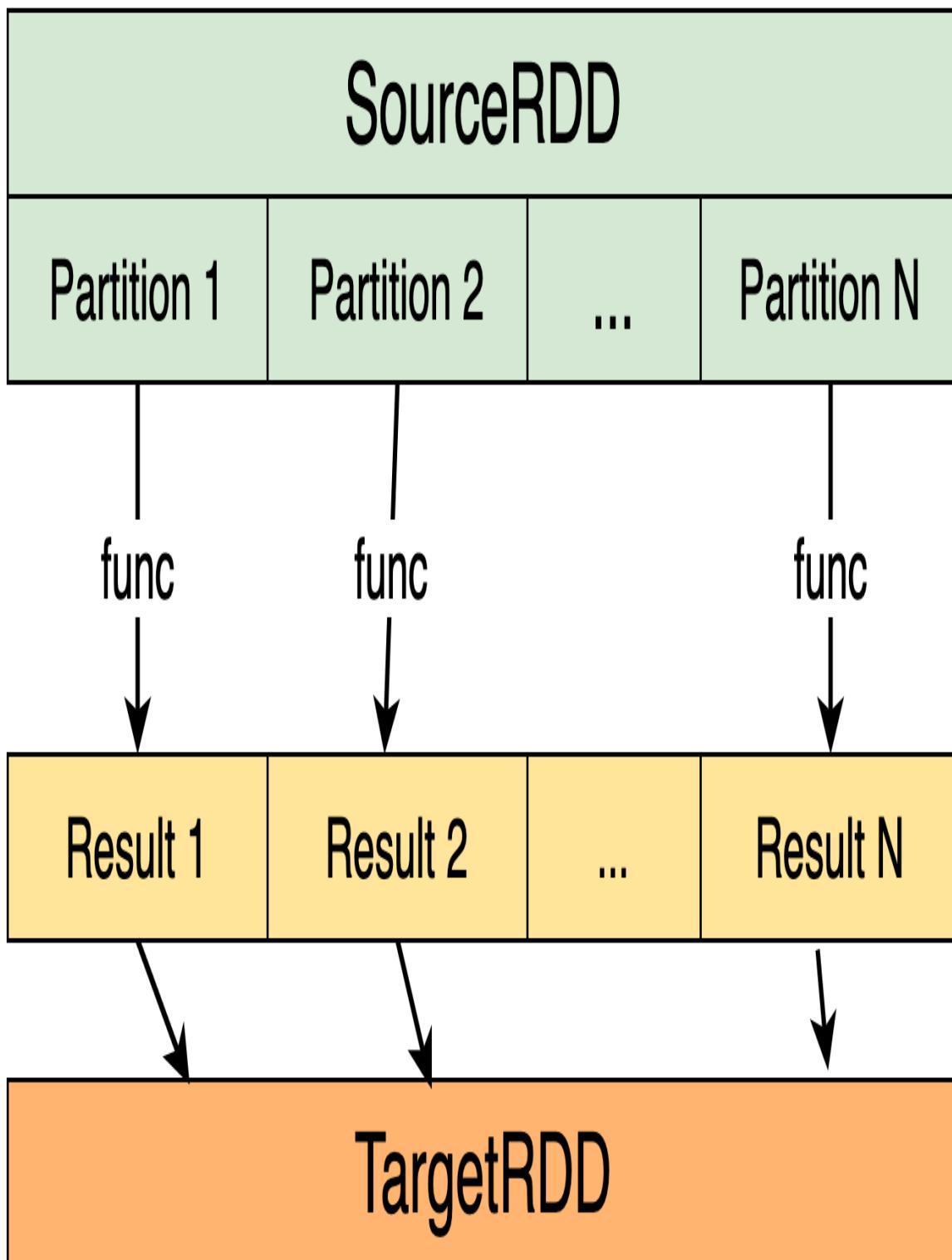


Figure 3-11. The mapPartition() Transformation

To understand the logic of the `mapPartition()` transformation, I present a concrete and simple example. Consider a source `RDD[Integer]` with `100,000,000,000` elements and assume that your `RDD` is partitioned into `10,000` chunks (the number of partitions = `10,000`). So, each partition will have about `10,000,000` elements. So, if you have enough cluster resources which can run `10,000` mappers in parallel, then each mapper will receive a partition, which is about `10,000,000` elements. Since you will be processing one partition at time (many partitions can be processed at the same time by Spark's parallelism), then you have a chance to filter elements and summarize your partition into a single desired data structure (such as a tuple, list, or dictionary).

Let's say that you want to find `(minimum, maximum, count)` for the source `RDD` of numbers. Then each mapper will find a local `(minimum, maximum, count)` per partition, and then eventually, you can find the final `(minimum, maximum, count)` for all of the partitions. Here the target data type is a triplet:

```
T = (int, int, int) = (minimum, maximum, count)
```

The `mapPartitions()` is an ideal transformation when you want to map each partition into small amount of condensed/reduced information. This transformation gives you a chance to filter out undesired elements of a source `RDD` and then summarize your desired elements into an ideal data structure (such as list, tuple, or dictionary).

The following shows the main flow of `mapPartitions()` transformation:

- First define a function, which accepts a single partition of source `RDD` (as `RDD[Integer]`) and returns a data type `T`, where

```
T = (int, int, int) = (minimum, maximum, count)
```

Let N be the number of partitions for your source RDD. Therefore given a partition p (where $p \in \{1, 2, \dots, N\}$), then `mapPartitions()` will compute $(\text{minimum}_p, \text{maximum}_p, \text{count}_p)$ per partition p .

```
def find_min_max_count(single_partition):
    # find (minimum, maximum, count) by iterating single_partition
    return [(minimum, maximum, count)]
#end-def
```

- Next, apply the transformation:

```
# source RDD: source_rdd = RDD[Integer]
# target RDD: min_max_count_rdd = RDD(int, int, int)
min_max_count_rdd = source_rdd.mapPartitions(find_min_max_count)
min_max_count_list = min_max_count_rdd.collect()
print(min_max_count_list)
[
    (min1, max1, count1),
    (min2, max2, count2),
    ...
    (minN, maxN, countN)
]
```

- Finally, we need to collect the content of `min_max_count_rdd` and find the final $(\text{minimum}, \text{maximum}, \text{count})$:

```
# minimum = min(min1, min2, ..., minN)
minimum = min(min_max_count_list)[0]
# maximum = max(max1, max2, ..., maxN)
maximum = max(min_max_count_list)[1]
# count = (count1+count2+...+countN)
count = sum(min_max_count_list)[2]
```

The complete solution is given below:

```
def find_min_max_count(single_partition_iterator):
    first_time = False
    for n in single_partition_iterator:
        if (first_time == False):
            minimum = n;
            maximum = n;
            count = 1
```

```

        first_time = True
    else:
        maximum = max(n, maximum)
        minimum = min(n, minimum)
        count = count + 1
    #end-for
    return [(minimum, maximum, count)]
#end-def

```

Next, let's create an RDD[Integer] and then apply the mapPartitions() transformation:

```

integers = [1, 2, 3, 1, 2, 3, 70, 4, 3, 2, 1]
# spark : SparkSession
source_rdd = spark.sparkContext.parallelize(integers)
# source RDD: source_rdd = RDD[Integer]
# target RDD: min_max_count_rdd = RDD[int, int, int]
min_max_count_rdd = source_rdd.mapPartitions(find_min_max_count)

min_max_count_list = min_max_count_rdd.collect() ❶
# compute the final values:
minimum = min(min_max_count_list)[0]
maximum = max(min_max_count_list)[1]
count = sum(min_max_count_list)[2]

```

- ❶ The collect() is scalable here, because the number of partitions will be in thousands and not in millions.

In summary, mapPartitions() has the following properties:

- It does implement Summarization Design Pattern: summarize each partition of source RDD into a compact element — such as a dictionary, list of objects, list of tuples, tuples — of target RDD
- mapPartitions() can be used as an alternative to map() and foreach()
- mapPartitions() should be used when you want to combine many source RDD elements into a desired element of target RDD (summarize a partition into a single desired data element)

- `mapPartitions()` can be called for each single partition while `map()` and `foreach()` is called for each element in an RDD
- Programmer can do the initialization on per-partition basis rather than each element basis

Therefore, if you have large amount of data, which should be reduced to small amount of information (this is called a summarization data pattern), then the `mapPartitions()` transformation is a possible option. For example to find Min-Max and Top-10, you may use the `mapPartitions()` transformation.

Next, I discuss a very important topic — how to handle and process an empty partition— when using the `mapPartitions` transformation.

Handling Empty Partitions

In previous section, we used the `mapPartitions(func)` transformation, which partitions input data into many partitions and then applies the function `func` (provided by the programmer) to each partition in parallel. What if any of these partitions are empty: it means that there is a partition, but there is no data (source RDD elements) to iterate. We do need to write our function `func` (partition handler) in such a way that to handle an empty partition properly and gracefully. You can not just ignore them. Ignoring will not solve the problem.

What if there is an exception (corrupted records after a network failure mid-transfer) for a Spark partitioner in partitioning the data, then some partitions might be empty. Also, empty partitions may happen for many other reasons: one reason can be that the partitioner does not have enough data to put for a given partition. This case can happen. Regardless of how empty partitions are created, we do need to handle all of them gracefully.

I will show the concept of an “**empty partition**” by a simple example. First, we define a function, `debug_partition()` (Listing 3.6 — to be used for testing and teaching only — not to be used in production environments), to

show the content of each partition: this is the function, which will be invoked per partition.

```
def debug_partition(iterator):
    #print("type(iterator)=", type(iterator))
    elements = []
    print("begin partition ===")
    for x in iterator:
        elements.append(x)
    print("elements = ", elements)
#end-def
```

TIP

Debugging Partition Elements

You should note that displaying or debugging content of a partition can be costly and should be avoided in production environments. I have included `print` statements for teaching and debugging purposes only.

Next, we define an RDD and partition it in such a way to create empty partitions (source Listing 4.7).

```
>>> sc
<SparkContext master=local[*] appName=PySparkShell>
>>> numbers = [1, 2, 3, 4, 5]
>>> rdd = sc.parallelize(numbers, 7) ❶
>>> rdd.collect()
[1, 2, 3, 4, 5]
>>> rdd.getNumPartitions()
7
```

- ❶ force to create empty partitions

Next, we examine each partition by using the `debug_partition()` function:

```
>>> rdd.foreachPartition(debug_partition)
elements =  [4]
elements =  [3]
elements =  [2]
```

```
elements = []
elements = [] // ❶
elements = [5]
elements = [1]
```

❶ an empty partition

From this test program we observe the following for using the `mapPartitions()`:

- A partition can be empty (with no RDD elements at all). You must handle empty partitions. You can not ignore them; ignoring empty partitions will create exceptions (if not handled properly)
- Your custom function must handle empty partitions gracefully (must return a proper value); empty partitions can not be just ignored.
- The data type of iterator (which represents a single partition and passed as a parameter to the `mapPartitions()`) is the `itertools.chain`. The `itertools.chain` is an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence.

Now the question is how to handle an empty n PySpark? The following pattern (Listing 3.8) may be used to handle an empty partition gracefully.

```
# This is the template function
# to handle a single partition
#
# source RDD: RDD[T]
#
# parameter: iterator
#
def func(iterator): ❶
    print("type(iterator)=", type(iterator))
    # ('type(iterator)=' , <type 'itertools.chain'>)
    #
    try:
        first_element = next(iterator) ❷
```

```

# if you are here it means that
# the partition is NOT Empty
... process the partition
... and return a proper result

except StopIteration: ❸
    # if you are here, it means that this
    # partition is Empty; now, you need
    # to handle and return a proper result
#end-def

```

- ❶ iterator represent a single partition of elements of type T
- ❷ Try to get the first element (as first_element of type T) for a given partition. If this fails (throws an exception), then control will go to the except (exception happened section).
- ❸ You will be here when a given partition is empty. You can not just ignore empty partitions, you must return a proper value

To handle an empty partition to find (min, max, count), we will rewrite the partition handler function.

```

def find_min_max_count_revised(single_partition_iterator):
    try:
        first_element = next(single_partition_iterator)
        # if you are here it means that
        # the partition is NOT Empty
        # process the partition and return a proper result
        minimum = n;
        maximum = n;
        count = 1
        #
        for n in single_partition_iterator:
            maximum = max(n, maximum)
            minimum = min(n, minimum)
            count = count + 1
        #end-for
        return [(minimum, maximum, count)]
    except StopIteration:
        # if you are here, it means that this
        # partition is Empty; now, you need

```

```

# to handle it gracefully and return
# a proper result
# return a value that we can filter it out later ❶
    return [(+1, -1, 0)]
#endif

```

- ❶ we return `[min=+1, max=-1, count=0]` as a fake value (since `min > max`, which can not be true), which will be filtered out by the `filter()` function

Next, we drop out the value for empty partitions:

```

integers = [1, 2, 3, 1, 2, 3, 70, 4, 3, 2, 1]
# spark : SparkSession
source_rdd = spark.sparkContext.parallelize(integers, 4)
# source RDD: source_rdd = RDD[Integer]
# target RDD: min_max_count_rdd = RDD(int, int, int)
min_max_count_rdd = source_rdd.mapPartitions(find_min_max_count_revised)

# filter out fake values returned from empty partitions
min_max_count_rdd_filtered = min_max_count_rdd.filter(lambda x: x[1] >= x[0]) ❷

min_max_count_list = min_max_count_rdd.collect()
# compute the final values:
minimum = min(min_max_count_list)[0]
maximum = max(min_max_count_list)[1]
count = sum(min_max_count_list)[2]

```

- ❷ keep results where `maximum >= minimum`, this will drop the redundant/fake triplets created for empty partitions

Benefits of `mapPartitions()` Transformation

The Spark's `mapPartitions()` is an efficient transformation, which implements Summarization Design Pattern and its benefits are listed below.

Low processing overhead

we are applying the mapper function once per RDD partition rather than per RDD elelemt (this limits the number of function calls). Therefore, the number of functions calls are reduced to the number of partitions rather than number of elements. Note that for some transformations, such as `map()` and `flatMap()`, there is an overhead of invoking a function call for each of the element of data collection residing in a partition.

Filter, Map, and Aggregation at the same time

this transformation provides opportunity to combine `map()`/`flatMap()` operation with a `filter()` operation. This combination would yield in higher efficiency of Spark Job since the overhead of setting up and managing a multiple data transformation steps could be avoided. Basically, as you iterate partition elelemts, you may drop undesired elelemts (the `filter()` function) and then map and aggregate your desired elements into your desired data type (such as list, tuple, dictionary, or custom data type).

Efficient Local Aggregation

Since `mapPartitions()` works on the partitions level, it gives the opportunity to the user to perform filtering and aggregation at a partition level. This local aggregation on a partition level greatly reduces the amount of shuffled data. With `mapPartitions()`, we are reducing a partition into a small contained data structure. It is evident that reduction in sort and shuffle of data results in improvement in efficiency and reliability of reduce operations.

Avoidance of Explicit Filtering Step

The `mapPartitions()` enable us to squeeze in the `filter()` steps during iteration of a partition (comprised of thousands or millions of elements). As you iterate a single partition elelemts, you may drop the ones not needed as well as aggregate on the desired values. Indeed, you may apply multiple filters at the same time. With `mapPartitions()`, you will not need additional `filter()` transformations.

Avoidance of Repetitive Heavy Initialization

With `mapPartitions()` you may use broadcasted variables (shared among all cluster nodes) and initialize your desired data structures required for aggregation of partition elements. If you need a heavy initialization, then you will not pay a heavy price, since the number of partitions are limited to the number of partitions. When using narrow transformations (like `map()` and `flatMap()`), then initialization and creation of such data structures becomes very inefficient since they incur a major overhead of repetitive initialization and de-initialization. But, in case of `mapPartitions()` usage, this heavy initialization would be executed only once (at the beginning of a function) and would suffice for all the data records residing in a partition. An example of a heavy initialization could be the initialization of a database (relational or HBase) connection to read/update/insert a record.

Summary

- Spark offers a lot of simple and powerful transformations (such as `map()`, `flatMap()`, `filter()`, `mapPartitions()`, ...) which you can convert one form of data into another. Spark transformations enable us to perform ETL (Extract, Transform and Load) with powerful transformations in a simple way.
- All of the Spark transformations do not have the same performance, therefore, you need to select the transformations in such a way, which fits your data and performance considerations. For example, in general, `reduceByKey()` has a better performance and scalability over `groupByKey()` for a lot of generated (`key, value`) pairs.
- Spark transformations are “lazy evaluated”: means that when we call the “action” (such as `count()`, `collectAsMap()`, or `collect()`) it executes all the transformations based on lineage graph. Lazy evaluation means that Spark does not evaluate each

transformation as they arrive, but instead queues them together and evaluate all at once, as an “action” is called. The benefit of “lazy evaluation” approach is that Spark can make optimization (such as improving cluster network I/O) decisions after it had a chance to look at the DAG (graph of transformations and actions) in entirety. This would not be possible if it were to execute everything as soon as it got it.

Chapter 4. Reductions in Spark

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

The focus of this chapter is to present the concept of reduction transformations in Spark. Typically, a reduction is over a set of values per key. Most of data algorithms and ETL require reduction of values by some keys (such as finding mean and median over a set of stock values). This chapter covers

- Reduction concepts
- What is a monoid?
- Reductions in Spark
- Most important Spark reduction transformations:
 - `reduceByKey()`
 - `combineByKey()`
 - `groupByKey()`

- `aggregateByKey()`
- `sortByKey()`

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 4](#).

The main goal of this chapter is to present reduction transformations on Resilient Distributed Datasets (RDDs). This chapter covers how to work with pair RDDs of `(key, value)` pairs, which are a common data abstraction required for many operations in Spark. Pair RDDs are commonly used to perform aggregations (sum, average, median, T-test, ...), and often we will do some initial ETL (extract, transform, and load) to get our data into a `'(key, value)'` form. With pair RDDs you may perform any desired aggregation over a set of values.

Spark supports powerfull reducer transformations and actions. All reducers by keys are denoted by `<reducer_name>ByKey()` transformations, which accepts a source `RDD[(K, V)]` and creates a target `RDD[(K, C)]` (for some transformations such as `reduceByKey()`, the `V` and `C` are the same. The function of `<reducer_name>ByKey()` transformation is to reduce all values of a given key (for all unique keys).

The reduction by key can be simply

- An average of all values
- Sum and count of all values
- Mode and median of all values
- Standard deviation of all values

- T-test of all values

For some of the reduction operations (such as median — which is the “middle” of a sorted list of numbers), you do need all values at the same time before finding the median. But for some other functions, such as sum and count of all values, the reducer does not need all values at the same time. For example, if you want to find the median of all values, then `groupByKey()` will be a good choice, but this transformation does not do well if a key has lots of values (which might cause an OOM problem). On the otherhand if you want to find the sum and count of all values, then the `reduceByKey()` will be a good choice, which merges the values for each key using an associative and commutative reduce function.

The purpose of this chapter is to show the most important Spark’s reduction transformations by simple working PySpark examples. Spark has many reduction transformations, but we will focus on the transformations used by most of the Spark applications.

Since pair RDDs are required for reducer transormations by keys, next I present creating pair RDDs.

Creating Pair RDDs

Given a set of keys and their associated values, a reduction transformation is a transformation to reduce values of each key using an algorithm (sum of value, median of values, ...). The reduction transformations presented in this chapter will work on `(key, value)` pairs. This means that RDD elements must conform to `(key, value)` pairs. There are several ways to create pair RDDs in Spark. The simple way to way create pair RDDs is by a `map()` function that returns `(key, value)` pairs. Also, to create a set of `(key, value)` pairs, you may use `parallelize()` on collections (such as list of tuples and dictionaries). To perform any reductions by keys (such as `reduceByKey()`, `groupByKey()`, ...), your RDD must be a pair RDD, where each element is a `(key, value)` pair.

Example: Using Collections

You may use Python's collections to create pair RDDs. The following illustrates how to create pair RDDs:

```
>>> key_value = [('A', 2), ('A', 4), ('B', 5), ('B', 7)]
>>> pair_rdd = spark.sparkContext.parallelize(key_value)
>>> pair_rdd.collect() ❶
[('A', 2), ('A', 4), ('B', 5), ('B', 7)]
>>> pair_rdd.count()
4
>>> hashmap = pair_rdd.collectAsMap()
>>> hashmap
{'A': 4, 'B': 7}
```

- ❶ `pair_rdd` has two keys as `{'A', 'B'}`

Example: Using `map()` Transformation

Suppose you have weather-related data and you want to create pairs of `(city_id, temperature)`. Assume that your input has the following format:

```
<city_id>,<,><lattitude>,<,><longtitue>,<,>temprature>
```

With a `map()` transformation and a simple Python function, you can create your desired pair RDD.

- Create a function to create `(key, value)` pair

```
def create_key_value(rec):
    tokens = rec.split(",")
    city_id = tokens[0]
    temprature = tokens[3]
    return (city_id, temprature) ❶
```

- ❶ key is `city_id` and value is `temprature`
 - Use `map()` to create pair RDD:

```

input_path = <your-temprature-data-path>
rdd = spark.sparkContext.textFile(input_path)
pair_rdd = rdd.map(create_key_value)
# or we may write it using lambda as
pair_rdd = rdd.map(lambda rec: create_key_value(rec))

```

The are many other ways to create (key, value) pair RDDs: such as `map()`, `reduceByKey()`, `combineByKey()`, etc. For example, `reduceByKey()` accepts a source RDD of (K, V) and produces a target RDD of (K, V). On the otherhand `combineByKey()` accepts a source RDD of (K, V) and produces a target RDD of (K, C) where V and C can be different data types (if desired).

Reducer transformations

Typically, a reducer transformation reduces the data size from a larger batch of values (such as list of numbers) to a smaller one (such as sum, median, or average of the list of numbers). An example of a reducer by key can be:

- find sum and average of all values
- find mean, mode and median of all values
- calculate mean and standard deviation of all values
- find (min, max, count) for all values
- find Top-10 of all values

In a nutshell, the reduce transformation roughly corresponds to the fold operation (also termed reduce, accumulate, aggregate) in functional programming. Reducer transformations are either applied to all data elements (such as finding sum of all elements) or to all elements per key (such as finding sum of all elements per key).

A simple addition reduction over a set of numbers {47, 11, 42, 13} for a single partition is illustrated in Figure 4.1.

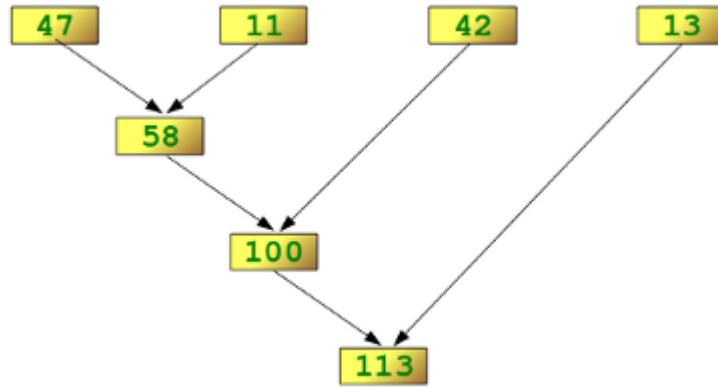


Figure 4-1. Reduction Concept

Another addition reduction concept is illustrated by the Figure 4.2. This is a reduction, which adds the elements of 2 partitions (note that Spark manages data using partitions — called chunks — that helps parallelize distributed data processing with minimal network traffic for sending data between executors). The final reduced values for Partition-1 and Partition-2 are 21 and 18. Each partition performs local reductions and finally the result of two partitions are reduced.

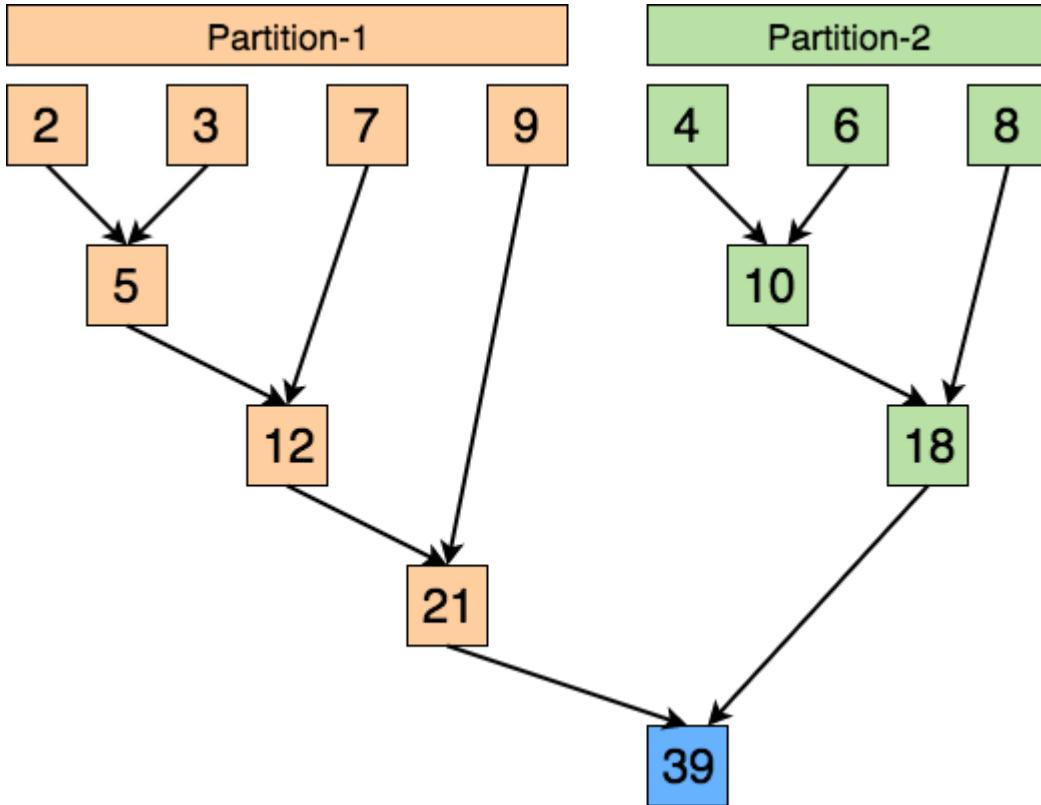


Figure 4-2. Reduction Concept

Reducer is a core concept in functional programming, which reduces a set of objects (such as numbers, strings, lists, ...) into a single value (such as sum of numbers, concatenation of string objects). Spark and MapReduce paradigm use the reducer concept to aggregate a set of values into a single value for a given set of keys. In the simplest form consider the following (`key, value`) pairs (where `key` is a `String` and `value` is a list of `Integers`)

```
(key1, [1, 2, 3])
(key2, [40, 50, 60, 70, 80])
(key3, [8])
```

The most simple reducer will be an addition function over a set of values per key. Once we apply the addition reducer, the result will be:

```
(key1, 6)
(key2, 300)
(key3, 8)
```

Or you may reduce each `(key, value)` to `(key, pair)` where `pair` is `(sum-of-values, count-of-values)`:

```
(key1, (6, 3))
(key2, (300, 5))
(key3, (8, 1))
```

Reducers are designed to operate concurrently and independently: meaning that there is no synchronization between reducers. The more resources a Spark cluster has, reductions can be done faster. In the worst possible case, if we have only one reducer, then reduction will work as a queue operation. In general, a cluster will offer many reducers (depending on the resource availability) for the reduction transformation.

In MapReduce programming paradigm, the programmer defines a mapper and a reducer with the following `map()` and `reduce()` signatures:

- `map: (K~1~, V~1~) -> [(K~2~, V~2~)]`
- `reduce: (K~2~, [V~2~]) -> [(K~3~, V~3~)]`

The `map()` function maps a `(key~1~, value~1~)` pair into a set of `(key~2~, value~2~)` pairs. After all maps are completed, the sort and shuffle automatically is done (provided by the MapReduce paradigm and not done by the programmer). The the sort and shuffle phase of MapReduce paradigm is very similar to the Spark's `groupByKey()` transformation

The `reduce()` function reduces a `(key~2~, [value~2~])` pair into a set of `(key~3~, value~3~)` pairs

The convention `[...]` is used to denote a **list of objects** (or an **iterable list of objects**). Therefore, we can say that a reducer transformation takes a list of values and reduces it to a tangible result (such as sum of values, average of values, or your desired data structure).

In MapReduce and distributed algorithms, reduction (the so called `reduce()` function) step is a required operation in solving a problem. Spark provides an easy to use rich set of reduction transformations. Throught the chapter

we'll discuss Spark's reduction transformations (such as `reduceByKey()`, `groupByKey()`, `aggregateByKey()`, and `combineByKey()`) on a given list of `(key, value)` pairs, typically emitted by mappers or generated by an ETL program. In general, the `combineByKey()` is more general than `reduceByKey()` and `aggregateByKey()`.

The `groupByKey()` transformation is very simple to use and its reduction transformation concept is illustrated by the Figure 4.3. In this example, we have four unique keys {A, B, C, P} and their associated values are grouped as a list of integers. In this example,

- Source RDD:
 - `RDD[(String, Integer)]`
 - Each element is a pair of `(String, Integer)`
- Target RDD:
 - `RDD[(String, [Integer])]`
 - Each element is a pair of `(String, [Integer])`, where value is a list/iterable of integers.

Informally and in a nutshell, Spark's `groupByKey()` transformation works very similar to the SQL's `GROUP BY` statement.

Source: RDD[(String, Integer)]

(A, 5)
(B, 8)
(A, 6)
(B, 4)
(C, 44)
(C, 66)
(A, 9)
(C, 77)
(C, 55)
(P, 100)

Target: RDD[(String, [Integer])]

groupByKey()

(A, [9, 5, 6])
(C, [44, 77, 55, 66])
(B, [4, 8])
(P, [100])

Figure 4-3. The groupByKey() transformation example

You should note that, by default, Spark reductions do not sort the reduced values. For example looking at Figure 3, the reduced value for key B can be [4, 8] or [8, 4]. If desired, you may sort the values before the final reduction. Therefore, if your reduction algorithm requires sorting, then you should sort values explicitly.

Next, we focus on Spark's reduction transformations on RDDs. In general most of the Spark applications will require several reductions by keys

Spark's Reductions

Spark is a fast and general engine for large scale data processing. Spark provides a high-level MapReduce API (such as `map()` and `reduce()`) and beyond (such as `filter()` and many other useful reduction by key transformations). Indeed, Spark's API is a superset (by providing natural join and merge functionalities) of Hadoop's classic MapReduce API. Spark's operations (transformations and actions) is much more powerful, higher level, easy to use, and faster than Hadoop's classic MapReduce paradigm. According to Spark documentation, you may run Spark programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk. In summary, Spark is easier, richer, and faster than Hadoop's classic MapReduce programming model. The API in this chapter is based on Spark-3.0.0.

Spark offers a set of high-level and powerful by key reducers. Some of the most important and common reducers are listed in Table 1.

T
a
b
l
e

4
-
l
. *S*
p
a
r
k,
s

R
e
d
u
c
t
i
o
n
s

Transformation Purpose

`reduceByKey()` Combine values with the same key

`groupByKey()` Group values with the same key

<code>aggregateByKey()</code>	Aggregate the values of each key, using given combine functions and a neutral “zero value”.
<code>combineByKey()</code>	Generic function to combine the elements for each key using a custom set of aggregation functions.

We will discuss these reduction transformations in the context of PySpark examples.

Internally , the `aggregateByKey()`, `reduceByKey()` and `groupByKey()` are implemented by `combineByKey()`. The `aggregateByKey()` transformation is similar to `reduceByKey()` but you can provide initial values (per partition) when performing aggregation. Using `reduceByKey()` will provide the most optimized performance (without writing 3 additional functions — `create_combiner`, `merge_value`, `merge_combiners` — that you have to provide for the `combineByKey()`). If you can not (such as when the input and result types differ from each other) use `reduceByKey()`, then consider using `combineByKey()`, which you have to provide 3 additional small functions.

Finally, if solving the group-by-key aggregation by `reduceByKey()` or `combineByKey()` is very hard and complex, then you may use the `groupByKey()`. While the use of `combineByKey()` takes a little more work than using a `groupByKey()` call, but avoiding `groupByKey()` can improve your spark job performance by reducing the amount of data sent across the network. If you are going to use `groupByKey()`, then make sure that you have enough memory in your cluster to handle all of values per key. When possible, use `combineByKey()` or `reduceByKey()` transformation to reduce the amount of shuffle data.

What is a Reduction?

According to a couple of dictionaries: reduction is defined as:

- the act of making something smaller in size, amount, number, etc.

- the act of reducing something
- an amount by which something is reduced
- the act or process of reducing : the state of being reduced
- the action or fact of making a specified thing smaller or less in amount, degree, or size.
- the process of converting an amount from one denomination to a smaller one, or of bringing down a fraction to its lowest terms.

Our focus will be on reductions, where the source RDD is of the form $\text{RDD}[(K, V)]$ (an RDD where elements are pairs of (K, V) — this is called a pair RDD). In Spark and MapReduce paradigm, the reduction is the process of applying some function $f()$ on the values (V_1, V_2, \dots, V_n) for every key K in the $\text{RDD}[(K, V)]$ (called a pair RDD). The function $f()$ can be something as trivial as summation of the values or can be as complex as your requirement.

Therefore, we will assume that each RDD has a set of keys and for each key (such as K) we have a set of values as illustrated below.

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

Of course this a simplistic view of a reducer. In real applications, values for the same key (here denoted as K) can come from many different partitions and each partition can come from a different server. Note that there is no order (such as ascending or descending) between the values

$\{ V_1, V_2, \dots, V_n \}$

for a given key K . In Spark, based on your selected transformation (such as `groupByKey()`, `reduceByKey()`, or `combineByKey()`) sort and shuffle phase can be done very differently, which might have a different efficiency and scale-out.

Spark's Reduction Transformations

Spark (Table 1.2) provides the following `<reduction-name>ByKey()` transformation functions over a set of `(key, value)` pairs (partial listing):

T
a
b
l
e

4
-
2
. *S*
p
a
r
k,
s

R
e
d
u
c
t
i
o
n

T
r
a
n
s
f

*o
r
m
a
t
i
o
n
S*

Reduction Transformation Description

<code>groupByKey()</code>	Group the values for each key in the RDD into a single sequence
<code>reduceByKey()</code>	Merge the values for each key using an associative and commutative reduce function
<code>combineByKey()</code>	Generic function to combine the elements for each key using a custom set of aggregation functions
<code>aggregateByKey()</code>	Aggregate the values of each key, using given combine functions and a neutral “zero value”
<code>foldByKey()</code>	Merge the values for each key using an associative function and a neutral “zero value” ...)
<code>countByKey()</code>	Count the number of elements for each key, and return the result to the master as a Map
<code>sampleByKey()</code>	Return a subset of this RDD sampled by key (via stratified sampling)
<code>sortByKey()</code>	Sort the RDD by key, so that each partition contains a sorted range of the elements in ascending order
<code>subtractByKey()</code>	Return an RDD with the pairs from <code>this</code> whose keys are not in <code>other</code>

These group of transformation functions act on (key, value) pairs represented by RDDs. For example, in Java programming language, (key , value) pairs

are represented by `JavaPairRDD<K,V>`, but since Python is a type-less language, `(key, value)` pairs are represented as `(key, value)`, which is a tuple of 2 elements. Therefore, Spark provides several ways to do reductions on data. Here, I will discuss the performance of each reduction function with respect to the size of values per given unique key. It has been well documented that, for example, the performance of Spark's `reduceByKey()` is much better than `groupByKey()` when aggregation or reduction is done over a lot of values per given key.

In this chapter, we will look only into reductions of data over a set of given unique keys. For example, given the following `(key, value)` pairs for `key=K`:

{ (K, V~1~), (K, V~2~), ..., (K, V~n~) }

We are assuming that the key K has a list of n ($n > 0$) values:

[V~1~, V~2~, ..., V~n~]

To keep it simple, the goal of reduction is to generate the following pair (or a set of new `(key, value)` pairs):

(K, R)

where

$f(V~1~, V~2~, \dots, V~n~) \rightarrow R$

where the function `f()` is called a `reducer` or `reduction` function. Spark provides a set of transformations (such as `groupByKey()`, `reduceByKey()`, `combineByKey()`, `aggregateByKey()`, ...) to apply function `f()` over a list of values: [V~1~, V~2~, ..., V~n~]. To find the reduced value, R, we have many options in Spark, but the performance and scalability of these transformation will differ based on the number of values processed over a set of keys. Spark does not impose any ordering among the values ([V~1~, V~2~, ..., V~n~]) to be reduced.

Simple Warmup Example

Suppose we have a list of pairs: (K, V) where K (as a key) is a String and V (as a value) is an Integer number:

```
[  
  ('alex', 2), ('alex', 4), ('alex', 8),  
  ('jane', 3), ('jane', 7),  
  ('rafa', 1), ('rafa', 3), ('rafa', 5), ('rafa', 6),  
  ('clint', 9)  
]
```

In this example, we have 4 unique keys:

```
{ 'alex', 'jane', 'rafa', 'clint' }
```

Suppose we want to combine (add the values) the values per key. The result of this reduction will be:

```
[  
  ('alex', 14),  
  ('jane', 10),  
  ('rafa', 15),  
  ('clint', 9)  
]
```

where

```
key: alex => 14 = 2+4+8  
key: jane => 10 = 3+7  
key: rafa => 15 = 1+3+5+6  
key: clint => 9 (single value, no operation is done)
```

There are so many ways to add these numbers to get the desired result. How did we arrive with these reduced (key, value) pairs? For this example, we may use any of the Spark transformations. Aggregating the values per key or combining the values per key is a reduction function. In classic MapReduce paradigm, this is called a “reduce by key” (or simply reduce) function. The MapReduce’s framework calls the application’s (user defined) “reduce” function once for each unique key in the sorted order of keys. The “reduce” function can iterate through the values that are associated with that key and

produce zero or more outputs as (key, value) pairs. The “reduce” function solves the problem of combining the elements of each unique key to a single value. Note that in some applications, the result might be more than a single value.

Here I present 4 different solutions using Spark’s transformations. For all solutions, we will use the following Python `data` and `key_value_pairs` (as `RDD[(String, Integer)]`), which represents a set of (`key=String`, `value=Integer`) pairs.

```
>>> data = ❶
[
    ('alex', 2), ('alex', 4), ('alex', 8),
    ('jane', 3), ('jane', 7),
    ('rafa', 1), ('rafa', 3), ('rafa', 5), ('rafa', 6),
    ('clint', 9)
]
>>>
>>> key_value_pairs = spark.SparkContext.parallelize(data) ❷
>>> key_value_pairs.collect()
[
    ('alex', 2), ('alex', 4), ('alex', 8),
    ('jane', 3), ('jane', 7),
    ('rafa', 1), ('rafa', 3), ('rafa', 5), ('rafa', 6),
    ('clint', 9)
]
```

- ❶ Python collection: list of pairs
- ❷ `key_value_pairs` is an `RDD[(String, Integer)]`

Solution by `reduceByKey()`

Adding values for a given key is pretty straightforward. Add every 2 values and keep going. This is the most efficient solution since combiners are used at worker levels and finally the partition values are added. A reducing addition (+) function is an associative binary operation. The source and target RDDs for `reduceByKey()` transformation can be stated as:

```
source RDD: RDD[(K, V)]
target RDD: RDD[(K, V)]
```

Note that source and target data types of RDD values (V) are the same (this is a limitation on the `reduceByKey()` — this limitation can be removed by using the `combineByKey()` or `aggregateByKey()`).

Using Lambda Expressions

This solution uses `reduceByKey()` and Lambda Expressions (anonymous function):

```
# a is (an accumulated) value for key=K
# b is a value for key=K
sum_per_key = key_value_pairs.reduceByKey(lambda a, b: a+b)
sum_per_key.collect()
[('jane', 10), ('rafa', 15), ('alex', 14), ('clint', 9)]
```

Using Functions

Instead of using Lambda Expressions, you may use a defined function, such as `add`:

```
from operator import add
sum_per_key = key_value_pairs.reduceByKey(add)
sum_per_key.collect()
[('jane', 10), ('rafa', 15), ('alex', 14), ('clint', 9)]
```

Adding values per key by `reduceByKey()` is an optimized solution, since aggregation will happen in all partitions before final aggregation of the all partitions. According to Spark: `reduceByKey()` merges the values for each key using an associative and commutative reduce function. This means that combiners (optimized mini-reducers) are used in all cluster nodes before merging the values per partitions.

Solution by `groupByKey()`

We can solve this problem by using the `groupByKey()` transformation, but this solution will not have an ideal performance since we will move lots of data to the reducer nodes.

```
sum_per_key = key_value_pairs
    .groupByKey() ❶
    .mapValues(lambda values: sum(values)) ❷
sum_per_key.collect()
[('jane', 10), ('rafa', 15), ('alex', 14), ('clint', 9)]
```

- ❶ Group values (similar to SQL's GROUP BY) per key, now each key will have a set of Integer values; for example these three pairs `{('alex', 2), ('alex', 4), ('alex', 8)}` will be reduced to a single pair of `('alex', [2, 4, 8])`
- ❷ Add values per key using Python's `sum()` function

The source and target RDDs for `groupByKey()` transformation can be stated as:

```
source RDD: RDD[(K, V)] ❶
target RDD: RDD[(K, [V])] ❷
```

- ❶ Value is a type `V`
- ❷ Value is an iterable/list of `V` as `[V]`

Note that source and target data types are not the same. The value data type for source RDD is `V`, while the the value data type for taget RDD is `[V]` (as a iterable/list of `V` — denoted as `[V]`).

Solution by `aggregateByKey()`

In simplest form, the `aggregateByKey()` transformation is defined as:

```

aggregateByKey(zero_value, seq_func, comb_func)

source RDD: RDD[(K, V)]
target RDD: RDD[(K, C)]

```

V and C can be different data types.

According to Spark: `aggregateByKey()` aggregates the values of each key, using given combine functions and a neutral “zero value”. This function can return a different result type, C, than the type of the values in the source RDD, V. Thus, we need one operation for merging a V into a C (per partition) and one operation for merging two C's (merging values of two partitions) into a single C. The former operation is used for merging values within a single partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new C. Note that C and V can be different data types. For this example both are Integer data types.

Sum of values is presented by using the `aggregateByKey()` transformation:

```

# zero_value -> C
# seq_func: (C, V) -> C
# comb_func: (C, C) -> C
#
>>> sum_per_key = key_value_pairs.aggregateByKey(
... 0, ❶
... (lambda C, V: C+V), ❷
... (lambda C1, C2: C1+C2) ❸
...
>>> sum_per_key.collect()
[('jane', 10), ('rafa', 15), ('alex', 14), ('clint', 9)]
>>>

```

- ❶ `zero_value` : initial value, applied per partition
- ❷ `seq_func` : used on single partition
- ❸ `comb_func` : combining values of partitions

Solution by combineByKey()

The `combineByKey()` transformation is the most general and powerful among all reduce by key transformations. In its simplest form, the `combineByKey()` transformation is defined as:

```
combineByKey(create_combiner, merge_value, merge_combiners)

source RDD: RDD[(K, V)]
target RDD: RDD[(K, C)]
```

V and C can be different data types.

Generic function to combine the elements for each key using a custom set of aggregation functions.

The `combineByKey()` transformation turns an `RDD[(K, V)]` into a result of type `RDD[(K, C)]`, for a “combined type” C. Note that V and C can be different data types (this is the power of `combineByKey()`), but for this example, both are Integer data types.

The `combineByKey()` interface allows you to customize combining behavior. This transformation enable us to create a custome combined data type C as well as customizing the reduction and combining behavior.

To use this transformation, we have to provide three functions:

- `create_combiner`, which turns a single V into a C (e.g., creates a one-element list). This is used within a single partition to initialize a C.
- `merge_value`, to merge a V into a C (e.g., adds it to the end of a list). This is used within a single partition to aggregate values into a C.
- `merge_combiners`, to combine two C’s into a single C (e.g., merges the lists). This is used in merging values from two partitions.

```
>>> sum_per_key = key_value_pairs.combineByKey(
...     (lambda v: v), ❶
```

```

...
    (lambda C,v: C+v), ②
...
    (lambda C1,C2: C1+C2) ③
...
)
>>> sum_per_key.collect()
[('jane', 10), ('rafa', 15), ('alex', 14), ('clint', 9)]

```

- ❶ `create_combiner` : initial value per partition
- ❷ `merge_value` : used on single partitions
- ❸ `merge_combiners` : combine partitions into final result

Overall, the `combineByKey()` transformation is the most powerful reduction in Spark, since the data type of values (V) of source RDD can be different from the data type of values (C) of target RDD. For example, `reduceByKey()` is a very special case of `combineByKey()`: V and C are the same data types. For example, using `combineByKey()`, V can be an Integer data type, while C can be a pair of (`Float`, `Integer`) or other composite data types.

To see the real power of the `combineByKey()` transformation, lets find mean of values per key. To solve this, we create a combined data type (C) as (`sum`, `count`), which will hold the sum of values and their associated count:

```

# C = combined type as (sum, count)
>>> sum_count_per_key = key_value_pairs.combineByKey(
...     (lambda v: (v, 1)),
...     (lambda C,v: (C[0]+v, C[1]+1)),
...     (lambda C1,C2: (C1[0]+C2[0], C1[1]+C2[1]))
...
)
>>> mean_per_key = sum_count_per_key.mapValues(lambda C: C[0]/C[1])

```

Given 3 partitions named {P1, P2, P3}, the following figure shows how to create a Combiner (data type C), how to Merge a value into a Combiner, and finally how to merge two combiners.

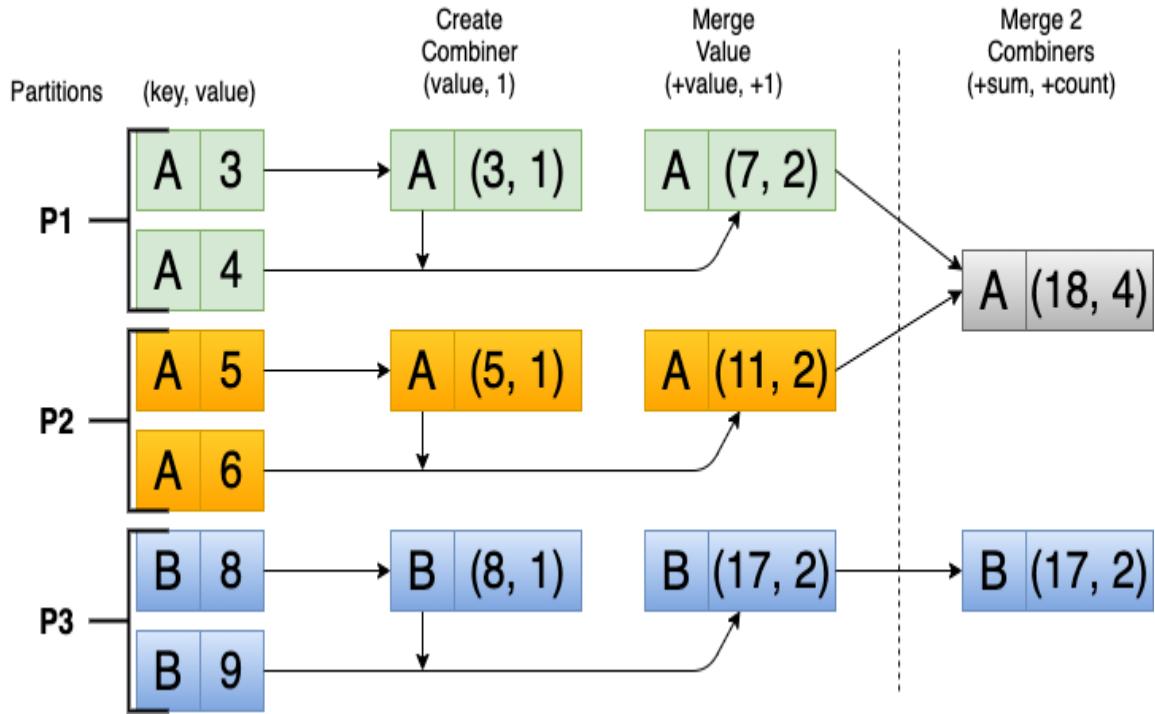


Figure 4-4. The `combineByKey()` transformation example

Next, I will discuss the concept of a Monoid, which will help us to understand the concept of combiners in reduction transformations.

What is a Monoid?

Since Spark's reductions execute on a partition by partition basis (i.e., your reducer function is distributed rather than being a sequential function), you need to make sure that your reducer function is semantically correct. To write proper reducers, which will generate correct output and results, we do need to understand the concept of a **monoid**. When reducing values, if your reducer function is not a monoid, then your final result will not be a correct value. This will be demonstrated shortly. In a nutshell, we can say that reducers are morphisms of monoids. The first step in creating a proper reducer is to identify the monoid. I will define monoid and provide some related examples shortly.

In algebra, a monoid is an algebraic structure with a single associative binary operation and an identity element (also called a **Zero** element). For

our purposes, I will provide an informal definition of a monoid:

$M = (T, f, \text{Zero})$ is a monoid, where

- T is a data type
- $f()$ is a binary operation: $f: (T, T) \rightarrow T$
- $\text{Zero}: T$ (an instance of T)

The Zero is an identity (neutral) element of type T and does not necessarily mean number zero. With the properties specified below, the triple (T, f, Zero) is called a monoid. Here are the monoidic properties:

Let a, b, c, Zero be type of T

Then the following properties must hold:

- **Binary operation:**

$f: (T, T) \rightarrow T$

- **Neutral element:**

for all a in T :

$$\begin{aligned}f(\text{Zero}, a) &= a \\f(a, \text{Zero}) &= a\end{aligned}$$

- **Associativity:**

for all a, b, c in T :

$$f(f(a, b), c) = f(a, f(b, c))$$

Not every binary operation in the world is a monoid. For example, the mean (average of numbers over a set of Integers) function is not a monoid and the proof is given below. Below we prove that the `mean()` is not an associative function and therefore it is not a monoid.

```
mean(10, mean(30, 50)) != mean(mean(10, 30), 50)
```

where

```
mean(10, mean(30, 50))
= mean (10, 40)
= 25
```

```
mean(mean(10, 30), 50)
= mean (20, 50)
= 35
```

25 != 35

Therefore, `mean()` function over a set of Integers is not a monoid. Let's look at some examples.

Monoid Examples

To help you understand monoids, here are some monoid examples:

- Integers with addition:

$$((a + b) + c) = (a + (b + c))$$

$$0 + n = n$$

$$n + 0 = n$$

The Zero element for addition is number 0.

- Integers with multiplication:

$$((a * b) * c) = (a * (b * c))$$

$$1 * n = n$$

$$n * 1 = n$$

The Zero element for multiplication is number 1.

- Strings with concatenation:

$$(a + (b + c)) = ((a + b) + c)$$

$$"" + s = s$$

$$s + "" = s$$

The Zero element for concatenation is an empty string of size 0.

- Lists with concatenation:

$$\text{List}(a, b) + \text{List}(c, d) = \text{List}(a, b, c, d)$$

- Sets with their union:

$$\text{Set}(1,2,3) + \text{Set}(2,4,5)$$

$$= \text{Set}(1,2,3,2,4,5)$$

$$= \text{Set}(1,2,3,4,5)$$

$$S + [] = S$$

$$[] + S = S$$

The Zero element is an empty set [].

Non-Monoid Examples

Here are some non-monoid examples:

- Integers with mean function:

$$\text{mean}(\text{mean}(a,b),c) \neq \text{mean}(a, \text{mean}(b,c))$$

- Integers with subtraction:

$$((a - b) - c) \neq (a - (b - c))$$

- Integers with division:

$$((a / b) / c) \neq (a / (b / c))$$

- Integers with mode:

```
mode(mode(a, b), c) != mode(a, mode(b, c))
```

- Integers with median:

```
median(median(a, b), c) != median(a, median(b, c))
```

Therefore, when writing a reducer, you need to make sure that your reduction function is a monoid, otherwise your reduced value might not be correct. This is because all algorithms operate in parallel on partitioned data: this means that writing distributed algorithms on Spark are much different than writing sequential algorithms on a single server. In some cases, it is possible to convert a non-monoid into a monoid. For example, to find mean of numbers, with a simple change to our data structures we are able to find the correct mean of numbers. There is no algorithm to convert a non-monoid structure to a monoid automatically.

Next, I introduce a simple problem (movies and users) and then provide solutions using reduce by key transformations.

Movie Problem

The goal of this example is to present a basic problem and then provide solutions by using different Spark reduction transformations by means of PySpark. For all reduction transformations, I have carefully selected the data types such that they form a **monoid**.

The movie problem is stated as: given a set of users, movies, and ratings, the goal is to find an average rating of all movies by a user. Therefore if a user (with userID=100) has rated 4 movies (rating is in the range of 1 to 5):

```
(100, "Lion King", 4.0)
(100, "Crash", 3.0)
(100, "Dead Man Walking", 3.5)
(100, "The Godfather", 4.5)
```

Then we want to generate the following output:

(100, 3.75)

where

$$\begin{aligned} 3.75 &= \text{mean}(4.0, 3.0, 3.5, 4.5) \\ &= (4.0 + 3.0 + 3.5 + 4.5) / 4 \\ &= 15.0 / 4 \end{aligned}$$

For this example, note that `reduceByKey()` transformation over a set of ratings will not always produce the correct output, since the mean of means is not equal to the mean of all input numbers. Average (or mean) is not an algebraic monoid over a set of float/integer numbers) and here is a simple proof:

$$\begin{aligned} \text{mean}(1, 2, 3, 4, 5, 6) \\ &= (1 + 2 + 3 + 4 + 5 + 6) / 6 \\ &= 21 / 6 \\ &= 3.5 \text{ [correct result]} \end{aligned}$$

Now, let's make a mean function as distributed function (we used 3 partitions here):

```
Partition-1: (1, 2, 3)
Partition-2: (4, 5)
Partition-3: (6)
```

Next, we compute the mean of partitions:

```
mean(1, 2, 3, 4, 5, 6)
= mean (
    mean(Partition-1),
    mean(Partition-2),
    mean(Partition-3)
)

mean(Partition-1)
= mean(1, 2, 3)
= mean( mean(1,2), 3)
```

```

= mean( (1+2)/2, 3)
= mean(1.5, 3)
= (1.5+3)/2
= 2.25

```

```

mean(Partition-2)
= mean(4,5)
= (4+5)/2
= 4.5

```

```

mean(Partition-3)
= mean(6)
= 6

```

Once all partitions are processed, therefore:

```

mean(1, 2, 3, 4, 5, 6)
= mean (
    mean(Partition-1),
    mean(Partition-2),
    mean(Partition-3)
)
= mean(2.25, 4.5, 6)
= mean(mean(2.25, 4.5), 6)
= mean((2.25 + 4.5)/2, 6)
= mean(3.375, 6)
= (3.375 + 6)/2
= 9.375 / 2
= 4.6875 [incorrect result]

```

To compute mean of ratings per user, we can use a monoid data structure (which supports associativity and commutativity) such as a pair of (`sum`, `count`), where `sum` is the total sum of all numbers — ratings — we have visited (per partition) and `count` is the number of ratings we have visited so far:

Let's define: `mean(pair(sum, count)) = sum / count`

```

mean(1,2,3,4,5,6)
= mean (mean(1,2,3), mean(4,5), mean(6))
= mean( pair(1+2+3, 1+1+1), pair(4+5, 1+1), pair(6,1))
= mean( pair(6, 3), pair(9, 2), pair(6,1))
= mean( mean(pair(6, 3), pair(9, 2)), pair(6,1))

```

```

= mean( pair(6+9, 3+2), pair(6,1))
= mean( pair(15, 5), pair(6,1))
= mean( pair(15+6, 5+1))
= mean( pair(21, 6))
= 21 / 6 = 3.5 [correct result]

```

Note that mean of different partitions is not associative, but by using monoid (we force associativity and commutativity— defined below) we can achieve associativity. Therefore, you may apply the

```

# a = (sum1, count1)
# b = (sum2, count2)
# f(a, b) = a + b
#           = (sum1+sum2, count1+count2)
#
reduceByKey(lambda a, b: f(a, b))

```

when your function $f()$ is commutative and associative. For example, the addition (+) operation is commutative and associative, but the average function does not satisfy the Commutative and Associative properties.

- **Commutative:** ensuring that the result would be independent of the order of elements in the RDD being aggregated

$$f(A, B) = f(B, A)$$

- **Associative:** ensuring that any two elements associated in the aggregation at a time does not effect the final result

$$f(f(A, B), C) = f(A, f(B, C))$$

Therefore, to find the average per key, we can use `reduceByKey()`, but we have to change our combined data structure to be a monoid, which are presented in the following sections.

Input Data Set to Analyze

To show different Spark solutions to our problem, we use a data set from [MovieLens](#). Consider a set of data (file named as `ratings.csv`), which has `users`, `movies`, and `ratings`. I downloaded the input from [MovieLens](#). According to MovieLens: “these datasets will change over time”. So at the time of your download, these data sizes might have changed. For simplicity, I am assuming that you have downloaded and unzipped the files at `/tmp/movielens/` directory. Note that, there is not any requirement to put the files under my suggested location and you may place your files at your preferred directory and hence update your input-paths accordingly.

The data we want to analyze has the following properties:

- 22,000,000 ratings
- 580,000 tag applications
- 33,000 movies
- 240,000 users

For details, please visit the following links:

- [Latest MovieLens Data](#)
- [Latest MovieLens README](#)

The data download can be done from [this link from GroupLens](#).

Note that, the full movie dataset (`ml-latest.zip`) is 264 MB. If you want to run/test/debug the programs listed here by a small movies data set (the smaller set is more manageable), then you may download it from the [Latest MovieLens Small Dataset](#).

Ratings Data File Structure (`ratings.csv`)

All ratings are contained in the file `ratings.csv`. Each line of this file after the header row represents one rating of one movie by one user, and has the following format:

```
<userId><,><movieId><,><rating><,><timestamp>
```

Let's understand the input file `ratings.csv`:

- The lines within this file are ordered first by `userId`, then, within `userId`, by `movieId`.
- Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).
- Timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970 (this field is ignored in our analysis).

After unzipping the downloaded file, you should have the following files:

```
# ls -l /tmp/movielens/
      8,305  README.txt
      725,770  links.csv
     1,729,811  movies.csv
    620,204,630  ratings.csv
    21,094,823  tags.csv
```

Find out the number of records:

```
# wc -l /tmp/movielens/ratings.csv
 22,884,378 /tmp/movielens/ratings.csv
```

Next, examine the content of `ratings.csv`:

```
# head -6 /tmp/movielens/ratings.csv
userId,movieId,rating,timestamp
1,169,2.5,1204927694
1,2471,3.0,1204927438
1,48516,5.0,1204927435
2,2571,3.5,1436165433
2,109487,4.0,1436165496
```

Since we are using RDDs, we do not need the metadata associated with data. Therefore, I removed the first line (as header line) from the `ratings.txt`

file:

```
# tail -n +2 ratings.csv > ratings.csv.no.header
# wc -l ratings.csv ratings.csv.no.header
22,884,378 ratings.csv
22,884,377 ratings.csv.no.header
```

Solution Using aggregateByKey()

To find average rating by user, first we do the following step: map each record into (key, value) pairs as:

(userID-as-key, rating-as-value)

The simplest way to add up your values per key is to use the `reduceByKey()`. But, we can not use the `reduceByKey()` to find average rating by user since the mean/average function is not a **monoid** over a set of ratings (as float numbers). To preserve a monoid operation, we use a pair data structure (a tuple of 2 elements) to hold a pair of values: (`sum`, `count`) where `sum` is the aggregated sum of ratings and `count` is the number of ratings we have added (i.e., `sum`) so far.

Let's prove that the pair structure (`sum`, `count`) with an "addition" operator over a set of numbers is a monoid:

- **Zero Element**

The Zero element is (0.0, 0)

$$\begin{aligned}f(A, \text{Zero}) &= A \\f(\text{Zero}, A) &= A\end{aligned}$$

$$A = (\text{sum}, \text{count})$$

$$\begin{aligned}f(A, \text{Zero}) \\&= (\text{sum}+0.0, \text{count}+0) \\&= (\text{sum}, \text{count})\end{aligned}$$

= A

$$\begin{aligned}f(\text{Zero}, A) \\= (0.0+\text{sum}, 0+\text{count}) \\= (\text{sum}, \text{count}) \\= A\end{aligned}$$

- **Commutative:** ensuring that the result would be independent of the order of elements in the RDD being aggregated

$$f(A, B) = f(B, A)$$

$$\begin{aligned}A &= (\text{sum1}, \text{count1}) \\B &= (\text{sum2}, \text{count2})\end{aligned}$$

$$\begin{aligned}f(A, B) \\= (\text{sum1}+\text{sum2}, \text{count1}+\text{count2}) \\= (\text{sum2}+\text{sum1}, \text{count2}+\text{count1}) \\= f(B, A)\end{aligned}$$

- **Associative:** ensuring that any two elements associated in the aggregation at a time does not effect the final result

$$f(f(A, B), C) = f(A, f(B, C))$$

$$\begin{aligned}A &= (\text{sum1}, \text{count1}) \\B &= (\text{sum2}, \text{count2}) \\C &= (\text{sum3}, \text{count3})\end{aligned}$$

$$\begin{aligned}f(f(A, B), C) \\= f((\text{sum1}+\text{sum2}, \text{count1}+\text{count2}), (\text{sum3}, \text{count3})) \\= (\text{sum1}+\text{sum2}+\text{sum3}, \text{count1}+\text{count2}+\text{count3}) \\= (\text{sum1}+(\text{sum2}+\text{sum3}), \text{count1}+(\text{count2}+\text{count3})) \\= f(A, f(B, C))\end{aligned}$$

Therefore, we frequently use `aggregateByKey()` to do more complicated calculations (like averages). Note that the `aggregateByKey()` is more suitable for compute aggregations for keys, example aggregations such as `sum`, `avg`, `standard deviation`, etc.

First take a look at the signature of `aggregateByKey()` in simple form:

```
aggregateByKey(zero_value, seq_func, comb_func)
```

To use `aggregateByKey()`, programmer has to provide the following 3 basic functions. Below, `C` is a combined data type:

```
aggregateByKey : RDD[(K, V)] --> RDD[(K, C)]
```

```
zero_value -> C ❶  
seq_func(C, V) -> C ❷  
comb_func(C, C) -> C ❸
```

- ❶ Create a `C` from `zero_value` (so called an initial value) per partition
- ❷ Merge a `V` and a `C` into a single `C` (inside a partition)
- ❸ Combine two `C`'s into a single `C` (combining two partitions)

`C` is a combined data structure, which in our case here, denotes a pair of (`sum`, `count`). Aggregate the values of each key, using given combine functions and a neutral “zero value” (the “zero value” is really not the zero value such as 0 — also it can be real zero if desired), but a starting initial value per partition). This function can return a different result type, `C`, than the type of the values in this RDD, `V`. Thus, we need one operation for merging a `V` into a `C` and one operation for merging two's. The former operation is used for merging values within a partition, and the latter is used for merging values between partitions. To avoid memory allocation, both of these functions are allowed to modify and return their first argument instead of creating a new `C`.

To make things simple, we define a very basic python function, `create_pair()`, which accepts a record of movie rating data and return a pair of (`userID`, `rating`):

```
# define a function, which accepts a CSV record
# and returns a pair of (userID, rating)
# Parameters: rating_record (as CSV String)
# rating_record = "userID,movieID,rating,timestamp"
def create_pair(rating_record):
    tokens = rating_record.split(",")
    userID = tokens[0]
    rating = float(tokens[2])
    return (userID, rating)
#end-def
```

Next we test the Python function:

```
key_value_1 = create_pair("3,2394,4.0,920586920")
print key_value_1
('3', 4.0)
#
key_value_2 = create_pair("1,169,2.5,1204927694")
print key_value_2
('1', 2.5)
```

Here is a PySpark solution using `aggregateByKey()`. The combined type (as `C`) to denote values for the `aggregateByKey()` is a pair of (`sum-of-ratings`, `count-of-ratings`).

```
# spark : an instance of SparkSession
ratings_path = "/tmp/movielens/ratings.csv.no.header"
rdd = spark.sparkContext.textFile(ratings_path)
# load user-defined python function
ratings = rdd.map(lambda rec : create_pair(rec)) ❶
ratings.count()
#
# C = (C[0], C[1]) = (sum-of-ratings, count-of-ratings)
# zero_value -> C = (0.0, 0)
# seq_func: (C, V) -> C
# comb_func: (C, C) -> C
sum_count = ratings.aggregateByKey( ❷
    (0.0, 0), ❸
    (lambda C, V: (C[0]+V, C[1]+1)), ❹
```

```
(lambda C1, C2: (C1[0]+C2[0], C1[1]+C2[1])) ❸  
)
```

- ❶ Source RDD is `ratings = [(userID, rating), ...] = RDD[(String, Float)]`
- ❷ Target RDD is `sum_count = [(userID, (sum-of-ratings, count-of-ratings)), ...] = RDD[(String, (Float, Integer))]`
- ❸ `zero_value` : initial value per partition
- ❹ `seq_func` : used within a single partition
- ❺ `comb_func` : used to combine results of partitions

Let's break down what's happening by each line. Call the `aggregateByKey()` function and create a result set "template" with the initial values.

We're starting the data out as `(0.0, 0)` which will hold our sum of ratings (as `0.0`) and count of records (as `0`). For each row of data we're going to do some adding. Note that `(0.0, 0)` is so called a `zero_value`, which is the initial value per partition.

`C` is the new template, so `C[0]` is referring to our "sum" element (sum-of-ratings) where `C[1]` is the "count" element (count-of-ratings).

`V` is a row's worth of the original data. So you have to pull the right element from the original data. `V` is the rating.

Final step, you're combining RDDs if they were processed on multiple partitions on different machines. Simply add `C1` values to `C2` values based on the template we made.

The data in `sum_count` RDD will end up looking like:

```
sum_count  
= [(userID, (sum-of-ratings, count-of-ratings)), ...]
```

```

= RDD[(String, (Float, Integer))]

[
  (100, (40.0, 10)),
  (200, (51.0, 13)),
  (300, (340.0, 90)),
  ...
]

```

Therefore, `userID=100` have rated 10 movies and sum of all his ratings is `40.0` and `userID=200` have rated 13 movies and sum of all his ratings is `51.0`, and so on.

So in order to get the actual average, we need to call the `mapValues()` transformation and divide the first entry (sum) by the second entry (count).

```

# x = (sum-of-ratings, count-of-ratings)
# x[0] = sum-of-ratings
# x[1] = count-of-ratings
# avg = sum-of-ratings / count-of-ratings
average_rating = sum_count.mapValues(lambda x: (x[0]/x[1])) ❶

```

- ❶ `average_rating = RDD[(String, Float)] = RDD[(userID, average-rating)]`

Results

Let's examine `average_rating` RDD:

```

average_rating
[
  (100, 4.00),
  (200, 3.92),
  (300, 3.77),
  ...
]

```

Next, I present the logic behind `aggregateByKey()`.

How does aggregateByKey() work?

Per key, each partition is initialized with the zero value, which is an initial combined data type (C). Then the zero value is merged with the first value in the partition, which creates a new C . Then the second value is merged with the resulting C and this continues until we merge all values. If the same key goes to multiple partitions, then these values are combined together, which results in a new C .

Figures 4.5 and 4.6 show how `aggregateByKey()` works with different zero values. The zero value is applied per key per partition. This means that if a key X is in N partitions, then the zero-value is applied N times (each of these N partitions will be initialized to zero-value for key X).

Figures 4.5 show that how `aggregateByKey()` works with `zero-value= (0, 0)`.

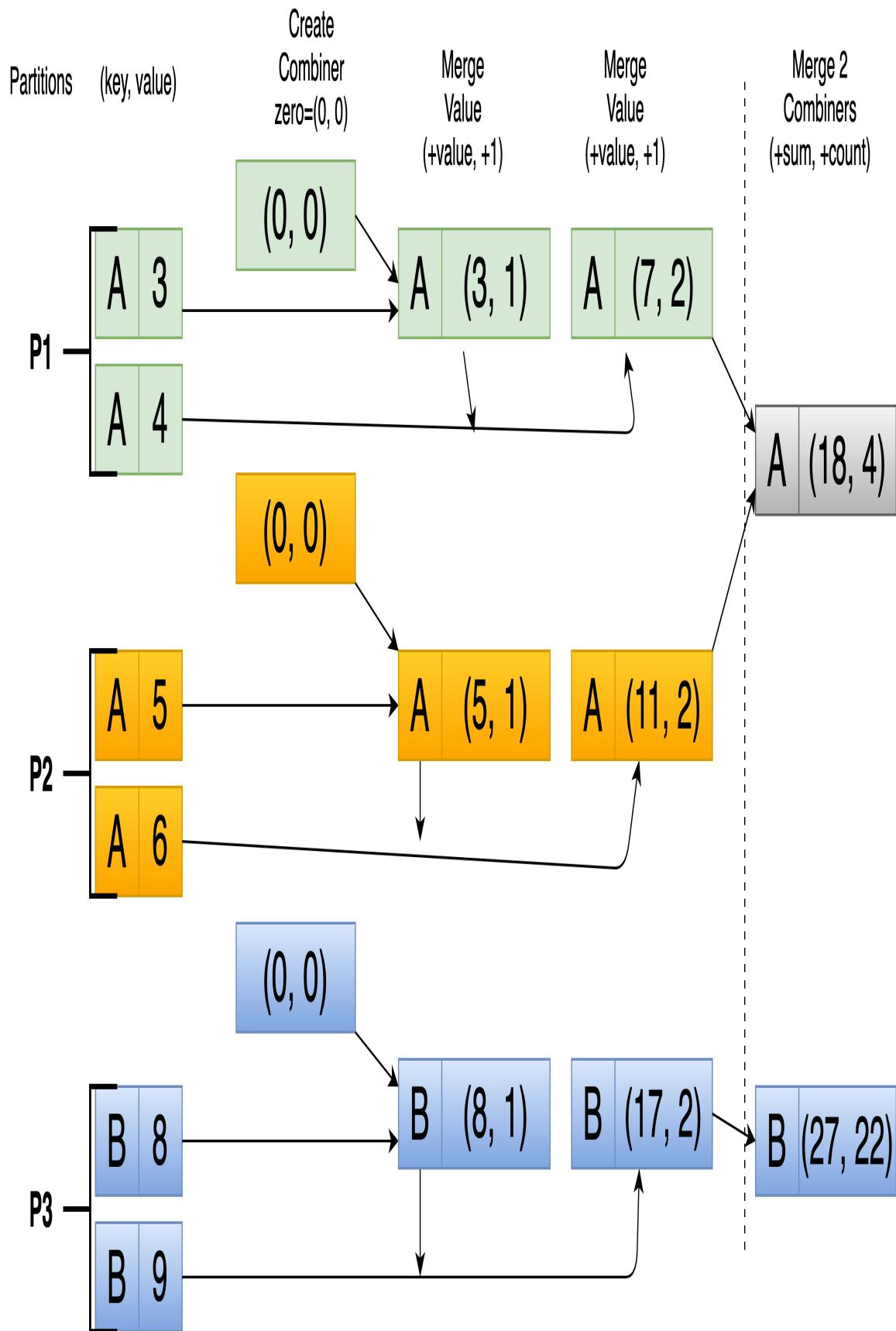
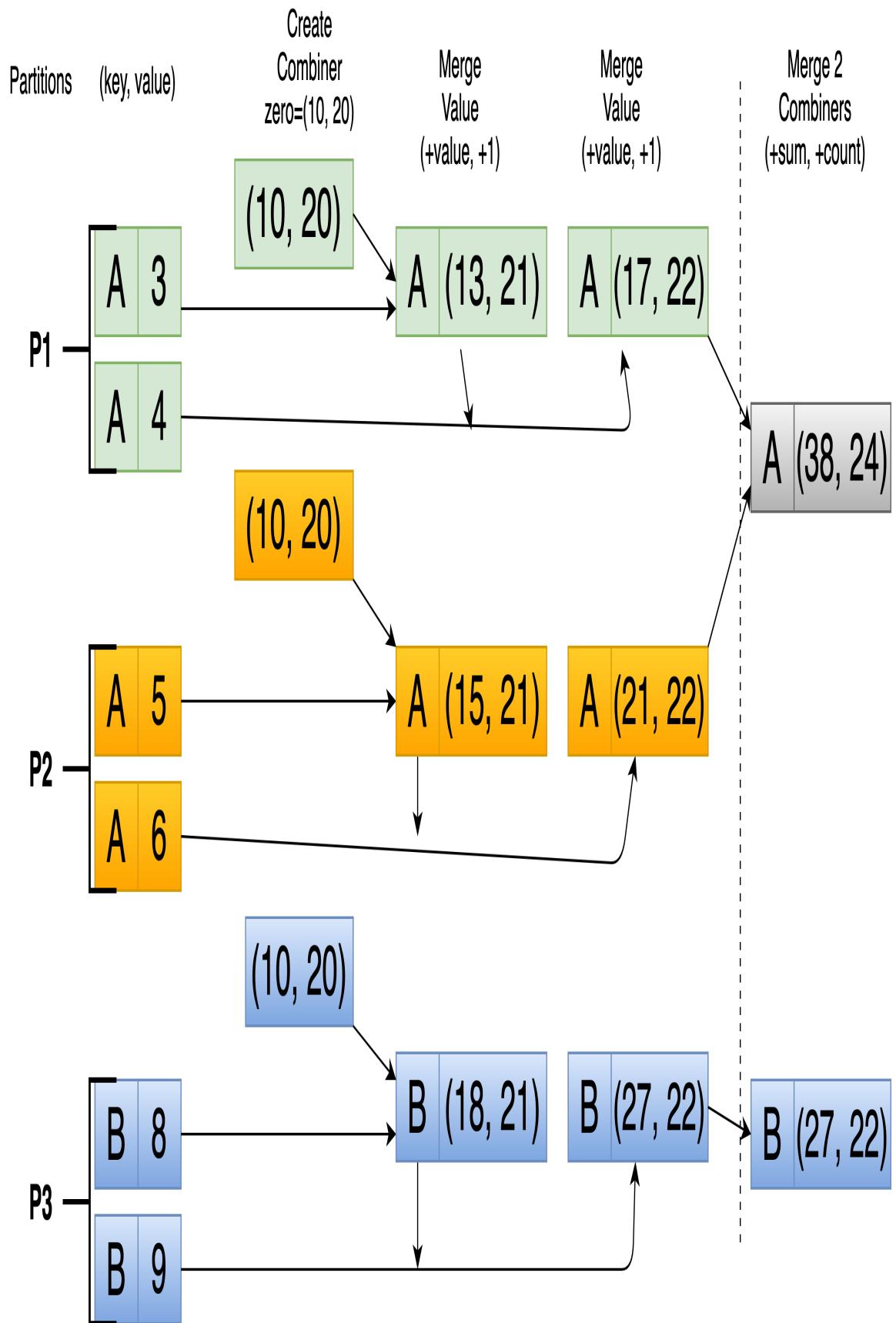


Figure 4-5. aggregateByKey() with zero-value=(0, 0)

Note that the zero-value is used per key per partition. Therefore you should select the zero-value in such a way to not give you wrong results.

Figures 4.6 show that how `aggregateByKey()` works with `zero-value=(10, 20)`. `aggregateByKey()` with `zero-value=(10, 20)`



Next, I present a complete solution to the movies problem by using `aggregateByKey()`.

PySpark Solution using `aggregateByKey()`

Here, I present a solution by using the `aggregateByKey()` transformation. To save space, I have trimmed the output generated by PySpark shell.

Step 1: Read Data and Create Pairs

Here we read data and create (key, value) pairs, where key is a userID and value is a rating.

```
# ./bin/pyspark
SparkSession available as 'spark'.
>>># create_pair() returns a pair (userID, rating)
>>># rating_record = "userID,movieID,rating,timestamp"
>>> def create_pair(rating_record):
...     tokens = rating_record.split(",")
...     return (tokens[0], float(tokens[2]))
...
>>> key_value_test = create_pair("3,2394,4.0,920586920")
>>> print key_value_test
('3', 4.0)
>>> ratings_path = "/tmp/movielens/ratings.csv.no.header"
>>> rdd = spark.sparkContext.textFile(ratings_path)
>>> rdd.count()
22884377
>>> ratings = rdd.map(lambda rec : create_pair(rec))
>>> ratings.count()
22884377
>>> ratings.take(3)
[(u'1', 2.5), (u'1', 3.0), (u'1', 5.0)]
```

Step 2: Use `aggregateByKey()` to Sum Up Ratings

Once (key, value) pairs are created, we now can apply the `aggregateByKey()` transformation to sum up the ratings. The initial value of `(0.0, 0)` is used

per partition: where 0.0 is the sum of the ratings and 0 is the number of raters.

```
>>> # C is a combiner data structure as (sum, count)
>>> sum_count = ratings.aggregateByKey( ❶
...     (0.0, 0), ❷
...     (lambda C, V: (C[0]+V, C[1]+1)), ❸
...     (lambda C1, C2: (C1[0]+C2[0], C1[1]+C2[1]))) ❹

>>> sum_count.count()
247753

>>> sum_count.take(3)
[
  (u'145757', (148.0, 50)),
  (u'244330', (36.0, 17)),
  (u'180162', (1882.0, 489))
]
```

- ❶ Target RDD will be `RDD[(String, (Float, Integer))]`
- ❷ `zero_value`: C is initialized to $(0.0, 0)$ per partition
- ❸ `seq_func`: add a single value of V to a C (used in a single partition)
- ❹ `comb_func`: combine values of partitions (add two C's to create a single C)

You have an option of using Python functions instead of lambda expressions. To compute `sum_count` with functions, we need to write the following functions:

```
# C = (sum, count)
# V is a single value of type Float
def seq_func(C, V):
    return (C[0]+V, C[1]+1)
#end-def

# C1 = (sum1, count1)
# C2 = (sum2, count2)
def comb_func(C1, C2):
```

```
    return (C1[0]+C2[0], C1[1]+C2[1])
#end-def
```

Now, we can compute `sum_count` by the defined functions:

```
sum_count = ratings.aggregateByKey(
    (0.0, 0),
    seq_func,
    comb_func
)
```

Step 3: Find Average Rating

The previous step created RDD elements of the following type:

```
(userID, (sum-of-ratings, number-of-ratings))
```

Next, we do the final calculation to find the average rating:

```
>>> # x refers to a pair of (sum-of-ratings, number-of-ratings)
>>> # where
>>> #     x[0] denotes sum-of-ratings
>>> #     x[1] denotes number-of-ratings
>>
>>> average_rating = sum_count.mapValues(lambda x:(x[0]/x[1]))
>>> average_rating.count()
247753

>>> average_rating.take(3)
[
(u'145757', 2.96),
(u'244330', 2.1176470588235294),
(u'180162', 3.8486707566462166)
]
>>>
```

Next, I present a solution to the movies problem by using `groupByKey()`.

Complete PySpark Solution by `groupByKey()`

For a given set of (K, V) pairs, `groupByKey()` has the following signature:

```
groupByKey(numPartitions=None, partitionFunc=<function portable_hash>)
groupByKey : RDD[(K, V)] --> RDD[(K, [V])]
```

Let the source RDD be as `RDD[(K, V)]`, then the `groupByKey()` transformation groups the values for each key (say K) in the RDD into a single sequence as `[V]` (list/iterable of `V`'s). Hash-partitions the resulting RDD with the existing partitioner/parallelism level. The ordering of elements within each group is not guaranteed, and may even differ each time the resulting RDD is evaluated.

Based on the API, you may customize the number of partitions (`numPartitions`) and partitioning function (`partitionFunc`).

Note that this operation may be very expensive. If you are grouping large number of values in order to perform an aggregation (such as a sum or average or a statistical function) over each key, using `combineByKey()`, `aggregateByKey()` or `reduceByKey()` will provide much better scalability and performance. Also note that the `groupByKey()` transformation is assuming that the data for a key will fit in memory, if you have large amount of data for a give key which will not fit in memory, then you might get “out of memory” error.

When possible, you should avoid using `groupByKey()`. While `groupByKey()` and `reduceByKey()` transformations can produce the correct answer, the `reduceByKey()` works much better (i.e., scales-out better) on a large dataset. That’s because Spark knows it can combine output with a common key on each partition before shuffling the data.

Here are more functions to prefer over `groupByKey()`:

- `combineByKey()` can be used when you are combining elements but your return type may differ from your input value type.
- `foldByKey()` merges the values for each key using an associative function and a neutral “zero value” (initial value per partition).

PySpark Solution using groupByKey()

Here, I present a complete solution by using the `groupByKey()` transformation:

Step 1: Read Data and Create Pairs

Here we read data and create (key, value) pairs, where key is a userID and value is a rating.

```
>>># spark : SparkSession
>>> def create_pair(rating_record):
...     tokens = rating_record.split(",")
...     return (tokens[0], float(tokens[2]))
...
>>> key_value_test = create_pair("3,2394,4.0,920586920")
>>> print key_value_test
('3', 4.0)

>>> ratings_path = "/tmp/movielens/ratings.csv.no.header"
>>> rdd = spark.sparkContext.textFile(ratings_path)
>>> rdd.count()
22884377
>>> ratings = rdd.map(lambda rec : create_pair(rec)) ❶
>>> ratings.count()
22884377
>>> ratings.take(3)
[
(u'1', 2.5),
(u'1', 3.0),
(u'1', 5.0)
]
```

- ❶ `ratings` is an `RDD[(String, Float)] = RDD[(userID, rating)]`

Step 2: Use groupByKey() to Group Ratings

Once (key, value) pairs are created, we now can apply the `groupByKey()` transformation to group all ratings for a user. This step creates (`userID`,

$[R_1, \dots, R_n]$), where R_1, \dots, R_n are all of the rating for a unique user ID.

As you will notice below, the `groupByKey()` transformation works exactly as SQL's GROUP BY semantics. It groups values of the same key as iterable of values.

```
>>> ratings_grouped = ratings.groupByKey() ❶
>>> ratings_grouped.count()
247753
>>> ratings_grouped.take(3)
[
  (u'145757', <ResultIterable object at 0x111e42e50>), ❷
  (u'244330', <ResultIterable object at 0x111e42dd0>),
  (u'180162', <ResultIterable object at 0x111e42e10>)
]
>>> ratings_grouped.mapValues(lambda x: list(x)).take(3) ❸
[
  (u'145757', [2.0, 3.5, ..., 3.5, 1.0]),
  (u'244330', [3.5, 1.5, ..., 4.0, 2.0]),
  (u'180162', [5.0, 4.0, ..., 4.0, 5.0])
]
```

- ❶ `ratings_grouped` is an `RDD[(String, [Float])] = RDD[(String, [rating1, rating2, ...])]`
- ❷ The full name of `ResultIterable` is
`pyspark.resultiterable.ResultIterable`
- ❸ For debugging, convert `ResultIterable` object to list of Integers

Step 3: Find Average Rating

To find average rating per user, we sum up all ratings and then calculate the average.

```
>>> # x refers to all ratings for a user as [R1, ..., Rn]
>>> # x : ResultIterable object
>>> average_rating = ratings_grouped.mapValues(lambda x: sum(x)/len(x)) ❶
>>> average_rating.count()
```

```
247753
>>> average_rating.take(3)
[
    (u'145757', 2.96),
    (u'244330', 2.12),
    (u'180162', 3.85)
]
>>>
```

- ❶ `average_rating` is $\text{RDD}[(\text{String}, \text{Float})] = \text{RDD}[(\text{userID}, \text{average-rating})]$

I presented multiple solutions to the same movie problem by using `reduceByKey()`, `aggregateByKey()` `combineByKey()`, and `groupByKey()`. This means that there are many ways to solve the same data problem. When possible `reduceByKey()` is preferable to all others. But for solving some problems (such as finding median of values per key), if you need all values at the same time, then `groupByKey()` is a preferred transformation. Next I examine the “Shuffle Step” in reduction transformations.

Shuffle Step in Reductions

Once all mappers have completed emitting (key, value) pairs, then the MapReduce’s magic happens: the sort and shuffle step. The sort and shuffle basically groups data by their associated keys and sends the results to reducers. This step is different (from the efficiency and scalability point of view) for different transformations.

In a nutshell, shuffling is a process of redistributing data across partitions that may or may not cause moving data across JVM processes or even over the wire (between executors on separate servers). Shuffling is the process of data transfer between stages (to be explained shortly).

What is the shuffle in general? I am going to explain the “shuffling” concept by an example. Imagine that you have a 100-nodes Spark cluster and each

node has a list of (URL, frequency) pairs in a table and you want to calculate the total frequency per URL. This way you would set the “URL” as your key, and for each pair you would emit “frequency” as a value. After this you would sum up frequencies for each key (i.e., URL), which would be an answer to your question — total amount of frequencies for each unique URL. But when you store the data across the cluster, how can you sum up the values for the same key stored on different servers? The only way to do so is to make all the values for the same key be on the same server, after this you would be able to sum them up. This process is called shuffling.

There are many transformations (such as `reduceByKey()` and `join()`) that require shuffling of the data across the cluster, for instance table join – to join two RDDs on the field “chromosomeID”, you must be sure that all the data for the same values of “chromosomeID” for both of the RDDs are stored in the same chunks. These examples show that shuffling is important, required, and expensive operation. Shuffling data for `groupByKey()` is different from shuffling of `reduceByKey()` data. This difference in shuffling means that each transformation has a different performance. Therefore, it is very important to properly select and use reduction transformations.

I will illustrate the shuffling concept by a simple word count example. PySpark solves the word count problem by:

```
# spark : SparkSession
# we use 5 partitions for textFile(), flatMap() and map()
# we use 3 partitions for the reduceByKey() reduction
rdd = spark.sparkContext.textFile("input.txt", 5) \
    .flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b, 3) \ ❶
    .collect()
```

❶ 3 is the number of partitions

According to Spark documentation, RDD operations are compiled into a DAG (directed acyclic graph) of RDD objects, where each RDD points to the parent it depends on (illustrated as Figure 6.4). At shuffle boundaries, the

DAG is partitioned into so-called stages (Stage-1, Stage-2, ...) that are going to be executed in order. Since shuffling involves copying data across executors and servers, making the shuffle is a complex and costly operation. Since shuffling is a costly operation, we have to be careful in selecting proper reductions.

Stage-1

Stage-2

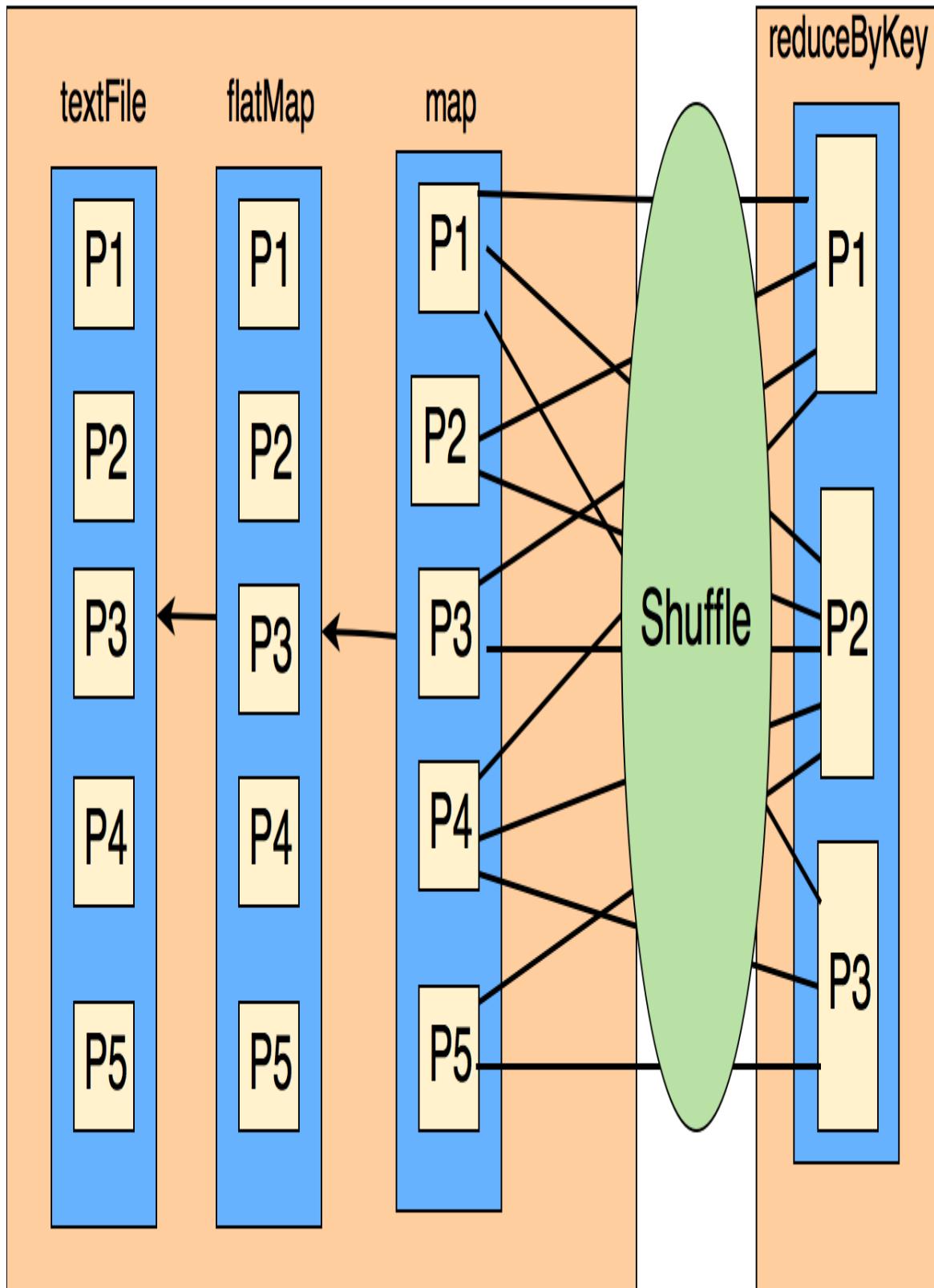


Figure 4-6. Spark's Shuffle Concept

Since we directed the `reduceByKey()` transformation to create 3 partitions, the result RDD will be partitioned into 3 chunks as depicted by Figure 4.7.

Shuffle Step for `groupByKey()`

The `groupByKey()` shuffle step is pretty straightforward. It does not merge the values for the key but directly the shuffle step happens and large volume of data gets sent to each partition (no change is done to the initial data values). The merging of values for each key happens after the shuffle step. For `groupByKey()`, a lot of data needs to be stored on final worker node (reducer) therefore resulting in OOM (out of memory error — if there are lots of data per key). The shuffle step is demonstrated below. Note that after `groupByKey()`, you do need to call `mapValues()` to generate your final desired output.

groupByKey(): Shuffle Step

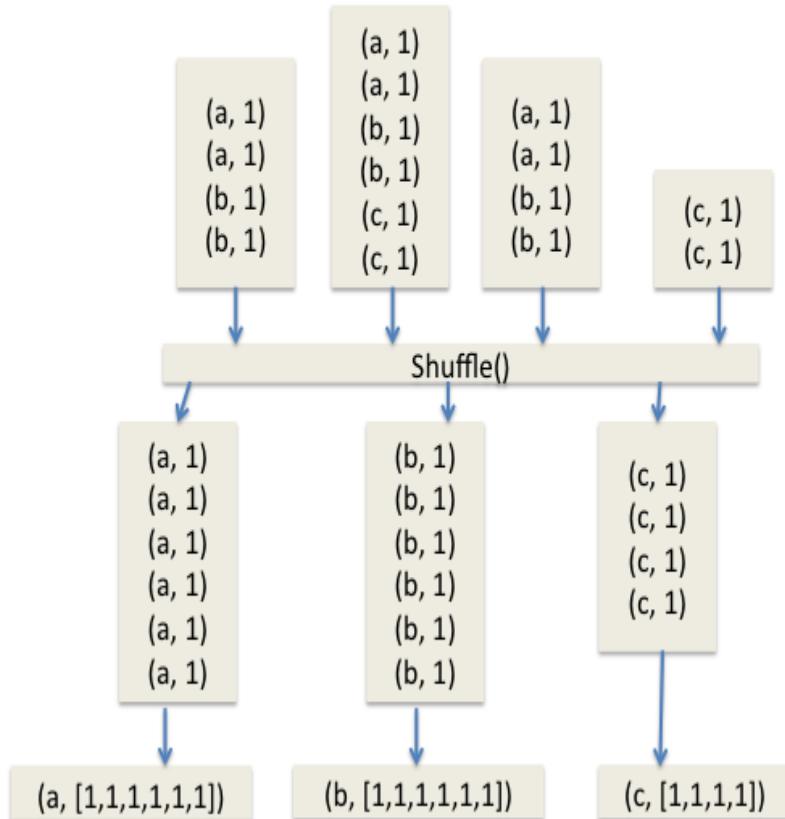


Figure 4-7. Shuffle Step for `groupByKey()`

The `groupByKey()` call makes no attempt at merging or combining values, so it's an expensive operation due to moving large amount of data over network.

Shuffle Step for `reduceByKey()`

Per worker node, data is combined so that at each partition there is at most one value for each key. Then shuffle happens and it is sent over the network to some particular executor for some action such as reduce. Note that after `reduceByKey()`, you do need to call `mapValues()` to generate you

final desired output. In general, a `reduceByKey()` can be replaced by a pair of `groupByKey()` and `mapValues()`.

reduceByKey(): Shuffle Step

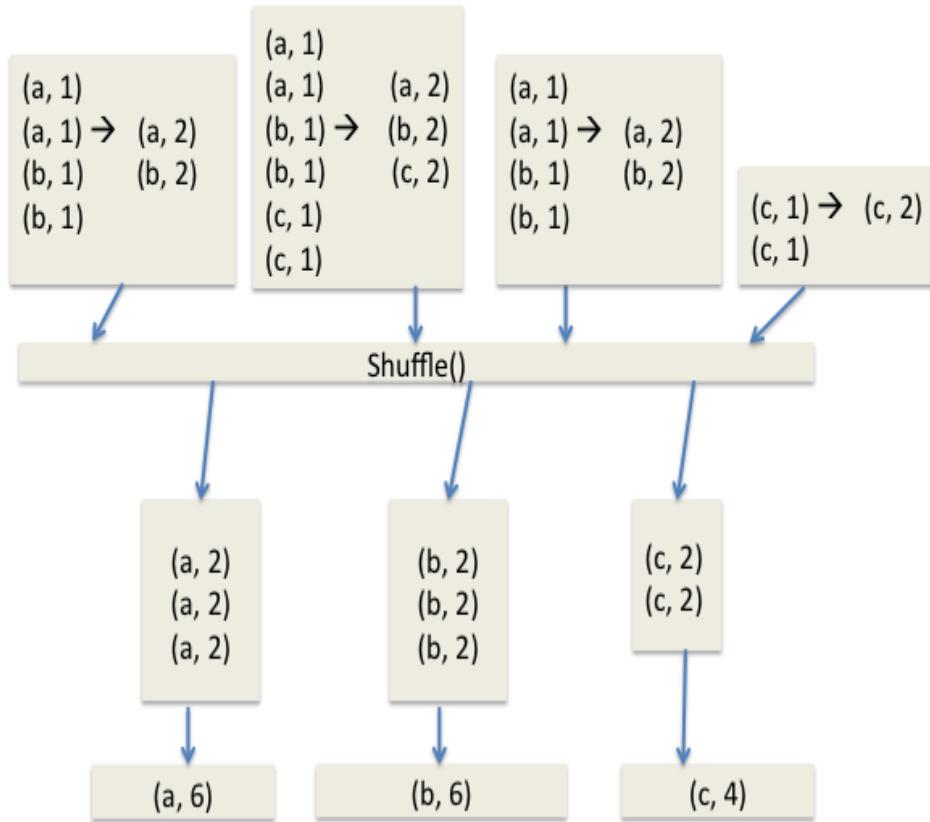


Figure 4-8. Shuffle Step for `reduceByKey()`

Complete PySpark Solution using `reduceByKey()`

In its simplest form, `reduceByKey()` transformation has the following signature (source and target data types — `V` — must be the same):

```
reduceByKey(func, numPartitions=None, partitionFunc)
reduceByKey: RDD[(K, V)] --> RDD[(K, V)]
```

The `reduceByKey()` transformation, merges the values for each key using an *associative* and *commutative* reduce function. This will also perform the merging locally on each mapper before sending results to a reducer, similarly to a "combiner" in MapReduce. Output will be partitioned with `'numPartitions` partitions, or the default parallelism level if `numPartitions` is not specified. Default partitioner is hash-partition.

Since we want to find the average rating for all movies rated by a user, and we know that mean of means is not a mean function (mean is not a monoid), therefore we will add up all ratings for a user and keep track of the number of movies rated: then (`sum_of_ratings`, `count_of_movies`) is a monoid over an addition function, but at the end we need one more `mapValues()` transformation to find the actual average rating by dividing `sum_of_ratings` over `count_of_movies`. The complete solution using `reduceByKey()` is given below. Note that `reduceByKey()` is efficient and scalable than a `groupByKey()` transformation, since merge and combine are done locally before sending data for the final reduction.

Step 1: Read Data and Create Pairs

Here we read data and create (key, value) pairs, where key is a userID and value is a pair of (rating, 1). To use `reduceByKey` for finding average, we need to find the (`sum_of_ratings`, `number_of_raters`). This is because mean function is not a “monoid”, so we create (`sum_of_ratings`, `number_of_raters`) to act as a monoid.

- Read input data and create an `RDD[String]`:

```
>>># spark : SparkSession
>>> ratings_path = "/tmp/movielens/ratings.csv.no.header"
>>># rdd: RDD[String]
>>> rdd = spark.sparkContext.textFile(ratings_path)
>>> rdd.take(3)
[
  u'1,169,2.5,1204927694',
  u'1,2471,3.0,1204927438',
```

```
    u'1,48516,5.0,1204927435'  
]
```

- Transform RDD[String] to RDD[(String, (Float, Integer))]

```
>>> def create_combined_pair(rating_record):  
...     tokens = rating_record.split(",")  
...     userID = tokens[0]  
...     rating = float(tokens[2])  
...     return (userID, (rating, 1))  
...  
>>># ratings: RDD[(String, (Float, Integer))]  
>>> ratings = rdd.map(lambda rec : create_combined_pair(rec)) ❶  
>>> ratings.count()  
22884377  
  
>>> ratings.take(3)  
[  
  (u'1', (2.5, 1)),  
  (u'1', (3.0, 1)),  
  (u'1', (5.0, 1))  
]
```

- ❶ Create pair RDD

Step 2: Use reduceByKey() to Sum up Ratings

Once (userID, (rating, 1)) pairs are created, we now can apply the `reduceByKey()` transformation to sum up all ratings and raters for a user. The output of this step will be : (userID, (sum_of_ratings, number_of_raters)).

```
>>># x refers to (rating1, frequency1)  
>>># y refers to (rating2, frequency2)  
>>># x = (x[0] = rating1, x[1] = frequency1)  
>>># y = (y[0] = rating2, y[1] = frequency2)  
>>># x + y = (rating1+rating2, frequency1+frequency2)  
>>># ratings is the source RDD ❶  
>>> sum_and_count = ratings.reduceByKey(lambda x, y: (x[0]+y[0],x[1]+y[1])) ❷  
>>> sum_and_count.count()  
247753  
>>> sum_and_count.take(3) ❸
```

```
[  
    (u'145757', (148.0, 50)),  
    (u'244330', (36.0, 17)),  
    (u'180162', (1882.0, 489))  
]
```

- ❶ Source RDD (`ratings`) is `RDD[(String, (Float, Integer))]`
- ❷ Target RDD (`sum_and_count`) is `RDD[(String, (Float, Integer))]`
- ❸ Notice that the data types for the source and target are the same

Step 3: Find Average Rating

To find average rating per user, we divide “sum of ratings” by the “number of raters”.

```
>>> # x refers to (sum_of_ratings, number_of_raters)  
>>> x = (x[0] = sum_of_ratings, x[1] = number_of_raters)  
>>> avg = sum_of_ratings / number_of_raters = x[0] / x[1]  
>>> avgRating = sum_and_count.mapValues(lambda x : x[0] / x[1])  
>>> avgRating.take(3)  
[  
    (u'145757', 2.96),  
    (u'244330', 2.1176470588235294),  
    (u'180162', 3.8486707566462166)  
]>>>
```

Complete PySpark Solution using `combineByKey()`

The `combineByKey()` is a general and extended version of `reduceByKey()` where the result type can be different than the values being aggregated. The `reduceByKey()` ’s limitation is that the reduced values data types must be the same data type as input (as defined in the Spark documentation). This means that, given the following:

```

# let rdd represents (key, value) pairs
# where value is of type T
rdd2 = rdd.reduceByKey(lambda x, y: func(x,y))

```

Then `func(x,y)` must create a value of type T.

Overall, the `combineByKey()` transformation is just such an optimization which aggregates values for a given key before sending it to the designated reducer. When using the `combineByKey()`, values are aggregated and merged into one value at each partition then each partition value is merged into a single value. Note that the type of the combined (result) value does not have to match the type of the original value (which solves the limitation of `reduceByKey()`). Using `reduceByKey()` or `combineByKey()`, in shuffle step, data is combined so each partition outputs at most one value for each key to send over the network.

For a given set of (K, V) pairs, `combineByKey()` has the following signature (this transformation has many different versions, here we show the simplest form):

```

combineByKey(create_combiner, merge_value, merge_combiners)
combineByKey : RDD[(K, V)] --> RDD[(K, C)]

```

`V` and `C` can be different data types.

This is a generic function to combine the elements for each key using a custom set of aggregation functions. It converts an `RDD[(K, V)]` into a result of type `RDD[(K, C)]`, for a “combined type” C. Note that C is a combined data structure. It can be a simple data type such as Integer and String or it can be a composite data structure such as pair (key, value) or triplet (x, y, z) or any desired data structure. The C being any data type, makes `combineByKey()` a very powerful reducer.

Let the source RDD be `RDD[(K, V)]`. Then we have to provide three basic functions:

- `create_combiner`:
which turns a V (a single value) into a C (e.g., creates a one-element

list of type C). This is done once per partition.

`create_combiner: (V) -> C`

- `merge_value`:

to merge a V into a C (e.g., adds it to the end of a list). This is applied to every element within a single partition.

`merge_value: (C, V) -> C`

- `merge_combiners`:

to combine two C's into a single one (e.g., merges the lists). This is applied for two partitions.

`merge_combiners: (C, C) -> C`

To avoid memory allocation, both `merge_value` and `merge_combiners` are allowed to modify and return their first argument instead of creating a new C (this can avoid creating new objects, which can be costly if you have a lot of data). Finally, note that V and C can be different data types (in `reduceByKey()`, V and C have to be the same data types).

In addition, users can control (by providing additional parameters) the partitioning of the output RDD, the serializer that is used for the shuffle, and whether to perform map-side aggregation (if a mapper can produce multiple items with the same key). The `combineByKey()` transformation is more general and you have the flexibility to specify whether you would like to perform map-side combine. However, it is a little bit more complex to use (at least you need to provide 3 small custom functions).

PySpark Solution using `combineByKey()`

To solve “average rating by user”, we use a pair of (`sum_of_ratings`, `number_of_raters`) as a “combined data structure.”

Step 1: Read Data and Create Pairs

Here we read data and create (key, value) pairs, where key is a userID and value is rating.

```
>>># spark : SparkSession
>>># create and return a pair of (userID, rating)
>>> def create_pair(rating_record):
...     tokens = rating_record.split(",")
...     return (tokens[0], float(tokens[2]))
...
>>> key_value_test = create_pair("3,2394,4.0,920586920")
>>> print key_value_test
('3', 4.0)

>>> ratings_path = "/tmp/movielens/ratings.csv.no.header"
>>> rdd = spark.sparkContext.textFile(ratings_path) ❶
>>> rdd.count()
22884377
>>> ratings = rdd.map(lambda rec : create_pair(rec)) ❷
>>> ratings.count()
22884377
>>> ratings.take(3)
[
    (u'1', 2.5),
    (u'1', 3.0),
    (u'1', 5.0)
]
```

❶ `rdd : RDD[String]`

❷ `ratings : RDD[(String, Float)]`

Step 2: Use `combineByKey()` to Sum up Ratings

Once `(userID, rating)` pairs are created, we now can apply the `combineByKey()` transformation to sum up all ratings and number of raters for a user. The output of this step will be :

`(userID, (sum_of_ratings, number_of_raters))`

```

>>># v is a rating from (userID, rating)
>>># C represents (sum_of_ratings, number_of_raters)
>>># C[0] denotes sum_of_ratings
>>># C[1] denotes number_of_raters
>>># ratings : source RDD ①
>>> sum_count = ratings.combineByKey( ②
    (lambda v: (v, 1)), ③
    (lambda C,v: (C[0]+v, C[1]+1)), ④
    (lambda C1,C2: (C1[0]+C2[0], C1[1]+C2[1])) ⑤
)
>>> sum_count.count()
247753
>>> sum_count.take(3)
[
  (u'145757', (148.0, 50)),
  (u'244330', (36.0, 17)),
  (u'180162', (1882.0, 489))
]

```

- ① $\text{RDD}[(\text{userID}, \text{rating})] = \text{RDD}[(\text{String}, \text{Float})]$
- ② $\text{RDD}[(\text{userID}, (\text{sum-of-ratings}, \text{count-of-ratings}))] = \text{RDD}[(\text{String}, (\text{Float}, \text{Integer}))]$
- ③ `create_combiner`: which turns a V (a single value) into a C as (V, 1)
- ④ `create_combiner`: to merge a V (rating) into a C as (sum, count)
- ⑤ `merge_combiners`: to combine two C's into a single C

Step 3: Find Average Rating

To find average rating per user, we divide “sum of ratings” by the “number of raters”.

```

>>># x = (sum_of_ratings, number_of_raters)
>>># x[0] = sum_of_ratings
>>># x[1] = number_of_raters
>>># avg = sum_of_ratings / number_of_raters
>>> average_rating = sum_count.mapValues(lambda x:(x[0] / x[1]))
>>> average_rating.take(3)
[

```

```
(u'145757', 2.96),  
(u'244330', 2.1176470588235294),  
(u'180162', 3.8486707566462166)  
]
```

Comparison of Reductions

This chapter presented some of the most important Spark's reduction transformations (listed below) by simple concrete examples. We discussed the following reducers:

- `reduceByKey()`
- `groupByKey()`
- `combineByKey()`
- `aggregateByKey()`

Let's compare four of the most important `<reducer-name>ByKey()` transformations. Note that, in the following table, V and C can be different data types.

T
a
b
l
e

4
-
3
. *C*
o
m
p
a
r
i
s
o
n

o
f
R
e
d
u
c
t
i
o
n
s

Reduction	Source RDD	Target RDD
reduceByKey()	RDD[(K, V)]	RDD[(K, V)]
groupByKey()	RDD[(K, V)]	RDD[(K, [V])]
aggregateByKey()	RDD[(K, V)]	RDD[(K, C)]
combineByKey()	RDD[(K, V)]	RDD[(K, C)]

We learned that some of the reduction transformations (such as `reduceByKey()` and `combineByKey()`) are preferable over `groupByKey()` and this is due to the shuffle step for `groupByKey()` is more expensive than the shuffle step for `reduceByKey()` and `combineByKey()`. When possible, use `reduceByKey()` over `groupByKey()`. Overall, for large volume of data, `reduceByKey()` and `combineByKey()` will scale out better than `groupByKey()`.

This chapter tried to answer few of the key questions such as :

- Which reduction transformation to use?
- Which transformation is more efficient?
- When to use `reduceByKey()` over `groupByKey()`?

To understand reduction transformation we need to understand the following:

- The underlying architecture of the reduction transformations
- The “shuffle phase” of the reduction transformations (the most important)

We learned that in Shuffle Step of `reduceByKey()`: the data is combined (and less data is sent over network) so that at each partition there is at most one value for each key and then shuffle happens and it is sent over the network to some particular executor for some action such as reduce. While in

`groupByKey()` Shuffle Step: it does not merge the values for the key but directly the shuffle step happens and lots of data gets sent to each partition, almost same as the initial data. In `groupByKey()` the merging of values for each key happens after the shuffle step and lots of data needs to be stored on final worker node (reducer) thereby resulting (may be) in out of memory problem.

While both of these Spark transformations (`reduceByKey()` and `groupByKey()`) will produce the correct answer, the `reduceByKey()` works much better on a large dataset. That's because Spark knows it can combine output with a common key on each partition before shuffling the data. You may use `combineByKey()` when you are combining elements but your return type differs from your input value type.

When possible, you should replace `groupByKey()` with `reduceByKey()` to improve scalability and performance (in some cases). The `reduceByKey()` transformation performs map side combine which can reduce network IO and shuffle size where as `groupByKey()` will not perform any map side combine at all.

We found out that the `aggregateByKey()` transformation is more suitable for compute aggregations for keys, example aggregations such as sum, average, variance, etc. The important rule here is that the extra computation spent for map side combine can reduce the size sent out to other worker nodes and driver. If your requirements satisfies this rule, you probably should use `aggregateByKey()` (at minimum, you need to implement three basic functions: `create_combiner`, `merge_value`, and `merge_combiners`—these functions were discussed in the early sections of this chapter).

Summary

- The `reduceByKey()` transformation is more efficient when we run this on large data set. This transformation's output type has to be the same as input value types.

- The `combineByKey()` transformation is a generic reduction and does not have restrictions of the `reduceByKey()`: output type can be different from input types.
- When possible, avoid `groupByKey()` on big data, which can cause “out of memory” and “out of disk space” Problems.
- When possible, use `reduceByKey()` or `combineByKey()` over `groupByKey()`
- Make sure that your reducer is a monoid, otherwise you might get wrong reduced values.

Chapter 5. Partitioning Data

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In this chapter we focus on partitioning data, which pays an important role in performance of Spark transformations and SQL queries run time on Amazon Athena and Google BigQuery. Data partitioning in Spark is for the purpose of parallelism and independence of tasks, but the purpose of data partitioning in Amazon Athena and Google BigQuery is to analyze slice of a data rather than the whole data.

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 5](#).

What is data partitioning

According to thefreedictionary.com: “Partitioning is the act of dividing or partitioning; separation by the creation of a boundary that divides or keeps apart”. Data partitioning is used in Spark, Amazon Athena, and Google BigQuery to improve performance of query executions. To scale-out big data solutions, data is divided into partitions that can be managed, accessed, and executed separately and in parallel.

In a nutshell, partitioning data can improve manageability, scalability, reduce contention, and optimize performance. Imagine some data (such as hourly temperature per city) for all countries (7 continents and 195 countries), and the goal is to query and analyze data for a given continent, countries or for several countries. If you do not partition your data accordingly, then you have to read out all data and find proper countries in the query and then apply your `map()` and `reduce()` transformations to find your final result. This is not very smart (due to query performance penalty) at all since for every query you are loading and reading all of the given data. The smart way is just to load a slice of data, which will make your query to finish in the least amount of time.

Partitions in Spark

Let’s say your data is in HDFS (Hadoop Distributed File System, which is distributed among many cluster nodes) or in Amazon S3 (Simple Storage Service, which is an object storage service (might be distributed in many nodes as well). How does your distributed data and Spark partitions work? In reality your physical data is distributed across storage as partitions residing in either HDFS, S3, or cloud storage (see Figure 5.1). As your physical data is distributed as partitions across the physical cluster, Spark treats each partition as a high-level logical data abstraction (such as RDDs and DataFrames) in memory (and disk if there is not a sufficient memory). Spark cluster will optimize partition access and will read the partition closest to it in the network, observing data locality.

Spark Logical Model, Partitions, Distributed Data

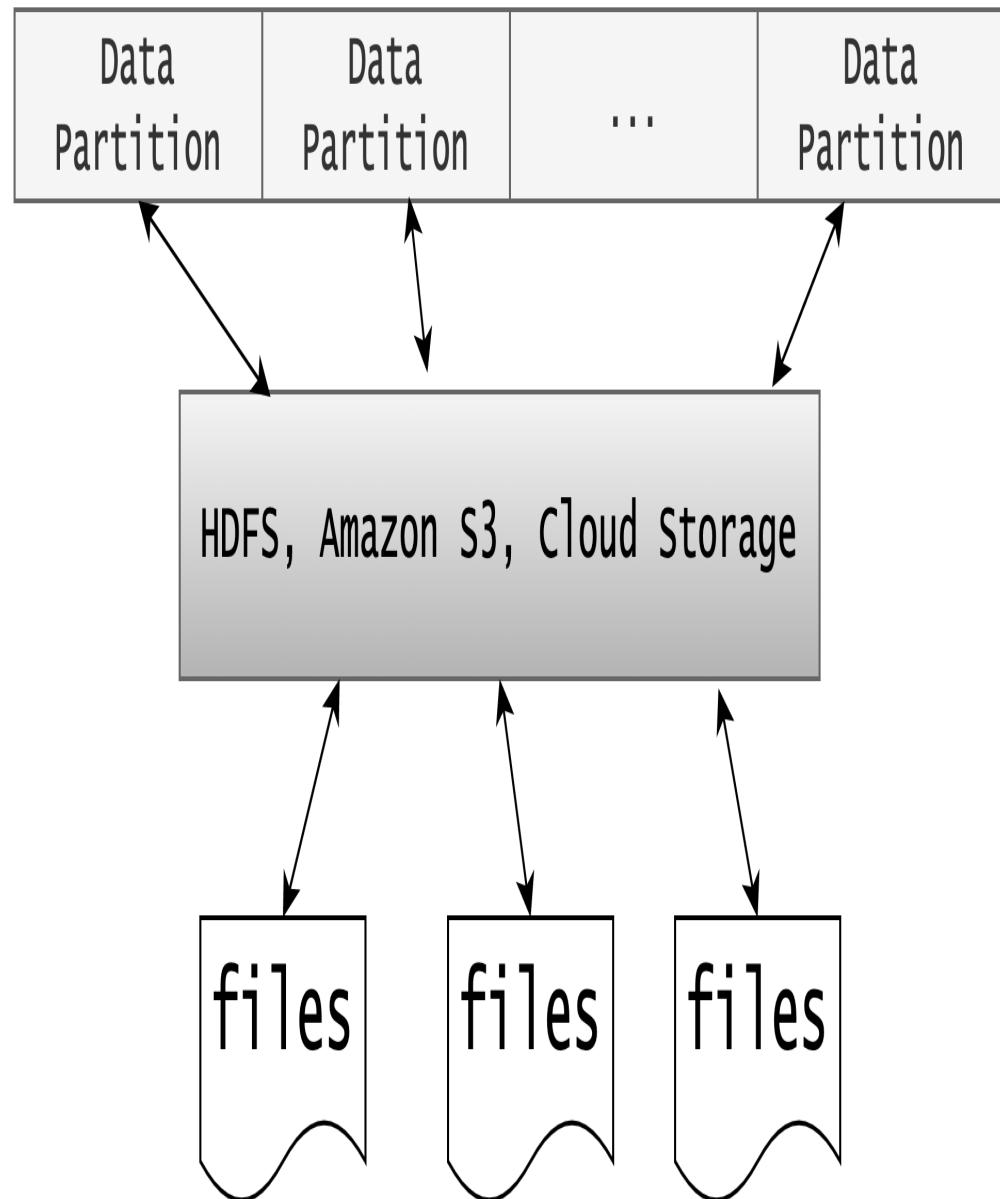


Figure 5-1. Spark Logical Partitioning

The question is what is the main purpose of partitioning data? The main purpose of partitioning data is for maximal efficient parallelism (execute many tasks at the same time using cluster nodes via Spark executors). Spark Executors are launched at the start of a Spark application in coordination with the Spark Cluster Manager. Spark Executors are worker nodes' processes in charge of running individual tasks in a given Spark job/application. Breaking up data (distributed scheme) into partitions allows Spark executors to process only data that is close to them, therefore minimizing network bandwidth. That is, each spark executor's core is assigned its own data partition to work on (see Figure 5.2).

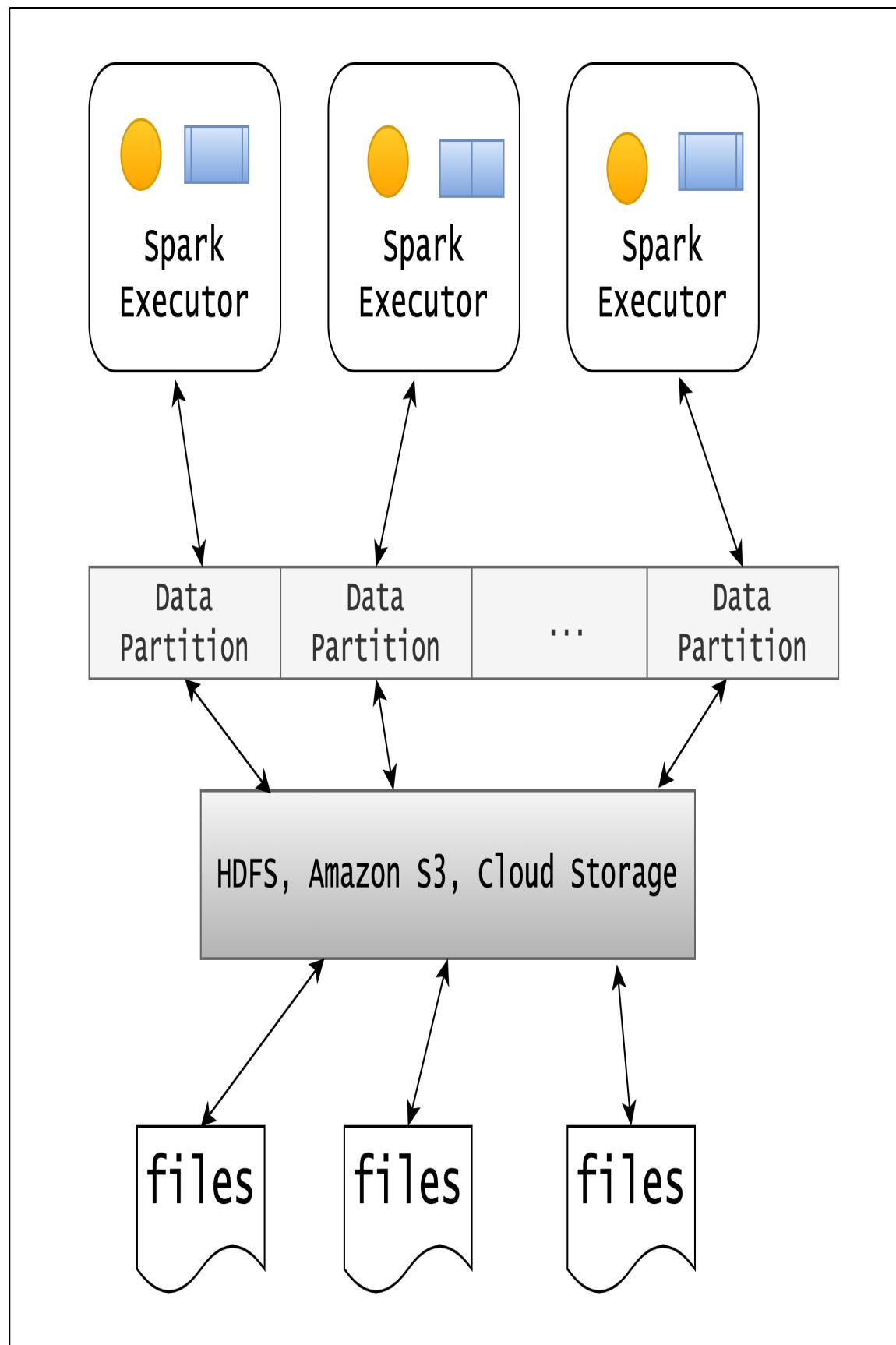


Figure 5-2. Spark Partitioning in Action

Spark's resilient distributed datasets (RDD) are a collection of various data that are so big in size, they cannot fit into a single node and should be partitioned across various nodes. Partitioning is an important concept in Spark as it determines how the entire hardware resources are accessed when executing any job. The optimal partitioning should utilize hardware resources as much as possible by maximizing parallelism for data transformations. The following factors affect data partitioning choices like:

- Available resources — Number of cores on which task can run on.
- External data sources — Size of local collections, input file systems (such as HDFS, S3, ...)
- Transformations used to derive RDDs and DataFrames — There are a number of rules to determine the number of partitions when an RDD/DataFrame is derived from another RDD/DataFrame.

Let's see how partitioning work in Spark computing environment. When Spark reads a data file, the entire content of the file is partitioned into multiple smaller chunks (you RDD is partitioned into a set of chunks, called partitions). Note that any RDD for that matter is partitioned by Spark into multiple partitions. Then when we apply a transformation (such as `map()`, `reduceByKey()`, ...) on a RDD, the transformation is applied to each of its partition. Spark spawns a single Task for a single partition, which will run inside the executor JVM. Each stage contains as many tasks as partitions of the RDD and will perform the transformations (such as `map()`, `reduceByKey()`, ...) pipelined in the stage. The partitioning process and its execution is illustrated by Figure 5.1.

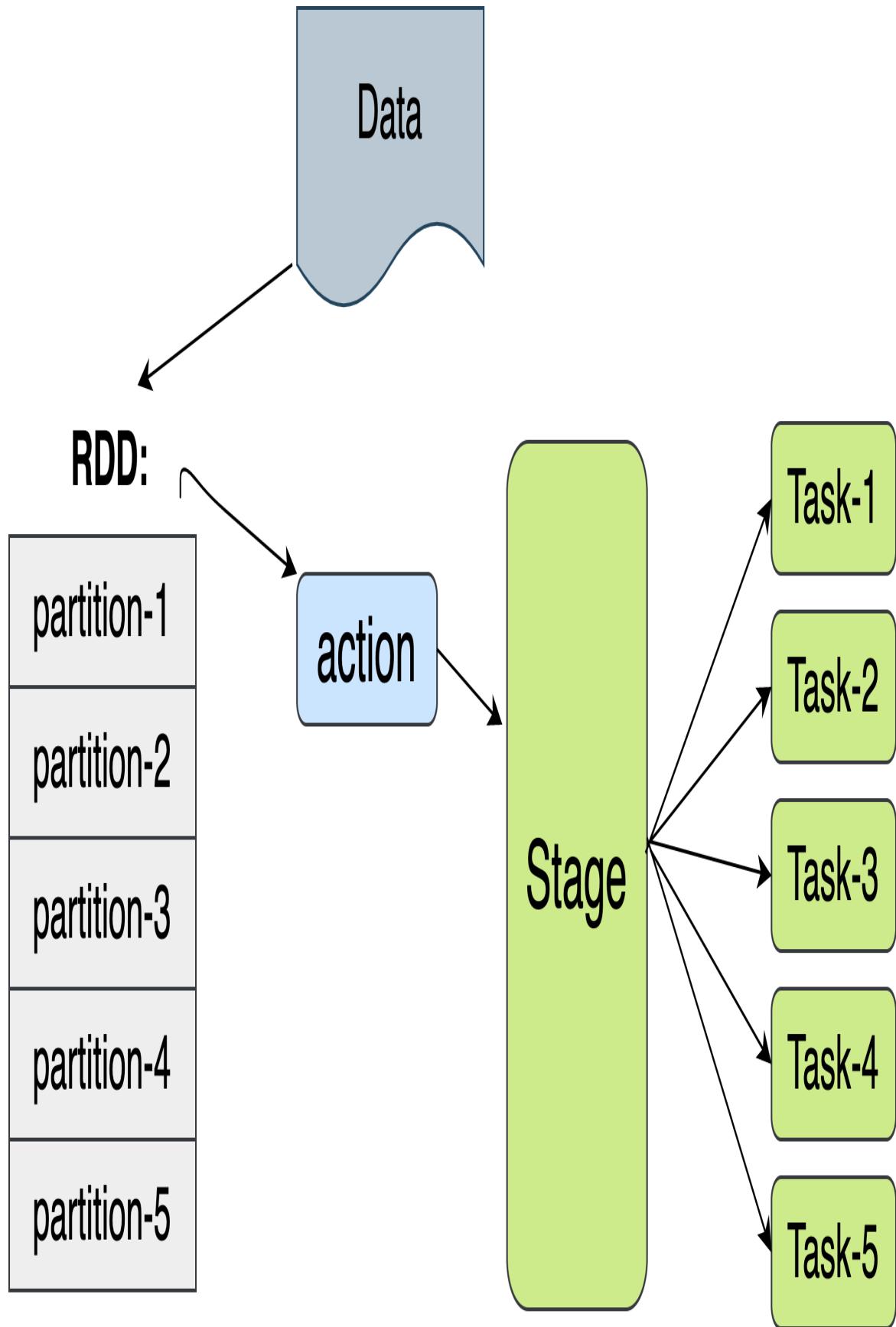


Figure 5-3. Partitioning Data in Spark

Example of Partitioning

For example, if an RDD has 10 billion elements, then Spark may partition it into 10,000 chunks (also called partitions — number of partitions is 10,000). In this case, each partition will have about one million elements ($10,000 \times 1 \text{ million} = 10 \text{ billion}$). Then these partitions can be sent to cluster nodes for further transformations such as mappers and reducers. Partitioning data enable Spark to run many transformations independently and at the same time (by maximizing parallelism).

Partitions in Spark do not span multiple machines. This means that each partition is sent to a single worker machine. Tuples in the same partition are guaranteed to be on the same machine. Spark assigns one task per partition and each worker can process one task at a time.

Proper partitioning can improve the performance of your data analysis and improper partitioning will hurt the performance of your data analysis. For example, imagine a Spark cluster with 501 nodes (one master and 500 worker nodes) and consider an RDD with 10 billion elements. The proper number partitions would be over 500 (such as 1000) such that all cluster nodes are utilized at the same time. What if your number partitions is 100 and each worker can accept at most 2 tasks, then most of your worker nodes (about 400 of them) will be idle and useless. The more you utilize the worker nodes, the faster your query will run.

Spark has a default and custom partitioner. When you create an RDD, you may let the Spark to set the number of partitions (default), or you may set it explicitly (custom). The number of partitions in the default case depends on the cluster size and available resources. Most of the times, the default case will work just fine, but if you are an experienced Spark programmer, then you may set the number of partitions explicitly.

Below, I'll provide examples to show default and custom partitioning.

Default Partitioning

The default partitioning of an RDD and DataFrame happens when a programmer does not set the number of partition explicitly. In this case, the number of partitions depends on data and resources available in the cluster.

DEFAULT NUMBER OF PARTITIONS

For production environments, most of the time, the default partitioner will work well. Make sure that no cluster nodes/executors are idle.

When you create an RDD or a DataFrame, there is an option for setting the number of partitions. For example, when creating an RDD, you may set the number of partitions with using the following API (`numSlices` represents the number of custom partitions) to distribute a local Python collection to form an RDD

```
SparkContext.parallelize(collection, numSlices=None)
```

You may use `textfile()` to read a text file from a file system (such as HDFS, S3, ...), and return it as an RDD of Strings. If you do not set the `minPartitions` parameter, then Spark will set it to the default number of partitions (based on data size and available resources in the cluster)

```
SparkContext.textFile(name, minPartitions=None, use_unicode=True)
```

Example of Default Partitioning

Here I will create an RDD from a collection without setting the number of partitions.

To understand partitioning, I am introducing a simple debugger function to display elements of each partition:

```
>>> def debug(iterator):
...     elements = []
...     for x in iterator:
```

```
...     elements.append(x)
...     print("elements=", elements)
```

Next, I create an RDD and then display content of each partition:

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> rdd = sc.parallelize(numbers)
>>> num_partitions = rdd.getNumPartitions()
>>> num_partitions
8
>>> rdd.foreachPartition(debug)
elements= [1]
elements= [11, 12]
elements= [4]
elements= [2, 3]
elements= [10]
elements= [8, 9]
elements= [7]
elements= [5, 6]
```

Note that in real data analysis, each partition may have millions of elements, but to understand partitions and partitioning, I used a toy data.

Custom Partitioning

The custom partitioning happens when a programmer explicitly sets the number of partitions.

SETTING THE NUMBER OF PARTITIONS

For production environments, you need to understand your data and your cluster before you set the number of partitions explicitly. Make sure that no cluster nodes/executors are idle.

Example of Custom Partitioning

Here I will create an RDD from a collection and set the number of partitions explicitly.

```
>>> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> rdd = sc.parallelize(numbers, 3) ❶
>>> rdd.getNumPartitions()
3
>>> 3
3
```

- ❶ we set the number of partitions to 3

Next, let's debug the created RDD and view the content of all partitions:

```
>>> rdd.foreachPartition(debug)
elements= [5, 6, 7, 8]
elements= [1, 2, 3, 4]
elements= [9, 10, 11, 12]
```

Further, let's apply `mapPartitions()` transformation on this RDD:

```
>>> def adder(iterator):
...     yield sum(iterator)
...
>>> rdd.mapPartitions(adder).collect()
[10, 26, 42]
```

Partitioning for SQL Queries

Amazon Athena and Google BigQuery are serverless services to query data by SQL. Given a query, proper data partitioning enable us to read, scan, and query slice(s) of a data rather than reading and analyzing the whole data. Querying slice of data (by proper data partitioning at the fields level) is faster than querying the whole data.

Partitioning data by specific fields (which are used in SQL's WHERE clause) plays a crucial role when querying data in Amazon Athena or Google's BigQuery, since it limits the volume of data scanned, dramatically accelerating queries and reducing costs (since cost is based on the amount of data scanned).

Consider some data which involves continents, countries, and cities. By looking at the data, you can see that a continent has a list of countries, and each country has a set of cities. If you are going to query this data by continent, country, and city, then it makes a lot of sense to partition your data by three data fields: (`continent`, `country`, `city`). The simple partitioning solution will be to create one folder per continent and then partition each continent by country folders, and finally partition each country by cities. For example, the following query will only scan `<root-dir>/continent=north_america/country=usa/city=Cupertino` rather than the entire directory structure under `<root-dir>/`.

```
SELECT <some-fields-from-my_table>
  FROM my_table
 WHERE continent = 'north_america' AND
       country = 'usa' AND
       city = 'Cupertino'
```

partitioning will enable us to scan a very limited set of data rather than the whole data. For example, if you have query which involves “United States”, then you will scan only one folder rather than scanning all 195 folders. In big data analysis, partitioning data by directories is very effective since we do not have an indexing mechanism like relational tables. Therefore partitioning can be considered a very simple indexing mechanism for big data analysis.

According to [Amazon Athena Documentation](#): “By partitioning your data, you can restrict the amount of data scanned by each query, thus improving performance and reducing cost. Athena leverages Hive for partitioning data. You can partition your data by any key. A common practice is to partition the data based on time, often leading to a multi-level partitioning scheme. For example, a customer who has data coming in every hour might decide to partition by year, month, date, and hour. Another customer, who has data coming from many different sources but loaded one time per day, may partition by a data source identifier and date.”

For example, in Amazon Athena, you may create a partitioned table as:

```
CREATE EXTERNAL TABLE world_tempreature(
    day_month_year DATE,
    temperature DOUBLE
)
PARTITIONED BY (
    continent string,
    country string,
    city string
)
STORED AS PARQUET
LOCATION 's3://<bucket-name>/dev/world_tempreature/'
tblproperties ("parquet.compress"="SNAPPY");
```

Therefore, if you query a partitioned table and specify the partition in the WHERE clause, then Amazon Athena scans the data only from that partition.

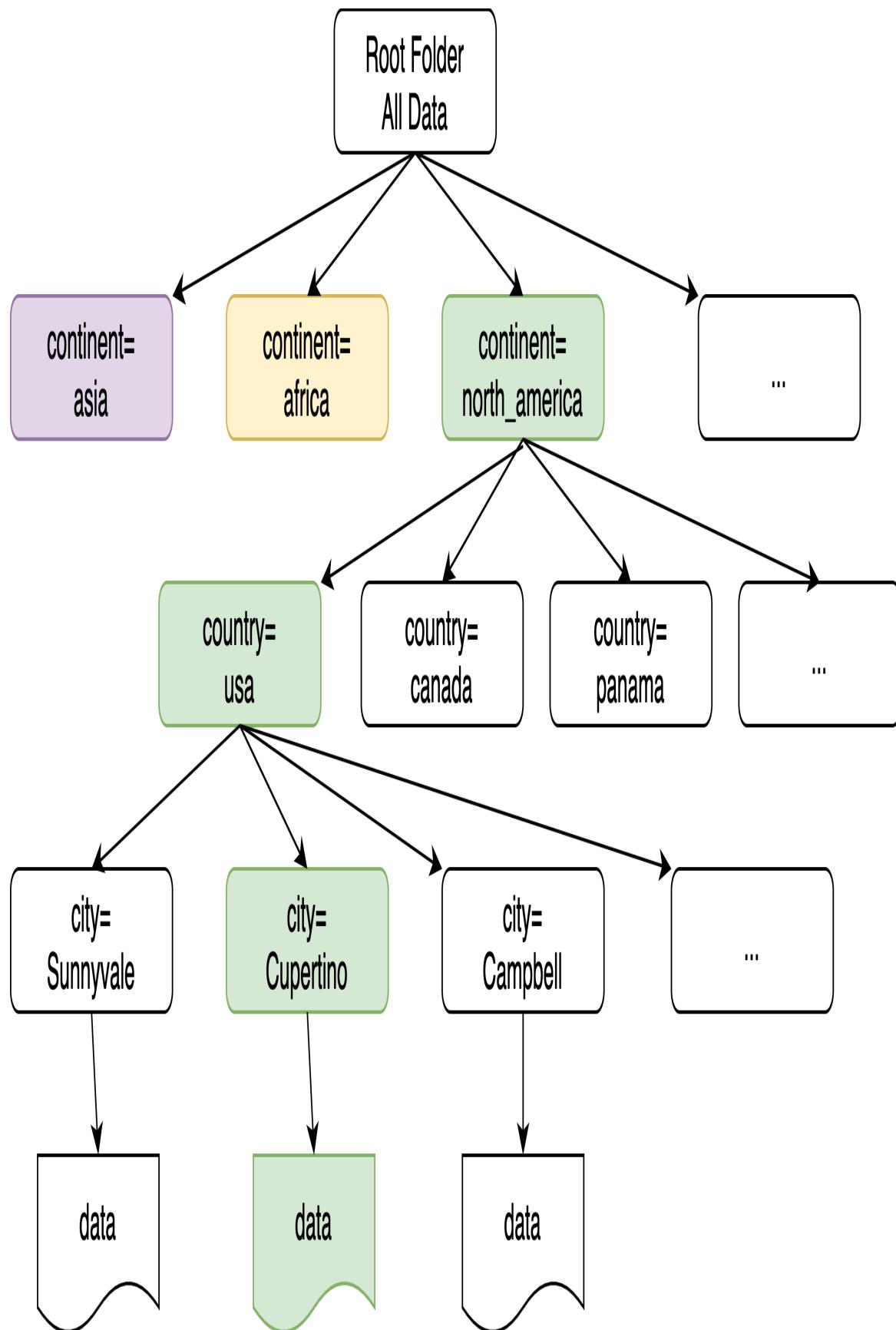


Figure 5-4. Partitioned Data

Note that if you are going to query data by `year`, `month`, and `day`, then you may partition the same data into another form, where partition fields will be `year`, `month`, and `day`. Therefore your schema will change into the following table:

```
CREATE EXTERNAL TABLE world_temperature_by_date(
    day_month_year DATE,
    continent string,
    country string,
    city string,
    temperature DOUBLE
)
PARTITIONED BY (
    year INTEGER,
    month INTEGER,
    day, INTEGER
)
STORED AS PARQUET
LOCATION 's3://<bucket-name>/dev/world_temperature_by_date/'
tblproperties ("parquet.compress"="SNAPPY");
```

Now with this new schema, you may issue SQL queries as:

```
SELECT <some-fields>
    FROM world_temperature_by_date
    WHERE year = 2020 AND
        month = 8 AND
        day = 16
```

Therefore, to partition your data effectively, you need to understand the queries that you will execute against your table (data expressed as a table).

2. Why partition data

What do you achieve by partitioning data: by partitioning your data, you can restrict the amount of data scanned by each query, thus improving performance and reducing cost. For example, Amazon Athena leverages Spark and Hive for partitioning data. You can partition your data by any key.

Therefore, if you are using Amazon Athena or Google BigQuery, you may just select and use specific folders for your query rather than using the entire data set for all countries.

If your data is represented in a table (as a Spark's DataFrame), then partitioning is a way of dividing a table (a DataFrame abstraction) into related parts based on the values of particular columns like chromosome (for genomics data), country, city, date, and department. Each table in the [Hive](#), [Amazon Athena](#), or [Presto](#) can have one or more partition keys to identify a particular partition. Using partition it is easy to execute queries on slices of the data rather than loading the entire data for analysis.

For example, genomics data records include a chromosome (total of 25 chromosomes), which are labeled as {chr1, chr2, ..., chr22, chrX, chrY, chrMT}. Since in most of the genomics analysis, you do not mix chromosomes, it makes a lot of sense to partition genomics data by chromosome ID, which can reduce the analysis time by just loading the desired chromosome.

Simple Problem

I will demonstrate the Partitioning concept by an example. Consider the following customer data, where each record has the following format:

```
<customer_id><,><date><,><transaction_id><,><item><,><transaction_value>
<date>=DAY/MONTH/YEAR
```

Further, assume that your goal is to analyze data by a given year (as YEAR) or by by a combination of year (as YEAR) and month (as MONTH). It looks that partitioning data is a good idea since you can limit scanning data for analysis by selecting specific folders (by year or by year and month). Let's say that the goal is to create partitioned data by (<year>, <month>) for all given records. Therefore after partitioning data, the partitioned data will look like:



Figure 5-5. Partitioned Data

Partitioning Data in Spark

How do we accomplish partitioning data in Spark. Spark offers a simple DataFrame API for partitioning data. Let `df` denote a DataFrame for our example data, as illustrated below

`df:`

```
<customer_id><,><date><,><transaction_id><,><item><,>
<transaction_value> ... sample values....
```

then we can partition our data by `DataFrameWriter.partitionBy()` method. Data can be partitioned in a text format (as row-based) or can be partitioned in a binary format such as Parquet (as column-based). The following subsections show how to perform partitioning data.

Partition as Text Format

The following code snippet shows how to partition data (represented as a DataFrame) into text format.

```
# df: a DataFrame with four columns:
#   <customer_id>
#   <date> (as DAY/MONTH/YEAR)
#   <transaction_id>
#   <item>
#   <transaction_value>
#
```

```

# partition data by YEAR and MONTH
# Create two additional columns as "year" and month
df
    .withColumn("year", ...) // add a year column ①
    .withColumn("month", ...) // add a month column ②
    .write ③
    .partitionBy("year", "month") ④
    .text(output_path) ⑤

```

- ① add a year column
- ② add a month column
- ③ Get a DataFrameWriter object
- ④ Partition your data by your desired columns
- ⑤ Save each partition as a text format

A complete solution for partitioning data is given by 3 files:

- `partition_data_as_text_by_year_month.py`
 - PySpark solution for partitioning data
- `partition_data_as_text_by_year_month.sh`
 - sample shell script to invoke PySpark program
- `partition_data_as_text_by_year_month.log`
 - sample run and detail outputs

The main steps in this program are:

- STEP-1: Read data and create a DataFrame (as `df`) with four columns: {`<customer_id>`, `<year>`, `<transaction_id>`, `<transaction_value>`}.

```

# create a DataFrame, note that toDF() returns a
# new DataFrame with new specified column names
# columns = ('customer_id', 'year', 'transaction_id', 'transaction_value')
df = spark.read.option("inferSchema", "true")\
    .csv(input_path)\n    .toDF('customer_id', 'year', 'transaction_id', 'transaction_value')

```

- STEP-2: Then use `df.write.partitionBy()` to create partitioned data.

```

# partition data
df.write.partitionBy('customer_id', 'year')\
    .parquet(output_path)

```

If you want to just create a single partitioned file per partition, then we can repartition data, before partitioning. Spaks `repartition(numPartitions, *cols)` returns a new DataFrame partitioned by the given partitioning expressions. The resulting DataFrame is hash partitioned.

```

# partition data
df.repartition('customer_id', 'year')\
    .write.partitionBy('customer_id', 'year')\
    .parquet(output_path)

```

Threfore `.repartition('customer_id', 'year')` creates a single output file per ('customer_id', 'year').

Sample Input

I will use the `customers.txt` as an input.

```

$ cat customers.txt
c1,01/10/2019,T0011,book,40.67
c1,01/10/2019,T0012,toy,32.34
c2,05/11/2019,T0012,toy,12.34
c2,07/11/2019,T0012,clothes,310.34
...

```

After running the PySpark program, partitioned data is created under the output path:

```
$ ls -1R /tmp/partition_demo/
_SUCCESS
customer_id=c1
customer_id=c2

/tmp/partition_demo//customer_id=c1:
year=2018
year=2019

/tmp/partition_demo//customer_id=c1/year=2018:
part-00000-8905097e-a6d3-4cb7-8b40-879073ec51bc.c000.snappy.parquet

/tmp/partition_demo//customer_id=c1/year=2019:
part-00000-8905097e-a6d3-4cb7-8b40-879073ec51bc.c000.snappy.parquet
...
```

Partition as Parquet Format

Partitioning data as a **Parquet** Format has advantages: data aggregation can be done faster than text data since Parquet stores data in columnar format. Parquet stores metadata as well. If desired, you may partition your data by other columnar formats (**ORC** and **CarbonData**).

The following code snippet shows how to partition data (represented as a DataFrame) into Parquet format.

```
# df: a DataFrame with four columns:
#   <customer_id>
#   <year>
#   <transaction_id>
#   <transaction_value>
#
# partition data by "customer_id" and "year"
df.write ①
    .partitionBy("customer_id", "year") ②
    .parquet(output_path) ③
```

- ➊ Get a DataFrameWriter object

- ② Partition your data by your desired columns
- ③ Save each partition as a Parquet format

A complete solution for partitioning data is given by 3 files:

- `partition_data_by_customer_and_year.py`
 - PySpark solution for partitioning data
- `partition_data_by_customer_and_year.sh`
 - sample shell script to invoke PySpark program
- `partition_data_by_customer_and_year.log`
 - sample run and detail outputs

The main steps in this program are:

- STEP-1: Read data and create a DataFrame (as `df`) with four columns: {`<customer_id>`, `<year>`, `<transaction_id>`, `<transaction_value>`}.

```
# create a DataFrame, note that toDF() returns a
# new DataFrame with new specified column names
# columns = ('customer_id', 'year', 'transaction_id', 'transaction_value')
df = spark.read.option("inferSchema", "true")\
    .csv(input_path)\n    .toDF('customer_id', 'year', 'transaction_id', 'transaction_value')
```

- STEP-2: Then use `df.write.partitionBy()` to create partitioned data.

```
# partition data
df.write.partitionBy('customer_id', 'year')\
    .parquet(output_path)
```

If you want to just create a single partitioned file per partition, then we can repartition data, before partitioning:

```
# partition data
df.repartition('customer_id', 'year')\
    .write.partitionBy('customer_id', 'year')\
        .parquet(output_path)
```

Sample Input

I will use the customers.txt as an input.

```
$ cat customers.txt
c1,2019,T0011,20.67
c1,2019,T0012,12.34
...
```

After running the PySpark program, partitioned data is created under the output path:

```
$ ls -1R /tmp/partition_demo/
_SUCCESS
customer_id=c1
customer_id=c2

/tmp/partition_demo//customer_id=c1:
year=2018
year=2019

/tmp/partition_demo//customer_id=c1/year=2018:
part-00000-8905097e-a6d3-4cb7-8b40-879073ec51bc.c000.snappy.parquet

/tmp/partition_demo//customer_id=c1/year=2019:
part-00000-8905097e-a6d3-4cb7-8b40-879073ec51bc.c000.snappy.parquet
...
```

How to partition data

You may partition your data by many tools. But, Spark provides a powerful and easy way to partition your big data. To partition your data, first create a

DataFrame, then use `pyspark.sql.DataFrameWriter.partitionBy` (Python method, in `pyspark.sql` module). The `partitionBy(*cols)` partitions the output by the given columns on the file system.

For example, if you have a DataFrame with 4 columns: (`c1`, `c2`, `c3`, `c4`) and you want to partition your data by column `c2` then write:

```
>>> data = [(1, 2, 3, 4), (5, 2, 7, 8), (9, 2, 11, 12),
           (13, 3, 15, 16), (17, 3, 19, 20)]
>>> column_names = ['c1', 'c2', 'c3', 'c4']
>>> df = spark.createDataFrame(data, column_names)
>>> df.show()
+---+---+---+---+
| c1| c2| c3| c4|
+---+---+---+---+
|  1|  2|  3|  4|
|  5|  2|  7|  8|
|  9|  2| 11| 12|
| 13|  3| 15| 16|
| 17|  3| 19| 20|
+---+---+---+---+
>>> output_path = '/tmp/partition/output1'
>>> df.write.partitionBy('c2').csv(output_path)
```

Now, we may examine the partitioned data:

```
$ ls -l /tmp/partition/output1/
total 0
-rw-r--r-- 1 mparsian wheel 0 Apr 3 12:56 _SUCCESS
drwxr-xr-x 8 mparsian wheel 256 Apr 3 12:56 c2=2
drwxr-xr-x 6 mparsian wheel 192 Apr 3 12:56 c2=3

$ ls -l /tmp/partition/output1/c2=2
-rw-r--r-- 1 mparsian wheel 6 Apr 3 12:56 part-00001-6298ea4f-248b-4b3a-934b-
334e9ae21f4e.c000.csv
-rw-r--r-- 1 mparsian wheel 6 Apr 3 12:56 part-00003-6298ea4f-248b-4b3a-934b-
334e9ae21f4e.c000.csv
-rw-r--r-- 1 mparsian wheel 8 Apr 3 12:56 part-00004-6298ea4f-248b-4b3a-934b-
334e9ae21f4e.c000.csv

$ ls -l /tmp/partition/output1/c2=3
-rw-r--r-- 1 mparsian wheel 9 Apr 3 12:56 part-00006-6298ea4f-248b-4b3a-934b-
334e9ae21f4e.c000.csv
```

```
-rw-r--r-- 1 mparsian wheel 9 Apr 3 12:56 part-00007-6298ea4f-248b-4b3a-934b-  
334e9ae21f4e.c000.csv

$ cat /tmp/partition/output1/c2=2/part-00001-6298ea4f-248b-4b3a-934b-  
334e9ae21f4e.c000.csv
1,3,4

$ cat /tmp/partition/output1/c2=2/part-00003-6298ea4f-248b-4b3a-934b-  
334e9ae21f4e.c000.csv
5,7,8

$ cat /tmp/partition/output1/c2=2/part-00004-6298ea4f-248b-4b3a-934b-  
334e9ae21f4e.c000.csv
9,11,12
```

If you want to create a single file (rather than many files — one file per partition) per partition, then you have to `repartition` your DataFrame, before partitioning it:

```
>>> output_path2 = '/tmp/partition/output2'
>>> df.repartition('c2')
    .write.partitionBy('c2')
    .csv(output_path2)
```

then you will see a sinle file per partition:

```
$ ls -l /tmp/partition/output2/
total 0
-rw-r--r-- 1 mparsian wheel 0 Apr 3 13:03 _SUCCESS
drwxr-xr-x 4 mparsian wheel 128 Apr 3 13:03 c2=2
drwxr-xr-x 4 mparsian wheel 128 Apr 3 13:03 c2=3

$ ls -l /tmp/partition/output2/c2=2/
-rw-r--r-- 1 mparsian wheel 20 Apr 3 13:03 part-00128-15c55246-8956-41e4-9996-  
32aea684d1e5.c000.csv

$ ls -l /tmp/partition/output2/c2=3/
-rw-r--r-- 1 mparsian wheel 18 Apr 3 13:03 part-00107-15c55246-8956-41e4-9996-  
32aea684d1e5.c000.csv

$ cat /tmp/partition/output2/c2=2/part-00128-15c55246-8956-41e4-9996-  
32aea684d1e5.c000.csv
1,3,4
5,7,8
```

9,11,12

```
$ cat /tmp/partition/output2/c2=3/part-00107-15c55246-8956-41e4-9996-  
32aea684d1e5.c000.csv  
13,15,16  
17,19,20
```

You may partition your data by more than one column. For example, if you have a DataFrame with 4 columns: (c1, c2, c3, c4) and you want to first partition by column c2 followed by column c4, then

How to query partitioned data

To have an optimized query performance, you should include the partitioned column(s) in your SQL queries. For example

Amazon Athena Example

“Amazon Athena is an interactive query service that makes it easy to analyze data in Amazon S3 using standard SQL. Athena is serverless, so there is no infrastructure to manage, and you pay only for the queries that you run.”
(source: <https://aws.amazon.com/athena/>).

To access and query your data in Athena by SQL query, you need to implement the following simple steps:

1. Understand your queries: then prepare your data partitioning accordingly. For example, if your SQL queries looks like:

```
select *  
from genome_table  
where chromosome = 'chr7' and ....
```

1. Then you may partition your data by chromosome column.

First prepare your data as a DataFrame (which includes a chromosome column). Then partition your data by chromosome and save it in S3 (called a

“location”).

```
# create a DataFrame
df = <dataframe-includes-chromosome-column>

# define your output location
s3_output_path = 's3://genomics_bucket01:/samples/'

# partition data by chromosome column
# and save it as a Parquet format
df.repartition("chromosome")\
    .write.mode("append")\
    .partitionBy("chromosome")\
    .parquet(s3_output_path)
```

1. Next you define your schema and point your schema to the same S3 location you created in the previous step

```
CREATE EXTERNAL TABLE `genome_table`(
    `sample_barcode` string,
    `allelcount` int,
    ...
)
PARTITIONED BY (
    `chromosome`
)
STORED AS PARQUET
LOCATION 's3://genomics_bucket01:/samples/'
tblproperties ("parquet.compress"="SNAPPY");
```

Note that the chromosome column is a data field defined in the “PARTITIONED BY” section.

1. Now that your schema is ready, you may execute/run your schema (this will create a metadata used by Amazon Athena).
2. Load your partitions:

```
MSCK REPAIR TABLE genome_table;
```

1. Once your partitions are ready, then you can start executing your SQL queries:

```
select sum(allelecount)
  from genome_table
 where chromosome = 'chr7';
```

Since we have partitioned our data by the `chromosome` column, only one directory `chromosome=chr7` will be read/scanned for the given SQL query.

Summary

- Data partitioning is the process of partitioning data by data fields into smaller pieces (chunks) in order to manage and access the data at a finer level.
- Data partitioning enable us to reduce the cost of storing a large amount of data as well as enhances the performance and manageability.
- When using serverless services such as Amazon Athena and Google BigQuery, you need to partition your data by the fields, mainly used in the `WHERE` clause of SQL queries. This means that we need to understand our queries before partitioning data accordingly.
- In a nutshell, data partitioning gives us the following advantages:
 - Improves query performance and manageability: for a given query, you just analyze slice(s) of data based on the query clause
 - Simplifies common ETL tasks: you can browse and view data based on the partitions.
 - Ad-hoc queries are easier and faster: since you do not need to analyze the whole data
 - Partitioning enable us to simulate partial indexing of relational tables.

Chapter 6. Feature Engineering in PySpark

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 6th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

This chapter covers algorithms — known as feature engineering — for working with features of data — such as car price, gene value, sugar level, hemoglobin, and education level — for building machine learning models. These set of algorithms (extracting, transforming and selecting features) are essential in building realistic machine learning models. Feature engineering is one of the most important topics in machine learning, because, the success and failure of a machine learning model to predict the future depends mainly on how you engineer features to get a better lift. Jason Brownlee states that the primary goal of feature engineering is the process of transforming raw data (without normalization, without dropping null values, and without altering the original data) into features that better represent the underlying business problem to the predictive models, resulting in improved model accuracy on unseen data.

Spark provides a comprehensive machine learning API for many well-known algorithms such as linear regression, logistic regression, decision trees and much more. The goal of this chapter is to provide fundamental tools and techniques in PySpark to be able to build any Machine Learning algorithms and pipelines. The techniques discussed and presented here can be applied to numerous Machine Learning algorithms such as Linear Regression and Logistic Regression. At the end of this chapter, I will show you how to use these provided techniques to build a Logistic Regression model from a given training data and then use the built model and query data for predicting labels.

This chapter introduces Spark’s powerful machine learning tools and utilities and provide the examples by using the PySpark API. The skills provided in this chapter may be used by an aspiring data scientist and data engineer. My goal is not to cover famous machine learning algorithms — such as linear regression, PCA, and logistic regression, since these are already covered in many books — but provide data utilities (normalize, standardize, string indexer, ...) that you may use them in cleaning data and building models for most of the machine learning algorithms.

No matter which machine learning algorithm you build and use, feature engineering is important. Machine learning enable us to find patterns in data—patterns by building models, then we use the built models to make predictions about new data points (called query data). To get those predictions right, we must construct the data set and transform the data correctly. This chapter covers these two key steps.

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 6](#).

This chapter will cover the following topics:

- Adding a new derived feature
- Creating and Applying UDF
- Tokenization
- String Indexer
- Normalization
- Standardization
- TF-IDF
- Bucketing
- Binarizer

Introduction to Feature Engineering

Let's say that your data is represented in a matrix of rows and columns — in machine learning, columns are called features (such as age, gender, education, testosterone, hemoglobin, hematocrit, ...) and each row represents an instance of features. The features in your data will directly influence the predictive models you build and use and the results you can achieve. It is believed that data scientists spend over 75% of their time on data preparation.

[Jason Brownlee](#) states that “feature engineering is the process of transforming raw data into features that better represent the underlying problem to the predictive models, resulting in improved model accuracy on unseen data”. In this chapter, my goal is to present generic feature engineering techniques by PySpark so that you can apply these data patterns in building better predictive models.

Where does feature engineering fit in building machine learning models? When do you apply feature engineering to your data? Let's take a look at the key steps for building and using a machine learning model:

1. Gather requirements for machine learning data
2. Select Data: Integrate data, de-normalize it into a dataset, collect it together.
3. Preprocess Data: Format it, clean it, sample it so you can work with it.
- 4. Transform Data: Feature Engineering happens here.**
5. Model Data: Create models, evaluate them and tune them.
6. Use query data and built model to predict

Feature engineering happens right before you build a model from your data. After selecting and cleaning data (for example, making sure that null values are filled with proper values), you **transform data** for building a model(s). The **transformation phase is feature engineering**: you convert string into numeric data, you bucketize your data, you normalize or standardize data, etc.

What this chapter covers is illustrated in Figure 6.1.

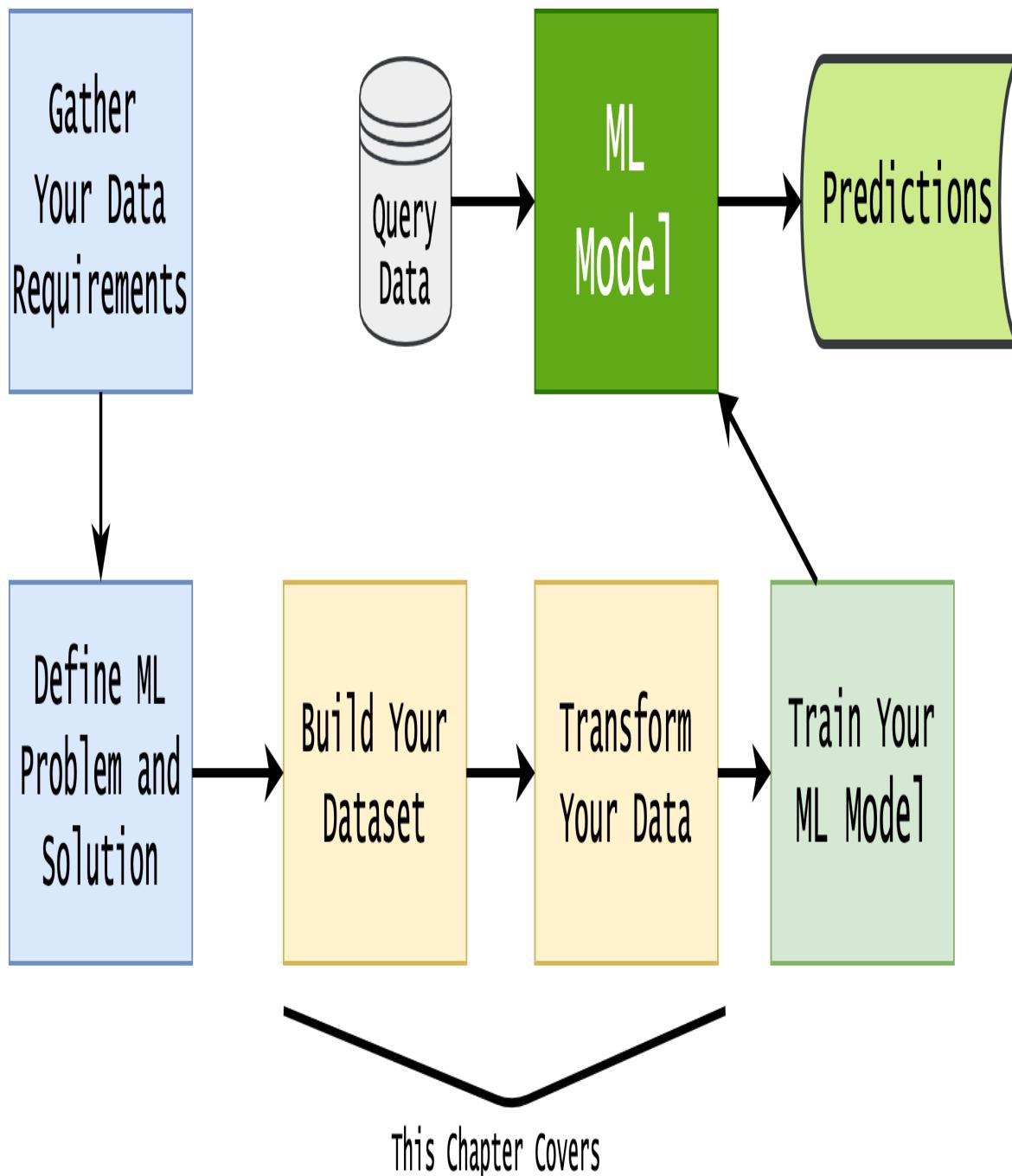


Figure 6-1. What this chapter covers

Spark API covers algorithms for working with features, which are roughly divided into these groups:

- Extraction: Extracting features from “raw” data
- Transformation: Scaling, converting, or modifying features
- Selection: Selecting a subset from a larger set of features

- Locality Sensitive Hashing (LSH): This class of algorithms combines aspects of feature transformation with other algorithms.

There can be many reasons for data transformation and feature engineering, but here I list two main reasons:

Mandatory transformations

These transformations are necessary to solve a problem (such as building a machine learning model) for data compatibility. Examples include:

- Converting non-numeric features into numeric. For example if a feature has non-numeric values, then average, sum, and median calculations will be impossible; likewise we can not perform matrix multiplication on a string, therefore, we must convert the string to some numeric representation.
- Resizing inputs to a fixed size. Some linear models and feed-forward neural networks have a fixed number of input nodes, so your input data must always have the same size. For example, image models need to reshape the images in their dataset to a fixed size.

Optional transformations

Optional data transformations may help the machine learning model to perform better. These transformations may include:

- Changing text to lowercase before applying other data transformations
- Tokenization and removing non-essential words such as “of”, “a”, “the”, “so”, ...
- Normalized numeric features: this might help you to build a better model

Adding a new feature

Sometimes you want to add a new derived feature (because you need that derived feature in your machine learning algorithm) to your data set, to add a new column or feature to your dataset, you may use the function `DataFrame.withColumn()`. This concept is demonstrated below:

```
# SparkSession available as 'spark'.
>>> column_names = ["emp_id", "salary"]
>>> records = [(100, 120000), (200, 170000), (300, 150000)]
>>> df = spark.createDataFrame(records, column_names)
>>> df.show()
+-----+
|emp_id|salary|
+-----+
|    100|120000|
|    200|170000|
|    300|150000|
+-----+
```

You may use Spark's `DataFrame.withColumn()` to add a new column/feature:

```
>>> df2 = df.withColumn("bonus", df.salary * 0.05)
>>> df2.show()
+-----+-----+
|emp_id|salary| bonus|
+-----+-----+
|    100|120000| 6000.0|
|    200|170000| 8500.0|
|    300|150000| 7500.0|
+-----+-----+
```

Applying UDF

UDF refers to User Defined Function. If PySpark does not provide your desired function, then you may define Python functions and register them as UDF, you may then apply it in your data transformations.

Therefore, we can define functions on PySpark as we would on Python but it would not be compatible with our spark DataDrame. To make your Python function compatible with PySpark, you use Spark's `udf` as illustrated below: To apply a UDF it is enough to add it as "decorator" of your Python function with a return data type of data associated with its output.

```
from pyspark.sql.functions import udf

>>> @udf("integer") ❶
... def tripled(num):
...     return 3*int(num)
...
>>> df2 = df.withColumn('tripled_col', tripled(df.salary))
>>> df2.show()
+-----+-----+
|emp_id|salary|tripled_col| ❷
+-----+-----+
|    100|120000|      360000|
|    200|170000|      510000|
|    300|150000|      450000|
+-----+-----+
```

❶ Function `tripled()` is a UDF and returns integer data type.

❷ `tripled_col` is a derived feature

Note that, if your features are represented as an RDD (each RDD element may represent an instance of your features), then you may use `RDD.map()` function to add a new feature to your feature set.

Pipeline

In machine learning algorithms, you may glue several stages together and run them together. Consider 3 stages called {Stage-1, Stage-2, Stage-3}, where output of Stage-1 is used as an input to Stage-2 and output of Stage-2 is used as an input to Stage-3. These 3 stages form a simple pipeline. Suppose we have to transform the data in the order shown in Table 6-1.:

T
a
b
l
e

6
-
I
.
P
i
p
e
l
i
n
e

S
t
a
g
e
s

Stage	Description
Stage-1	Label Encode or String Index the column <code>dept</code> * (create <code>dept_index</code> column)
Stage-2	Label Encode or String Index the column <code>education</code> (create <code>education_index</code> column)
Stage-3	One-Hot Encode the indexed column <code>education_index</code> (create <code>education_OHE</code> column)

Spark provides a pipeline API defined as `pyspark.ml.Pipeline(*, stages=None)`, which acts as an estimator. According to Spark documentation: “a Pipeline consists of a sequence of

stages, each of which is either an `Estimator` or a `Transformer`. When `Pipeline.fit()` is called, the stages are executed in order. If a stage is an `Estimator`, its `Estimator.fit()` method will be called on the input dataset to fit a model. Then the model, which is a transformer, will be used to transform the dataset as the input to the next stage. If a stage is a `Transformer`, its `Transformer.transform()` method will be called to produce the dataset for the next stage. The fitted model from a `Pipeline` is a `PipelineModel`, which consists of fitted models and transformers, corresponding to the pipeline stages. If `stages` is an empty list, the pipeline acts as an identity transformer.”

To understand the pipeline concept, first we create a `DataFrame` (as input data), and then create a simple pipeline using the `pyspark.ml.Pipeline`.

Let's create a sample `DataFrame` with three columns as shown below.

```
# spark : as SparkSession
# create a dataframe
df = spark.createDataFrame([
    (1, 'CS', 'MS'),
    (2, 'MATH', 'PHD'),
    (3, 'MATH', 'MS'),
    (4, 'CS', 'MS'),
    (5, 'CS', 'PHD'),
    (6, 'ECON', 'BS'),
    (7, 'ECON', 'BS'),
], ['id', 'dept', 'education'])
```

Next, let's view the sample data:

```
df.show()
>>> df.show()
+---+---+---+
| id|dept|education|
+---+---+---+
| 1| CS| MS|
| 2|MATH| PHD|
| 3|MATH| MS|
| 4| CS| MS|
| 5| CS| PHD|
| 6|ECON| BS|
| 7|ECON| BS|
+---+---+---+
```

Now that we have created the `DataFrame`, suppose we have to transform the data (as a `DataFrame`) by 3 defined stages (`{Stage-1, Stage-2, Stage-3}`). At each stage, we will pass the input and output column name and setup the pipeline by passing the defined stages in the list of the `Pipeline` object.

Spark's pipeline model then performs specific steps one by one in a sequence and gives us the final desired result. Below, we implement the pipeline for the 3 defined stages:

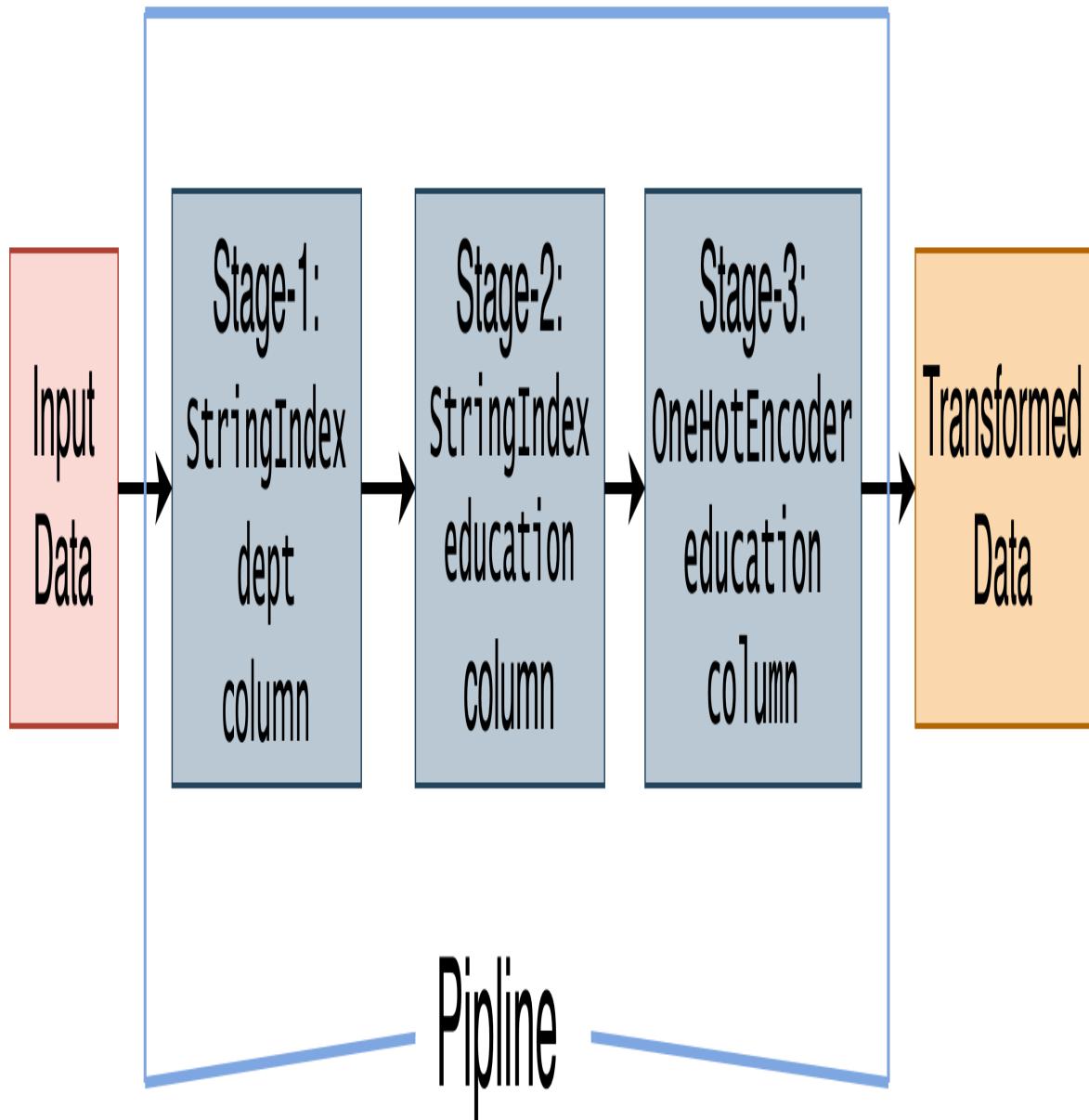


Figure 6-2. Pipeline with Three Stages

The three stages are implemented as:

```

from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer
from pyspark.ml.feature import OneHotEncoder

# Stage-1 : transform the `dept` column to numeric
stage_1 = StringIndexer(inputCol= 'dept', outputCol= 'dept_index')
#
# Stage-2 : transform the `education` column to numeric
stage_2 = StringIndexer(inputCol= 'education', outputCol= 'education_index')
#
# Stage-3 : one hot encode the numeric column `education_index`
stage_3 = OneHotEncoder(inputCols=[ 'education_index'], outputCols=[ 'education_OHE'])

```

Next, we define a pipeline with these three stages:

```
# setup the pipeline: glue the stages together
pipeline = Pipeline(stages=[stage_1, stage_2, stage_3])

# fit the pipeline model and transform the data as defined
pipeline_model = pipeline.fit(df)

# view the transformed data
final_df = pipeline_model.transform(df)
final_df.show(truncate=False)

+---+-----+-----+-----+-----+
|id |dept|education|dept_index|education_index|education_OHE|
+---+-----+-----+-----+-----+
|1  |CS   |MS      |0.0      |0.0      ||(2,[0],[1.0])|
|2  |MATH |PHD     |2.0      |2.0      ||(2,[],[])    |
|3  |MATH |MS      |2.0      |0.0      ||(2,[0],[1.0])|
|4  |CS   |MS      |0.0      |0.0      ||(2,[0],[1.0])|
|5  |CS   |PHD     |0.0      |2.0      ||(2,[],[])    |
|6  |ECON |BS      |1.0      |1.0      ||(2,[1],[1.0])|
|7  |ECON |BS      |1.0      |1.0      ||(2,[1],[1.0])|
+---+-----+-----+-----+-----+
```

Binarizer

Binarize data means to set feature values to 0 or 1 according to a threshold. Values greater than the threshold map to 1, while values less than or equal to the threshold map to 0. With the default threshold of 0, only positive values map to 1. Binarization is the process of thresholding numerical features to binary ``{0, 1}`` features.

Spark's Binarizer takes the parameters `inputCol` and `outputCol`, as well as the `threshold` for binarization. Feature values greater than the threshold are binarized to 1.0; values equal to or less than the threshold are binarized to 0.0.

First, let create a DataFrame with a single feature:

```
from pyspark.ml.feature import Binarizer

raw_df = spark.createDataFrame([
    (1, 0.1),
    (2, 0.2),
    (3, 0.5),
    (4, 0.8),
    (5, 0.9),
    (6, 1.1)
], ["id", "feature"])
```

Next, we create Binarizer with `threshold=0.5`: so any value less than or equal to threshold will map into `0.0` and any value greater than threshold will map into `1.0`.

```
>>> from pyspark.ml.feature import Binarizer
>>> binarizer = Binarizer(threshold=0.5, inputCol="feature", outputCol="binarized_feature")
```

Finally, we apply the defined Binarizer to a feature column:

```
binarized_df = binarizer.transform(raw_df)

>>> print("Binarizer output with Threshold = %f" % binarizer.getThreshold())
Binarizer output with Threshold = 0.500000

>>> binarized_df = binarizer.transform(raw_df)
>>> binarized_df.show(truncate=False)
+---+-----+
|id |feature|binarized_feature|
+---+-----+
|1  |0.1    |0.0              |
|2  |0.2    |0.0              |
|3  |0.5    |0.0              |
|4  |0.8    |1.0              |
|5  |0.9    |1.0              |
|6  |1.1    |1.0              |
+---+-----+
```

Imputer

Spark's `Imputer` is an imputation transformer for completing missing values. Real world datasets may contain missing values, often encoded as nulls, blanks, NaNs or other placeholders (for example in SQL, missing values are denoted by NULL). There are many methods to handle missing values:

- Delete instances if there is any missing feature (this might not be such a good idea since important information from other features will be lost)
- For a missing feature, find the average of that feature and replace missing values by the computed average
- A better strategy is to impute the missing values, i.e., to infer them from the known part of the data.

Spark's Imputer has the following signature:

```
class pyspark.ml.feature.Imputer(*, strategy='mean', missingValue=nan, inputCols=None, outputCols=None,
inputCol=None, outputCol=None, relativeError=0.001)
```

`Imputer` uses either the mean or the median of the columns in which the missing values are located. The input columns should be of numeric type. Currently `Imputer` does not support categorical features and may create incorrect values for a categorical feature.

Note that the mean/median/mode value is computed after filtering out missing values. All null values in the input columns are treated as missing, and so are also imputed. For computing median, `pyspark.sql.DataFrame.approxQuantile()` is used with a relative error of 0.001.

Imputer can impute custom values other than NaN by `.setMissingValue(custom_value)`. For example, `.setMissingValue(0)` will impute all occurrences of (0). Note all null values in the input columns are treated as missing, and so are also imputed.

The following example show how an imputer can be used. Suppose that we have a DataFrame with 3 columns `id`, `col1` and `col2`:

```
>>> df = spark.createDataFrame([
...     (1, 12.0, 5.0),
...     (2, 7.0, 10.0),
...     (3, 10.0, 12.0),
...     (4, 5.0, float("nan")),
...     (5, 6.0, None),
...     (6, float("nan"), float("nan")),
...     (7, None, None)
... ], ["id", "col1", "col2"])
>>> df.show(truncate=False)
+---+---+---+
|id |col1|col2|
+---+---+---+
|1  |12.0|5.0 |
|2  |7.0 |10.0|
|3  |10.0|12.0|
|4  |5.0 |NaN  |
|5  |6.0 |null |
|6  |NaN  |NaN  |
|7  |null|null |
+---+---+---+
```

Next, let's create an imputer and apply it to our created data:

```
>>> from pyspark.ml.feature import Imputer
>>> imputer = Imputer(inputCols=["col1", "col2"], outputCols=["col1_out", "col2_out"])
>>> model = imputer.fit(df)
>>> transformed = model.transform(df)
>>> transformed.show(truncate=False)
+---+---+---+---+
|id |col1|col2|col1_out|col2_out|
+---+---+---+---+
|1  |12.0|5.0 |12.0   |5.0    |
|2  |7.0 |10.0|7.0    |10.0   |
|3  |10.0|12.0|10.0   |12.0   |
|4  |5.0 |NaN  |5.0    |9.0    |
|5  |6.0 |null |6.0    |9.0    |
|6  |NaN  |NaN  |8.0    |9.0    |
|7  |null|null |8.0    |9.0    |
+---+---+---+---+
```

How did we get values **8.0** for missing values of `col1` and **9.0** for missing values of `col2`? It is easy: since the default strategy is “mean”, then we compute averages for missing values:

```
col1: (12.0+7.0+10.0+5.0+6.0) / 5 = 40 / 5 = 8.0
col2: (5.0+10.0+12.0) / 3 = 27.0 / 3 = 9.0
```

Based on your data requirements, you may set the missing values by using the `imputer()` function to the “median” of available feature values:

```
>>> imputer.setStrategy("median")
>>> model = imputer.fit(df)
>>> transformed = model.transform(df)
>>> transformed.show(truncate=False)
+---+---+---+---+
|id |col1|col2|col1_out|col2_out|
+---+---+---+---+
|1  |12.0|5.0 |12.0   |5.0    |
|2  |7.0  |10.0|7.0    |10.0   |
|3  |10.0 |12.0|10.0   |12.0   |
|4  |5.0  |NaN  |5.0    |10.0   |
|5  |6.0  |null |6.0    |10.0   |
|6  |NaN  |NaN  |7.0    |10.0   |
|7  |null |null |7.0    |10.0   |
+---+---+---+---+
```

How did we get values **7.0** for missing values of `col1` and **10.0** for missing values of `col2`? It is easy: we just compute median for missing values:

```
median(col1) =
median(12.0, 7.0, 10.0, 5.0, 6.0) =
median(5.0, 6.0, 7.0, 10.0, 12.0) =
7.0

median(col2) =
median(5.0, 10.0, 12.0) =
10.0
```

Tokenization

Tokenization is an algorithm (or set of algorithms) for splitting a phrase, sentence, paragraph, or an entire text document into smaller units, such as individual words, bigrams, or terms. Each of these smaller units are called **tokens**. For example, the lexical analyzer (as an algorithm in compiler writing) breaks programming syntaxes into a series of tokens, by removing any whitespace or comments in the source code. Therefore, tokenization is a process of splitting a string into words, symbols, or any other meaningful tokens (such as **bigram** or **N-grams**).

In Spark, you may use `Tokenizer` and `RegexTokenizer` (which you may define custom tokenization by defining regular expressions) to tokenize strings.

Tokenizer

Spark’s `Tokenizer` is a tokenizer that converts the input string to lowercase and then splits it by white spaces.

```
>>> docs = [(1, "a Fox jumped over FOX"),
           (2, "RED of fox jumped")]
```

```
>>> df = spark.createDataFrame(docs, ["id", "text"])
>>> df.show(truncate=False)
+---+-----+
|id |text      |
+---+-----+
|1  |a Fox jumped over FOX|
|2  |RED of fox jumpled   |
+---+-----+
```

Now, let's apply tokenizer:

```
>>> tokenizer = Tokenizer(inputCol="text", outputCol="tokens")
>>> tokenized = tokenizer.transform(df)
>>> tokenized.select("text", "tokens").withColumn("tokens_length",
countTokens(col("tokens"))).show(truncate=False)
+-----+-----+-----+
|text      |tokens          |tokens_length|
+-----+-----+-----+
|a Fox jumped over FOX|[a, fox, jumped, over, fox]|5
|RED of fox jumpled   |[red, of, fox, jumpled]|4
+-----+-----+-----+
```

RegexTokenizer

Spark's `RegexTokenizer` is a “regular expression” based tokenizer that extracts tokens either by using the provided regex pattern (in Java dialect) to split the text (default) or repeatedly matching the regex (if gaps is false).

```
>>> regexTokenizer = RegexTokenizer(inputCol="text", outputCol="tokens",
pattern="\W", minTokenLength=3)
>>> regex_tokenized = regexTokenizer.transform(df)
>>> regex_tokenized.select("text", "tokens").withColumn("tokens_length",
countTokens(col("tokens"))).show(truncate=False)
+-----+-----+-----+
|text      |tokens          |tokens_length|
+-----+-----+-----+
|a Fox jumped over FOX|[fox, jumped, over, fox]|4
|RED of fox jumpled   |[red, fox, jumpled]|3
+-----+-----+-----+
```

Tokenization with Pipeline

Here, we create a `DataFrame` with two columns, which then we will apply the `RegexTokenizer` function.

```
>>> docs = [(1, "a Fox jumped, over, the fence?"),
(2, "a RED, of fox?")]
>>> df = spark.createDataFrame(docs, ["id", "text"])
>>> df.show(truncate=False)
+---+-----+
|id |text      |
+---+-----+
|1  |a Fox jumped, over, the fence?|
+---+-----+
```

```
|2 |a RED, of fox? |
```

Next, we apply `RegexTokenizer` function:

```
>>> tk = RegexTokenizer(pattern=r'(?:(\p{Punct}|\s)+', inputCol="text", outputCol='text2')
>>> sw = StopWordsRemover(inputCol='text2', outputCol='text3')
>>> pipeline = Pipeline(stages=[tk, sw])
>>> df4 = pipeline.fit(df).transform(df)
>>> df4.show(truncate=False)
+-----+-----+-----+
|id |text          |text2           |text3          |
+-----+-----+-----+
|1 |a Fox jumped, over, the fence?|[a, fox, jumped, over, the, fence]|[[fox, jumped, fence]]|
|2 |a RED, of fox?    |[a, red, of, fox] |[[red, fox]]   |
+-----+-----+-----+
```

Standardization

One of the most popular techniques for scaling numerical data prior to modeling is standardization. Standardizing a dataset involves rescaling the distribution of values so that the mean of observed values (as a feature) is **0.00** and the standard deviation is **1.00**.

In general, many machine learning algorithms perform better — you can build a better realistic model — when numerical input variables — also called features — are scaled to a standard range. For example, machine algorithms such as linear regression and K-nearest neighbors require standardize values, otherwise the build models might be under-fit or over-fit (under performing algorithms). For example, when you use a weighted sum of the input and distance measures, it is better to standardize your features.

A value is standardized as follows:

$$y = (x - \text{mean}) / \text{standard_deviation}$$

Where the mean is calculated as:

$$\text{mean} = \frac{\sum(x)}{\text{count}(x)}$$

$$\bar{x} = \frac{1}{N} \sum_{i=1}^N x_i$$

And the standard deviation is calculated as:

$$\text{standard_deviation} = \sqrt{ \frac{\sum((x - \text{mean})^2)}{\text{count}(x)} }$$

$$sd = \sqrt{\frac{1}{N} \sum_{i=1}^N (x_i - \bar{x})^2}$$

Let $X = (1, 3, 6, 10)$. Then mean/average is calculated as:

$$\text{mean} = (1+3+6+10)/4 = 20/4 = 5.0$$

and standard deviation is calculated as:

```
standard_deviation
= sqrt ( ((1-5)^2 + (3-5)^2 + (6-5)^2 + (10-5)^2) / 4 )
= sqrt ((16+4+1+25)/4)
= sqrt(46/4)
= sqrt(11.5) = 3.39116
```

So the new standardized values will be :

```
y = (y1, y2, y3, y4) = (-1.1795, -0.5897, 0.2948, 1.4744)
where
y1 = (1 - 5.0) / 3.39116
y2 = (3 - 5.0) / 3.39116
y3 = (6 - 5.0) / 3.39116
y4 = (10 - 5.0) / 3.39116
```

As you can see the mean of standardized values (y) is **0.00** and its standard deviation is **1.00**.

Let's go over how to perform standardization in PySpark? How to standardize a single column — feature — in Spark using `StandardScaler`? Let's say that we are trying to standardize (`mean = 0.00, stddev = 1.00`) one column (`age`) in a dataframe. Below is the code in PySpark:

```
features = [('alex', 1), ('bob', 3), ('ali', 6), ('dave', 10)]
columns = ("name", "age")
samples = spark.createDataFrame(features, columns)
>>> samples.show()
+---+---+
|name|age|
+---+---+
|alex| 1|
| bob| 3|
| ali| 6|
|dave| 10|
+---+---+
```

Method-1: Using DataFrame Functions

```
>>> from pyspark.sql.functions import stddev, mean, col
>>> (samples
...     .select(mean("age").alias("mean_age"), stddev("age").alias("stddev_age")))
...     .crossJoin(samples)
...     .withColumn("age_scaled" , (col("age") - col("mean_age")) / col("stddev_age")))
...     .show(truncate=False)
+-----+-----+-----+
|mean_age|stddev_age| name|age|age_scaled|
```

```
+-----+-----+-----+-----+
| 5.0 | 3.9157800414902435|alex|1 | -1.0215078369104984|
| 5.0 | 3.9157800414902435|bob | 3 | -0.5107539184552492|
| 5.0 | 3.9157800414902435|ali | 6 | 0.2553769592276246 |
| 5.0 | 3.9157800414902435|dave|10 | 1.276884796138123 |
+-----+-----+-----+-----+
```

or alternatively, we may write it as:

```
>>> mean_age, sttdev_age = samples.select(mean("age"), stddev("age")).first()
>>> samples.withColumn("age_scaled", (col("age") - mean_age) / sttdev_age).show(truncate=False)
+-----+
| name|age|age_scaled   |
+-----+
| alex|1 |-1.0215078369104984|
| bob |3 |-0.5107539184552492|
| ali |6 | 0.2553769592276246 |
| dave|10 | 1.276884796138123 |
+-----+
```

Method-2: Using ML Functions

Use `pyspark.ml.feature.VectorAssembler` to transform to a vector:

```
>>> from pyspark.ml.feature import VectorAssembler
>>> from pyspark.ml.feature import StandardScaler
>>> vecAssembler = VectorAssembler(inputCols=['age'], outputCol="age_vector")
>>> samples2 = vecAssembler.transform(samples)
>>> samples2.show()
+-----+
| name|age|age_vector   |
+-----+
| alex| 1 | [1.0]|
| bob | 3 | [3.0]|
| ali | 6 | [6.0]|
| dave| 10 | [10.0]|
+-----+
>>> scaler = StandardScaler(inputCol="age_vector", outputCol="age_scaled",
    withStd=True, withMean=True)
```

```

>>> scalerModel = scaler.fit(samples2)
>>> scaledData = scalerModel.transform(samples2)
>>> scaledData.show(truncate=False)
+-----+-----+
|name|age|age_vector|age_scaled|
+-----+-----+
|alex|1 |[1.0] |[-1.0215078369104984]|
|bob |3 |[3.0] |[-0.5107539184552492]|
|ali |6 |[6.0] |[0.2553769592276246] |
|dave|10|[10.0] |[1.276884796138123] |
+-----+-----+

```

Standardization can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Also, unlike normalization, standardization does not have a bounding range. So, even if you have outliers in your data, they will not be impacted by standardization.

Normalization

Normalization is a scaling technique often applied as part of data preparation for machine learning. The goal of normalization is to change the values of numeric columns in the dataset to use a common scale, without distorting differences in the ranges of values or losing information. Normalization scales each numeric input variable separately to the range [0.00 .. 1.00], which is the range for floating-point values where we have the most precision. Therefore, Normalization of features is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0.00 and 1.00. This technique is also known as Min-Max scaling.

Here's the formula for normalization:

$$\tilde{X}_i = \frac{X_i - X_{\min}}{X_{\max} - X_{\min}}$$

Note that X_{\max} and X_{\min} are the maximum and the minimum values of the given feature, X_i , respectively.

To show the normalization process, let's a DataFrame with three features:

```

df = spark.createDataFrame([
    (100, 77560, 45),
    (200, 41560, 23),
    (300, 30285, 20),
    (400, 10345, 6),
    (500, 88000, 50)
], ["user_id", "revenue", "num_of_days"])

print("Before Scaling :")
df.show(5)
>>> df.show()
+-----+-----+
|user_id|revenue|num_of_days|
+-----+-----+
|    100| 77560|      45|
|    200| 41560|      23|
|    300| 30285|      20|
|    400| 10345|       6|
+-----+-----+

```

```

|    500| 88000|      50|
+-----+-----+

```

Next, we apply the `MinMaxScaler` to our features:

```

from pyspark.ml.feature import MinMaxScaler
from pyspark.ml.feature import VectorAssembler
from pyspark.ml import Pipeline
from pyspark.sql.functions import udf
from pyspark.sql.types import DoubleType

# UDF for converting column type from vector to double type
unlist = udf(lambda x: round(float(list(x)[0]),3), DoubleType())

# Iterating over columns to be scaled
for i in ["revenue", "num_of_days"]:
    # VectorAssembler Transformation - Converting column to vector type
    assembler = VectorAssembler(inputCols=[i],outputCol=i+"_Vect")

    # MinMaxScaler Transformation
    scaler = MinMaxScaler(inputCol=i+"_Vect", outputCol=i+"_Scaled")

    # Pipeline of VectorAssembler and MinMaxScaler
    pipeline = Pipeline(stages=[assembler, scaler])

    # Fitting pipeline on dataframe
    df = pipeline.fit(df).transform(df).withColumn(i+"_Scaled", unlist(i+"_Scaled")).drop(i+"_Vect")

print("After Scaling :")
df.show(5)

>>> for i in ["revenue", "num_of_days"]:
...     assembler = VectorAssembler(inputCols=[i],outputCol=i+"_Vect")
...     scaler = MinMaxScaler(inputCol=i+"_Vect", outputCol=i+"_Scaled")
...     pipeline = Pipeline(stages=[assembler, scaler])
...     df = pipeline.fit(df).transform(df).withColumn(i+"_Scaled", unlist(i+"_Scaled")).drop(i+"_Vect")
...

```

Next, we examine scaled values:

```

>>> df.show(5)
+-----+-----+-----+-----+
|user_id|revenue|num_of_days|revenue_Scaled|num_of_days_Scaled|
+-----+-----+-----+-----+
|    100| 77560|      45|      0.866|      0.886|
|    200| 41560|      23|      0.402|      0.386|
|    300| 30285|      20|      0.257|      0.318|
|    400| 10345|       6|      0.0|      0.0|
|    500| 88000|      50|      1.0|      1.0|
+-----+-----+-----+-----+

```

Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution. This can be useful in algorithms that do not assume any distribution of the data like linear regression, K-Nearest Neighbors, and Neural Networks.

Scaling a column using Pipeline

First, let's define a set of features and then we will apply a scaler in a pipeline:

```
>>> from pyspark.ml.feature import MinMaxScaler
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.feature import VectorAssembler
>>> triplets = [(0, 1, 100), (1, 2, 200), (2, 5, 1000)]
>>> df = spark.createDataFrame(triplets, ['x', 'y', 'z'])
>>> df.show()
+---+---+---+
| x | y | z |
+---+---+---+
| 0 | 1 | 100|
| 1 | 2 | 200|
| 2 | 5 |1000|
+---+---+---+
```

Next, we apply `MinMaxScaler` in a pipeline:

```
>>> assembler = VectorAssembler(inputCols=["x"], outputCol="x_vector")
>>> scaler = MinMaxScaler(inputCol="x_vector", outputCol="x_scaled")
>>> pipeline = Pipeline(stages=[assembler, scaler])
>>> scalerModel = pipeline.fit(df)
>>> scaledData = scalerModel.transform(df)
>>> scaledData.show(truncate=False)
+---+---+---+---+---+
|x |y |z |x_vector|x_scaled|
+---+---+---+---+---+
|0 |1 |100|[0.0] |[0.0] |
|1 |2 |200|[1.0] |[0.5] |
|2 |5 |1000|[2.0] |[1.0] |
+---+---+---+---+---+
```

MinMaxScaler on multiple columns

Next, we show how to apply a scaler (`MinMaxScaler`) on multiple columns:

```
>>> triplets = [(0, 1, 100), (1, 2, 200), (2, 5, 1000)]
>>> df = spark.createDataFrame(triplets, ['x', 'y', 'z'])
>>> df.show()
+---+---+---+
| x | y | z |
+---+---+---+
| 0 | 1 | 100|
| 1 | 2 | 200|
| 2 | 5 |1000|
+---+---+---+
>>> from pyspark.ml import Pipeline
>>> from pyspark.ml.feature import MinMaxScaler
>>> columns_to_scale = ["x", "y", "z"]
>>> assemblers = [VectorAssembler(inputCols=[col], outputCol=col + "_vector") for col in columns_to_scale]
>>> scalers = [MinMaxScaler(inputCol=col + "_vector", outputCol=col + "_scaled") for col in columns_to_scale]
>>> pipeline = Pipeline(stages=assemblers + scalers)
>>> scalerModel = pipeline.fit(df)
```

```

>>> scaledData = scalerModel.transform(df)
>>> scaledData.show(truncate=False)
+---+---+---+---+---+---+---+
| x | y | z | x_vector|y_vector|z_vector|x_scaled|y_scaled|z_scaled |
+---+---+---+---+---+---+---+
| 0 | 1 | 100 | [0.0] | [1.0] | [100.0] | [0.0] | [0.0] | [0.0] |
| 1 | 2 | 200 | [1.0] | [2.0] | [200.0] | [0.5] | [0.25] | [0.1111111111111111] |
| 2 | 5 | 1000 | [2.0] | [5.0] | [1000.0] | [1.0] | [1.0] | [1.0] |
+---+---+---+---+---+---+---+

```

Extra Post-processing: you can recover the columns in their original names with some post-processing. For example:

```

from pyspark.sql import functions as f

names = {x + "_scaled": x for x in columns_to_scale}
scaledData = scaledData.select([f.col(c).alias(names[c]) for c in names.keys()])

```

The output will be:

```

scaledData.show()
+---+---+---+
| y | x | z |
+---+---+---+
| [0.0]| [0.0] | [0.0] |
| [0.25]| [0.5] | [0.1111111111111111] |
| [1.0]| [1.0] | [1.0] |
+---+---+---+

```

Normalization using Normalizer

The purpose of a normalizer is to calculate distance between features. The most commonly used distance metrics are “Manhattan distance” and the “Euclidean distance”. The Normalizer takes a parameter p from the user which represents the power norm.

For example, Manhattan norm (Mahnatan distance) $p = 1$; Euclidean norm (Euclidean distance) $p = 2$;

```

L1: z = || x ||1 = sum(|xi|) for i =1, ..., n
L2: z = || x ||2 = sqrt(sum(xi^2)) for i=1,..., n

```

```

from pyspark.ml.feature import Normalizer
# Let us create an object of the class Normalizer product
ManhattanDistance=Normalizer().setP(1)
    .setInputCol("features").setOutputCol("Manhattan Distance")
EuclideanDistance=Normalizer().setP(2)
    .setInputCol("features").setOutputCol("Euclidean Distance")
# Let us transform
ManhattanDistance.transform(scaleDF).show()
+---+---+---+
| id | features | Manhattan Distance |
+---+---+---+
| 0 | [1.0,0.1,-1.0] | [0.47619047619047... |
| 1 | [2.0,1.1,1.0] | [0.48780487804878... |

```

```

|  0|[1.0,0.1,-1.0]||[0.47619047619047...|
|  1|[2.0,1.1,1.0]||[0.48780487804878...|
|  1|[3.0,10.1,3.0]||[0.18633540372670...|
+---+-----+-----+
EuclideanDistance.transform(scaleDF).show()
+-----+-----+
| id|    features| Euclidean Distance|
+-----+-----+
|  0|[1.0,0.1,-1.0]||[0.70534561585859...|
|  1|[2.0,1.1,1.0]||[0.80257235390512...|
|  0|[1.0,0.1,-1.0]||[0.70534561585859...|
|  1|[2.0,1.1,1.0]||[0.80257235390512...|
|  1|[3.0,10.1,3.0]||[0.27384986857909...|
+-----+-----+

```

String Indexing

String indexing means converting strings to numerical values. Most of the ML algorithms require converting categorical features (such as strings) into numerical ones.

According to Spark documentation, StringIndexer is a label indexer that maps a string column of labels to an ML column of label indices. If the input column is numeric, we cast it to string and index the string values. The indices are in [0, numLabels). By default, this is ordered by label frequencies so the most frequent label gets index 0. The ordering behavior is controlled by setting stringOrderType.

Apply StringIndexer to a Single Column

I have a PySpark dataframe

```

+-----+-----+-----+
|address|      date|name|food|
+-----+-----+-----+
|1111111|20151122045510| Yin|gre |
|1111111|20151122045501| Yin|gre |
|1111111|20151122045500| Yln|gra |
|1111112|20151122065832| Yun|ddd |
|1111113|20160101003221| Yan|fdf |
|1111111|20160703045231| Yin|gre |
|1111114|20150419134543| Yin|fdf |
|1111115|20151123174302| Yen|ddd |
|2111115|      20123192| Yen|gre |
+-----+-----+-----+

```

that I want to transform to use with `pyspark.ml`. I can use a `StringIndexer` to convert the name column to a numeric category:

```

indexer = StringIndexer(inputCol="name", outputCol="name_index").fit(df)
df_ind = indexer.transform(df)
df_ind.show()
+-----+-----+-----+-----+
|address|      date|name|name_index|food|
+-----+-----+-----+-----+

```

```
+-----+-----+-----+
|1111111|20151122045510| Yin|    0.0|gre |
|1111111|20151122045501| Yin|    0.0|gre |
|1111111|20151122045500| Yln|    2.0|gra |
|1111112|20151122065832| Yun|    4.0|ddd |
|1111113|20160101003221| Yan|    3.0|fdf |
|1111111|20160703045231| Yin|    0.0|gre |
|1111114|20150419134543| Yin|    0.0|fdf |
|1111115|20151123174302| Yen|    1.0|ddd |
|2111115|     20123192| Yen|    1.0|gre |
+-----+-----+-----+
```

Apply StringIndexer to several columns

How do we apply `StringIndexer` to several columns? The simple way to do it is to combine several `StringIndex` on a list and use a `Pipeline` to execute them all:

```
from pyspark.ml import Pipeline
from pyspark.ml.feature import StringIndexer

indexers = [ StringIndexer(inputCol=column, outputCol=column+"_index").fit(df)
             for column in list(set(df.columns)-set(['date'])) ]

pipeline = Pipeline(stages=indexers)
df_indexed = pipeline.fit(df).transform(df)

df_indexed.show()
+-----+-----+-----+-----+-----+-----+
|address|      date|food|name|food_index|name_index|address_index|
+-----+-----+-----+-----+-----+-----+
|1111111|20151122045510| gre| Yin|    0.0|    0.0|    0.0|
|1111111|20151122045501| gra| Yin|    2.0|    0.0|    0.0|
|1111111|20151122045500| gre| Yln|    0.0|    2.0|    0.0|
|1111112|20151122065832| gre| Yun|    0.0|    4.0|    3.0|
|1111113|20160101003221| gre| Yan|    0.0|    3.0|    1.0|
|1111111|20160703045231| gre| Yin|    0.0|    0.0|    0.0|
|1111114|20150419134543| gre| Yin|    0.0|    0.0|    5.0|
|1111115|20151123174302| ddd| Yen|    1.0|    1.0|    2.0|
|2111115|     20123192| ddd| Yen|    1.0|    1.0|    4.0|
+-----+-----+-----+-----+-----+-----+
```

Vector assembler

The main function of vector assembler is to concatenate all the features into a single vector which can be further passed to the estimator or machine learning algorithm. Spark's VectorAssembler is a feature transformer that merges multiple columns into a vector column.

Consider the following DataFrame:

```
df.show()
+-----+
|col1|col2|col3|
+-----+
```

```

| 7.0| 8.0| 9.0|
| 1.1| 1.2| 1.3|
| 4.0| 5.0| 6.0|
| 2| 3| 4|
| 5.0| NaN|null|
+-----+

```

Next, we apply `VectorAssembler` to three features (`col1`, `col2`, `col3`) and merge 3 columns into a vector column (named as `features`):

```

from pyspark.ml.feature import VectorAssembler
input_columns = ["col1", "col2", "col3"]
assembler = VectorAssembler(inputCols=input_columns, outputCol="features")
# use the transform method to transform dataset into a vector
transformed = assembler.transform(df)
transformed.show()
+-----+-----+
|col1|col2|col3|    features|
+-----+-----+
| 7.0| 8.0| 9.0|[7.0,8.0,9.0]|
| 1.1| 1.2| 1.3|[1.1,1.2,1.3]|
| 4.0| 5.0| 6.0|[4.0,5.0,6.0]|
| 2| 3| 4|[2.0,3.0,4.0]|
| 5.0| NaN|null|[5.0,NaN,NaN]|
+-----+

```

If you want to skip the rows, which has `NaN` or `null` values, you may do it by `VectorAssembler.setParams(handleInvalid="skip")`:

```

assembler2 = VectorAssembler(inputCols=input_columns, outputCol="features")
    .setParams(handleInvalid="skip")

assembler2.transform(df).show()
+-----+-----+
|col1|col2|col3|    features|
+-----+-----+
| 7.0| 8.0| 9.0|[7.0,8.0,9.0]|
| 1.1| 1.2| 1.3|[1.1,1.2,1.3]|
| 4.0| 5.0| 6.0|[4.0,5.0,6.0]|
| 2| 3| 4|[2.0,3.0,4.0]|
+-----+

```

Bucketing

Data binning—also called Discrete binning or bucketing—is a data pre-processing technique used to reduce the effects of minor observation errors. The original data values which fall into a given small interval, a bin, are replaced by a value representative of that interval, often the central value. For example if a car price value is so scattered, then you may use bucketing instead of actual car prices.

Spark's Bucketizer transforms a column of continuous features to a column of feature buckets, where the buckets are specified by users.

Consider this example: there's no linear relationship between latitude and the housing values, but you may suspect that individual latitudes and housing values are related, but the relationship is not linear. Therefore you might bucketize the latitudes; for example (the example was borrowed from [Representation: Cleaning Data](#)), you may create buckets as:

```
Bin-1: 32 < latitude <= 33  
Bin-2: 33 < latitude <= 34  
...  
...
```

Binning technique can be applied on both categorical and numerical data. The following examples show both types of binning.

- Numerical Binning Example:

T
a
b
l
e

6
-
2
. *N*
u
m
e
r
i
c
a
l
B
i
n
n
i
n
g

E
x
a
m
p
l
e

Value	Bin
0-10	Very Low
11-30	Low
31-70	Mid
71-90	High

91-100

Very High

- Categorical Binning Example

T
a
b
l
e

6
-
3
. *C*
a
t
e
g
o
r
i
c
a
l
B
i
n
n
i
n
g

E
x
a
m
p
l
e

Value	Bin
India	Asia
China	Asia

Japan	Asia
Spain	Europe
Italy	Europe
Chile	South America
Brazil	South America

Binning is used genomics data as well: we bucketize human genome chromosomes (1, 2, 3, ..., 22, X, Y, MT). For instance chromosomes 1 has 250 million positions, which we may bucketize into 101 buckets as:

```
for id in (1, 2, 3, ..., 22, X, Y, MT):
    chr_position = (chromosome-<id> position)
    # chr_position range is from 1 to 250,000,000
    bucket = chr_position % 101
    # where
    #     0 <= bucket <= 100
```

Bucketing is a most straight forward approach for converting the continuous variables into categorical variable. To understand this, let's look at an example below. In PySpark the task of bucketing can be easily accomplished using the `Bucketizer` class.

To use the `Bucketizer` class, firstly, we shall accomplish the task of creating bucket borders. Let us define a list of bucket borders as the following example. Next, let us create a object of the `Bucketizer` class. Then we will apply the `transform` method to our defined Dataframe `dataframe`.

First, Let's create a sample dataframe for demo purpose:

```
>>> data = [('A', -99.99), ('B', -0.5), ('C', -0.3),
...   ('D', 0.0), ('E', 0.7), ('F', 99.99)]
>>>
>>> dataframe = spark.createDataFrame(data, ["id", "features"])
>>> dataframe.show()
+---+-----+
| id|features|
+---+-----+
|  A| -99.99|
|  B|   -0.5|
|  C|   -0.3|
|  D|    0.0|
|  E|    0.7|
|  F|  99.99|
+---+-----+
```

Next, we apply the `Bucketizer` to create buckets:

```

>>> bucket_borders=[-float("inf"), -0.5, 0.0, 0.5, float("inf")]
>>> from pyspark.ml.feature import Bucketizer
>>> bucketer = Bucketizer().setSplits(bucket_borders).setInputCol("features").setOutputCol("bucket")
>>> bucketer.transform(dataframe).show()
+-----+
| id|features|bucket|
+-----+
| A| -99.99|    0.0|
| B|   -0.5|    1.0|
| C|   -0.3|    1.0|
| D|    0.0|    2.0|
| E|    0.7|    3.0|
| F|  99.99|    3.0|
+-----+

```

QuantileDiscretizer

Spark's QuantileDiscretizer takes a column with continuous features and outputs a column with binned categorical features. The number of bins is set by the numBuckets parameter. It is possible that the number of buckets used will be smaller than this value, for example, if there are too few distinct values of the input to create enough distinct quantiles.

You can use both together like:

```

from pyspark.ml.feature import Bucketizer
from pyspark.ml.feature import QuantileDiscretizer
data = [(0, 18.0), (1, 19.0), (2, 8.0), (3, 5.0), (4, 2.2)]
df = spark.createDataFrame(data, ["id", "hour"])
print(df.show())
+---+
| id|hour|
+---+
| 0|18.0|
| 1|19.0|
| 2| 8.0|
| 3| 5.0|
| 4| 2.2|
+---+

qds = QuantileDiscretizer(numBuckets=5, inputCol="hour", outputCol="buckets", relativeError=0.01,
handleInvalid="error")
bucketizer = qds.fit(df)
bucketizer.setHandleInvalid("skip").transform(df).show()
+---+
| id|hour|buckets|
+---+
| 0|18.0|    3.0|
| 1|19.0|    3.0|
| 2| 8.0|    2.0|
| 3| 5.0|    2.0|
| 4| 2.2|    1.0|
+---+

```

Logarithm Transformation

In a nutshell, logarithm (commonly denoted by \log) transformation compresses range of large numbers and expands the range of small numbers. In mathematics, the logarithm is the inverse function to exponentiation and defined as (b is called the base number):

$$\log_b(x) = y \rightarrow b^y = x$$

In feature engineering, logarithm transformation (or sometimes called \log transform) is one of the most commonly used mathematical transformations. logarithm transformation helps us to handle skewed data and after transformation, the distribution becomes more approximate to normal. This means that by using the logarithm transformation on a feature, you can reduce skewness in data distribution and make it more normal. Another benefit of using logarithm transformation is that it decreases the effect of the outliers, due to the normalization of magnitude differences and hence the machine learning model become more robust. For example, the natural logarithm (base e) of number 4000 is 8.2940496401.

In using logarithm transformation, the data must have only positive values, otherwise you receive an error. Sometimes, you may add $1 - \log(x+1)$ — to your data before transforming it. So, you ensure the output of the transformation to be positive.

Spark provides the logarithm function in any base and defined as:

```
pyspark.sql.functions.log(arg1, arg2=None)
Description: Returns the first argument-based logarithm
of the second argument. If there is only one argument,
then this takes the natural logarithm of the argument.
```

The logarithm transformation is illustrated below. First, we create a DataFrame:

```
>>> data = [('gene1', 1.2), ('gene2', 3.4), ('gene1', 3.5), ('gene2', 12.6)]
>>> df = spark.createDataFrame(data, ["gene", "value"])
>>> df.show()
+---+---+
| gene|value|
+---+---+
|gene1| 1.2|
|gene2| 3.4|
|gene1| 3.5|
|gene2| 12.6|
+---+---+
```

Then we apply the logarithm transformation on a feature labeled as `value`:

```
>>> from pyspark.sql.functions import log
>>> df.withColumn("base-10", log(10.0, df.value))
    .withColumn("base-e", log(df.value)).show()
+---+---+---+
| gene|value| base-10| base-e|
+---+---+---+
|gene1| 1.2| 0.0791812460476249| 0.1823215567939546|
|gene2| 3.4| 0.531478917042255| 1.2237754316221157|
|gene1| 3.5| 0.5440680443502756| 1.252762968495368|
```

gene2	12.6	1.1003705451175627	2.533696813957432
-------	------	--------------------	-------------------

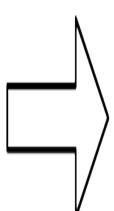
One Hot Encoding

To build a model in Machine learning, it requires that all input features and output predictions to be numeric. Therefore, this implies that if your data contains categorical data — such as education degree {BS, MBA, MS, MD, PHD} — , you must encode it to numbers before you can build and evaluate a model.

The Figure 6.1 illustrates the concept of One-Hot-Encoding. Each categorial value is converted to a binary vector.

Label Encoding

One Hot Encoding



Cancer Type	Biomarker		
Benign	3	1	0
Premalignant	6	0	1
Malignant	12	0	1

	Benign	Premalignant	Malignant	Biomarker
Benign	1	0	0	3
Premalignant	0	1	0	6
Malignant	0	0	1	12

Figure 6-3. One Hot Encoding Example

One hot encoder maps the label indices to a binary vector representation with at the most a single one-value. This method is used when you need to use categorical features but the algorithm expects continuous features. To understand this encoding method: consider a feature called “Safety-Level”, which has 5 categorial values (represented in the following table). The first column shows the feature and the rest of columns shows a binary vector (representing with at most one 1 in it) as a One Hot Encoded (as a vector with binary values).

*T
a
b
l
e
6
-
4*

*.
U
n
d
e
r
s
t
a
n
d
i
n
g*

*O
n
e
H
o
t
E
n
c
o
d
i
n
g*

Safety-Level (text)	Very-Low	Low	Medium	High	Very-High
Very-Low	1	0	0	0	0

Low	0	1	0	0	0
Medium	0	0	1	0	0
High	0	0	0	1	0
Very-High	0	0	0	0	1

The spark one hot encoder takes the indexed label/category from the string indexer and then encodes it into a sparse vector. Lets add the one hot encoder to the example above and see how this would work.

```
>>> from pyspark.sql.types import *
>>>
>>> schema = StructType().add("id","integer")\
...     .add("safety_level","string")\
...     .add("engine_type","string")
>>> schema
StructType(List(StructField(id,IntegerType,true),
                StructField(safety_level,StringType,true),
                StructField(engine_type,StringType,true)))
>>> data = [
...     (1,'Very-Low','v4'),
...     (2,'Very-Low','v6'),
...     (3,'Low','v6'),
...     (4,'Low','v6'),
...     (5,'Medium','v4'),
...     (6,'High','v6'),
...     (7,'High','v6'),
...     (8,'Very-High','v4'),
...     (9,'Very-High','v6')
... ]
>>>
>>> df = spark.createDataFrame(data, schema=schema)
>>> df.show(truncate=False)
+---+-----+-----+
|id |safety_level|engine_type|
+---+-----+-----+
|1  |Very-Low    |v4          |
|2  |Very-Low    |v6          |
|3  |Low          |v6          |
|4  |Low          |v6          |
|5  |Medium       |v4          |
|6  |High         |v6          |
|7  |High         |v6          |
|8  |Very-High   |v4          |
|9  |Very-High   |v6          |
+---+-----+-----+
```

Apply the `OneHotEncoder` transformation to the `safety_level` and `engine_type` features. In Spark, we can not apply `OneHotEncoder` to string columns directly. We need to first convert string columns to numeric values. For that we will use `StringIndexer`, after that we can apply the `OneHotEncoder` transformation.

First, we apply `StringIndexer` to the `safety_level` feature:

```
>>> from pyspark.ml.feature import StringIndexer
>>> safety_level_indexer = StringIndexer(inputCol="safety_level", outputCol="safety_level_index")
>>> safety_level_indexer
StringIndexer_fb96b3df2be3
>>> df1 = safety_level_indexer.fit(df).transform(df)
>>> df1.show()
>>> safety_level_indexer = StringIndexer(inputCol="safety_level", outputCol="safety_level_index")
>>> df1 = safety_level_indexer.fit(df).transform(df)
>>> df1.show()
+---+-----+-----+
| id|safety_level|engine_type|safety_level_index|
+---+-----+-----+
| 1| Very-Low| v4| 3.0|
| 2| Very-Low| v6| 3.0|
| 3| Low| v6| 1.0|
| 4| Low| v6| 1.0|
| 5| Medium| v4| 4.0|
| 6| High| v6| 0.0|
| 7| High| v6| 0.0|
| 8| Very-High| v4| 2.0|
| 9| Very-High| v6| 2.0|
+---+-----+-----+
```

Next, we apply `StringIndexer` to `engine_type` feature:

```
>>> engine_type_indexer = StringIndexer(inputCol="engine_type", outputCol="engine_type_index")
>>> df2 = engine_type_indexer.fit(df).transform(df)
>>> df2.show()
+---+-----+-----+
| id|safety_level|engine_type|engine_type_index|
+---+-----+-----+
| 1| Very-Low| v4| 1.0|
| 2| Very-Low| v6| 0.0|
| 3| Low| v6| 0.0|
| 4| Low| v6| 0.0|
| 5| Medium| v4| 1.0|
| 6| High| v6| 0.0|
| 7| High| v6| 0.0|
| 8| Very-High| v4| 1.0|
| 9| Very-High| v6| 0.0|
+---+-----+-----+
```

Apply `OneHotEncoder` to `safety_level_index` and `engine_type_index` columns

```
>>> from pyspark.ml.feature import OneHotEncoder
>>> onehotencoder_safety_level = OneHotEncoder(inputCol="safety_level_index",
outputCol="safety_level_vector")
>>> df11 = onehotencoder_safety_level.fit(df1).transform(df1)
>>> df11.show(truncate=False)
+---+-----+-----+-----+
|id |safety_level|engine_type|safety_level_index|safety_level_vector|
+---+-----+-----+-----+
|1 |Very-Low| v4| 3.0| [(4,[3],[1.0])]|
```

|2 |Very-Low| v6| 3.0| [(4,[3],[1.0])]|

```

| 3 | Low | v6 | 1.0 | (4,[1],[1.0]) |
| 4 | Low | v6 | 1.0 | (4,[1],[1.0]) |
| 5 | Medium | v4 | 4.0 | (4,[],[]) |
| 6 | High | v6 | 0.0 | (4,[0],[1.0]) |
| 7 | High | v6 | 0.0 | (4,[0],[1.0]) |
| 8 | Very-High | v4 | 2.0 | (4,[2],[1.0]) |
| 9 | Very-High | v6 | 2.0 | (4,[2],[1.0]) |
+---+-----+-----+-----+
[source%autofit, python]

>>> onehotencoder_engine_type = OneHotEncoder(inputCol="engine_type_index",outputCol="engine_type_vector")
>>> df12 = onehotencoder_engine_type.fit(df2).transform(df2)
>>> df12.show(truncate=False)
+-----+-----+-----+-----+
| id | safety_level | engine_type | engine_type_index | engine_type_vector |
+-----+-----+-----+-----+
| 1 | Very-Low | v4 | 1.0 | (1,[],[]) |
| 2 | Very-Low | v6 | 0.0 | (1,[0],[1.0]) |
| 3 | Low | v6 | 0.0 | (1,[0],[1.0]) |
| 4 | Low | v6 | 0.0 | (1,[0],[1.0]) |
| 5 | Medium | v4 | 1.0 | (1,[],[]) |
| 6 | High | v6 | 0.0 | (1,[0],[1.0]) |
| 7 | High | v6 | 0.0 | (1,[0],[1.0]) |
| 8 | Very-High | v4 | 1.0 | (1,[],[]) |
| 9 | Very-High | v6 | 0.0 | (1,[0],[1.0]) |
+-----+-----+-----+-----+

```

How about applying encoding to multiple columns at the same time:

```

>>> indexers = [StringIndexer(inputCol=column, outputCol=column+"_index").fit(df) for column in
list(set(df.columns)-set(['id']))]

>>> from pyspark.ml import Pipeline
>>> pipeline = Pipeline(stages=indexers)
>>> df_indexed = pipeline.fit(df).transform(df)
>>> df_indexed.show()
+-----+-----+-----+-----+
| id | safety_level | engine_type | safety_level_index | engine_type_index |
+-----+-----+-----+-----+
| 1 | Very-Low | v4 | 3.0 | 1.0 |
| 2 | Very-Low | v6 | 3.0 | 0.0 |
| 3 | Low | v6 | 1.0 | 0.0 |
| 4 | Low | v6 | 1.0 | 0.0 |
| 5 | Medium | v4 | 4.0 | 1.0 |
| 6 | High | v6 | 0.0 | 0.0 |
| 7 | High | v6 | 0.0 | 0.0 |
| 8 | Very-High | v4 | 2.0 | 1.0 |
| 9 | Very-High | v6 | 2.0 | 0.0 |
+-----+-----+-----+-----+

>>> encoder = OneHotEncoder(
...     inputCols=[indexer.getOutputCol() for indexer in indexers],
...     outputCols=[
...         "{0}_encoded".format(indexer.getOutputCol()) for indexer in indexers]
... )
>>>

>>> from pyspark.ml.feature import VectorAssembler

```

```

>>> assembler = VectorAssembler(
...     inputCols=encoder.getOutputCols(),
...     outputCol="features"
... )
>>>
>>> pipeline = Pipeline(stages=indexers + [encoder, assembler])
>>>
>>> pipeline.fit(df).transform(df).show()
+-----+-----+-----+-----+-----+
| id|safety_level|engine_type|safety_level_index|engine_type_index|safety_level_index_encoded|engine_type_index_encoded|
+-----+-----+-----+-----+-----+
|  1|Very-Low|      v4|        3.0|        1.0|          (4,[3],[1.0])|
|  1,[[,[]])|(5,[3],[1.0])|
|  2|Very-Low|      v6|        3.0|        0.0|          (4,[3],[1.0])|
|  1,[0],[1.0])|(5,[3,4],[1.0,1.0])|
|  3|Low|      v6|        1.0|        0.0|          (4,[1],[1.0])|
|  1,[0],[1.0])|(5,[1,4],[1.0,1.0])|
|  4|Low|      v6|        1.0|        0.0|          (4,[1],[1.0])|
|  1,[0],[1.0])|(5,[1,4],[1.0,1.0])|
|  5|Medium|      v4|        4.0|        1.0|          (4,[],[])
|  1,[[,[]))|(5,[],[])
|  6|High|      v6|        0.0|        0.0|          (4,[0],[1.0])|
|  1,[0],[1.0])|(5,[0,4],[1.0,1.0])|
|  7|High|      v6|        0.0|        0.0|          (4,[0],[1.0])|
|  1,[0],[1.0])|(5,[0,4],[1.0,1.0])|
|  8|Very-High|    v4|        2.0|        1.0|          (4,[2],[1.0])|
|  1,[[,[]))|(5,[2],[1.0])|
|  9|Very-High|    v6|        2.0|        0.0|          (4,[2],[1.0])|
|  1,[0],[1.0])|(5,[2,4],[1.0,1.0])|
+-----+-----+-----+-----+

```

There is another way to do all of the data transformations: we may use `Pipeline` to simplify the transformations processes:

First create required stages:

```

>>> safety_level_indexer = StringIndexer(inputCol="safety_level", outputCol="safety_level_ndex")
>>> engine_type_indexer = StringIndexer(inputCol="engine_type", outputCol="engine_type_index")
>>> onehotencoder_safety_level = OneHotEncoder(inputCol="safety_level_ndex",
outputCol="safety_level_vector")
>>> onehotencoder_engine_type = OneHotEncoder(inputCol="engine_type_index",
outputCol="engine_type_vector")

```

Then, create a `Pipeline` and pass all defined stages:

```

>>> pipeline = Pipeline(stages=[safety_level_indexer,
...                             engine_type_indexer,
...                             onehotencoder_safety_level,
...                             onehotencoder_engine_type
...                           ])
>>>
>>> df_transformed = pipeline.fit(df).transform(df)

```

```
>>> df_transformed.show(truncate=False)
+-----+-----+-----+-----+-----+-----+-----+
| id | safety_level | engine_type | safety_level_ndex | engine_type_index | safety_level_vector | engine_type_vector |
+-----+-----+-----+-----+-----+-----+-----+
| 1 | Very-Low | v4 | 3.0 | 1.0 | [4,[3],[1.0]] | [(1,[],[])] |
| 2 | Very-Low | v6 | 3.0 | 0.0 | [4,[3],[1.0]] | [(1,[0],[1.0])] |
| 3 | Low | v6 | 1.0 | 0.0 | [4,[1],[1.0]] | [(1,[0],[1.0])] |
| 4 | Low | v6 | 1.0 | 0.0 | [4,[1],[1.0]] | [(1,[0],[1.0])] |
| 5 | Medium | v4 | 4.0 | 1.0 | [4,[],[]] | [(1,[],[])] |
| 6 | High | v6 | 0.0 | 0.0 | [4,[0],[1.0]] | [(1,[0],[1.0])] |
| 7 | High | v6 | 0.0 | 0.0 | [4,[0],[1.0]] | [(1,[0],[1.0])] |
| 8 | Very-High | v4 | 2.0 | 1.0 | [4,[2],[1.0]] | [(1,[],[])] |
| 9 | Very-High | v6 | 2.0 | 0.0 | [4,[2],[1.0]] | [(1,[0],[1.0])] |
+-----+-----+-----+-----+-----+-----+-----+
```

TF-IDF

TF-IDF is an abbreviation for Term Frequency- Inverse Document Frequency. TF-IDF is a measure of originality of a word — sometimes called a term — by the number of times a word appears in a document with the number of documents the word appears in. TF-IDF technique is used analyzing documents in natural language processing (such as search engines and feature engineering). TF-IDF is a feature vectorization method used in text mining to reflect the importance of a term to a document in the corpus (set of documents).

Term frequency $TF(t,d)$ is the number of times that term t appears in document d , while document frequency $DF(t, D)$ is the number of documents that contains term t . If a term (such as “of”, “the”, and “as”) appears very often across the corpus, it means it does not carry special information about a particular document — usually these kind of words may be dropped from the text analysis. Before we go deeper on TF-IDF transformation, let’s define basic definitions in Table 6-5.

T

a

b

l

e

6

$-$

5

$.$

T

F

$-$

I

D

F

N

o

t

a

t

i

o

n

s

Notation	Description
t	term
d	document
D	corpus (set of finite documents)
$ D $	the number of the documents in corpus
$TF(t, d)$	Term Frequency: the number of times that term t appears in document d
$DF(t, D)$	Document Frequency: the number of documents that contains term t
$IDF(t, D)$	Inverse Document Frequency: is a numerical measure of how much information a term provides

Inverse document frequency, IDF, is a numerical measure of how much information a term provides:

$$\text{IDF}(t, D) = \log \left(\frac{|\mathcal{D}| + 1}{|\mathcal{D}| - \text{DF}(t, D) + 1} \right)$$

$$\text{IDF}(t, D) = \log \left(\frac{|\mathcal{D}| + 1}{|\mathcal{D}| - \text{DF}(t, D) + 1} \right)$$

where $|\mathcal{D}|$ denotes the total number of documents in the corpus.

Let's say N is the number of documents in a corpus, since logarithm is used, if a term appears in all documents, its IDF value becomes 0. This is what happens a term appears in all documents:

$$\text{IDF}(t, D) = \log \left(\frac{(N+1)}{(N+1)} \right) = \log(1) = 0$$

Note that a smoothing term is applied to avoid dividing by zero for terms outside the corpus. The TF-IDF measure is simply the product of TF and IDF:

TF-IDF is defined as:

$\text{TF}(t, d)$ = the number of times that term t appears in document d

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

where

- t denotes the term(s)
- d denotes a document
- D denotes the corpus

$$\text{TF}_{ij} = \frac{n_{ij}}{\sum_k n_{kj}}$$

$$\text{IDF}_i = \log \left(\frac{|\mathcal{D}|}{|\{d : t_i \in d\}|} \right)$$

Before, providing TF-IDF in Spark, I want to show the values for TF-IDF in a very small settings with 2 documents (corpus size is 2 and $D = \{\text{doc1}, \text{doc2}\}$).

```
documents = spark.createDataFrame([
    ("doc1", "Ada Ada Spark Spark Spark"),
    ("doc2", "Ada SQL")],["id", "document"])

TF(Ada, doc1) = 2
TF(Spark, doc1) = 3
TF(Ada, doc2) = 1
TF(SQL, doc2) = 1

DF(Ada, D) = 2
DF(Spark, D) = 1
DF(SQL, D) = 1
```

Note that logarithm base is e for all calculations.

```

IDF(Ada, D) = log ( (|D|+1) / (DF(t,D)+1) )
              = log ( (2+1) / (DF(Ada, D)+1) )
              = log ( 3 / (2+1)) = log(1)
              = 0.00

IDF(Spark, D) = log ( (|D|+1) / (DF(t,D)+1) )
              = log ( (2+1) / (DF(Spark, D)+1) )
              = log ( 3 / (1+1) )
              = log (1.5)
              = 0.40546510811

TF-IDF(Ada, doc1, D) = TF(Ada, doc1) x IDF(Ada, D)
                      = 2 x 0.0
                      = 0.0

TF-IDF(Spark, doc1, D) = TF(Spark, doc1) x IDF(Spark, D)
                      = 3 x 0.40546510811
                      = 1.21639532433

```

for calculations: see [TF-IDF, Feature Extraction](#).

In Spark, `HashingTF` and `CountVectorizer` are the two algorithms which used to generate term frequency vectors. The following example show how to create required transformations:

```

>>> from pyspark.ml.feature import HashingTF, IDF, Tokenizer

>>> sentences = spark.createDataFrame([
...     (0.0, "Hi I heard about Spark and Java"),
...     (0.0, "I wish Java could use case classes"),
...     (1.0, "fox jumped over fence"),
...     (1.0, "red fox jumped over")
... ], ["label", "text"])

>>>
>>> tokenizer = Tokenizer(inputCol="text", outputCol="words")
>>> words_data = tokenizer.transform(sentences)
>>> words_data.show(truncate=False)
+-----+-----+
|label|text           |words          |
+-----+-----+
|0.0 |Hi I heard about Spark and Java|[hi, i, heard, about, spark, and, java]|
|0.0 |I wish Java could use case classes|[i, wish, java, could, use, case, classes]|
|1.0 |fox jumped over fence|[fox, jumped, over, fence]|
|1.0 |red fox jumped over|[red, fox, jumped, over]|
+-----+-----+

```

Next we create raw features:

```

>>> hashingTF = HashingTF(inputCol="words", outputCol="raw_features", numFeatures=16)
>>> featurized_data = hashingTF.transform(words_data)
>>> featurized_data.show(truncate=False)
+-----+-----+-----+
|label|text           |words          |raw_features |
+-----+-----+-----+
|0.0 |Hi I heard about Spark and Java|[hi, i, heard, about, spark, and, java]|([16,[0,1,6,11,12,15],|

```

```

[1.0,1.0,1.0,1.0,2.0,1.0]) |
|0.0 | I wish Java could use case classes|[i, wish, java, could, use, case, classes]|(16,
[0,6,11,12,13,15],[1.0,1.0,1.0,1.0,1.0,2.0])|
|1.0 | fox jumped over fence          |[fox, jumped, over, fence]           |(16,[0,1,6,8],
[1.0,1.0,1.0,1.0])          |
|1.0 | red fox jumped over          |[red, fox, jumped, over]           |(16,[1,4,6,8],
[1.0,1.0,1.0,1.0])          |
+-----+-----+-----+
-----+

```

The we apply IDF transformation:

```

>>> idf = IDF(inputCol="raw_features", outputCol="features")
>>> idf_model = idf.fit(featurized_data)
>>> rescaled_data = idf_model.transform(featurized_data)
>>> rescaled_data.show(truncate=False)
+-----+-----+-----+
-----+
-----+
-----+
|label|text              |words                |raw_features
|features
|
+-----+-----+-----+
-----+
-----+
|0.0 |Hi I heard about Spark and Java |[hi, i, heard, about, spark, and, java] |(16,[0,1,6,11,12,15],
[1.0,1.0,1.0,2.0,1.0]) |(16,[0,1,6,11,12,15],
[0.22314355131420976,0.22314355131420976,0.0,0.5108256237659907,1.0216512475319814,0.5108256237659907])|
|0.0 | I wish Java could use case classes|[i, wish, java, could, use, case, classes]|(16,
[0,6,11,12,13,15],[1.0,1.0,1.0,1.0,2.0])|(16,[0,6,11,12,13,15],
[0.22314355131420976,0.0,0.5108256237659907,0.5108256237659907,0.9162907318741551,1.0216512475319814])|
|1.0 | fox jumped over fence          |[fox, jumped, over, fence]           |(16,[0,1,6,8],
[1.0,1.0,1.0,1.0])          |(16,[0,1,6,8],
[0.22314355131420976,0.22314355131420976,0.0,0.5108256237659907])
|
|1.0 | red fox jumped over          |[red, fox, jumped, over]           |(16,[1,4,6,8],
[1.0,1.0,1.0,1.0])          |(16,[1,4,6,8],
[0.22314355131420976,0.9162907318741551,0.0,0.5108256237659907])
|
+-----+-----+-----+
-----+
-----+
-----+
>>> rescaled_data.select("label", "features").show(truncate=False)
+-----+-----+
-----+
-----+
|label|features
|
+-----+-----+
-----+
|0.0 |(16,[0,1,6,11,12,15],
[0.22314355131420976,0.22314355131420976,0.0,0.5108256237659907,1.0216512475319814,0.5108256237659907])|
|0.0 |(16,[0,6,11,12,13,15],
[0.22314355131420976,0.0,0.5108256237659907,0.5108256237659907,0.9162907318741551,1.0216512475319814])|
|1.0 |(16,[0,1,6,8],[0.22314355131420976,0.22314355131420976,0.0,0.5108256237659907])
|
|1.0 |(16,[1,4,6,8],[0.22314355131420976,0.9162907318741551,0.0,0.5108256237659907])
|

```

```
+-----+-----+
-----+
```

The next example shows how to do TF-IDF with using `CountVectorizer`, which extracts a vocabulary from document collections and generates a `CountVectorizerModel`. To understand `CountVectorizer`, let's look at a simple example: let each row of a DataFrame represent a document:

```
>>> df = spark.createDataFrame(
...     [(0, ["a", "b", "c"]), (1, ["a", "b", "b", "c", "a"])],
...     ["label", "raw"]
... )
>>> df.show()
+-----+-----+
|label|      raw|
+-----+-----+
|    0| [a, b, c]|
|    1|[a, b, b, c, a]|
+-----+-----+


>>> from pyspark.ml.feature import CountVectorizer
>>> cv = CountVectorizer().setInputCol("raw").setOutputCol("features")
>>> model = cv.fit(df)
>>> transformed = model.transform(df)
>>> transformed.show(truncate=False)
+-----+-----+
|label|raw      |features          |
+-----+-----+
|0    |[a, b, c] | (3,[0,1,2],[1.0,1.0,1.0])|
|1    |[a, b, b, c, a]| (3,[0,1,2],[2.0,2.0,1.0])|
+-----+-----+
```

Note that in `vectors` column, for example in second row,

- 3 : vector length
- [0, 1, 2] : vector indices (`index(a)=0`, `index(b)=1`, `index(c)=2`)
- [2.0,2.0,1.0] : vector values

`HashingTF` converts documents to vectors of fixed size.

```
>>> hashing_TF = HashingTF(inputCol="raw", outputCol="features", numFeatures=128)
>>> result = hashing_TF.transform(df)
>>> result.show(truncate=False)
+-----+-----+
|label|raw      |features          |
+-----+-----+
|0    |[a, b, c] |(128,[40,99,117],[1.0,1.0,1.0])|
|1    |[a, b, b, c, a]|(128,[40,99,117],[1.0,2.0,2.0])|
+-----+-----+
```

Note that the size of the vector generated through `CountVectorizer` depends on the training corpus and the document whereas the one generated through `HashingTF` has fixed size (we set it to 128). By using `CountVectorizer`, each raw feature is mapped to an index, but `HashingTF` might suffer from potential hash collision — two or more terms may be mapped to same index — in order to avoid hash collision we can increase the target feature dimension.

FeatureHasher

Feature hashing projects a set of categorical or numerical features into a feature vector of specified dimension (typically substantially smaller than that of the original feature space). This is done using the [hashing trick](#) to map features to indices in the feature vector.

```
>>> from pyspark.ml.feature import FeatureHasher
>>> df = spark.createDataFrame([
...     (2.1, True, "1", "fox"),
...     (2.1, False, "2", "gray"),
...     (3.3, False, "2", "red"),
...     (4.4, True, "4", "fox")
... ], ["number", "boolean", "string_number", "string"])
>>>
>>> df
DataFrame[number: double, boolean: boolean, string_number: string, string: string]

input_columns = ["number", "boolean", "string_number", "string"]

>>> featurized = hasher.transform(df)
>>> featurized.show(truncate=False)
+-----+-----+-----+-----+
|number|boolean|string_number|string|features
+-----+-----+-----+-----+
|2.1   |true    |1           |fox   |(256,[22,40,71,156],[1.0,1.0,2.1,1.0]) |
|2.1   |false   |2           |gray  |(256,[71,91,109,130],[2.1,1.0,1.0,1.0])|
|3.3   |false   |2           |red   |(256,[71,91,130,205],[3.3,1.0,1.0,1.0])|
|4.4   |true    |4           |fox   |(256,[40,71,84,156],[1.0,4.4,1.0,1.0]) |
+-----+-----+-----+-----+
```

SQL Transformer

Spark's `SQLTransformer` implements the transformations which are defined by SQL statement. Rather than registering your DataFrame as a table and then querying the table, you can directly apply the SQL transformation to your data represented as a DataFrame. Currently, `SQLTransformer` has a limited functionality and can be applied to a single DataFrame as `_THIS_`, which represents the underlying table of the input dataset.

`SQLTransformer` supports statements like:

```
SELECT salary, salary * 0.06 AS bonus
  FROM _THIS_
 WHERE salary > 10000
```

```
SELECT dept, location, SUM(salary) AS sum_of_salary
FROM __THIS__
GROUP BY dept, location
```

The following example shows how to use `SQLTransformer`:

```
>>> from pyspark.ml.feature import SQLTransformer
>>> df = spark.createDataFrame([
...     (10, "d1", 27000),
...     (20, "d1", 29000),
...     (40, "d2", 31000),
...     (50, "d2", 39000)], ["id", "dept", "salary"])

>>>
>>> df.show()
+---+---+---+
| id|dept|salary|
+---+---+---+
| 10| d1| 27000|
| 20| d1| 29000|
| 40| d2| 31000|
| 50| d2| 39000|
+---+---+---+

>>> sql_query = "SELECT dept, SUM(salary) AS sum_of_salary FROM __THIS__ GROUP BY dept"
>>> sqlTrans = SQLTransformer(statement=sql_query)
>>> sqlTrans.transform(df).show()
+---+---+
|dept|sum_of_salary|
+---+---+
| d2|      70000|
| d1|      56000|
+---+---+
```

References of Feature Engineering

Below is a list of web sites, which might help you for your feature engineering:

- What is feature engineering
- Data Manipulation: Features
- Representation: Feature Engineering
- Want to Build Machine Learning Pipelines? A Quick Introduction using PySpark
- TF-IDF, Term Frequency-Inverse Document Frequency

Chapter 7. Graph Algorithms

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 7th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

This chapter introduces Spark’s powerful graph API and provide the following examples by using the GraphFrames API:

- Introduces GraphFrames API in PySpark
- How to Use GraphFrames
- Shows how to build graphs and query them
- Provides basic graph algorithms in PySpark
- Shows how to use “motif finding” for identifying graph patterns
- PageRank Algorithm
- Flight Data Analysis

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 7](#).

Introduction to Graphs

Graphs are data structures (a data structure is a specific way of organizing data in a computer program so that it can be used effectively) to visually illustrate relationships in the data. There are some type of data (such as social networks data: FaceBook, Twitter), which can be best expressed by Graphs. For example, the following graph (Figure 7.1) represents a small set of airports (by vertices — identified by the airport codes such as SJC, LAX, ...) and shows the relationship between originating airports and their destinations (by edges).

In a nutshell, a Graph is a data structure consisting of nodes (or vertices) and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph. Graphs are used to solve many real-life problems. Graphs are used to represent networks.

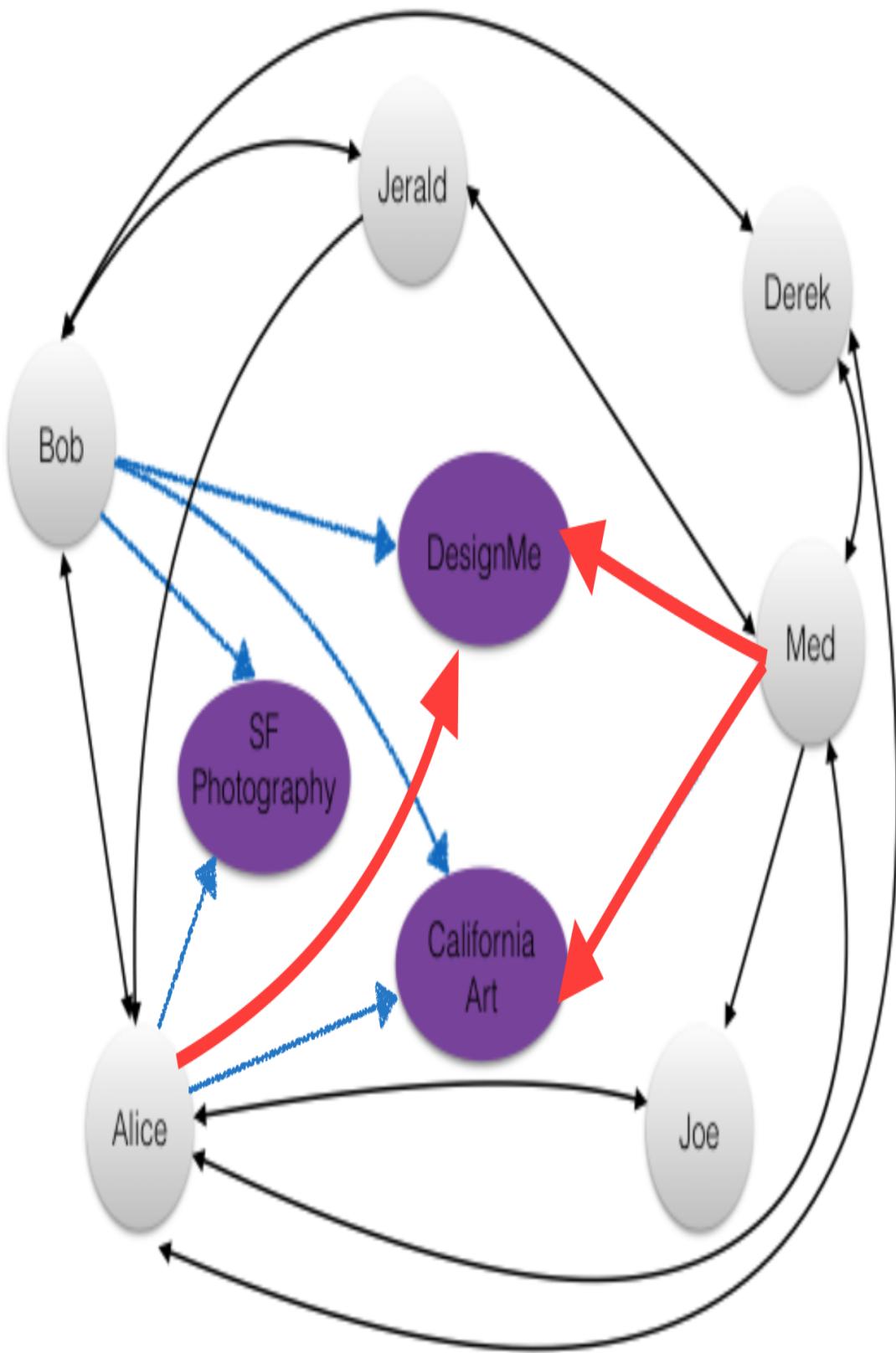


Figure 7-1. Directed Graph Example

Graphs and matrices are used by social network analysts to represent and analyze information about patterns of ties among social actors (users, friends, and “friends of friends”) and objects (such as products, stories, genes, etc.). In network analysis, data is usually modeled as a graph or set of graphs. A graph is a data structure that has a finite set of vertices (also called nodes), together with a finite set of lines (a connection from one node to another one), called edges, that join some or all of these nodes.

The following (Figure 7.2) is an undirected graph with 6 vertices (also called nodes) labeled as {A, B, C, D, E, F} and the connectors between nodes are called edges. In this example, the nodes might represent cities and edges might represent the distance between the cities. By definition, an **undirected graph** is graph with a set of vertices that are connected together, where all the edges are bidirectional.

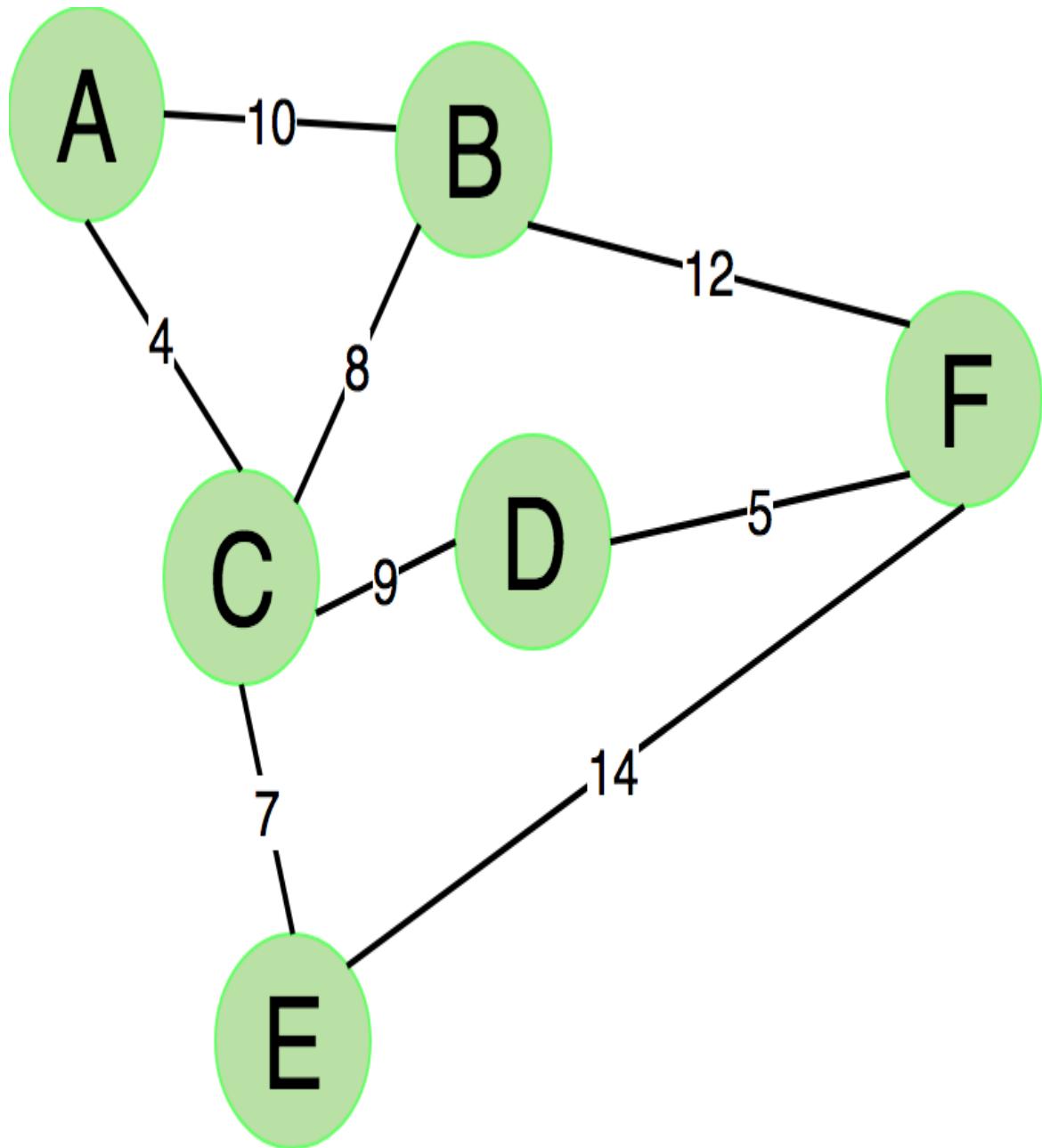


Figure 7-2. Undirected Graph Example

Informally, a graph is a pair (V, E) , where

- V is a set of nodes, called vertices
- E is a collection of pairs of vertices, called edges
- V (vertices) and E (edges) are positions and store elements

How do we identify nodes and edges? In general, each node is identified by a unique identifier and a set of associated attributes. An edge is identified by two node identifiers (source and target nodes) and a set of associated attributes. A path represents a sequence of edges between the two vertices

Example of a graph

- A vertex represents an airport and stores the three-letter airport code and other vital information (city, state, ...)
- An edge represents a flight route between two airports and stores the mileage of the route

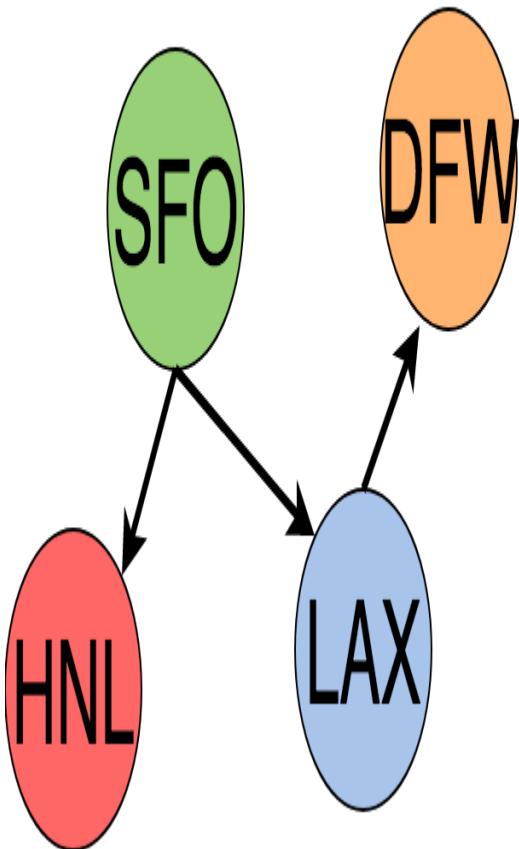
An edge can be directed or undirected:

- Directed edge
 - ordered pair of vertices (u, v)
 - first vertex u is the source
 - second vertex v is the destination
- Undirected edge
 - unordered pair of vertices (u, v)

A graph can be directed or undirected:

- Directed graph (see left image on Figure 7.3)
 - all the edges are directed
- Undirected graph (see right image on Figure 7.3)
 - all the edges are undirected

Directed Graph



Undirected Graph

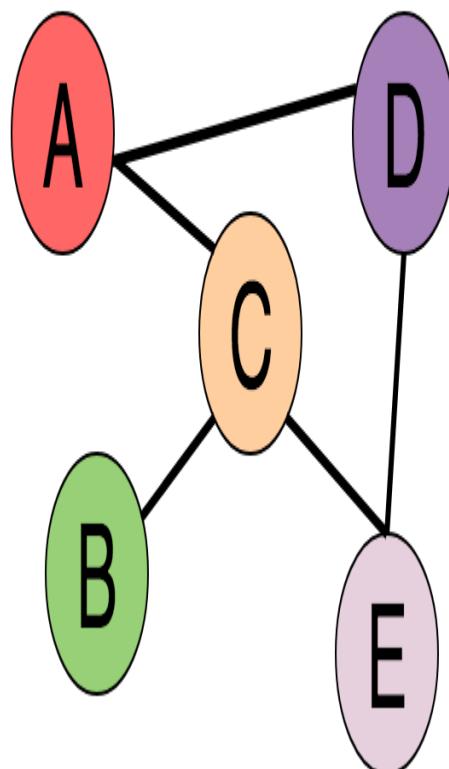


Figure 7-3. Directed and Undirected Graph Example

To convert a directed graph to an undirected graph, you add an additional edge for every directed edge. Therefore if there is a directed edge as (u, v) , then you add an edge as (v, u) .

To understand graphs in detail, refer to [Graph data structure and algorithms](#).

Spark's Graph API

Spark offers two distinct and powerful API for implementing graph algorithms such as PageRank, Shortest Path Problem, Connected Components, and triangle counting. Spark's graph API are: **GraphX** and **GraphFrames**. The **GraphX** is part of Spark's base code and is built by using

RDDs (low level API), while **GraphFrames** (as an open-source external library) are built by DataFrames (high level API).

Spark provides two distinct APIs for supporting graphs:

- **GraphX API** (based on RDDs)
 - Available only for Scala and Java (not available for Python)
 - General-purpose graph processing library
 - Optimized for fast distributed computing
 - Library of algorithms: PageRank, Connected Components, ...
 - Based on Lower-level RDD-based API (vs. DataFrames)
- **GraphFrame** (based on DataFrames)
 - Available only for Python, Scala, and Java
 - A third-party graph processing library for Spark.
 - A graph library based on Spark's DataFrames.
 - GraphFrames provides the scalability and high performance of DataFrames, and they provide a uniform API for graph processing
 - GraphFrames provide both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames.
 - Simplify interactive graph queries
 - Support “motif-finding” for graph's structural pattern search
 - Benefit from DataFrame optimizations

- Provides powerful tools for running queries on graph algorithms.

GraphFrames and GraphX are compared in the following table.

T
 a
 b
 l
 e

7
-

I

.

G
 r
 a

p
 h

F

r

a

m

e

s

v

s

.

G
 r
 a

p
 h

X

Feature

GraphFrames

GraphX

Built on	DataFrames	RDDs
Languages	Scala, Java, Python	Scala, Java
Use Cases	Algorithms and Queries	Algorithms
Vertex/Edge Attributes	Any number of DataFrame columns	Any type (VD, ED)
Return types	GraphFrame or DataFrame	Graph<VD,ED> or RDD
Motif finding	Yes	No direct support

As of this writing to date, there is no GraphX API for Python programming language, for details, see [SPARK-3789](#) (Python bindings for GraphX). The GraphX library is only available for Java and Scala. But, we can use [GraphFrames](#) in PySpark, which provides DataFrame based graph processing, and optionally interface selected GraphX functions under the covers. GraphFrames API benefit from the scalability and high performance of DataFrames, and they provide a uniform Python API for graph processing (graph queries, algorithms, and motif finding).

Therefore we can state that: **GraphX is to RDDs as GraphFrames are to DataFrames.**

GraphFrames API

[GraphFrames](#) is a third-party package for Spark which provides DataFrame-based Graphs. It provides high-level APIs in Python, Scala, and Java. The GraphFrames package provides both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames. This extended functionality includes “motif finding”, DataFrame based serialization, and highly expressive graph queries. GraphFrames is a graph API based on Spark’s DataFrames and provides the following features:

- Supports only directed graphs: if your application requires an undirected graph, then you need to convert your graph to an

undirected one.

- Unifies graph algorithms, graph queries, and DataFrames
- Available in Python, Scala, and Java
- Powerful queries: GraphFrames allow users to phrase queries in the familiar, powerful APIs of Spark SQL and DataFrames.
- Saving and loading graphs: GraphFrames fully support DataFrame data sources, allowing writing and reading graphs using many formats like Parquet, JSON, and CSV.
- Motif finding: find structural patterns in a graph

`GraphFrame` (full name of the class is `graphframes.GraphFrame`) is the main class, which builds a graph using GraphFrames API. The `GraphFrame` class is defined as:

```
class GraphFrame {  
    def vertices: DataFrame ①  
    def edges: DataFrame ②  
    def find(pattern: String): DataFrame ③  
    def degrees(): DataFrame ④  
    def pageRank(): GraphFrame ⑤  
    def connectedComponents(): GraphFrame ⑥  
    ...  
}
```

- ❶ `vertices` is a DataFrame
- ❷ `edges` is a DataFrame
- ❸ Find a defined pattern for a given graph: motif finding: searching the graph for structural patterns
- ❹ Find degrees of vertices
- ❺ PageRank algorithm for graph

⑥ Connected Components algorithm for graph

As we can see from the `GraphFrame` class definition, the `vertices` and `edges` are Spark's DataFrames. High level architecture of GraphFrame is illustrated in Figure 7.4.

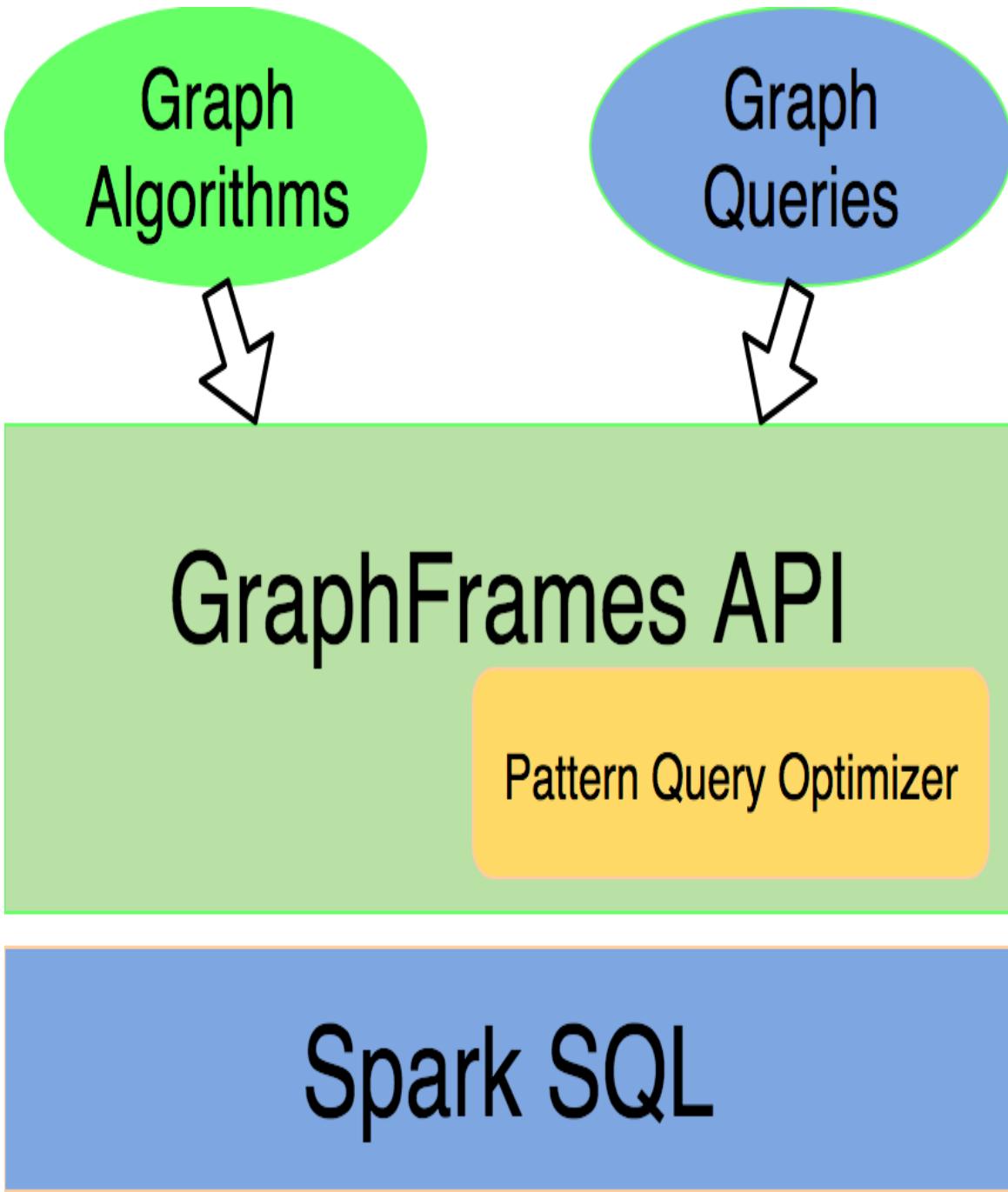


Figure 7-4. GraphFrames Architecture

How to Use GraphFrames

This section, uses GraphFrames API to build graphs. Since GraphFrames is an external package API (not as the main component of Spark API), to use it

in PySpark shell, we have to explicitly make it available. To use GraphFrames, you need to download and install GraphFrames.

GraphFrames is a collaborative effort among UC Berkeley, MIT, and Databricks. You can find the latest distribution of the GraphFrames package [here](#).

The following example shows how to run the PySpark shell with the GraphFrames package. The GraphFrames Python API documentation is defined [here](#).

We use the `--packages` argument to download the graphframes package and any dependencies automatically. The following example (output is trimmed to fit the page) shows how to run the PySpark shell with GraphFrames using a specific version (`0.6.0-spark2.3-s_2.11`) of the GraphFrames package. To use a different version, just change the last part of the `--packages` argument.

```
$ export GF="graphframes:graphframes:0.6.0-spark2.3-s_2.11"
$ ./bin/pyspark --packages $GF
...
graphframes#graphframes added as a dependency
...
Welcome to Spark version 2.4.3
SparkSession available as 'spark'.
>>>
```

Once the GraphFrames is downloaded, next, we can start using the GraphFrames API. The following example shows how to create a GraphFrame, query it, and run the PageRank algorithm. In GraphFrames API, a graph is represented as an instance of `GraphFrame(v, e)`, where `v` represents vertices (as a DataFrame) and `e` represents edges (as a DataFrame). Example is provided below.

Let's create a simple graph and then compute the PageRank algorithm for all of the nodes in a given graph. Consider the following graph:

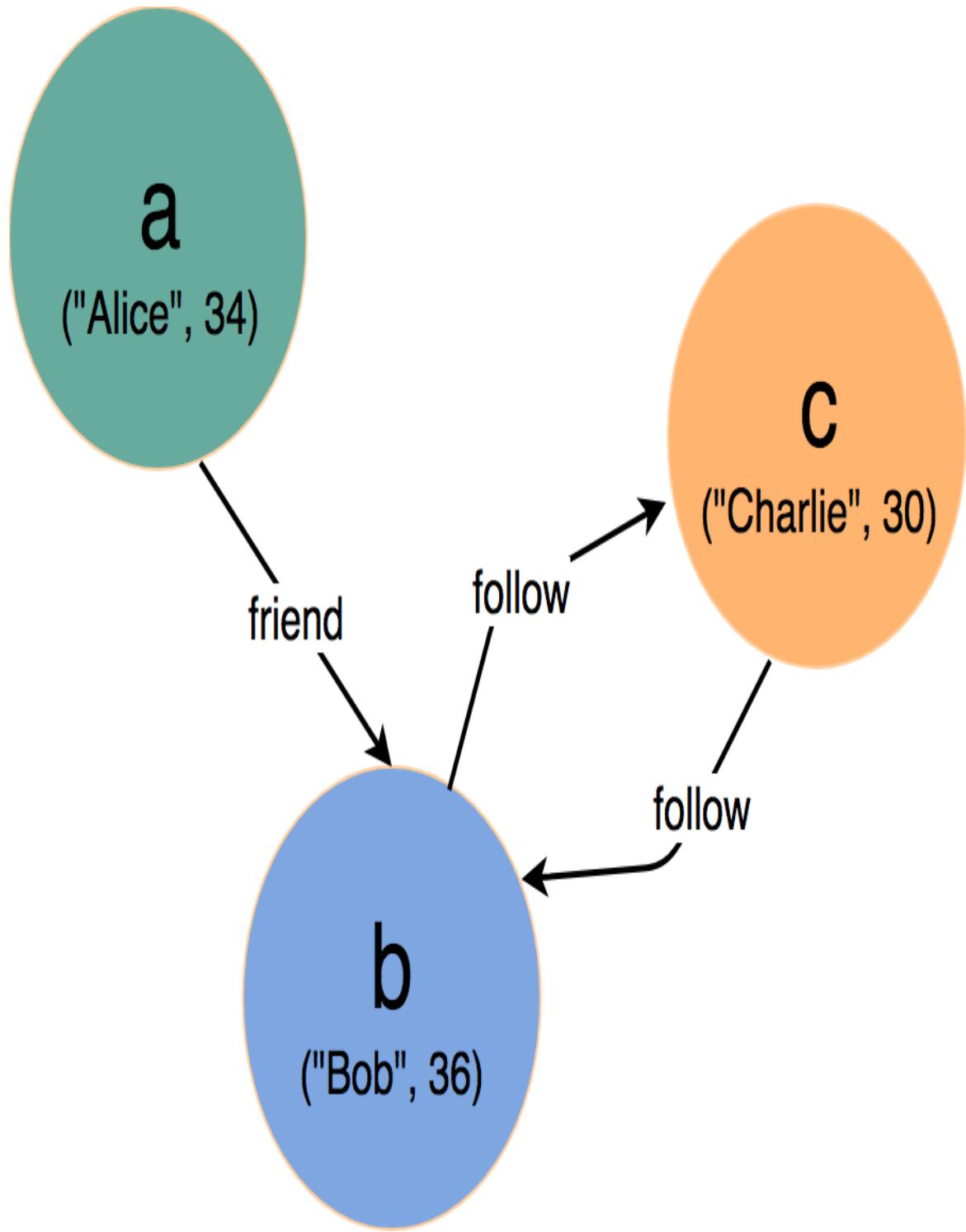


Figure 7-5. Simple Graph

In the following steps, we will build a graph using the GraphFrames API and apply some simple graph query and algorithms.

Step-1: Create Vertices

Create a Vertex DataFrame with unique ID column “id”. The “id” column is required by the GraphFrames API, which uniquely identifies all vertices for the graph to be built. You may have an additional attributes per node: in the following example, we have two additional columns “name” and “age”.

Below, we create a DataFrame with 3 columns as `DataFrame["id", "name", "age"]` of vertices:

```
# spark is an instance of a SparkSession
>>> vertices = [("a", "Alice", 34), \
    ("b", "Bob", 36), \
    ("c", "Charlie", 30)] ❶
>>> column_names = ["id", "name", "age"]
>>> v = spark.createDataFrame(vertices, column_names) ❷
>>> v.show()
+---+---+---+
| id| name|age|
+---+---+---+
| a| Alice| 34|
| b| Bob| 36|
| c| Charlie| 30|
+---+---+---+
```

❶ A Python collection representing vertices

❷ `v` represents vertices as a DataFrame

Step-2: Create Edges

Create an Edge DataFrame with “src” and “dst” columns. This step creates edges as a DataFrame as `DataFrame["src", "dst", "relationship"]`. Note that “src” and “dst” columns are required, which represent source and destination vertex IDs. You may have any additional attributes other than “src” and “dst” columns: for example, in the following example, the additional attribute is called “relationship”:

```
>>> edges = [("a", "b", "friend"), \
    ("b", "c", "follow"), \
    ("c", "b", "follow")] ❶
```

```

>>> column_names = ["src", "dst", "relationship"]
>>> e = sqlContext.createDataFrame(edges, column_names) ②
>>> e.show()
+---+---+-----+
|src|dst|relationship|
+---+---+-----+
|  a|  b|    friend|
|  b|  c|    follow|
|  c|  b|    follow|
+---+---+-----+

```

- ❶ A Python collection representing edges
- ❷ e represents edges as a DataFrame

Step-3: Create a Graph

Create a GraphFrame using GraphFrames API, a graph is built as an instance of a GraphFrame, which is a pair of vertices (as v) and edges (as e):

```

>>> from graphframes import GraphFrame ❶
>>> graph = GraphFrame(v, e) ❷
>>> graph ❸
GraphFrame(v:[id: string, name: string ... 1 more field],
            e:[src: string, dst: string ... 1 more field])

```

- ❶ Import the required class GraphFrame
- ❷ Build a graph as an instance of GraphFrame using v (vertices) and e (edges)
- ❸ Inspect the built graph

The **GraphFrame** class is defined as:

```
class graphframes.GraphFrame(v, e)
```

Description:

Represents a graph with vertices
and edges stored as DataFrames.

Parameters:

- v DataFrame holding vertex information.
Must contain a column named "id" that stores unique vertex IDs.
- e DataFrame holding edge information.
Must contain two columns "src" and "dst" storing source vertex IDs and destination vertex IDs of edges, respectively.

Step-4: Query Graph: Get in-degree of each vertex. Once the graph is built, then you can start issuing queries and applying algorithms. To get the “in degree” of nodes, you may write:

```
>>> graph.inDegrees.show()  
+-----+  
| id|inDegree|  
+-----+  
| c |      1|  
| b |      2|  
+-----+
```

The `GraphFrame.inDegrees()` return the in-degree of each vertex in the graph, returned as a DataFrame with two columns:

- `id`: the ID of the vertex
- `inDegree`: storing the in-degree of the vertex as an integer

Note that vertices with 0 in-edges are not returned in the result.

Step-5: Query Graph: Count the number of “follow” connections in the graph.

```
>>> graph.edges.filter("relationship = 'follow'").count()  
2
```

Step-6: Run PageRank algorithm (we will discuss the PageRank algorithm in this chapter), and show results.

```

>>> pageranks = graph.pageRank(resetProbability=0.01, maxIter=20)    ①
>>> pageranks.vertices.select("id", "pagerank").show()      ②
+-----+
| id |      pagerank |
+-----+
| b | 1.0905890109440908 |
| a |      0.01 |
| c | 1.8994109890559092 |
+-----+

```

- ① Run the PageRank algorithm for the given graph for 20 iterations
- ② Show the PageRank values for each node of the given graph

GraphFrames Attributes and Structures

GraphFrame's functions can be categorized into two categories:

- Attributes
- Structures

The complete list of all functions (called as GraphOps) can be found in [GraphFrames API](#). But note that that all of those functions can not be used with DataFrames. When you know how to work with DataFrames, you can also apply `sort()`, `groupBy()` or `filter()` on output of each of the functions and get more summaries. If you want to learn more about DataFrames, I do recommend you to have a look at the chapter 7 of this book.

Graph Attributes

Using GraphFrames, you can get lot of details about the graph such as list and number of edges, nodes, neighbors per nodes, in degree and out degree score per each node. The basic graph functions that can be used in PySpark are the following:

- vertices
- edges

- inDegrees
- outDegrees
- degrees

For example, if a `graph` is an instance of a `GraphFrame`, then you can get vertices and edges as DataFrames:

```
vertices_as_dataframes = graph.vertices    ①
edges_as_dataframes = graph.edges    ②
```

- ① The `Graphframe.vertices` attribute returns vertices as a DataFrame
- ② The `Graphframe.edges` attribute returns edges as a DataFrame

Graph's Algorithms

`GraphFrame` API provides a set of graph algorithms such as finding a particular pattern (using “motif finding”) out of the graph, which is usually an expensive operation. But since Spark uses MapReduce and distributed algorithms, these expensive operations run much faster but still consider these will be time consuming processes. The followings algorithms are supported:

- Motif finding
- Subgraphs
- Breadth-first search (BFS)
- Connected components
- Strongly connected components
- Label Propagation
- PageRank
- Shortest paths

- Triangle count

Finding Triangles

This section provides efficient solutions to find, count, and list all triangles for a given graph or set of graphs using Spark API GraphFrame (based on DataFrames).

Before we define some metrics using the count of triangles, we need to define a triad and a triangle. Let $T = (a, b, c)$ be a set of three distinct nodes in a graph identified by G , then T is a triad if two of those nodes are connected ($\{(a, b), (a, c)\}$) and it is a triangle if all three nodes are connected ($\{(a, b), (a, c), (b, c)\}$).

In graph analysis, there are three important metrics:

- Global clustering coefficient
- Transitivity ratio, defined as:

$$T(G) = 3 \times (m) / (n)$$

where

m = (number of triangles on the graph)
 n = (number of connected triads of vertices)

- Local clustering coefficient

In order to calculate these three metrics in a large graph, we need to know the number of triangles in the graph. For details on these metrics, see [Clustering Coefficient](#). A graph might have hundreds of millions of nodes (e.g., users in a social network) and edges (the relationships between these users), and counting the number of triangles is very time consuming.

Triangle Counting with MapReduce

Chapter 16 of *Data Algorithms* book by Mahmoud Parsian provided two MapReduce solutions that find, count, and list all triangles for a given graph

or set of graphs. These solutions are given in Java, MapReduce, and Spark.

Triangle Counting with GraphFrames

The `GraphFrame` package provides a convenient method for Triangle count. `GraphFrame.triangleCount()` method computes the number of triangles passing through each vertex. The following example shows how to build a graph from nodes and edges and then find the number of triangles passing through each node.

Step-1: Build a Graph

First, let's define vertices:

```
>>> # SparkSession available as 'spark'.
>>> # Display the vertex and edge DataFrames
>>> vertices = [( 'a', 'Alice', 34), \
    ('b', 'Bob', 36), \
    ('c', 'Charlie', 30), \
    ('d', 'David', 29), \
    ('e', 'Esther', 32), \
    ('f', 'Fanny', 36), \
    ('g', 'Gabby', 60)]
```

Next, define the edges between nodes:

```
>>> edges = [( 'a', 'b', 'friend'), \
    ('b', 'c', 'follow'), \
    ('c', 'b', 'follow'), \
    ('f', 'c', 'follow'), \
    ('e', 'f', 'follow'), \
    ('e', 'd', 'friend'), \
    ('d', 'a', 'friend'), \
    ('a', 'e', 'friend')]
```

Once we have vertices and edges, then we can build a graph:

```
>>> v = spark.createDataFrame(vertices, ["id", "name", "age"]) ❶
>>> e = spark.createDataFrame(edges, ["src", "dst", "relationship"]) ❷
>>> from graphframes import GraphFrame
>>> graph = GraphFrame(v, e) ❸
```

- ① “id” column is required for a vertex
- ② “src” and “dst” columns are required for an edge
- ③ graph is built as a GraphFrame object

Next, we examine the built graph, vertices and edges:

- Examine graph:

```
>>> graph
GraphFrame(v:[id: string, name: string ... 1 more field],
            e:[src: string, dst: string ... 1 more field])
```

- Examine vertices:

```
>>> graph.vertices.show()
+---+-----+---+
| id |     name|age|
+---+-----+---+
| a  | Alice  | 34 |
| b  | Bob    | 36 |
| c  | Charlie| 30 |
| d  | David  | 29 |
| e  | Esther | 32 |
| f  | Fanny  | 36 |
| g  | Gabby  | 60 |
+---+-----+---+
```

- Examine edges:

```
>>> graph.edges.show()
+---+---+-----+
|src|dst|relationship|
+---+---+-----+
| a | b | friend |
| b | c | follow |
| c | b | follow |
| f | c | follow |
| e | f | follow |
| e | d | friend |
| d | a | friend |
```

```

| a| e|      friend|
+---+-----+

```

Step-2: Count Triangles

To count triangles, we use the `GraphFrame.triangleCount()` method, which counts the number of triangles passing through each vertex in this graph.

```

>>> results = g.triangleCount()
>>> results.show()
+-----+
|count| id|    name|age|
+-----+
|   0|  g| Gabby| 60|
|   0|  f| Fanny| 36|
|   1|  e| Esther| 32|
|   1|  d| David| 29|
|   0|  c|Charlie| 30|
|   0|  b|    Bob| 36|
|   1|  a| Alice| 34|
+-----+

```

To show only vertex ID and the number of triangles passing through each vertex, we can write:

```

>>> results.select("id", "count").show()
+-----+
| id|count|
+-----+
|  g|   0|
|  f|   0|
|  e|   1|
|  d|   1|
|  c|   0|
|  b|   0|
|  a|   1|
+-----+

```

Motif Finding

Motif in graphs refer to a pattern of vertices and their relationships. For example, a motif might refer to a pattern of triangles (3 nodes, which have a very strong relationship between them). Motifs also can represent subgraphs of a graph. According to [wikipedia](#): “all networks, including biological networks, social networks, technological networks (e.g., computer networks and electrical circuits) and more, can be represented as graphs, which include a wide variety of subgraphs. One important local property of networks are so-called network motifs, which are defined as recurrent and statistically significant sub-graphs or patterns.”

The good news is that the GraphFrames API provides strong support for “motif finding”, which enable us to find structural patterns and their relationships among the graph nodes and edges. Therefore, we can state that motif finding refers to searching for structural patterns in a graph.

How do we express motifs for a graph? GraphFrame motif finding uses a simple Domain-Specific Language (DSL) for expressing structural queries. For example, the following query:

```
graph.find("(a)-[e1]->(b); (b)-[e2]->(a)")
```

will search for pairs of vertices { a, b } connected by edges in both directions (bi-directional nodes). It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. In this case, the returned columns will be "a, b, e1, e2" (where e1 represents an edge from a to b and e2 represents an edge from b to a).

In GraphFrames framework, DSL for expressing structural patterns are defined as:

- The basic unit of a pattern is an edge. An edge connects one node to another one. For example, "(a)-[e]->(b)" expresses an edge e from vertex a to vertex b. Note that vertices are denoted by parentheses (a) and (b), while edges are denoted by square brackets [e]. An anonymous edge can be denoted by [] (an

anonymous edge is an edge, which is not needed, and used in identifying interested vertices).

- A pattern is expressed as a union of edges. Edge patterns can be joined with semicolons (“;”). For example, the motif "(a)-[e1]->(b); (b)-[e2]->(c)" specifies two edges (e1 and e2) from "a to b" and "b to c".
- Within a pattern, names can be assigned to vertices and edges. For example, "(a)-[e]->(b)" has three named elements: vertices {a, b} and an edge e. These names serve two purposes:
 - The names can identify common elements among edges. For example, "(a)-[e1]->(b); (b)-[e2]->(c)" specifies that the same vertex b is the destination of edge e1 and source of edge e2.
 - The names are used as column names in the result DataFrame. If a motif contains named vertex a, then the result DataFrame will contain a column a which is a StructType with sub-fields equivalent to the schema (columns) of GraphFrame.vertices. Similarly, an edge e in a motif will produce a column e in the result DataFrame with sub-fields equivalent to the schema (columns) of GraphFrame.edges.
- It is acceptable to omit names for vertices or edges in motifs when not needed. For example, motif "(a)-[]->(b)" expresses an edge between vertices "a, b" but does not assign a name to the edge. There will be no column for the anonymous edge in the result DataFrame. Similarly, the motif "(a)-[e]->()" indicates an out-edge of vertex a but does not name the destination vertex; and "()-[e]->(b)" indicates an in-edge of vertex b but does not name the source vertex

- An edge can be negated by using an exclamation point “!” to indicate that the edge should not be present in the graph. For example,, the following motif

(a)-[]->(b); !(b)-[]->(a)

finds edges from a to b for which there is no edge from b to a (“a” follows “b”, but “b” does not follow “a”).

Triangle Counting with Motif Finding

This section will focus on triangle counting using motifs. GraphFrame is an external library for Spark, which provides DataFrame based Graphs. GraphFrame functionality includes motif finding, DataFrame based serialization, and highly expressive graph queries. GraphFrame’s motif finding enable us to find structural patterns (such as triangles) in a very easy manner by defining a motif. For example, to find triangles, it is just enough to define a triangle: let “{a, b, c}” denote 3 nodes in a graph, then we can define a motif for a triangle as:

Triangle definition by motif:
triangle: a -> b -> c -> a

This definition includes 3 vertices
(a, b, and c) such that:

a is connected to b (as an edge a -> b)
b is connected to c (as an edge b -> c)
c is connected to a (as an edge c -> a)

You can build more complex relationships involving edges and vertices using motifs. In using GraphFrame motifs, the basic unit of a pattern is an edge. For example, the following expression

(a)-[e]->(b)

expresses an edge e from vertex a to vertex b . Note that vertices are denoted by parentheses (a) and (b) , while edges are denoted by square brackets $[e]$. It is important to note that the GraphFrame uses a **directed graph** (see Figure 11.5). By definition, a **directed graph** is graph with a set of vertices that are connected together, where all the edges are directed from one vertex to another. To use an undirected graph, you need to provide two edges for connected nodes. For example, if there is an edge, which connects a to b $[a \rightarrow b]$, then you need to define an additional edge, which connects b to a $[b \rightarrow a]$.

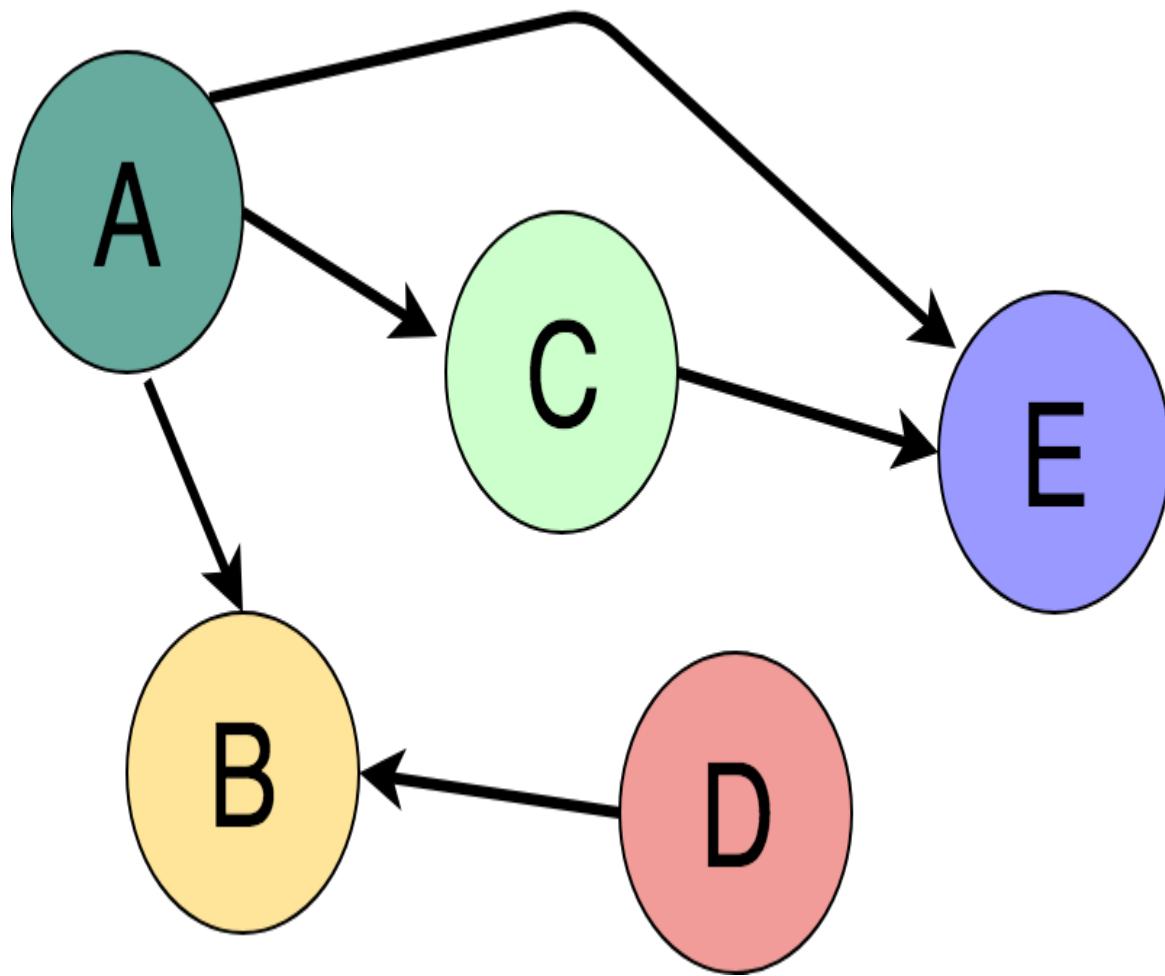


Figure 7-6. Directed Graph Example

Assume that our graph is undirected: if we have an edge $[a \rightarrow b]$, then we will have another edge as: $[b \rightarrow a]$.

To understand the concept of “motif finding”, Let g be a GraphFrame object, then we will write several motifs to find the optimal way of identifying triangles.

Trial-1

Find triangle as "a -> b -> c -> a". Since a triangle can be defined many ways, we will have duplicate output.

```

triangles = g.find("(a)-[e1]->(b);
                    (b)-[e2]->(c);
                    (c)-[e3]->(a)")
triangles.show()
+-----+-----+-----+-----+
|     a|     e1|      b|     e2|      c|     e3|
+-----+-----+-----+-----+
|[1,1]||[1,2,][2,2][2,4,][4,4][4,1,]|
|[2,2]||[2,1,][1,1][1,4,][4,4][4,2,]|
|[1,1]||[1,4,][4,4][4,2,][2,2][2,1,]|
|[4,4]||[4,1,][1,1][1,2,][2,2][2,4,]|
|[2,2]||[2,4,][4,4][4,3,][3,3][3,2,]|
|[2,2]||[2,4,][4,4][4,1,][1,1][1,2,]|
|[4,4]||[4,2,][2,2][2,3,][3,3][3,4,]|
|[4,4]||[4,2,][2,2][2,1,][1,1][1,4,]|
|[2,2]||[2,3,][3,3][3,4,][4,4][4,2,]|
|[3,3]||[3,2,][2,2][2,4,][4,4][4,3,]|
|[3,3]||[3,4,][4,4][4,2,][2,2][2,3,]|
|[4,4]||[4,3,][3,3][3,2,][2,2][2,4,]|
|[5,5]||[5,6,][6,6][6,7,][7,7][7,5,]|
|[6,6]||[6,5,][5,5][5,7,][7,7][7,6,]|
|[5,5]||[5,7,][7,7][7,6,][6,6][6,5,]|
|[7,7]||[7,5,][5,5][5,6,][6,6][6,7,]|
|[6,6]||[6,7,][7,7][7,5,][5,5][5,6,]|
|[7,7]||[7,6,][6,6][6,5,][5,5][5,7,]|
+-----+-----+-----+-----+

```

This trial finds triangles, but there is problem with output as duplicates (since our graph is undirected).

Trial-2:

Find triangle as "a -> b -> c -> a" and filter (to remove duplicate triangles) the results by: ("e1.src < e1.dst"):

```

triangles = g.find("(a)-[e1]->(b);
                    (b)-[e2]->(c);
                    (c)-[e3]->(a)")
                    .filter("e1.src < e1.dst")
triangles.show()
+-----+-----+-----+-----+
|   a|   e1|     b|   e2|     c|   e3|
+-----+-----+-----+-----+
|[1,1]||[1,2,][[2,2]][2,4,][4,4][4,1,]|
|[1,1]||[1,4,][4,4][4,2,][2,2][2,1,]|
|[2,2]||[2,4,][4,4][4,3,][3,3][3,2,]|
|[2,2]||[2,4,][4,4][4,1,][1,1][1,2,]|
|[2,2]||[2,3,][3,3][3,4,][4,4][4,2,]|
|[3,3]||[3,4,][4,4][4,2,][2,2][2,3,]|
|[5,5]||[5,6,][6,6][6,7,][7,7][7,5,]|
|[5,5]||[5,7,][7,7][7,6,][6,6][6,5,]|
|[6,6]||[6,7,][7,7][7,5,][5,5][5,6,]|
+-----+-----+-----+

```

Still the output is not the desired ones, since we have duplicates in our results.

Trial-3:

In this trial we want to make sure that "[**a** -> **b** -> **c** -> **a**]" is the same as: "[**b** -> **c** -> **a** -> **b**]" and "[**c** -> **a** -> **b** -> **c**]" by adding two additional filters. These filters enable us to uniquely identify all triangles without duplicates.

```

triangles = g.find("(a)-[e1]->(b);
                    (b)-[e2]->(c);
                    (c)-[e3]->(a)") ❶
                    .filter("e1.src < e1.dst") ❷
                    .filter("e2.src < e2.dst") ❸
triangles.show()
+-----+-----+-----+-----+
|   a|   e1|     b|   e2|     c|   e3|
+-----+-----+-----+-----+
|[1,1]||[1,2,][[2,2]][2,4,][4,4][4,1,]|
|[2,2]||[2,3,][3,3][3,4,][4,4][4,2,]|
|[5,5]||[5,6,][6,6][6,7,][7,7][7,5,]|
+-----+-----+-----+

```

- ❶ find triangles {a -> b -> c -> a}
- ❷ make sure that e1.src and e1.dst are not the same
- ❸ make sure that e2.src and e2.dst are not the same

Finding Unique Triangles with Motif

The goal of this section is to build a GraphFrame from given vertices and edges and then find “unique triangles” from the built graph.

Input

The required components for building a graph (using GraphFrame) are vertices and edges. Assume that vertices and edges are defined by two files:

- `sample_graph_vertices.txt`
- `sample_graph_edges.txt`

Let's examine these input files:

```
$ head -4 sample_graph_vertices.txt
vertex_id
0
1
2

$ head -4 sample_graph_edges.txt
edge_weight,from_id,to_id
0,5,15
1,18,8
2,6,1
```

To comply with GraphFrames API, the following clean up and filtering will be done for vertices and edges:

- rename “vertex_id” to “id”
- drop the column “edge_weight”

- rename “from_id” to “src”
- rename “to_id” to “dst”

Output

The expected output will be unique triangles from the built graph. Note that given 3 vertices {a, b, c} of a triangle, it can be represented by the following 6 representations:

```
a -> b -> c -> a
a -> c -> b -> a
b -> a -> c -> b
b -> c -> a -> b
c -> a -> b -> c
c -> b -> a -> c
```

The goal is to output only one of the representation.

Algorithm

The complete PySpark solution is presented as `unique_triangles_finder.py`. Using GraphFrames motif and DataFrames, the solution is pretty simple:

- Create a DataFrame for vertices: `vertices_df`
- Create a DataFrame for edges: `edges_df`
- Build a graph as a GraphFrame
- Apply a motif which is a triangle pattern
- Filter out duplicate triangles

Building `vertices_df` is straightforward. In building `edges_df`, to make sure that our graph is undirected, if there is a connection from “src” vertex to “dst” vertex, then we add an extra edge from “dst” to “src”. This way we will be able to find all triangles.

Now, let's focus on how to find all triangles, which might have duplicates as well:

```
graph = GraphFrame(vertices_df, edges_df)
# find all triangles, which might have duplicates
motifs = graph.find("(a)-[]->(b);
                      (b)-[]->(c);
                      (c)-[]->(a)")
print("motifs.count()", motifs.count())
42
```

Now our `motifs` represent all triangles with duplication. To remove the duplication, we can use DataFrame's powerful filtering mechanism: now remove duplicate triangles; keep only one representation of a triangle $\{a, b, c\}$ where $a > b > c$.

```
unique_triangles = motifs[(motifs.a > motifs.b) &
                           (motifs.b > motifs.c)] ❶
unique_triangles.count()
7
unique_triangles.show(truncate=False)
+-----+
| a | b | c |
+-----+
|[42]| [32] | [30]|
|[5] | [31] | [15]|
|[8] | [22] | [18]|
|[8] | [22] | [17]|
|[7] | [39] | [28]|
|[52] | [51] | [50]|
|[73] | [72] | [71]|
+-----+
```

❶ Remove duplicate triangles

Note that the `motifs.count()` was 42 (since a triangle can be represented in 6 different ways — as I depicted earlier) and `unique_triangles.count()` is 7 ($6 \times 7 = 42$)

The combination of GraphFrames and DataFrames are very powerful tool in solving graph-related problems and beyond.

Motif Finding Examples

In the preceding section, I demonstrated how to find triangles using motifs. Below, I present more examples on Motif finding.

Find Bidirectional Vertices

Using motifs, you can build more complex relationships involving graph's edges and vertices . The following example finds the pairs of vertices with edges in both directions between them. The result is a DataFrame, in which the column names are motif keys. Let `graph` be an instance of a GraphFrame, then finding bidirectional vertices can be expressed as:

```
# search for pairs of vertices with edges
# in both directions between them:
#
bidirectional = graph.find("(a)-[e1]->(b);"
                           "(b)-[e2]->(a)") ❶
```

- ❶ `bidirectional` will have columns “a”, “e1”, “b”, and “e2”

Since the result is a DataFrame, more complex queries can build on top of the motif. Let us find all the reciprocal relationships in which one person is older than 30:

```
older_than_30 = bidirectional.filter("b.age > 30 or a.age > 30")
```

Subgraphs

A subgraph is a graph whose vertices and edges are subsets of another graph. You can build subgraphs by filtering a subset of edges and vertices. For example, let `graph` be an instance of GraphFrame, then the following

subgraph only contains people who are friends and follower's age is less than the friend's age.

```
paths = graph.find("(a)-[e]-(b)")\
    .filter("e.relationship = 'follow'")\
    .filter("a.age < b.age")

# The `paths` variable contains the vertex
# information, which we can extract:
selected_edges = paths.select("e.src", "e.dst", "e.relationship")

# Construct the subgraph
sample_subgraph = GraphFrame(g.vertices, selected_edges)
```

Friend recommendation

GraphFrames “motif finding” enable us to make powerful graph queries. For example, to recommend whom to follow, we might search for triplets of users A, B, C where “A follows B” and “B follows C”, but “A does not follow C”. This can be expressed as:

```
# g is an instance of a GraphFrame
# Motif: "A -> B", "B -> C", but not "A -> C"
results = g.find("(A)-[]-(B);"
                  "(B)-[]-(C);"
                  !(A)-[]-(C)")

# Filter out loops (with DataFrame operation)
results_filtered = results.filter("A.id != C.id")

# Select recommendations for A to follow C
recommendations = results_filtered.select("A", "C")
```

Product Recommendations

Let the following motif represents a use case where a customer who bought product (p) also purchased the other two other products, (a), and (b), as well. This relationship can be expressed by the following Figure:

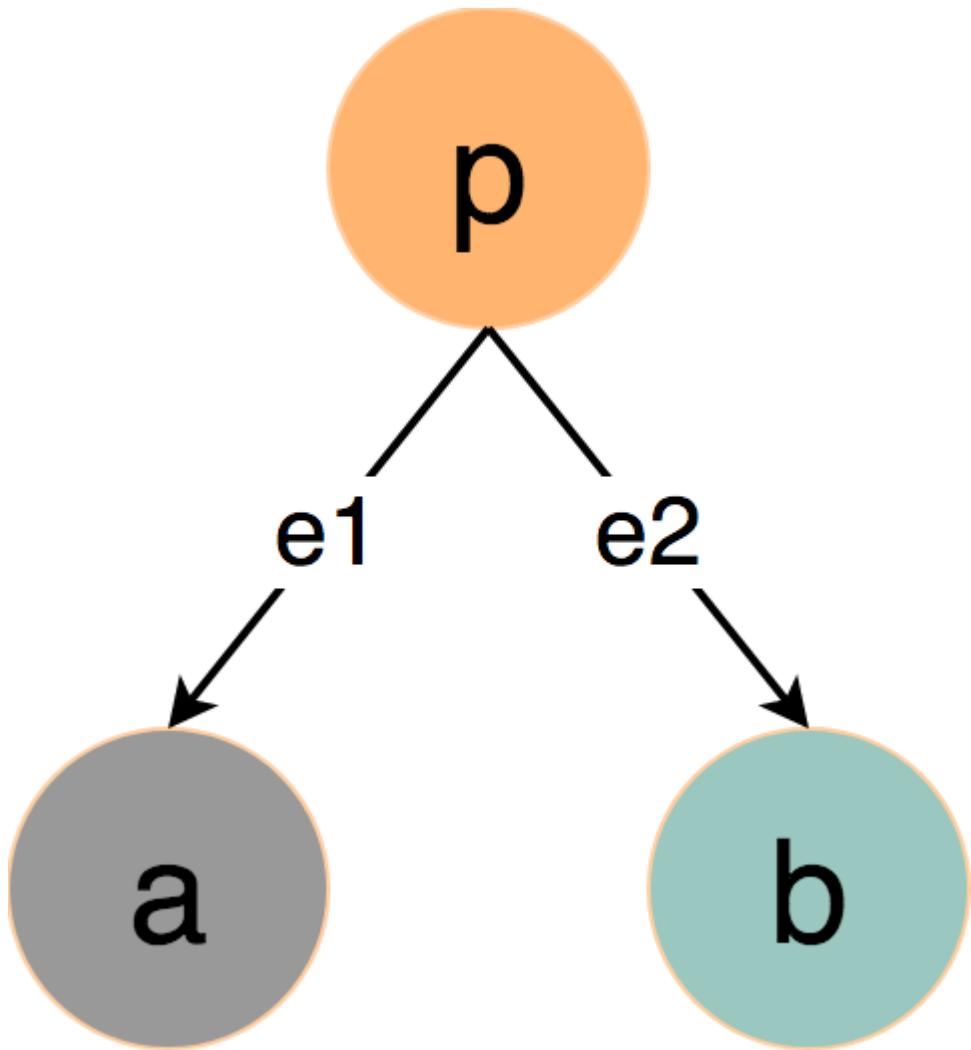


Figure 7-7. Product Recommendation

Then the following motif finding example specifies two separate edges from product p (to a and b). This is a situation where a customer buys a product (p) and then also buys either of, or both, (a) and (b). Therefore, this motif can be expressed as:

```

graph = GraphFrame(vertices, edges)
motifs = graph.find("(p)-[e1]->(a);"
                     "(p)-[e2]->(b)")
    .filter("(a != b)")
  
```

We can also apply filters to the result of motif finding; for example, we have specified the value of the vertex p as 1200 (denoting a product with “id” of 1200) in the following filter:

```
motifs.filter("p.id == 1200").show()
```

The following example shows how to find strong relationships between two products. In this example, we specify edges from “p to a” and “a to b”, and another one from “b to a”. This pattern typically represents the case in which when a customer buys a product (p), and may also buy (a) and then go on to buy (b). This can be indicative of some prioritization on the items being purchased. In addition, “motif finding” enable us to find the strongly connected products, which indicates a close relationship between products (a) and (b). Motif is expressed as:

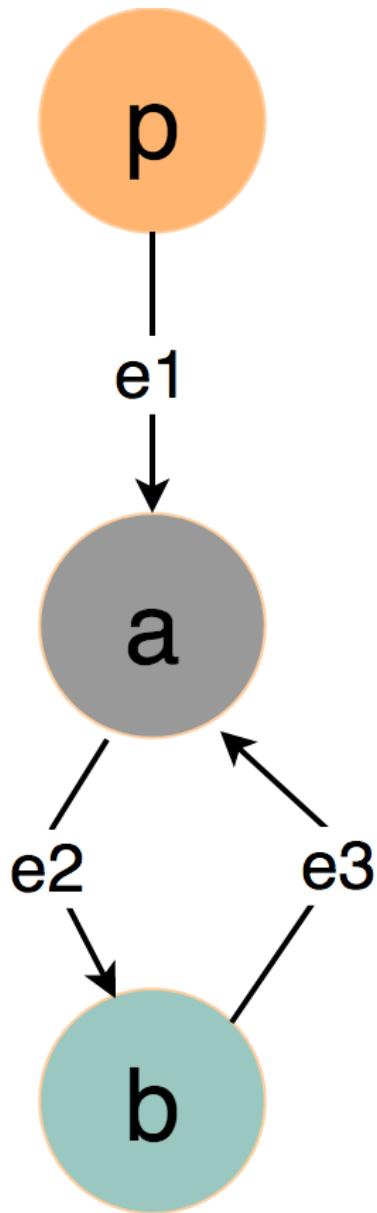


Figure 7-8. Product Relationships

The motive for strong relationships can be expressed as:

```

graph = GraphFrame(vertices, edges)
strong_motifs = graph.find("(p)-[]->(a);"
                           "(a)-[]->(b);"
                           "(b)-[]->(a)")
strong_motifs.show()
  
```

Not that in motif definition the notation [e] denotes an edge labeled as e, while [] denotes just an edge without a name.

Gene Analysis

This section presents a graph problem of Gene-to-Gene Interaction and then provides a solution using GraphFrames. Gene is a unit of heredity that is transferred from a parent to offspring and is held to determine some characteristic of the offspring. Gene relationships have been analyzed for Down Syndrome with Labeled Transition Graphs (a directed graph where vertices represent genes and an edge represents a relationship between genes). Let XAB2, ERCC8, and POLR2A denote three genes. For example, three vertices (XAB2, ERCC8, POLR2A) and two edges can be represented by the following raw data (the relationships are: “Biochemical Activity” and “Reconstituted Complex”):

```
XAB2,ERCC8,Reconstituted Complex  
XAB2,POLR2A,Affinity Capture-Western
```

One important analysis is to find motifs between specific vertices. For example, in [1] they have listed some motifs, which can help detect diseases like Down Syndrome or Alzheimers Disease:

Hedgehog signaling pathway (HSP) is illustrated by the Figure 11.8.

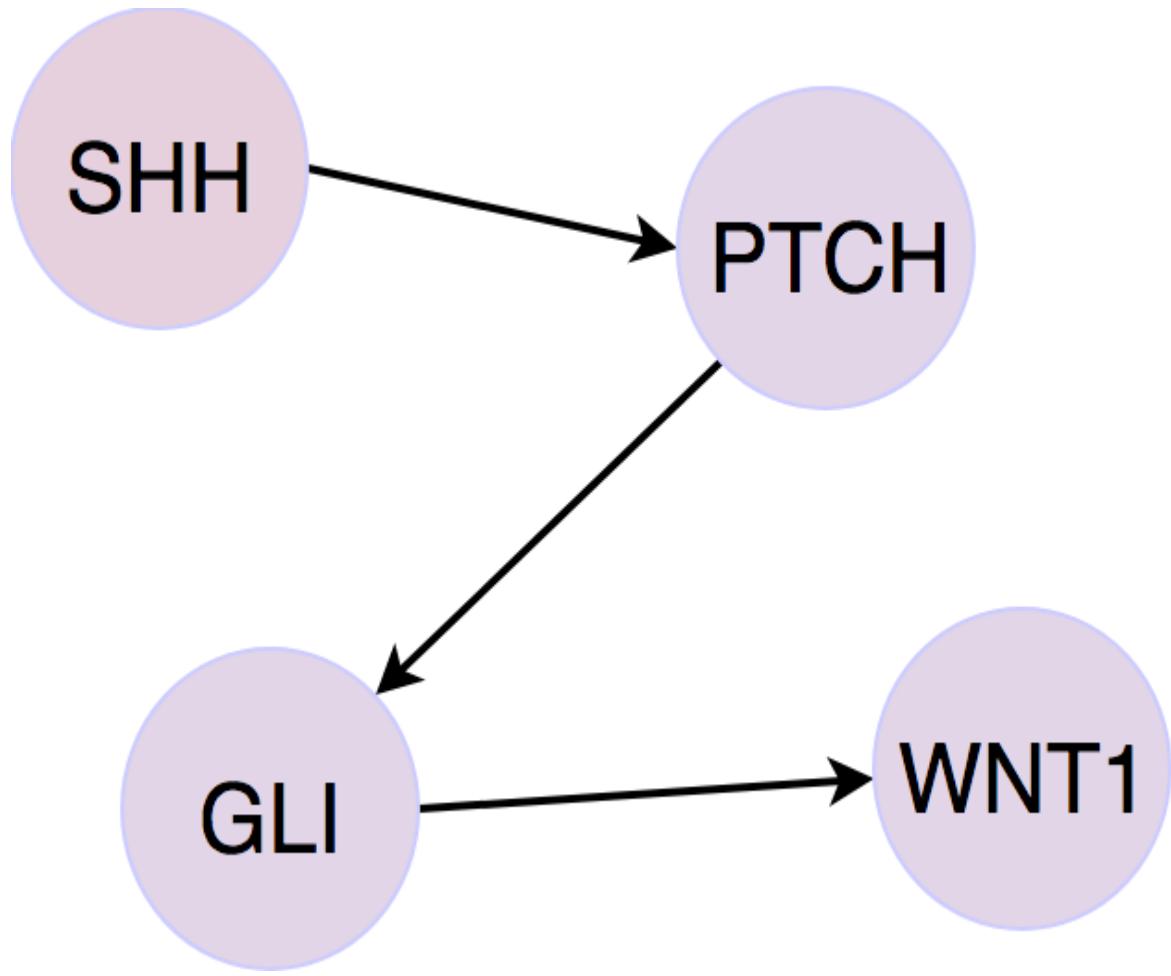


Figure 7-9. Hedgehog Signaling Pathway Relationship

Alzheimers Disease is illustrated by the Figure 11.9.

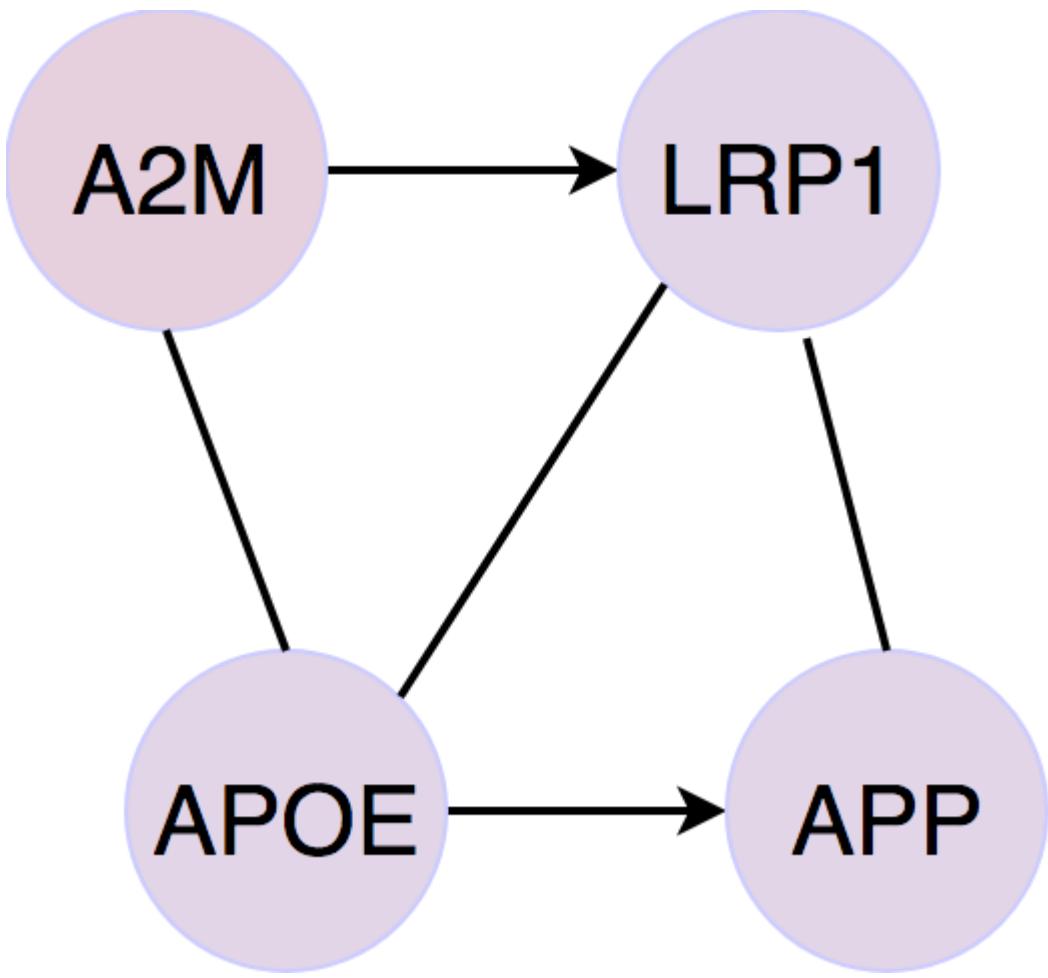


Figure 7-10. Alzheimers Disease Relationship

These patterns (illustrated by Figure 11.8 and 11.9) and relationships can be easily detected by the GraphFrame’s “motif finding” feature. Further we can find the most important genes (using the PageRank algorithm) or finding gene communities (by running Label Propagation algorithm for many iterations). We will demonstrate all of these plus more by using the GraphFrames API.

Input for Gene-to-Gene Analysis

We will use the following input format to build a graph. For this input, we have only edges, but vertices will be extracted from the same input as well.

<source-gene>,<><destination-gene>,<><type-of-relationship>

For example, four input records are listed below.

```
BRCA1,BRCA1,Biochemical Activity  
SET,TREX1,Co-purification  
SET,TREX1,Reconstituted Complex  
PLAGL1,EP300,Reconstituted Complex
```

Since we have only input for edges, we will derive the vertices from the edges.

Motif Finding for Genes

In the preceding section, I listed two structural patterns Hedgehog signaling pathway (HSP) and Alzheimers Disease. To express HSP, we write:

```
hsp = graph.find(  
    "(shh)-[e1]->(ptch); " +  
    "(ptch)-[e2]->(gli); " +  
    "(gli)-[e3]->(wnt1)")  
.filter("shh.id = 'SHH'")  
.filter("ptch.id = 'PTCH'")  
.filter("gli.id = 'GLI'")  
.filter("wnt1.id = 'WNT1'")
```

This is very powerful and straightforward: search for 3 nodes connected to each other and further restrict them to specific nodes.

Other Graph Algorithms

GraphFrames and GraphX provide solutions for important graph algorithms, which I have listed some of them using GraphFrames API. Once the GraphFrame has been created, there are many interesting out of the box analytics. Let `graph` be an instance of GraphFrame, then we can write the following:

- Simple Degree Algorithms:
 - Vertex in-Degree
`graph.inDegrees().show()`

- Vertex out-Degree
`graph.outDegrees().show()`

- Vertex Degree
`graph.degrees().show()`

- Complex Graph Algorithms:

- Triangle Count

```
graph.triangleCount().run().select("id",
"count").show()
```

- Label Propagation

```
graph.labelPropagation().maxIter(10).run().show()
```

- PageRank algorithm

```
graph.pageRank()
  .maxIter(0).resetProbability(0.15)
  .run()
  .vertices()
  .show()
```

- Find Shortest Paths w.r.t. Landmarks

```
graph.shortestPaths()
  .landmarks(getLandmarks())
  .run()
  .show()
```

Social Recommendation

These days, recommender systems are popular among social networks (such as Twitter and FaceBook) and shopping sites (such as Amazon). How do they do it? In this chapter, we will use distributed graph algorithms (such as

motif finding). This section will focus on building a simple Social recommendation system using Spark's GraphFrames.

Social networks introduces new types of data (represented as graphs) and metadata that can be leveraged by recommender systems (users and objects can be denoted by graph nodes and social relationships can be expressed as graph edges).

This sections is based on a blog [Using GraphFrames for Social Recommendation](#) by Hamed Firooz. The purpose of this section is to show how one can use power of pattern matching in GraphFrames for recommendation in social networks. In a social graph/network, let's assume that we have two types on objects: users and tables (these will be represented as vertexes in a graph). Tables can be chatroom or point of interests. Users can follow each other and it is one way connection (similar to Twitter, where "follow" relationship is one way, in some social networks — like FaceBook — the relationship is bi-directional). Tables contain two types of data: public and private. A user can only "follow" a table which gives him/her access to public messages or a user can be a "member" of table which means access to all messages and also privilege to send message to other members and follower of the table.

Let's consider the following sample graph for 6 users and 3 tables.

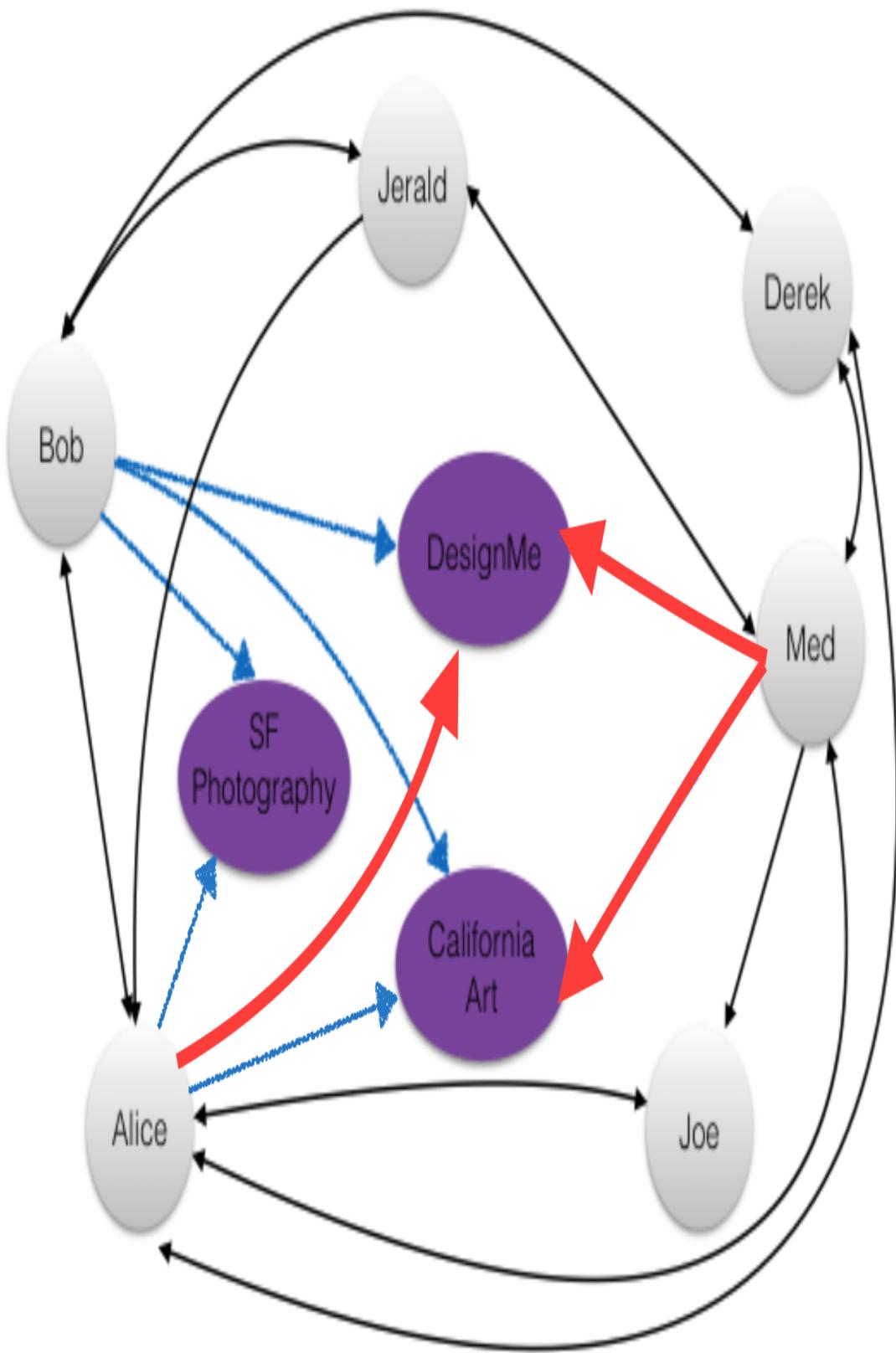


Figure 7-11. Sample Social Graph

Since GraphFrames uses DataFrame (note that DataFrame is a higher abstraction than RDD) concept rather than RDDs, we will look at its definition, before providing a solution using GraphFrames. A DataFrame is a distributed collection of data, which is organized into named columns(similar to relational data). Conceptually, it is equivalent to relational tables with good distributed optimization techniques. There are many ways to create a DataFrame. For example, You may SparkSession (as a factory class) to generate a DataFrame.

Next, we discuss on building a Graph with GraphFrame. How do we represent our social graph in GraphFrames? GraphFrames represent graphs as vertices (e.g., users and tables) and edges (e.g., relationships between users and tables). This is how we define a graph using GraphFrames API:

```
# create nodes as a DataFrame
vertices = ...

# create edges as a DataFrame
edges = ...

# build a graph from vertices and edges
graph = GraphFrame(vertices, edges)
```

Once a GraphFrame is built, then we can run algorithms and queries against it. Also, GraphFrames provide powerful API for running queries and graph algorithms. With GraphFrames, you can easily search for patterns (using “motif finding”) within graphs, find important vertices (by the PageRank algorithm), find connected friends and families (by ConnectedComponents algorithm), and more.

Now, we will use the built graph to generate some social recommendations: Suppose, we want to recommend person B to follow person A if the following conditions four are satisfied:

1. A and B are not connected.
A does not follow B, and
B does not follow A.

2. A and B have at least 4 nodes in common.
This means they both are connected to at least 4 nodes. This means that A and B have at least 4 common friends.
3. At least two of those four nodes are table.
4. A is member of the above tables

How does GraphFrame define a “motif”? GraphFrame motif finding uses a Domain Specific Language (DSL) for expressing structural pattern and queries. For example,

```
graph.find("(a)-[e1]->(b);  
          (b)-[e2]->(c);  
          (c)-[e3]->(a)")
```

will search for triangles as pairs of vertices “a, b, and c” such that

```
{ (a, b), (b, c), (c, a) }
```

It will return a DataFrame of all such structures in the graph, with columns for each of the named elements (vertices or edges) in the motif. In this case, the returned columns will be "a, b, c, e1, e2, e3". To express negation in motif finding, the exclamation (“!”) character is used: an edge can be negated to indicate that the edge should not be present in the graph. For example, the following motif

```
"(a)-[]->(b); !(b)-[]->(a)"
```

finds edges from “a to b” for which there is no edge from “b to a”.

Our social recommendation can be achieved by GraphFrames “motif finding”:

```
one_hub_connection = graph.find(  
    "(a)-[ac1]->(c1); (b)-[bc1]->(c1); " +  
    "(a)-[ac2]->(c2); (b)-[bc2]->(c2); " +  
    "(a)-[ac3]->(c3); (b)-[bc3]->(c3); " +
```

```

"(a)-[ac4]->(c4); (b)-[bc4]->(c4); " +
"!(a)-[]->(b); !(b)-[]->(a)") ❶
    .filter("c1.type = 'table'") ❷
    .filter("c2.type = 'table'") ❷
    .filter("a.id != b.id") ❸
    .filter("c1.id != c2.id") ❹
    .filter("c2.id != c3.id") ❹
    .filter("c3.id != c4.id") ❹

recommendations = one_hub_connection
    .select("a", "b")
    .distinct()
recommendations.show()
recommendations.printSchema()

```

- ❶ make sure **a** and **b** are not connected
- ❷ make sure that at least two of those four nodes are “table”
- ❸ make sure **a** is not the same as **b**
- ❹ make sure 4 friends are not the same

The output will be:

	a	b

```

root
|-- a: struct (nullable = false)
|   |-- id: string (nullable = false)
|   |-- name: string (nullable = false)
|   |-- type: string (nullable = false)
|-- b: struct (nullable = false)
    |-- id: string (nullable = false)
    |-- name: string (nullable = false)
    |-- type: string (nullable = false)

```

As you can see, GraphFrame has a very powerful “motif finding” tool. The motif expresses the following rules:

- we are interested in finding two nodes {a, b}, which are connected to 4 other nodes {c1, c2, c3, c4}, which are expressed as:

```
(a)-[ac1]->(c1);  
(b)-[bc1]->(c1);  
(a)-[ac2]->(c2);  
(b)-[bc2]->(c2);  
(a)-[ac3]->(c3);  
(b)-[bc3]->(c3);  
(a)-[ac4]->(c4);  
(b)-[bc4]->(c4);
```

- a and b are not connected to each other: expressed as:

```
!(a)-[]->(b);  
!(b)-[]->(a)
```

- at least two of those four nodes are table: expressed as two filters:

```
filter("c1.type = 'table'")  
filter("c2.type = 'table'")
```

- make sure that a and b are different users: expressed as

```
filter("a.id != b.id")
```

Since there are many ways to traverse the graph for a given motif, we want to make sure that we eliminate the duplicate entries by:

```
recommendation = one_hub_connection  
.select("a", "b")  
.distinct()
```

PageRank Algorithm

This section is devoted to a single algorithm: the computation of PageRank for a given graph. This algorithm made Google stand out from other search engines, and it is still an essential part of how search engines know what pages the user is likely want to see. Extensions of the PageRank have been used to fight against spam. For details on PageRank, you should read Ian Rogers' web page "The Google PageRank Algorithm and How It Works".

Connected Components

Introduction

Given millions of DNA samples, and given genomic relationships (sibling, parent-offspring, twins, 2nd-degree, 3rd-degree) between every 2 samples , how do you find connected families? Given social networks (such as FaceBook or Twitter), how do you identify connected communities? To solve these kind of problems, you can use "Connected Components" algorithms.

Following [Connected Component in Wikipedia](#): "In graph theory, a connected component (or just component) of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super-graph." For example, the graph shown below) has three connected components. Note that a vertex with no incident edges is itself a connected component. A graph that is itself connected has exactly one connected component, consisting of the whole graph.

An application of connected component can be to find groups of people who are more closely related in a huge set of data. Another example would be to find the communities of people in a social network, which are closely related to each other.

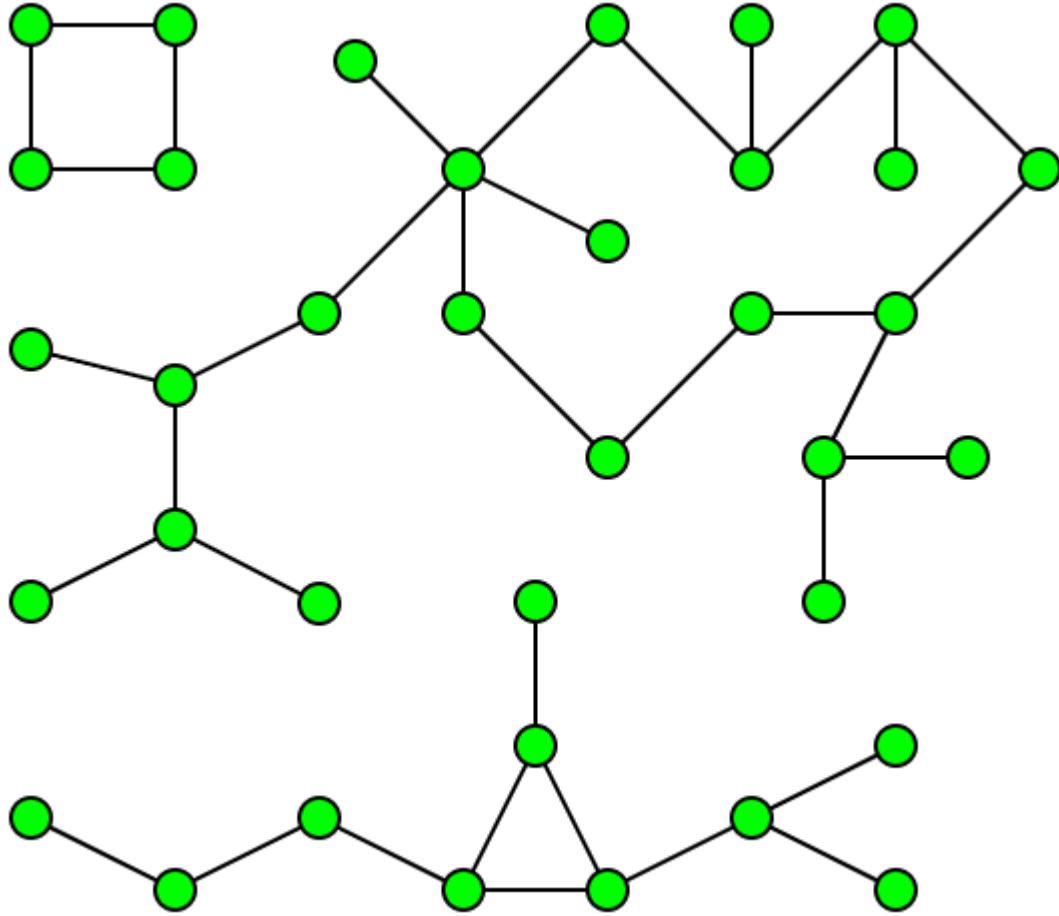


Figure 7-12. Connected Components Example

For a given graph, finding and identifying connected components is at the heart of many graph applications. For example, consider the problem of identifying family clusters in a set of DNA samples. We can represent each DNA sample by a vertex and add an edge between each pair of samples that are deemed “connected” (parent-offspring, sibling, ...). The connected components of this graph correspond to different classes of related families.

Connected Components Algorithm

The goal of Connected Components Algorithm is to identify independent disconnected subgraphs. Before we present the Connected Components Algorithm, we informally define the concept of Connected Components. Let G be a graph defined as a set of vertices V and E as a set of edges, where each edge is a pair of vertices:

$$G = (V, E)$$

Next, we define the concept of a path from one vertex to another one. Given a graph G , a path from $x \in V$ to $y \in V$ can be described by a sequence of vertices,

$$x = u_0, u_1, u_2, \dots, u_n = y,$$

where we have an edge from u_i to $u_{\{i+1\}}$ for each $0 \leq i \leq n-1$. Note that Vertices can repeat allowing the path to cross itself or fold onto itself.

Now, we can define a “connected component”: a graph G is connected if there is a path between every pair of vertices. In a nutshell, we can say that a connected component of a graph is a sub-graph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the super-graph. The smallest connected component can be a single vertex, which does not connect to any other Vertex.

Given a graph, how do we identify its connected components. Following **Connected Components algorithm**: “It is straightforward to compute the connected components of a graph in linear time (in terms of the numbers of the vertices and edges of the graph) using either breadth-first search (BFS) or depth-first search (DFS). In either case, a search that begins at some particular vertex v will find the entire connected component containing v (and no more) before returning. To find all the connected components of a graph, loop through its vertices, starting a new breadth first or depth first search whenever the loop reaches a vertex that has not already been included in a previously found connected component.”

Connected Components in Spark

The connected components algorithm labels each connected component of the graph with the ID of its lowest-numbered vertex.

To use the Connected Components algorithm, first you need to build a graph as an instance of GraphFrame. Then we call the following method to find the Connected Components:

```
vertices = <DataFrame-representing-vertices>
edges = <DataFrame-representing-edges>
graph = GraphFrame(vertices, edges)
connectedComponents = graph.connectedComponents(...)
```

The `connectedComponents()` method computes the connected components of the graph has the following signature:

```
connectedComponents(
    algorithm='graphframes',
    checkpointInterval=2,
    broadcastThreshold=1000000
)

Parameters:
algorithm - connected components algorithm
    to use (default: "graphframes"); supported
    algorithms are "graphframes" and "graphx".

checkpointInterval - checkpoint interval in
    terms of number of iterations (default: 2)

broadcastThreshold - broadcast threshold
    in propagating component assignments
    (default: 1000000)

Returns:
DataFrame with new vertices column "component"
```

The following is an example of `GraphFrame.connectedComponents()`:

```
vertices = <DataFrame-representing-vertices>
edges = <DataFrame-representing-edges>
#
graph = GraphFrame(vertices, edges)
connected_components = graph.connectedComponents()
connected_components.explain(extended=True)
#
connected_components
    .groupBy('component')
    .count()
    .orderBy('count', ascending=False)
    .show()
#
connected_components.select("id", "component")
```

```
.orderBy("component")
.show()
```

Facebook Circles

In this section we will use “motif finding” to analyze Facebook friends and relationships. The data we will use is generated by a Facebook application and data has been anonymized.

Input

For input, we use data from SNAP: [Stanford Network Analysis Project](#). According to the SNAP, this dataset consists of “circles” (or “friends lists”) from Facebook. Facebook data was collected from survey participants using a Facebook app. The dataset includes node features (profiles), circles, and ego networks. For this example, you can download the vertices file from [here](#) and the edges from [here](#).

Let’s take at a look at the downloaded data:

```
$ wc -l stanford_fb_edges.csv stanford_fb_vertices.csv  
88,235 stanford_fb_edges.csv  
4,039 stanford_fb_vertices.csv
```

Therefore, we have 4039 vertices and 88235 edges. Then, we examine the content of vertices and edges. As you can observe below, these files have headers and ideal to read them and create DataFrames. I have renamed the headers of vertices and edges files to follow the GraphFrames guidelines.

```
$ head -5 stanford_fb_edges.csv  
src,dst  
0,1  
0,2  
0,3  
0,4  
  
$ head -5 stanford_fb_vertices.csv  
id,birthday,hometown_id,work_employer_id,education_school_id,education_year_id  
1098,None,None,None,None,None  
1142,None,None,None,None,None  
1304,None,None,None,None,None  
1593,None,None,None,None,None
```

Build Graph

Since we have vertices and edges as CSV files with headers, first, we will build Dataframes for vertices and edges.

Build vertices as a GraphFrame:

```
>>> vertices_path = 'file:///tmp/stanford_fb_vertices.csv'
>>> vertices = spark ①
    .read ②
    .format("csv") ③
    .option("header", "true") ④
    .option("inferSchema", "true") ⑤
    .load(vertices_path) ⑥

>>> vertices.count()
4039

>>> vertices.printSchema()
root
 |-- id: integer (nullable = true)
 |-- birthday: string (nullable = true)
 |-- hometown_id: string (nullable = true)
 |-- work_employer_id: string (nullable = true)
 |-- education_school_id: string (nullable = true)
 |-- education_year_id: string (nullable = true)

>>> vertices.show(3, truncate=False)
+-----+-----+-----+-----+
|id   |birthday|hometown|work_      |education_|education|
|    |         |_id       |employer_id|school_id |_year_id |
+-----+-----+-----+-----+
|1098|None     |None      |None       |None      |None      |
|1142|None     |None      |None       |None      |None      |
|1917|None     |None      |None       |None      |72        |
+-----+-----+-----+-----+
```

- ① spark is a an instance of SparkSession
- ② DataFrameReader to read input file
- ③ type of file to be read

- ④ CSV file has a header
- ⑤ automatically infers column types; it requires one extra pass over the data and is false by default
- ⑥ input path

Build edges as a GraphFrame:

```
>>> edges_path = 'file:///tmp/stanford_fb_edges.csv'
>>> vertices = spark ①
        .read ②
        .format("csv") ③
        .option("header", "true") ④
        .option("inferSchema", "true") ⑤
        .load(edges_path) ⑥

>>> edges.count()
88234

>>> edges.printSchema()
root
| -- src: integer (nullable = true)
| -- dst: integer (nullable = true)

>>> edges.show(4, truncate=False)
+----+
|src|dst|
+----+
|0  |1  |
|0  |2  |
|0  |3  |
|0  |4  |
+----+
```

- ① spark is a an instance of SparkSession
- ② DataFrameReader to read input file
- ③ type of file to be read

- ④ CSV file has a header
- ⑤ automatically infers column types; it requires one extra pass over the data and is false by default
- ⑥ input path

Once we have our two DataFrames, we will now create the GraphFrame”

```
from graphframes import GraphFrame
graph = GraphFrame(vertices, edges)
>>> graph
GraphFrame(v:[id: int, birthday: string ... 4 more fields],
e:[src: int, dst: int])
>>> graph.triplets.show(3, truncate=False)
+-----+-----+-----+
| src | edge | dst |
+-----+-----+-----+
|[0, None, None, None, None]| [0, 1] | [1, None, None, None, None] |
|[0, None, None, None, None]| [0, 2] | [2, None, None, None, None] |
|[0, None, None, None, None]| [0, 3] | [3, 7, None, None, None] |
+-----+-----+-----+
```

Motif Finding

Now that the graph is built as a GraphFrame, then we can do some analysis:

Analysis: Same Birthday

Here, we find all connected vertices with the same birthday identifier

```
same_birthday = graph.find("(a)-[]-(b)")
    .filter("a.birthday = b.birthday")
print "count: %d" % same_birthday.count()
selected = same_birthday.select("a.id", "b.id", "b.birthday")
```

Analysis: Find Triangles

Here, we counts the number of triangles passing through each vertex in this graph.

```

>>> triangle_counts = graph.triangleCount()
>>> triangle_counts.show(5, truncate=False)
+-----+-----+-----+-----+-----+
|count|id |birthday|hometown_id|work_employer_id|education |education |
|     ||    |         |hometown_id|                 |_school_id|_year_id |
+-----+-----+-----+-----+-----+
|80   |148|None   |None      |None       |None     |None    |
|361  |463|None   |None      |None       |None     |None    |
|312  |471|None   |None      |None       |52       |None    |
|399  |496|None   |None      |None       |52       |None    |
|38   |833|None   |None      |None       |None     |None    |
+-----+-----+-----+-----+-----+

```

Analysis: Find Friends of Friends

The following graph query finds “friends of friends” who are not connected to each other, but graduated the same year from the same school:

```

from pyspark.sql.functions import col

friends_of_friends = graph.find("(a)-[]->(b);"
                                 "(b)-[]->(c);"
                                 !(a)-[]->(c)") \
    .filter("a.education_school_id = c.education_school_id")
    .filter("a.education_year_id = c.education_year_id")

filtered = friends_of_friends
    .filter("a.id != c.id") \
    .select(col("a.id").alias("source"), "a.education_school_id", \
            "a.education_year_id", col("c.id").alias("target"), \
            "c.education_school_id", "c.education_year_id")

>>> filtered.show(5)
+-----+-----+-----+-----+-----+
|source|education |education|target|education |education |
|source|_school_id|_year_id |      |_school_id|_year_id |
+-----+-----+-----+-----+-----+
|  3|    None|    None|  246|    None|    None|
|  3|    None|    None|   79|    None|    None|
|  3|    None|    None|  290|    None|    None|
|  5|    None|    None|  302|    None|    None|
|  9|    None|    None|  265|    None|    None|
+-----+-----+-----+-----+-----+

```

Analysis: Run PageRank

Finally, we run the PageRank algorithm on our graph:

```
>>> page_rank =  
    graph.pageRank(resetProbability=0.15, tol=0.01)  
    .vertices  
    .sort('pagerank', ascending=False)  
  
>>> page_rank.select("id", "pagerank")  
    .show(5, truncate=False)  
+----+-----+  
| id | pagerank |  
+----+-----+  
| 1911 | 37.59716511250488 |  
| 3434 | 37.555460465662755 |  
| 2655 | 36.34549422981058 |  
| 1902 | 35.816887526732344 |  
| 1888 | 27.459048061380063 |  
+----+-----+
```

Analyzing Flight Data

The example provided here is inspired from the blog [Analyzing Flight Delays by Carol McDonald](#). This blog has provided solution for flight data analysis in Scala programming language using GraphFrames. The solution presented in this section is an equivalent solution in PySpark.

Data

The data for vertices (airports) and edges (flights) are provided in the JSON formats. We will read these data and create two DataFrames for vertices and edges. Once these DataFrames are created, then we can create a graph represented as an instance of GraphFrame.

Vertices

The data for vertices is provided as a JSON file `airports.json`.

Let's review the first 2 records of `airports.json` file:

```
{"id": "ORD", "City": "Chicago", "State": "IL", "Country": "USA"}  
{"id": "LGA", "City": "New York", "State": "NY", "Country": "USA"}
```

Edges

The data for edges (flight data) is provided as a JSON file `flightdata2018.json`

Let's review the first record of `flightdata2018.json` file:

```
{  
  "id": "ATL_BOS_2018-01-01_DL_104",  
  "fldate": "2018-01-01",  
  "month": 1,  
  "dofw": 1,  
  "carrier": "DL",  
  "src": "ATL",  
  "dst": "BOS",  
  "crsdephour": 9,  
  "crsdeptime": 850,  
  "depdelay": 0.0,  
  "crsarrtime": 1116,  
  "arrdelay": 0.0,  
  "crselapsedtime": 146.0,  
  "dist": 946.0  
}
```

Building Graph

To build a graph (as an instance of a `GraphFrame`), we have to create two `DataFrames`:

- vertices from `airports.json` file
- edges from `flightdata2018.json` file

The following code snippet creates vertices and edges as required `DataFrames`.

- **Build Vertices:**

```
airports_path = '/pyspark_book_project/code/chap11/airports.json'  
vertices = spark.read.json(airports_path)  
vertices.show(3)  
+-----+-----+-----+---+  
|     City|Country|State| id|
```

Chicago	USA	IL	ORD
New York	USA	NY	LGA
Boston	USA	MA	BOS

```
vertices.count()  
13
```

- **Build Edges:**

```
flights_path = '/pyspark_book_project/code/chap11/flightdata2018.json'
edges = spark.read.json(flights_path)
edges.select("src", "dst", "dist", "depdelay")
    .show(3)

+---+---+---+
|src|dst| dist|depdelay|
+---+---+---+
|ATL|BOS|946.0|      0.0|
|ATL|BOS|946.0|      8.0|
|ATL|BOS|946.0|      9.0|
+---+---+---+


edges.count()
282628
```

- **Build Graph:**

A graph (as an instance of GraphFrame) is created using the created vertices and edges:

```
from graphframes import GraphFrame
graph = GraphFrame(vertices, edges)
graph
GraphFrame(
v:[id: string, City: string ... 2 more fields],
e:[src: string, dst: string ... 12 more fields]
)
graph.vertices.count()
13
graph.edges.count()
282628
```

Flight Analysis

Now that we have created a graph (as an instance of a GraphFrame object), we can execute queries on the created graph. GraphFrame enable us to perform queries as well as execute algorithms (such as PageRank) on the created graph.

Now we can query the GraphFrame to answer the following questions:

- How many airports are there?

```
>>> num_of_airports = graph.vertices.count()  
>>> num_of_airports  
13
```

- How many flights are there?

```
>>> num_of_flights = graph.edges.count()  
>>> num_of_flights  
282628
```

- Which flight routes have the longest distance?

```
>>> from pyspark.sql.functions import col  
>>> graph.edges  
    .groupBy("src", "dst")  
    .max("dist")  
    .sort(col("max(dist)").desc())  
    .show(4)  
+---+---+  
|src|dst|max(dist)|  
+---+---+  
|MIA|SEA| 2724.0|  
|SEA|MIA| 2724.0|  
|BOS|SFO| 2704.0|
```

```
| SFO|BOS|  2704.0|
```

```
+----+-----+
```

- Which flight routes have the highest average delays?

```
>>> graph.edges  
    .groupBy("src", "dst")  
    .avg("depdelay")  
    .sort(col("avg(depdelay)").desc())  
    .show(5)
```

```
+----+-----+
```

```
| src|dst|  avg(depdelay)|
```

```
+----+-----+
```

```
| ATL|EWR|25.520159946684437|
```

```
| DEN|EWR|25.232164449818622|
```

```
| MIA|SFO|24.785953177257525|
```

```
| MIA|EWR|22.464104423495286|
```

```
| IAH|EWR| 22.38344914718888|
```

```
+----+-----+
```

- Which flight hours have the highest average delays?

```
>>> graph.edges  
    .groupBy("crsdephour")  
    .avg("depdelay")  
    .sort(col("avg(depdelay)").desc())  
    .show(5)
```

```
+----+-----+
```

```
| crsdephour|  avg(depdelay)|
```

```
+----+-----+
```

```
|      19|22.915831356645498|
```

```
|      20|22.187089292616932|
```

```
|      18|22.183962000558815|
```

```
|      17|20.553385253108907|
```

```

|      21| 19.89884280327656|
+-----+-----+

```

- What are the longest delays for flights that are greater than 1500 miles in distance?

```

>>> graph.edges
    .filter("dist > 1500")
    .orderBy(col("depdelay").desc())
    .show(3)

+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+
|arrdelay|carrier|crsarr|crsdep|crsdep|crselapsed|depdelay| | |
|dist|dofW|dst|      fldate|                  id|month|src|
|      |      |time|hour|time|time|      |      |      |
|      |      |      |      |      |      |      |
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+
| 1332.0|     AA| 1326|     10| 1012|    254.0| 1345.0|1562.0|
| 4|DFW|2018-06-28|BOS_DFW_2018-06-2...|      6|BOS|
| 1373.0|     AA| 1811|      9|   945|    326.0| 1283.0|2342.0|
| 1|MIA|2018-07-09|LAX_MIA_2018-07-0...|      7|LAX|
| 1234.0|     AA| 1427|     10| 1040|    407.0| 1242.0|2611.0|
| 3|LAX|2018-03-28|BOS_LAX_2018-03-2...|      3|BOS|
+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+

```

- What is the average delay for delayed flights departing from Atlanta (note that Atlanta airport is labeled as ALT)?

```

>>> graph.edges
    .filter("src = 'ATL' and depdelay > 1")
    .groupBy("src", "dst")
    .avg("depdelay")

```

```

    .sort(col("avg(depdelay)").desc())
    .show(3)
+---+---+-----+
|src|dst|      avg(depdelay)|
+---+---+-----+
|ATL|EWR| 58.1085801063022|
|ATL|ORD| 46.42393736017897|
|ATL|DFW| 39.454460966542754|
+---+---+-----+

```

To view the logical and physical plans for a DataFrame execution, you may use `DataFrame.explain()`:

```
explain(extended=False)
```

Description:

Prints the (logical and physical) plans to the console for debugging purpose.

For extended plan view, set `extended=True`

Let's take a look at the logical and physical plan for our recent query:

```

graph.edges
    .filter("src = 'ATL' and depdelay > 1")
    .groupBy("src", "dst")
    .avg("depdelay")
    .sort(col("avg(depdelay)").desc())
    .explain()

== Physical Plan ==
*(3) Sort [avg(depdelay)#870 DESC NULLS LAST], true, 0
+- Exchange rangepartitioning(avg(depdelay)#870 DESC NULLS LAST, 200)
  +- *(2) HashAggregate(keys=[src#33, dst#29],
    functions=[avg(depdelay#26)])
  +- Exchange hashpartitioning(src#33, dst#29, 200)
    +- *(1) HashAggregate(keys=[src#33, dst#29],
      functions=[partial_avg(depdelay#26)])
        +- *(1) Project [depdelay#26, dst#29, src#33]
          +- *(1) Filter (((isNotNull(src#33)
            && isNotNull(depdelay#26)))

```

```

    && (src#33 = ATL)) && (depdelay#26 > 1.0))
+- *(1) FileScan json [depdelay#26,dst#29,src#33]
Batched: false, Format: JSON, Location:
InMemoryFileIndex[file:/pyspark_book/code/chap11/flightdat...,
PartitionFilters: [], PushedFilters: [IsNotNull(src),
IsNotNull(depdelay), EqualTo(src,ATL),
GreaterThan(depdelay,1.0)], ReadSchema:
struct<depdelay:double,dst:string,src:string>

```

You should always filter RDD/DataFrame before applying an expensive transformation. Since filter will drop non-required elements/rows from RDD/DataFrame and hence can improve the performance for future transformations. Therefore, filter pushdown improves performance by reducing the amount of data passed between transformations when filtering data.

- What are the worst hours for delayed flights departing from Atlanta (denoted by ATL)?

```

>>> graph.edges
    .filter("src = 'ATL' and depdelay > 1")
    .groupBy("crsdephour")
    .avg("depdelay")
    .sort(col("avg(depdelay)").desc())
    .show(4)

+-----+-----+
| crsdephour |      avg(depdelay) |
+-----+-----+
|      23 | 52.83333333333336 |
|      18 | 51.57142857142857 |
|      19 | 48.93338815789474 |
|      17 | 48.383354350567465 |
+-----+-----+

```

- What are the four most frequent flight routes in the data set? or What is the count of flights for all possible flight routes, sorted? (note that we will use the DataFrame returned later)

```

>>> flight_route_count = graph.edges
    .groupBy("src", "dst")
    .count()
    .orderBy(col("count").desc())
    .show(4)

+---+---+---+
| src|dst|count|
+---+---+---+
| LGA|ORD| 4442|
| ORD|LGA| 4426|
| LAX|SFO| 4406|
| SFO|LAX| 4354|
+---+---+---+

```

- Which airports have the most incoming and outgoing flights?

To answer this question, we need to understand the concept of vertex degrees. By graph theory definitions, the degree of a vertex is the number of edges that touch the vertex. GraphFrames provides vertex `inDegree`, `outDegree`, and `degree` operations, which determine the number of incoming edges, outgoing edges, and total edges. Using the GraphFrames `degree` operation we can answer the following question.

```

>>> graph.degrees
    .orderBy(col("degree").desc())
    .show(3)

+---+---+
| id|degree|
+---+---+
| ORD| 64386|
| ATL| 60382|
| LAX| 53733|
+---+---+

```

- What are the most important airports, according to PageRank algorithm?

The GraphFrames API provides PageRank algorithm, which is based on the Google's PageRank algorithm. The PageRank algorithm is an iterative algorithm, which measures the importance of each vertex in a graph, by determining which vertices have the most edges with other vertices. Here, we use PageRank to determine which airports are the most important, by measuring which airports have the most connections to other airports with lots of connections.

Based on PageRank algorithm, the city of Chicago airport is the most important (since it has the highest page rank) among all airports examined.

```
# Run PageRank until convergence to tolerance "tol"
>>> ranks = graph.pageRank(resetProbability=0.15, tol=0.01)
>>> ranks
GraphFrame(
v:[id: string, City: string ... 3 more fields],
e:[src: string, dst: string ... 13 more fields]
)
>>> ranks.vertices.orderBy(col("pagerank").desc()).show(3)
+-----+-----+-----+-----+
|      City|Country|State| id|      pagerank|
+-----+-----+-----+-----+
|  Chicago|    USA|   IL|ORD| 1.4151923966632058|
|  Atlanta|    USA|   GA|ATL| 1.3342533126163776|
| Los Angeles|    USA|   CA|LAX| 1.197905124144182|
+-----+-----+-----+-----+
```

You may run the PageRank algorithm by fixed number of iterations rather using the tolerance level for convergence:

```
# Run PageRank for a fixed number of iterations.
results = graph.pageRank(resetProbability=0.15, maxIter=10)
```

For details on using the PageRank algorithm, refer to the [GraphFrames documentation](#).

- What are the flight routes with no direct connection?

To answer this question, we will use the GraphFrame's Motif Finding for Graph Pattern Queries. Motif finding searches for structural patterns (such as triangles) in a graph. To find flights with no direct connection, first we create a subgraph from the `flight_route_count` DataFrame that we created earlier, which gives us a subgraph with all the possible flight routes. Then we do a `find()` on the pattern shown here to search for flights from `a` to `b` and `b` to `c`, that do not have a flight from `a` to `c`. Finally we use a DataFrame filter to remove duplicates. This shows how Graph queries can be easily combined with DataFrame operations like `filter`.

The following code snippet finds flight routes with no direct connection:

Let's recall finding most frequent flight routes in the data set:

```
>>> flight_route_count = graph.edges
     .groupBy("src", "dst")
     .count()
     .orderBy(col("count").desc())
     .show(4)
+-----+
|src|dst|count|
+----+---+---+
|LGA|ORD| 4442|
|ORD|LGA| 4426|
|LAX|SFO| 4406|
|SFO|LAX| 4354|
+----+---+
```

Let's search for flights from `a` to `b` and `b` to `c`, that do not have a flight from `a` to `c`:

```
>>> sub_graph = GraphFrame(graph.vertices, flight_route_count)
>>> sub_graph
GraphFrame(
v:[id: string, City: string ... 2 more fields],
e:[src: string, dst: string ... 1 more field]
)
>>> results = sub_graph.find(
    "(a)-[]->(b);
```

```

        (b)-[]->(c);
        !(a)-[]->(c)"
    )
.filter("c.id != a.id")

```

Results are presented below:

```

>>> results.show(5)
+-----+-----+-----+
|      a|      b|      c|
+-----+-----+-----+
|[New York, USA, N...|[Denver, USA, CO,...|[San Francisco, U...
|[Los Angeles, USA...|[Miami, USA, FL,...|[New York, USA, N...
|[New York, USA, N...|[Denver, USA, CO,...|[Newark, USA, NJ,...|
|[New York, USA, N...|[Miami, USA, FL,...|[Newark, USA, NJ,...|
|[Newark, USA, NJ,...|[Atlanta, USA, GA...|[New York, USA, N...
+-----+-----+-----+

```

- Shortest Path Graph Algorithm

The “Shortest Path” algorithm computes the shortest paths from each vertex (airport) to the given sequence of landmark vertices (airports), Here we search for the shortest path from each airport to LGA , the results show that there are no direct flights from LAX, SFO, SEA , and EWR to LGA (the distances greater than 1).

```

>>> results = graph.shortestPaths(landmarks=["LGA"])
>>> results.show(5)
+-----+-----+-----+-----+
|      City|Country|State| id| distances|
+-----+-----+-----+-----+
|      Houston|     USA|     TX|IAH|[LGA -> 1]|
|      Charlotte|     USA|     NC|CLT|[LGA -> 1]|
|      Los Angeles|     USA|     CA|LAX|[LGA -> 2]|
|      Denver|     USA|     CO|DEN|[LGA -> 1]|
|      Dallas|     USA|     TX|DFW|[LGA -> 1]|
+-----+-----+-----+-----+

```

- Breadth First Search Graph Algorithm

The “Breadth-First Search” (BFS) algorithm finds the shortest path from beginning vertices to end vertices. The beginning and end vertices are

specified as DataFrame expressions, `maxPathLength` specifies the limit on the length of paths. Here we see that there are no Direct flights between LAX and LGA:

```

# bfs() signature:
# bfs(fromExpr, toExpr, edgeFilter=None, maxPathLength=10)
# it returns a DataFrame with one Row for each shortest
# path between matching vertices.
>>> paths = graph.bfs("id = 'LAX'", "id = 'LGA'", maxPathLength=1)
>>> paths.show()
+-----+-----+
|City|Country|State| id|
+-----+-----+
+-----+-----+
>>> paths = graph.bfs("id = 'LAX'", "id = 'LGA'", maxPathLength=2)
>>> paths.show(4)
+-----+-----+-----+
+-----+-----+
|           from|          e0|          v1|
|e1|              to|           |           |
+-----+-----+-----+
+-----+
|[Los Angeles, USA...|[0.0, UA, 1333, 8...|[Houston, USA, TX...|[0.0, UA, 1655,
1...|[New York, USA, N...
|[Los Angeles, USA...|[0.0, UA, 1333, 8...|[Houston, USA, TX...|[22.0, UA, 2233,
...|[New York, USA, N...
|[Los Angeles, USA...|[0.0, UA, 1333, 8...|[Houston, USA, TX...|[6.0, UA, 1912,
1...|[New York, USA, N...
|[Los Angeles, USA...|[0.0, UA, 1333, 8...|[Houston, USA, TX...|[0.0, UA, 2321,
1...|[New York, USA, N...
+-----+-----+-----+
+-----+

```

It is possible to combine “motif finding” with DataFrames operations. Here we want to find connecting flights between LAX and LGA using a “motif finding query”. We use a motif query to search for the pattern of:

- “a flying to b”,
 - “connecting through c”,

- then we use a DataFrame filter on the results for $a = \text{LAX}$ and $c = \text{LGA}$.

The results show some flights connecting through IAH for flights from LAX to LGA.

```
>>> graph.find("(a)-[ab]->(b);"
              "(b)-[bc]->(c)"
              ).filter("a.id = 'LAX'")
              .filter("c.id = 'LGA'")
              .limit(4)
              .select("a", "b", "c")
              .show()

+-----+-----+-----+
|       a|       b|       c|
+-----+-----+-----+
|[Los Angeles, USA...|[Houston, USA, TX...|[New York, USA, N...
+-----+-----+-----+
```

Combining a “motif finding” with DataFrame operations, we can narrow these results down further, for example flights with the arrival flight time before the departure flight time, and/or with a specific carrier.

Creating Undirected Graph

Given that GraphFrames deal with only directed graphs, for some applications it is desirable to work with undirected graphs (for example, Facebook users form a undirected graph since relationships between nodes are bidirectional). Now the question is, given a directed graph, how do we convert it into a undirected graph.

MP To complete... TBDL

for every $a \rightarrow b$, create $b \rightarrow a$, and then make edges distinct so that there will not duplicate edges such as $a \rightarrow b$ and $a \rightarrow b$

Summary

- Spark provides two distinct scale out Graph libraries:
 - GraphX (based on RDDs — not available in PySpark)
 - GraphFrames (based on DataFrames — available in PySpark)
- GraphX is Spark's API for graphs and graph-parallel computation and is available only for Java and Scala.
- GraphFrames is a external package for Spark, which provides DataFrame-based Graphs. It provides high-level APIs in Python, Scala, and Java.
- GraphX to RDDs as GraphFrames are to DataFrames
- GraphFrames provide:
 - Python, Java, and Scala APIs
 - Expressive graph queries by using “motif finding”
 - Query plan optimizers from Spark SQL
 - Graph algorithms

Chapter 8. Interacting with External Data Sources

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 8th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

In the previous chapters, we have explored interacting with the built-in data sources (RDDs and DataFrames) in Spark. In this chapter, we will focus on how Spark interfaces with external data sources such as Amazon S3, relational databases, Parquet files, and more. Specifically, I will discuss how Spark allows you to:

- Use user-defined functions (UDF) in analyzing data
- Reading data from external storage systems
- Creating RDDs and DataFrames from external storage systems
- Writing data to external storage systems

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 8](#).

Spark can read data from external storage systems like Linux files, Amazon S3, HDFS, Hive tables and relational databases (such as Oracle, MySQL, PostgreSQL) through its data source interface. The goal of this chapter is to show how to read/write data from/to external storage systems and then convert them into RDDs and DataFrames for further processing. Another goal of this chapter is to show that how Spark's data can be written back to external storage systems like files, Amazon S3, Hadoop DFS, Hive tables, HBase, and JDBC databases.

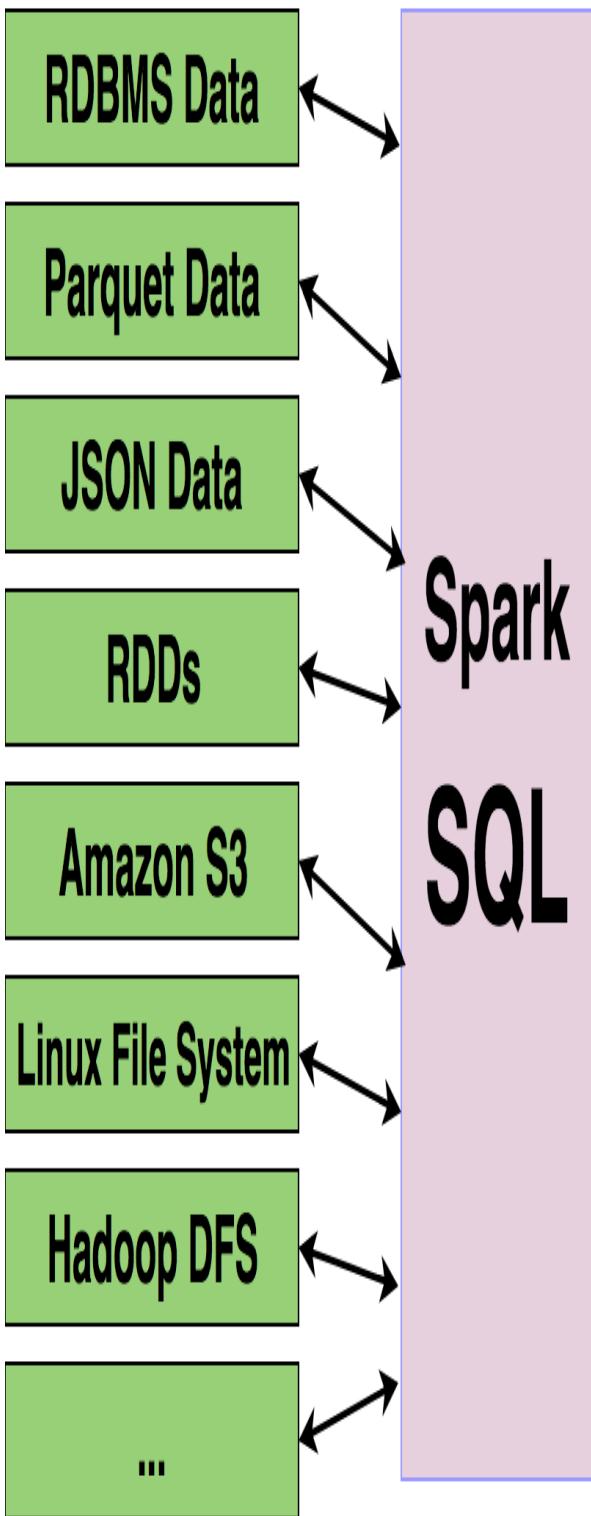
According to Spark documentation: “PySpark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, Avro files, Parquet files, SequenceFiles, and any other Hadoop InputFormat”.

There are so many data sources that Spark can read and write data, (see Figure 8.1) but we will limit this chapter to the following data sources (partial listing):

- Relational Databases
- Linux File System
- CSV Files
- JSON Files
- Hadoop DFS
- Amazon S3

- Parquet Files
- Avro Files

External Data Sources



DataFrame

	Column-1	Column-2	...
row-1			...
row-2			...
row-3			...
row-4			...
...			

Figure 8-1. Spark External Data Sources

TIP

DataFrameReader and DataFrameWriter

Spark's `DataFrameReader` class is an interface to read data from external data sources, such as text files, Parquet format, Hive tables or JDBC compliant database tables, into a `DataFrame`; and Spark's `DataFrameWriter` class is an interface to write a `DataFrame` into an external data source. `DataFrameReader` supports many file popular formats — such as Text, CSV, Parquet, ORC, ... --- and interface for new ones.

Relational Databases

A relational database is a collection of data items organized as a set of formally described tables (using `CREATE TABLE` statement) from which data can be accessed or reassembled in many different ways without having to reorganize the database tables. Open-source relational databases (such as MySQL and PostgreSQL) are currently the predominant choice in storing data like social media network records, financial records, medical records, personal information and manufacturing data. There are other well-known and licensed proprietary relational databases such as MS SQL Server and Oracle.

Relational Table Example: Employee

	id: INTEGER	name: STRING	age: INTEGER	salary: DOUBLE
row-1	99000100	alex	28	54500.50
row-2	99000200	jane	42	68500.80
row-3	99000300	joe	48	44900.20
row-4	99000400	rafael	31	126000.00
...

Figure 8-2. Relational Databases

Informally, a relational database table has a set of rows and named columns. Each row in a table can have its own unique key (called a primary key). Rows in a table can be linked to rows in other tables by adding a column for the unique key of the linked row (such columns are known as foreign keys).

PySpark provides two classes for reading (`DataFrameReader`) data from and writing (`DataFrameWriter`) data to relational databases. These two classes are defined as:

- **class pyspark.sql.DataFrameReader(spark)**
Interface used to load a DataFrame from external storage systems (e.g. file systems, key-value stores, etc). Use `spark.read()` to access this.
- **class pyspark.sql.DataFrameWriter(df)**
Interface used to write a DataFrame to external storage systems (e.g. file systems, key-value stores, etc). Use `DataFrame.write()` to access this.

Read from JDBC

PySpark enable us to read data from a relational database table and create a new DataFrame. You can read a table from any JDBC-compliant database. This reading task can be accomplished by the `pyspark.sql.DataFrameReader.load()` Python method. The `load()` method is defined as:

```
load(path=None, format=None, schema=None, **options)
Loads data from a data source and returns it as a :class:`DataFrame`.
```

Parameters:

- path - optional string or a list of string
for file-system backed data sources.
- format - optional string for format of the data
source. Default to ‘parquet’.
- schema - optional `pyspark.sql.types.StructType` for
the input schema or a DDL-formatted string
(for example `col-1 INT, col-2 DOUBLE`).
- options - all other string options

To read JDBC-compliant table, the format must be `format("jdbc")` and then you may pass the attributes of table as `options(<key>, <value>)`. For example, to construct a DataFrame representing the database table named `dept` accessible via JDBC URL `url` and connection properties, we pass these as additional `options(<key>, <value>)` explained in subsequent sections.

To read data from a JDBC-compliant relational database, I am assuming that you have access to a MySQL database server and you have sufficient privileges to read and write data (expressed in tables).

1. Create a Database Table

In this step, I will connect to MySQL database server and create a table called `dept` which will have 7 rows. For example, if you have installed a MySQL database on your MacBook, then MySQL client is available as `/usr/local/bin/mysql`. We execute `mysql` client program to enter into MySQL client shell. For details, refer to [The MySQL Command-Line Tool](#).

```
$ mysql -uroot -p ①
Enter password: <your-root-password> ②
Welcome to the MySQL monitor. Commands end with ; or \g.
Your MySQL connection id is 7
Server version: 5.7.18 MySQL Community Server (GPL)

Type 'help;' or '\h' for help.
Type '\c' to clear the current input statement.

mysql> show databases; ③
+-----+
| Database      | ④
+-----+
| information_schema |
| mysql          | ⑤
| performance_schema |
+-----+
3 rows in set (0.00 sec)
```

- ① Invoke MySQL's shell client
- ② Must enter a valid password for `root` user
- ③ List databases available in MySQL database server
- ④ These 3 databases are created by MySQL database server
- ⑤ The `mysql` database manages users, groups, and privileges

Creating and Selecting a Database

```
mysql> create database metadb; ①
mysql> use metadb; ②
Database changed
mysql>
mysql> show tables; ③
Empty set (0.00 sec)
```

- ① Create a new database called 'metadb'
- ② Make 'metadb' as your current default database
- ③ Show tables in 'metadb' database (since it is a new database, there will be no tables in it)

Next, we create a new table called 'dept' inside 'metadb' database:

```
mysql> create table dept ( ①
    ->     dept_number int,
    ->     dept_name varchar(128),
    ->     dept_location varchar(128),
    ->     manager varchar(128)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> show tables; ②
+-----+
| Tables_in_metadb |
+-----+
| dept           |
+-----+
1 row in set (0.00 sec)

mysql> desc dept; ③
+-----+-----+-----+-----+-----+-----+
| Field      | Type       | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| dept_number | int(11)   | YES  |     | NULL    |       |
| dept_name   | varchar(128)| YES  |     | NULL    |       |
| dept_location| varchar(128)| YES  |     | NULL    |       |
| manager     | varchar(128)| YES  |     | NULL    |       |
```

```
+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

- ① Table definition for 'dept', which has four columns
- ② List tables in 'metadb' database
- ③ Describe schema for 'dept' table

Finally, we insert 7 rows into the 'dept' table, by using the INSERT statement:

```
mysql> INSERT INTO dept
-> (dept_number, dept_name, dept_location, manager)
-> VALUES
-> (10, 'ACCOUNTING', 'NEW YORK, NY', 'alex'),
-> (20, 'RESEARCH', 'DALLAS, TX', 'alex'),
-> (30, 'SALES', 'CHICAGO, IL', 'jane'),
-> (40, 'OPERATIONS', 'BOSTON, MA', 'jane'),
-> (50, 'MARKETING', 'Sunnyvale, CA', 'jane'),
-> (60, 'SOFTWARE', 'Stanford, CA', 'jane'),
-> (70, 'HARDWARE', 'BOSTON, MA', 'sophia');
Query OK, 7 rows affected (0.01 sec)
Records: 7  Duplicates: 0  Warnings: 0
```

Next, we examine the content of the 'dept' table to make sure that it has 7 rows.

```
mysql> select * from dept;
+-----+-----+-----+-----+
| dept_number | dept_name | dept_location | manager |
+-----+-----+-----+-----+
|      10 | ACCOUNTING | NEW YORK, NY | alex   |
|      20 | RESEARCH   | DALLAS, TX  | alex   |
|      30 | SALES      | CHICAGO, IL | jane   |
|      40 | OPERATIONS | BOSTON, MA | jane   |
|      50 | MARKETING  | Sunnyvale, CA | jane   |
|      60 | SOFTWARE    | Stanford, CA | jane   |
|      70 | HARDWARE   | BOSTON, MA | sophia |
+-----+-----+-----+-----+
7 rows in set (0.00 sec)
```

Now, at this point, we are sure that there is a '`metadb`' database, which has a '`dept`' table with 7 records.

2. Read a Database Table into a DataFrame

Once you have a JDBC-compliant table (such as `dept`), then you can use `pyspark.sql.DataFrameReader` class's methods (combinations of `option()` and `load()`) to read the content of a table and create a new DataFrame. To perform this read, you need a JAR file, which is a MySQL JDBC Driver (you may download this JAR file from [mysql.com site](#). MySQL offers standard database driver (called **Connector J**) connectivity for using MySQL with applications and tools that are compatible with industry standards ODBC and JDBC. Any system that works with ODBC or JDBC can use MySQL. Let's put this JAR file (which contains the MySQL Driver class) here (you may place it where ever you prefer):

```
.../code/jars/mysql-connector-java-5.1.42.jar
```

Next, we get into PySpark shell by passing the JAR file to the `$SPARK_HOME/bin/pyspark` program:

```
export JAR=/pyspark_book/code/jars/mysql-connector-java-5.1.42.jar ❶
$SPARK_HOME/bin/pyspark --jars $JAR ❷
Python 3.7.2
Welcome to Spark version 2.4.0

SparkSession available as ``spark``.
>>> spark ❸
<pyspark.sql.session.SparkSession object at 0x10a5f2a50>
>>>
```

- ❶ The Driver class JAR for MySQL
- ❷ Start PySpark shell with loading MySQL Driver class
- ❸ Make sure that `SparkSession` is available

Next, we use SparkSession to read a relational table and create a new DataFrame:

```
dataframe_mysql = spark \①
    .read \②
    .format("jdbc") \③
    .option("url", "jdbc:mysql://localhost/metadb") \④
    .option("driver", "com.mysql.jdbc.Driver") \⑤
    .option("dbtable", "dept") \⑥
    .option("user", "root") \⑦
    .option("password", "mp22_pass") \⑧
    .load() ⑨
```

- ❶ spark is an instance of a SparkSession
- ❷ Returns a DataFrameReader that can be used to read data in as a DataFrame
- ❸ indicate that you are reading a JDBC-compliant data
- ❹ Database URL
- ❺ JDBC Driver (loaded from the JAR file)
- ❻ The database table name
- ❼ The database username
- ❽ The database password
- ❾ Loads data from a JDBC data source and returns it as a DataFrame

Next we examine the newly created DataFrame:

```
>>> dataframe_mysql.count() ①
7
>>> dataframe_mysql.show() ②
+-----+-----+-----+-----+
|dept_number| dept_name|dept_location|manager |
+-----+-----+-----+-----+
```

10	ACCOUNTING	NEW YORK, NY	alex
20	RESEARCH	DALLAS, TX	alex
30	SALES	CHICAGO, IL	jane
40	OPERATIONS	BOSTON, MA	jane
50	MARKETING	Sunnyvale, CA	jane
60	SOFTWARE	Stanford, CA	jane
70	HARDWARE	BOSTON, MA	sophia

- ❶ Count the number of DataFrame's rows
- ❷ Prints the first 20 rows to the console.

Next we examine the schema of newly created DataFrame:

```
>>> dataframe_mysql.printSchema ❶
<bound method DataFrame.printSchema of
DataFrame[
    dept_number: int,
    dept_name: string,
    dept_location: string,
    manager: string
]
>
```

- ❶ Prints out the schema in the tree format

SQL Queries on DataFrame

PySpark offers many ways to access your DataFrame by executing SQL queries on your data. First it offers many SQL-like methods such as `select(columns)`, `groupBy(columns)`, ``min()`, max(), ...). Then it offers a full fledge SQL queries on your DataFrame. In order to use full SQL queries on DataFrame data, first we need to register your DataFrame as a “table” and then issue SQL queries against that registered table. We will discuss DataFrame table registration shortly.`

First, we will execute some SQL-like queries by using DataFrame methods.

The following table selects all rows for two columns: “dept_number” and “manager”:

```
dataframe_mysql.select("dept_number", "manager") ①
    .show() ②
+-----+-----+
|dept_number|manager|
+-----+-----+
|      10| alex |
|      20| alex |
|      30| jane |
|      40| jane |
|      50| jane |
|      60| jane |
|      70| sophia |
+-----+-----+
```

- ① Select “dept_number” and “manager” columns from a DataFrame
- ② Display the selection result

The following table groups all rows by “manager” and then finds the minimum of “dept_number”:

```
>>> dataframe_mysql.select("dept_number", "manager")
    .groupBy("manager")
    .min("dept_number")
    .collect()
[
  Row(manager=u'jane', min(dept_number)=30),
  Row(manager=u'sophia', min(dept_number)=70),
  Row(manager=u'alex', min(dept_number)=10)
]
```

The following table groups all rows by “manager” and then finds the frequencies of grouped data:

```
>>> dataframe_mysql.select("dept_number", "manager")
    .groupBy("manager")
    .count()
    .collect()
```

```
[  
Row(manager=u'jane', count=4),  
Row(manager=u'sophia', count=1),  
Row(manager=u'alex', count=2)  
]
```

The following table groups all rows by “manager” and then finds the frequencies of grouped data, finally orders output by the “manager” column:

```
>>> dataframe_mysql.select("dept_number", "manager")  
    .groupBy("manager")  
    .count()  
    .orderBy("manager")  
    .collect()  
  
[  
Row(manager=u'alex', count=2),  
Row(manager=u'jane', count=4),  
Row(manager=u'sophia', count=1)  
]
```

SQL Queries

To execute the full fledge SQL queries against a DataFrame, first you have to register your DataFrame as a “table” by:

```
DataFrame.registerTempTable(<your-desired-table-name>)
```

Once your DataFrame is registered as a table, then you can execute regular SQL queries on that registered table (like as it is a relational database table).

```
>>> dataframe_mysql.registerTempTable("mydept") ❶  
>>> spark.sql("select * from mydept where dept_number > 30") ❷  
    .show() ❸  
+-----+-----+-----+-----+  
|dept_number|dept_name|dept_location|manager|  
+-----+-----+-----+-----+  
|      40|OPERATIONS|    BOSTON, MA|    jane|  
|      50| MARKETING|Sunnyvale, CA|    jane|  
|      60| SOFTWARE| Stanford, CA|    jane|  
|      70| HARDWARE|    BOSTON, MA| sophia|  
+-----+-----+-----+-----+
```

- ❶ Registers this DataFrame as a temporary table using the given name
- ❷ You can issue a SQL query to your registered table
- ❸ Prints the first 20 rows to the console

The next SQL query uses the “like” pattern matching for the dept_location column:

```
>>> spark.sql("select * from mydept where dept_location like '%CA'")  
    .show()  
+-----+-----+-----+-----+  
|dept_number| dept_name|dept_location|manager|  
+-----+-----+-----+-----+  
|      50| MARKETING|Sunnyvale, CA|   jane|  
|      60| SOFTWARE| Stanford, CA|   jane|  
+-----+-----+-----+-----+  
>>>
```

Next, let’s use “GROUP BY” on a SQL query:

```
>>> spark.sql("select manager, count(*) as count from mydept group by  
manager").show()  
+-----+  
|manager|count|  
+-----+  
|   jane|    4|  
| sophia|    1|  
|   alex|    2|  
+-----+
```

Write DataFrame to JDBC

A Spark’s DataFrame can be written or saved as a relational database table. We may use PySpark’s `DataFrameWriter` to write a DataFrame to external storage systems (files, relational databases, ...). We can use `DataFrameWriter.save()` to save the contents of the DataFrame to an external data source.

In the following example, we perform the following steps:

- STEP 1: create a list of triplets as ('name', 'age', 'salary') as a local Python collection
- STEP 2: convert a Python collection into an Spark **DataFrame**
- STEP 3: finally save/write created **DataFrame** as a relational database table.
- STEP 4: verify that the relational table (*triplets*) is created and check its content
- STEP 5: read back the content of newly created relational table *triplets* and create another new DataFrame
- STEP 6: perform some SQL queries on the DataFrame created in STEP 5

STEP 1 is accomplished by the following Python collection of triplets:

```
>>> triplets = [ ("alex", 60, 18000),
...               ("adel", 40, 45000),
...               ("adel", 50, 77000),
...               ("jane", 40, 52000),
...               ("jane", 60, 81000),
...               ("alex", 50, 62000),
...               ("mary", 50, 92000),
...               ("mary", 60, 63000),
...               ("mary", 40, 55000),
...               ("mary", 40, 55000)
...             ]
```

STEP 2 is accomplished by the `SparkSession.createDataFrame()` method.

```
>>> tripletsDF = spark.createDataFrame( ❶
...                                         triplets, ❷
...                                         ['name', 'age', 'salary'] ❸
...                                       )
>>> tripletsDF.show() ❹
+----+---+-----+
```

```

| name | age | salary |
+-----+----+-----+
| alex | 60 | 18000 |
| adel | 40 | 45000 |
| adel | 50 | 77000 |
| jane | 40 | 52000 |
| jane | 60 | 81000 |
| alex | 50 | 62000 |
| mary | 50 | 92000 |
| mary | 60 | 63000 |
| mary | 40 | 55000 |
| mary | 40 | 55000 |
+-----+----+-----+

```

- ❶ Create a new DataFrame
- ❷ Convert triplets into a DataFrame
- ❸ Impose a schema to created DataFrame
- ❹ Display content of a newly created DataFrame

STEP 3 is accomplished by converting a `DataFrame` into a relational table called *triplets*:

```

tripletsDF
  .write ❶
    .format("jdbc") ❷
      .option("driver", "com.mysql.jdbc.Driver") ❸
        .mode("overwrite") ❹
          .option("url", "jdbc:mysql://localhost/metadb") ❺
            .option("dbtable", "triplets") ❻
              .option("user", "root") ❼
                .option("password", "mp22_pass") ❽
                  .save() ❾

```

- ❶ Returns a `DataFrameWriter` that can be used to write to an external device
- ❷ indicate that you are writing to a JDBC-compliant database

- ③ JDBC Driver (loaded from the JAR file)
- ④ if table exists, then overwrite it
- ⑤ Database URL
- ⑥ The target database table name
- ⑦ The database username
- ⑧ The database password
- ⑨ save DataFrame data as a JDBC data source

When writing the content of a DataFrame to an external device, you can choose desired “mode”. Spark JDBC writer supports following modes:

- **append**: Append contents of this `:class:DataFrame` to existing data
- **overwrite**: Overwrite existing data
- **ignore**: Silently ignore this operation if data already exists
- **error** (default case): Throw an exception if data already exists

In STEP 4 we verify that the 'triplets' table is created (output is formatted to fit the page) in MySQL database server under the 'metadb' database:

```
$ mysql -uroot -p ①
Enter password: xxxxxxxx ②
Welcome to the MySQL Server version: 5.7.18

mysql> use metadb; ③
Database changed

mysql> desc triplets; ④
+-----+-----+-----+-----+-----+
```

```

| Field | Type      | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+
| name  | text       | YES  |     | NULL    |       |
| age   | bigint(20) | YES  |     | NULL    |       |
| salary | bigint(20) | YES  |     | NULL    |       |
+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)

```

```
mysql> select * from triplets; ⑤
```

```

+-----+-----+-----+
| name | age  | salary |
+-----+-----+-----+
| jane | 40   | 52000  |
| adel | 50   | 77000  |
| jane | 60   | 81000  |
| alex | 50   | 62000  |
| mary | 40   | 55000  |
| mary | 40   | 55000  |
| adel | 40   | 45000  |
| mary | 60   | 63000  |
| alex | 60   | 18000  |
| mary | 50   | 92000  |
+-----+-----+-----+
10 rows in set (0.00 sec)

```

- ① Enter into MySQL client shell
- ② Enter the password for the `root` user
- ③ Select the desired '`metadb`' database
- ④ Make sure that the `triplets` table is created
- ⑤ Display the content of the '`triplets`' table

In STEP 5, we read the '`triplets`' table back from the MySQL relational database to make sure that the table is readable:

```

tripletsDF_mysql =
spark ①
.read ②
.format("jdbc") ③

```

```

.option("url", "jdbc:mysql://localhost/metadb") ④
.option("driver", "com.mysql.jdbc.Driver") ⑤
.option("dbtable", "triplets") ⑥
.option("user", "root") ⑦
.option("password", "mp22_pass") ⑧
.load() ⑨

#
tripletsDF_mysql.show() ⑩
+-----+
|name|age|salary|
+-----+
|jane| 40| 52000|
|adel| 50| 77000|
|jane| 60| 81000|
|alex| 50| 62000|
|mary| 40| 55000|
|mary| 40| 55000|
|adel| 40| 45000|
|mary| 60| 63000|
|alex| 60| 18000|
|mary| 50| 92000|
+-----+

```

- ❶ spark is an instance of a `SparkSession`
- ❷ Returns a `DataFrameReader` that can be used to read data in as a `DataFrame`
- ❸ indicate that you are reading a JDBC-compliant data
- ❹ Database URL
- ❺ JDBC Driver (loaded from the JAR file)
- ❻ The database table name to be read
- ❼ The database username
- ❼ The database password
- ❽ Loads data from a JDBC data source and returns it as a `DataFrame`

⑩ Show the content of newly created DataFrame

In STEP 6, we execute some SQL queries on the newly created DataFrame.

The following query finds the minimum and maximum of the *salary* column:

```
>>> tripletsDF_mysql.registerTempTable("mytriplets") ❶
>>> spark.sql("select min(salary), max(salary) from mytriplets") ❷
    .show() ❸
+-----+-----+
|min(salary)|max(salary)|
+-----+-----+
|      18000|      92000|
+-----+-----+
```

- ❶ Registers this DataFrame as a temporary table using the given name “mytriplets”
- ❷ Execute the SQL statement and create a new DataFrame
- ❸ Display the result of SQL statement

Here, we aggregate the *age* column by using the SQL’s GROUP BY:

```
>>> spark.sql("select age, count(*) from mytriplets group by age").show()
+---+-----+
|age|count(1)|
+---+-----+
| 50|      3|
| 60|      3|
| 40|      4|
+---+-----+
```

Next, we sort the result of the previous SQL query:

```
>>> spark.sql("select age, count(*) from mytriplets group by age order by
age").show()
+---+-----+
|age|count(1)|
+---+-----+
```

40	4
50	3
60	3
+-----+	

Reading Text Files

Spark enable us to read text files and create a DataFrame. Consider the following text file:

```
$ cat people.txt
Alex,30,Tennis
Betty,40,Swimming
Dave,20,Walking
Jeff,77,Baseball
```

Let's first create an RDD[Row] (each element is a Row object):

```
from pyspark.sql import Row
#
def create_row(rec):
    p = rec.split(",")
    return Row(name=p[0], age=int(p[1]), hobby=p[2])
#end-def
input_path = "people.txt"
# Load a text file and convert each line to a Row.
records = spark.sparkContext.textFile(input_path) ❶
>>> records.collect()
[
    u'Alex,30,Tennis',
    u'Betty,40,Swimming',
    u'Dave,20,Walking',
    u'Jeff,77,Baseball'
]
people = records.map(create_row) ❷
people.collect()
[
    Row(age=30, hobby=u'Tennis', name=u'Alex'),
    Row(age=40, hobby=u'Swimming', name=u'Betty'),
    Row(age=20, hobby=u'Walking', name=u'Dave'),
    Row(age=77, hobby=u'Baseball', name=u'Jeff')
]
```

❶ records is an RDD[String]

❷ people is an RDD[Row]

Now that we have people as RDD[Row], it is straightforward to create a DataFrame:

```
people_df = spark.createDataFrame(people) ❶
>>> people_df.show()
+---+---+---+
| age | hobby | name |
+---+---+---+
| 30 | Tennis | Alex |
| 40 | Swimming | Betty |
| 20 | Walking | Dave |
| 77 | Baseball | Jeff |
+---+---+---+
people_df.printSchema() ❷
root
 |-- age: long (nullable = true)
 |-- hobby: string (nullable = true)
 |-- name: string (nullable = true)
```

❶ people_df is a DataFrame[Row]

❷ display the schema a created DataFrame

Next, we use a SQL query to manipualte the created DataFrame:

```
people_df.registerTempTable("people_table") ❶
>>> spark.sql("select * from people_table").show() ❷
+---+---+---+
| age | hobby | name |
+---+---+---+
| 30 | Tennis | Alex |
| 40 | Swimming | Betty |
| 20 | Walking | Dave |
| 77 | Baseball | Jeff |
+---+---+---+
>>> spark.sql("select * from people_table where age > 35").show() ❸
```

```
+---+-----+-----+
| age| hobby| name|
+---+-----+-----+
| 40| Swimming| Betty|
| 77| Baseball| Jeff|
+---+-----+
```

- ❶ Registers `people_df` DataFrame as a temporary table using the given name (“`people_table`”)
- ❷ `spark.sql(sql-query)` creates a new DataFrame
- ❸ `spark.sql(sql-query)` creates a new DataFrame

Saving text files can be done by `DataFrame.write()`.

Read/Write CSV Files

What is a CSV file? A CSV is a comma separated values file which allows data to be saved in a table structured format. The following is a simple example of a CSV file with header called `cats.with.header.csv`:

```
$ cat cats.with.header.csv
#name,age,gender,weight ❶
cuttie,2,female,6        ❷
mono,3,male,9           ❸
fuzzy,1,female,4         ❹
```

- ❶ Header record starts with “#”, describes columns
- ❷ 1st record
- ❸ 2nd record
- ❹ 3rd and last record

The following is a simple example of a CSV file without any header called `cats.no.header.csv`:

```
$ cat cats.no.header.csv
cuttie,2,female,6
mono,3,male,9
fuzzy,1,female,4
```

Reading CSV Files

In this section, we will use the two files (`cats.with.header.csv` and `cats.no.header.csv`) to show how Spark can read these CSV files. Spark offers many methods to load CSV files into a DataFrame and we will examine some them here by examples.

In the following example, using PySpark shell, we will read a CSV file with a header (name of columns separated by “,” — metadata for columns) and load it as a DataFrame.

```
# spark : pyspark.sql.session.SparkSession object
input_path = '/pyspark_book/code/chap08/cats.with.header.csv'
cats = spark ❶
    .read ❷
        .format("csv") ❸
        .option("header", "true") ❹
        .option("inferSchema", "true") ❺
        .load(input_path) ❻
```

- ❶ Create a new DataFrame as `cats` by using a `SparkSession`
- ❷ Returns a `DataFrameReader` that can be used to read data in as a DataFrame
- ❸ Specifies the input data source format is CSV
- ❹ Indicate that the input CSV file has a header (note that the header is not part of the actual data)
- ❺ Infer a DataFrame schema from input file

❶ Path for CSV file

Next, we display the content of newly created DataFrame and its inferred schema:

```
cats.show() ❶
+-----+
| name|age|gender|weight|
+-----+
| cuttie| 2|female|     6|
| mono| 3| male|     9|
| fuzzy| 1|female|     4|
+-----+

cats.printSchema ❷
<bound method DataFrame.printSchema of DataFrame ❸
[
  name: string,
  age: int,
  gender: string,
  weight: int
]>
>>> cats.count()
3
```

- ❶ Display the content of a DataFrame
- ❷ Display the schema of a DataFrame
- ❸ Display the size of a DataFrame

In the next example, I show how to read a CSV file without a header and create a new DataFrame:

```
input_path = '/pyspark_book/code/chap08/cats.no.header.csv'
cats2 = spark ❶
    .read ❷
        .format("csv") ❸
        .option("header", "false") ❹
```

```
.option("inferSchema", "true") ⑤  
.load(input_path) ⑥
```

- ❶ Create a new DataFrame as `cats` by using a `SparkSession`
- ❷ Returns a `DataFrameReader` that can be used to read data in as a `DataFrame`
- ❸ Specifies the input data source format is CSV
- ❹ Indicate that the input CSV file has no header
- ❺ Infer a `DataFrame` schema from input file
- ❻ Path for CSV file

Next, we display the content of newly created DataFrame and its inferred schema:

```
>>> cats2.show()  
+-----+-----+-----+  
| _c0|_c1| _c2|_c3| ❶  
+-----+-----+-----+  
| cuttie|  2|female|  6|  
| mono |  3| male |  9|  
| fuzzy|  1|female|  4|  
+-----+-----+-----+
```

- ❶ default column names

Next, we define a schema with four columns:

```
>>>  
>>> from pyspark.sql.types import StructType  
>>> from pyspark.sql.types import StructField  
>>> from pyspark.sql.types import StringType  
>>> from pyspark.sql.types import IntegerType  
>>>
```

```
>>> catsSchema = StructType([
... StructField("name", StringType(), True),
... StructField("age", IntegerType(), True),
... StructField("gender", StringType(), True),
... StructField("weight", IntegerType(), True)
... ])
```

Finally, we use the defined schema to read a CSV file and create a DataFrame:

```
>>> input_path = '/pyspark_book/code/chap08/cats.no.header.csv'
>>> cats3 = spark
    .read
    .format("csv")
    .option("header", "false")
    .option("inferSchema", "true")
    .load(input_path, schema = catsSchema)

>>>
>>> cats3.show()
+-----+-----+-----+
| name|age|gender|weight| ❶
+-----+-----+-----+
| cuttie| 2|female|     6|
| mono| 3| male|     9|
| fuzzy| 1|female|     4|
+-----+-----+-----+

>>> cats3.count()
3
```

❶ Explicit column names

The following example reads a header-less CSV file with a predefined schema:

```
>>> cats4 = spark
    .read ❶
    .csv("file:///tmp/cats.no.header.csv", ❷
          schema = catsSchema, ❸
          header = "false") ❹

>>>#
>>> cats4.show()
+-----+-----+-----+
```

```

|   name|age|gender|weight| ⑤
+-----+---+----+-----+
|cuttie|  2|female|     6|
|  mono|  3|  male|     9|
|fuzzy|  1|female|     4|
+-----+---+----+-----+

>>> cats4.printSchema
<bound method DataFrame.printSchema of DataFrame
[
  name: string,
  age: int,
  gender: string,
  weight: int
]>
>>> cats4.count()
3
>>>

```

- ❶ The `read` represents a `DataFrameReader`
- ❷ Read a CSV file
- ❸ Use the given schema for CSV file
- ❹ Indicate that CSV file has no header
- ❺ Explicit column names

Writing CSV Files

There are several ways that you can create CSV files from Spark. The easiest way to create CSV file is to use the `csv()` method of the `DataFrameWriter(df)` interface used to write a `DataFrame` to external storage systems (e.g. file systems, key-value stores, etc). To access `DataFrameWriter(df)`, use the `DataFrame.write` to access this.

The `DataFrameWriter.csv()` method is defined below (there are lots of parameters, I have just listed a small subset of them).

```
csv(path, mode=None, compression=None, sep=None, ...)  
Saves the content of the DataFrame in CSV format  
at the specified path.
```

Parameters:

- path – the path in any Hadoop supported file system
- mode – specifies the behavior of the save operation when data already exists.
 - "append": Append contents of this DataFrame to existing data.
 - "overwrite": Overwrite existing data.
 - "ignore": Silently ignore this operation if data already exists.
 - "error": Throw an exception if data already exists.
- compression – compression codec to use when saving to file...
- sep – sets a single character as a separator for each field and value. If None is set, it uses the default value
- ...

Writing CSV Files: Example-1

Let cats4 be a DataFrame:

```
>>> cats4.show()  
+-----+-----+-----+  
| name|age|gender|weight|  
+-----+-----+-----+  
| cuttie| 2|female|     6|  
| mono|  3| male|     9|  
| fuzzy| 1|female|     4|  
+-----+-----+-----+  
  
>>> cats4.write.csv("file:///tmp/cats4", sep = ';')
```

Now examine the /tmp/cats4 directory (this output has no headers data):

```
$ ls -l /tmp/cats4  
total 8  
-rw-r--r--  ...  0 Apr 12 16:46 _SUCCESS  
-rw-r--r--  ...  49 Apr 12 16:46 part-00000-...-c000.csv  
$ cat /tmp/cats4/part*  
cuttie;2;female;6
```

```
mono;3;male;9
fuzzy;1;female;4
```

Note that in output path, we see two types of files:

- zero-sized `_SUCCESS` file, which indicates that the `write` operation was success
- file(s) which begin with "`part-`", which represents an output from a single partition

Writing CSV Files: Example-2

Let save a DataFrame, and then examine the saved file(s):

```
cats4.write.csv("file:///tmp/cats48",
                sep = ';',
                header = 'true')

$ ls -l /tmp/cats48
total 8
-rw-r--r-- ... 0 Apr 12 16:49 _SUCCESS
-rw-r--r-- ... 72 Apr 12 16:49 part-00000-....c000.csv
$ cat /tmp/cats48/part*
name;age;gender;weight ❶
cuttie;2;female;6
mono;3;male;9
fuzzy;1;female;4
```

❶ header from a DataFrame

Read/Write JSON Files

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. JSON in a nutshell:

- is a syntax for storing and exchanging data
- is text, written with JavaScript object notation
- is a set of (key, value) pairs

The following is an example of a JSON data:

```
{  
    "first_name" : "John",  ❶  
    "last_name" : "Smith",  
    "age" : 23,  
    "gender" : "Male",  
    "cars": [ "Ford", "BMW", "Fiat" ] ❷  
}
```

- ❶ simple (key, value) pair
- ❷ an array value

Reading JSON Files

JSON data can be read by the `DataFrameReader.json()` method, which can take a set of parameters: such as path and schema. For details refer to [PySpark Documentation](#).

Consider the following JSON file:

```
$ cat $SPARK_HOME/examples/src/main/resources/employees.json  
{"name":"Michael", "salary":3000}  
 {"name":"Andy", "salary":4500}  
 {"name":"Justin", "salary":3500}  
 {"name":"Berta", "salary":4000}
```

Next, we will read this JSON data and convert it to a `DataFrame`:

```
data_path = 'examples/src/main/resources/employees.json'  
>>> df = spark.read.json(data_path)  
>>> df.show()  
+-----+-----+  
|    name|salary|  
+-----+-----+  
|Michael|  3000|  
|   Andy|  4500|  
| Justin|  3500|  
|  Berta|  4000|  
+-----+-----+
```

```

>>> df.printSchema
<bound method DataFrame.printSchema of DataFrame
[
  name: string,
  salary: bigint
]>
>>> df.count()
4

```

Also, you may use the `load()` method and pass your JSON files:

```

data_path = 'examples/src/main/resources/employees.json'
>>> df2 = spark.read.format('json')
      .load([data_path, data_path]) ❶
>>> df2.show()
+-----+-----+
|    name|salary|
+-----+-----+
|Michael|  3000|
|   Andy|  4500|
| Justin|  3500|
|  Berta|  4000|
|Michael|  3000|
|   Andy|  4500|
| Justin|  3500|
|  Berta|  4000|
+-----+-----+

>>> df2.printSchema
<bound method DataFrame.printSchema of DataFrame
[
  name: string,
  salary: bigint
]>
>>> df2.count()
8

```

- ❶ Note that `data_path` is loaded twice

Writing JSON Files

A `DataFrame` can be written as a JSON object. To write a `DataFrame` as a JSON object, we can use `DataFrameWriter.json()` method.

The `json()` method accepts a set of parameters and saves the content of the `DataFrame` in JSON format:

```
json(  
    path,  
    mode=None,  
    compression=None,  
    dateFormat=None,  
    timestampFormat=None  
)
```

Parameters:

- `path` – the path in any Hadoop supported file system
- `mode` – specifies the behavior of the save operation when data already exists. Possible values are:
`"append"`, `"overwrite"`, `"ignore"`, `"error"`
- `compression` – compression codec to use when saving to file.
- `dateFormat` – sets the string that indicates a date format.
- `timestampFormat` – sets the string that indicates a timestamp format.

Let's first create a `DataFrame`:

```
>>> data = [("name", "alex"), ("gender", "male"), ("state", "CA")]  
>>> df = spark.createDataFrame(data, ['key', 'value'])  
>>> df.show()  
+-----+  
| key|value|  
+-----+  
| name| alex|  
| gender| male|  
| state| CA|  
+-----+
```

Next, we write the created `DataFrame` as a JSON to an ouput path:

```
>>> df.write.json('/tmp/data')  
  
$ ls -l /tmp/data  
total 24
```

```
-rw-r--r-- ... 0 Apr 2 01:15 _SUCCESS
-rw-r--r-- ... 0 Apr 2 01:15 part-00000-....c000.json
-rw-r--r-- ... 0 Apr 2 01:15 part-00001-....c000.json
-rw-r--r-- ... 30 Apr 2 01:15 part-00002-....c000.json
-rw-r--r-- ... 0 Apr 2 01:15 part-00003-....c000.json
-rw-r--r-- ... 0 Apr 2 01:15 part-00004-....c000.json
-rw-r--r-- ... 32 Apr 2 01:15 part-00005-....c000.json
-rw-r--r-- ... 0 Apr 2 01:15 part-00006-....c000.json
-rw-r--r-- ... 29 Apr 2 01:15 part-00007-....c000.json
```

Note that we have 8 file names, which begin with "part-": this means that our `DataFrame` was represented by 8 partitions.

Next, let's take a look at the content of output path:

```
$ cat /tmp/data/part*
{"key": "name", "value": "alex"}
 {"key": "gender", "value": "male"}
 {"key": "state", "value": "CA"}
```

The `mode(saveMode)` specifies the behavior when data or table already exists; the possible values for the `saveMode` parameter are:

- “**append**”: Append contents of this `DataFrame` to existing data.
- “**overwrite**”: Overwrite existing data.
- “**error**” or “**errorIfExists**”: Throw an exception if data already exists.
- “**ignore**”: Silently ignore this operation if data already exists.

If you want to create a single file output, then you may put your `DataFrame` into a single partition, before writing it out:

```
>>> df.repartition(1).write.json('/tmp/data') ❶
```

- ❶ `repartition(numPartitions)` returns a new `DataFrame` partitioned by the given partitioning expressions.

Amazon S3

Amazon S3 (Simple Storage Service) is a storage web service offered by Amazon Web Services (AWS). Amazon S3 provides storage through web services interfaces. S3 objects are treated as web objects — that is, they are accessed via the Internet protocols using a URL identifier:

- Every S3 object has a unique URL, in this format:

`http://s3.amazonaws.com/bucket/key`

- An actual S3 object using this format looks like this:

`http://s3-us-east-1.amazonaws.com/project-dev/dna/sample123.vcf`

Where

- `project-dev` is the bucket name, and
- `dna/sample123.vcf` is a key.

Also S3 objects can be accessed by URI schemas such as `s3n`, `s3a`, and `s3`.

- S3 Native FileSystem (URI scheme: `s3n`) A native filesystem for reading and writing regular files on S3.
- The third generation, `s3a`: filesystem. Designed to be a switch-in replacement for `s3n`, this filesystem binding supports larger files and promises higher performance.
- S3 Block FileSystem (URI scheme: `s3`) A block-based filesystem backed by S3. Files are stored as blocks, just like they are in HDFS.

So, what are the differences between `s3`, `s3n`, and `s3a`? The difference between `s3` and `s3n/s3a` is that `s3` is a block-based overlay on top of Amazon S3, while `s3n/s3a` are not (they are object-based). The difference between `s3n` and `s3a` is that `s3n` supports objects up to 5GB in size, while

`s3a` supports objects up to 5TB and has higher performance (both are because it uses multi-part upload). `s3a` is the successor to `s3n`.

For example, using the `s3` URI schema, we can access the `sample72.vcf` file as:

```
s3://project-dev/dna/sample72.vcf
```

In general, to access any services from AWS, you have to be authenticated by AWS Services. There are many ways to get authenticated by AWS Services:

One method is to export access key and secret key from the command line:

```
export AWS_ACCESS_KEY_ID="AKIAI7405KPLUQGV0JWQ"  
export AWS_SECRET_ACCESS_KEY="LmuKE7afdasdfxK2vj1nfA0Bp"
```

Another method is to set your credentials using `SparkContext` object:

```
# spark : SparkSession  
sc = spark.sparkContext  
# set access key  
sc._jsc  
    .hadoopConfiguration()  
    .set("fs.s3.awsAccessKeyId",  
        "AKIAI7405KPLUQGV0JWQ")  
# set secret key  
sc._jsc  
    .hadoopConfiguration()  
    .set("fs.s3.awsSecretAccessKey",  
        "LmuKE7afdasdfxK2vj1nfA0Bp")
```

Reading from Amazon S3

You'll need to use the `s3`, `s3n` or `s3a` (for bigger S3 objects) URI schema for reading objects from S3.

Let `spark` be an instance of a `SparkSession`, then you may use the following to load a text file (an Amazon S3 Object) and returns a `DataFrame` (denoted as variable `df`) with a single string column named “value”.

```
s3_object_path = "s3n://bucket-name/object-path"
df = spark.read.text(s3_object_path)
```

The following example shows how to read an S3 object. First we use `boto3` library (`boto3` is the Amazon Web Services (AWS) SDK for Python, which allows Python developers to write software that makes use of Amazon services like S3 and EC2) to verify that S3 object does exist and then we read it using PySpark.

First, verify that S3 object (`s3://caselog-dev/tmp/csv_file_10_rows.csv`) does exist:

```
>>> import boto3
>>> s3 = boto3.resource('s3')
>>> bucket = 'caselog-dev'
>>> key = 'tmp/csv_file_10_rows.csv'
>>> obj = s3.Object(bucket, key)
>>> obj
s3.Object(bucket_name='caselog-dev', key='tmp/csv_file_10_rows.csv')
>>> obj.get()['Body'].read().decode('utf-8')
u'0,a,0.0\n1,b,1.1\n2,c,2.2\n3,d,\n4,,4.4\n,f,5.5\n,,\n7,h,7.7\n8,i,8.8\n9,j,9.9'
>>>
```

Now that S3 object does exist, we will load that S3 object and create a new `DataFrame[String]`:

```
>>> s3_object_path = "s3n://caselog-dev/tmp/csv_file_10_rows.csv" ①
>>> df = spark.read.text(s3_object_path) ②
>>> df.show() ③
+-----+
| value|
+-----+
| 0,a,0.0|
| 1,b,1.1|
| 2,c,2.2|
|   3,d,|
| 4,,4.4|
|   ,f,5.5|
|   ,,|
| 7,h,7.7|
| 8,i,8.8|
| 9,j,9.9|
+-----+
```

```
>>> df.printSchema ④
<bound method DataFrame.printSchema of DataFrame[value: string]>
>>>
```

- ❶ Define your S3 object path
- ❷ Use `SparkSession` (as `spark`) to load S3 object and create a `DataFrame`
- ❸ Show the content of newly created `DataFrame`
- ❹ Display the schema for newly created `DataFrame`

Writing to Amazon S3

```
>>># spark : SparkSession
>>> pairs_data = [("alex", 4), ("alex", 8),
                 ("rafa", 3), ("rafa", 6)]
>>> df = spark.createDataFrame(pairs_data,
                               ['name', 'number'])
```

Once your `DataFrame` is created, you may examine the content and its associated schema:

```
>>> df.show()
+-----+
| name | number |
+-----+
| alex |     4 |
| alex |     8 |
| rafa |     3 |
| rafa |     6 |
+-----+

>>> df.printSchema
<bound method DataFrame.printSchema of DataFrame
[
  name: string,
  number: bigint
]>
```

Next, we save data to Amazon S3 file system:

```
>>> df
    .write
    .format("csv")
    .mode("overwrite")
    .save("s3n://caselog-dev/output/pairs")
```

Then you will see that the following files created:

```
https://s3.amazonaws.com/caselog-dev/output/pairs/_SUCCESS
https://s3.amazonaws.com/caselog-dev/output/pairs/part-00000-....c000.csv
https://s3.amazonaws.com/caselog-dev/output/pairs/part-00001-....c000.csv
```

Now let's read it back:

```
>>> # Read S3 Object as Text:
>>> s3_object_path = "s3n://caselog-dev/output/pairs"
>>> df = spark.read.text(s3_object_path)
>>> df.show()
+-----+
| value|
+-----+
| alex,4|
| alex,8|
| rafa,3|
| rafa,6|
+-----+

>>> df.printSchema
<bound method DataFrame.printSchema of DataFrame[value: string]>
>>>
```

Read S3 Object as CSV:

```
>>> df2 = spark.read.format("csv").load(s3_object_path)
>>> df2.show()
+-----+
| _c0|_c1| ❶
+-----+
| alex| 4|
| alex| 8|
| rafa| 3|
```

| rafa | 6 |
+-----+

① Default column names

Read/Write Hadoop Files

What is Hadoop? Hadoop is an open source, MapReduce programming framework that supports the processing and storage of extremely huge data sets in a distributed computing environment. **Hadoop** is sponsored by the **Apache Software Foundation**.

According to Hadoop documentation: “The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly available service on top of a cluster of computers, each of which may be prone to failures.”

The Hadoop project includes these modules:

- Hadoop Common:
The common utilities that support the other Hadoop modules
- Hadoop Distributed File System (HDFS):
A distributed file system that provides high-throughput access to application data
- Hadoop YARN:
A framework for job scheduling and cluster resource management
- Hadoop MapReduce:
A YARN-based system for parallel processing of large data sets

In this section, our focus will be:

- How to read files from HDFS and create RDDs and DataFrames
- How to write RDDs and DataFrames into HDFS

I am assuming that you have access to a Hadoop cluster.

Read Hadoop Text Files

To show a complete story of reading a file from HDFS, first I will create a text file in HDFS, then I will use PySpark to read it as a DataFrame as well as an RDD.

Let `name_age_salary.csv` be a text file in a Linux file system (this file can be created with any text editor — note that \$ is the Linux operating system prompt):

```
export input_path="/pyspark_book/code/chap08/name_age_salary.csv"
$ cat $input_path
alex,60,18000
adel,40,45000
adel,50,77000
jane,40,52000
jane,60,81000
alex,50,62000
mary,50,92000
mary,60,63000
mary,40,55000
mary,40,55000
```

By using the `$HADOOP_HOME/bin/hdfs` command, next I create a `/test` directory in HDFS:

```
$ hdfs dfs -mkdir /test
```

Next I copy `name_age_salary.csv` to `hdfs://test/` directory:

```
$ hdfs dfs -put $input_path /test/
$ hdfs dfs -ls /test/
-rw-r--r-- 1 ... 140 ... /test/name_age_salary.csv
```

Next I examine the content of HDFS file:

```
$ hdfs dfs -cat /test/name_age_salary.csv
alex,60,18000
adel,40,45000
adel,50,77000
jane,40,52000
jane,60,81000
alex,50,62000
mary,50,92000
mary,60,63000
mary,40,55000
mary,40,55000
```

Now that we have created a file in HDFS, we will read it and create a DataFrame and an RDD from its content:

First, we read an HDFS file and create a DataFrame with default column names (_c0, _c1, _c2). The general format for HDFS schema URI is:

```
hdfs://<server>:<port>/<directories>/<filename>
```

```
server: NameNode-hostname
port: NameNode-port-number
```

In this example I use a Hadoop instance (as a **localhost** with Name Node using the **9000** port number), which is installed in my MacBook:

```
>>> uri = 'hdfs://localhost:9000/test/name_age_salary.csv'
>>> df = spark.read.csv(uri)
>>> df.show()
+---+---+---+
| _c0|_c1| _c2| ①
+---+---+---+
|alex| 60|18000|
|adel| 40|45000|
|adel| 50|77000|
|jane| 40|52000|
|jane| 60|81000|
|alex| 50|62000|
|mary| 50|92000|
|mary| 60|63000|
|mary| 40|55000|
```

```
|mary| 40|55000|
+----+---+-----+
```

❶ Default column names

Then examine the schema for newly created DataFrame:

```
>>> df.printSchema
<bound method DataFrame.printSchema of DataFrame
[
  _c0: string,
  _c1: string,
  _c2: string
]>
```

If you want to impose your explicit schema (column names and data types) on a DataFrame, then you may do the following:

```
>>> from pyspark.sql.types import StructType
>>> from pyspark.sql.types import StructField
>>> from pyspark.sql.types import StringType
>>> from pyspark.sql.types import IntegerType
>>>
>>> empSchema = StructType([
    ❶ StructField("name", StringType(), True),
    StructField("age", IntegerType(), True),
    StructField("salary", StringType(), True)
])
>>>
>>> uri = 'hdfs://localhost:9000/test/name_age_salary.csv'
>>> df2 = spark.read.csv(uri, schema = empSchema) ❷
>>>
>>> df2.show()
+----+---+-----+
|name|age|salary| ❸
+----+---+-----+
|alex| 60| 18000|
|adel| 40| 45000|
|adel| 50| 77000|
|jane| 40| 52000|
|jane| 60| 81000|
|alex| 50| 62000|
|mary| 50| 92000|
```

```

|mary| 60| 63000|
|mary| 40| 55000|
|mary| 40| 55000|
+---+---+-----+
>>> df2.printSchema
<bound method DataFrame.printSchema of DataFrame
[
    name: string,
    age: int,
    salary: string
]>

```

- ❶ Explicit schema definition
- ❷ Enforce an explicit schema
- ❸ Explicit column names

Also, you may read an HDFS file and create an `RDD[String]` from it:

```

>>> rdd = spark.sparkContext.textFile(uri)

>>> rdd.collect()
[
    u'alex,60,18000',
    u'adel,40,45000',
    u'adel,50,77000',
    u'jane,40,52000',
    u'jane,60,81000',
    u'alex,50,62000',
    u'mary,50,92000',
    u'mary,60,63000',
    u'mary,40,55000',
    u'mary,40,55000'
]

```

Write Hadoop Text Files

PySpark's API enable us to save our RDDs and DataFrames as files into HDFS.

First, we examine how to save an RDD into an HDFS:

```
>>> pairs = [('alex', 2), ('alex', 3),
              ('jane', 5), ('jane', 6)]
>>>
>>> rdd = spark.sparkContext.parallelize(pairs)
>>> rdd.collect()
[('alex', 2), ('alex', 3), ('jane', 5), ('jane', 6)]
>>> rdd.count()
4
>>> rdd.saveAsTextFile("hdfs://localhost:9000/test/pairs")
```

The `RDD.saveAsTextFile(path)` writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call the `toString()` method on each element to convert it to a line of text in the file.

Next, let's examine what is created in HDFS (output is formatted to fit the page):

```
$ hdfs dfs -ls hdfs://localhost:9000/test/pairs
Found 9 items
-rw-r--r-- ... 0 ... hdfs://localhost:9000/test/pairs/_SUCCESS
-rw-r--r-- ... 0 ... hdfs://localhost:9000/test/pairs/part-00000
-rw-r--r-- ... 12 ... hdfs://localhost:9000/test/pairs/part-00001
-rw-r--r-- ... 0 ... hdfs://localhost:9000/test/pairs/part-00002
-rw-r--r-- ... 12 ... hdfs://localhost:9000/test/pairs/part-00003
-rw-r--r-- ... 0 ... hdfs://localhost:9000/test/pairs/part-00004
-rw-r--r-- ... 12 ... hdfs://localhost:9000/test/pairs/part-00005
-rw-r--r-- ... 0 ... hdfs://localhost:9000/test/pairs/part-00006
-rw-r--r-- ... 12 ... hdfs://localhost:9000/test/pairs/part-00007

$ hdfs dfs -cat hdfs://localhost:9000/test/pairs/part*
('alex', 2)
('alex', 3)
('jane', 5)
('jane', 6)
```

The reason we got 8 `part-*` files is because the source RDD had 8 partitions:

```
>>> rdd.getNumPartitions()
8
```

If you want to create a single `part-*` file, then you should create a single RDD partition:

```
>>> rdd_single = spark.sparkContext.parallelize(pairs, 1)
>>> rdd_single.collect()
[('alex', 2), ('alex', 3), ('jane', 5), ('jane', 6)]
>>> rdd_single.getNumPartitions()
1
>>> rdd_single.saveAsTextFile("hdfs://localhost:9000/test/pairs_single")
```

Let's examine what is created in HDFS:

```
$ hdfs dfs -ls hdfs://localhost:9000/test/pairs_single
Found 2 items
-rw-r--r-- 0  hdfs://localhost:9000/test/pairs_single/_SUCCESS
-rw-r--r-- 48  hdfs://localhost:9000/test/pairs_single/part-00000

$ hdfs dfs -cat hdfs://localhost:9000/test/pairs_single/part-00000
('alex', 2)
('alex', 3)
('jane', 5)
('jane', 6)
```

We can save a DataFrame into HDFS by using a `DataFrameWriter`:

```
>>> pairs = [('alex', 2), ('alex', 3),
             ('jane', 5), ('jane', 6)]
>>>
>>> pairsDF = spark.createDataFrame(pairs)
>>> pairsDF.show()
+---+---+
| _1| _2|
+---+---+
|alex| 2|
|alex| 3|
|jane| 5|
|jane| 6|
+---+---+
>>> pairsDF.write.csv("hdfs://localhost:9000/test/pairs_df")
```

Next, we examine the HDFS:

```
$ hdfs dfs -ls hdfs://localhost:9000/test/pairs_df
Found 9 items
-rw-... 0 hdfs://localhost:9000/test/pairs_df/_SUCCESS
-rw-... 0 hdfs://localhost:9000/test/pairs_df/part-00000-....c000.csv
-rw-... 7 hdfs://localhost:9000/test/pairs_df/part-00001-....c000.csv
-rw-... 0 hdfs://localhost:9000/test/pairs_df/part-00002-....c000.csv
-rw-... 7 hdfs://localhost:9000/test/pairs_df/part-00003-....c000.csv
-rw-... 0 hdfs://localhost:9000/test/pairs_df/part-00004-....c000.csv
-rw-... 7 hdfs://localhost:9000/test/pairs_df/part-00005-....c000.csv
-rw-... 0 hdfs://localhost:9000/test/pairs_df/part-00006-....c000.csv
-rw-... 7 hdfs://localhost:9000/test/pairs_df/part-00007-....c000.csv

$ hdfs dfs -cat hdfs://localhost:9000/test/pairs_df/part*
alex,2
alex,3
jane,5
jane,6
```

You may save your DataFrame in different data formats into HDFS. For example to save your DataFrame as a **parquet** format, you may use the following template:

```
# df is an existing DataFrame object.
# format options are 'csv', 'parquet', 'json'
df.write.save(
    '/target/path/',
    format='parquet',
    mode='append'
)
```

Read/Write HDFS SequenceFiles

Hadoop offers to persist any file types, including “SequenceFile(s)” in HDFS. SequenceFiles are flat files consisting of binary (key, value) pairs. Hadoop defines a **SequenceFile** class as the **org.apache.hadoop.io.SequenceFile**. **SequenceFile** provides **SequenceFile.Writer**, **SequenceFile.Reader** and **SequenceFile.Sorter** classes for writing, reading and sorting respectively.

SequenceFile is the standard binary serialization format for Hadoop. It stores records of **Writable** (key, value) pairs, and supports splitting and compression. According to Hadoop documentation, **SequenceFile(s)** are a commonly used format in particular for intermediate data storage in MapReduce pipelines, since they are more efficient than text files. For details on reading and writing , refer to [Spark Documentation](#).

Read HDFS SequenceFiles

Spark supports reading **SequenceFile`'s** using the `'SparkContext.sequenceFile()'` method. For example, to read a **SequenceFile** with **Text** keys and **DoubleWritable** values in Python, we would do the following. Note that unlike Java/Scala, we do not pass the data types of (key, value) pairs to Spark API, Spark automatically converts Hadoop's **Text** to **String** and **DoubleWritable** to **Double**.

```
>>># spark: an instance of Sparksession  
rdd = spark.sparkContext.sequenceFile(path)
```

Write HDFS SequenceFiles

The PySpark's `RDD.saveAsSequenceFile()` method allows users to save an RDD of (key, value) pairs as a **SequenceFile**. For example, we can create an RDD from a Python collection, save it as a **SequenceFile**, and read it back using the following code snippet.

```
>>># spark: an instance of Sparksession  
pairs = [('key1', 10.0), ('key2', 20.0),  
          ('key3', 30.0), ('key4', 40.0)]  
rdd = spark.sparkContext.parallelize(pairs)  
rdd.saveAsSequenceFile('/tmp/sequencefile/')
```

Next, we read the newly created **SequenceFile** and convert it to and RDD of (key, value) pairs:

```
>>># spark: an instance of Sparksession  
rdd2 = spark.sparkContext.sequenceFile('/tmp/sequencefile/')
```

```
rdd2.collect()  
[(u'key1', 10.0),  
(u'key2', 20.0),  
(u'key3', 30.0),  
(u'key4', 40.0)  
]
```

Parquet Files

Parquet is a columnar data format, which is supported by many other data processing systems. According to Apache Parquet documentation: “Parquet is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model or programming language”. Characteristics of Parquet data format are :

- Self-describing (meta data included)
- Columnar format (ideal for fast analytics)
- Language-independent

Spark SQL provides support for both reading and writing Parquet files that automatically preserves the schema of the original data. When writing Parquet files, all columns are automatically converted to be nullable for compatibility reasons.

Figure 8.3 illustrates a logical table, and its associated row and column layouts.

Logical Table Representation

A	B	C
a1	b1	c1
a2	b2	c2
a3	b3	c3
a4	b4	c4

Columnar layout:

a1	a2	a3	a4	b1	b2	b3	b4	c1	c2	c3	c4
----	----	----	----	----	----	----	----	----	----	----	----

Row layout:

a1	b1	c1	a2	b2	c2	a3	b3	c3	a4	b4	c4
----	----	----	----	----	----	----	----	----	----	----	----

Figure 8-3. Logical Table, Row Layout, Column layout

Write Parquet Files

Using PySpark API, first we read a JSON file, then create a Parquet file.

Consider the following JSON file:

```
$ cat examples/src/main/resources/employees.json
>{"name":"Michael", "salary":3000}
>{"name":"Andy", "salary":4500}
>{"name":"Justin", "salary":3500}
>{"name":"Berta", "salary":4000}
```

Using DataFrameReader, JSON is read into a DataFrame object as `peopleDF`:

```
>>> input_path = "examples/src/main/resources/employees.json"
>>> peopleDF = spark.read.json(input_path)
>>> peopleDF.show()
+-----+-----+
| name | salary |
+-----+-----+
| Michael | 3000 |
| Andy | 4500 |
| Justin | 3500 |
| Berta | 4000 |
+-----+-----+

>>> peopleDF.printSchema()
root
 |-- name: string (nullable = true)
 |-- salary: long (nullable = true)

>>>
```

DataFrames can be saved as Parquet files, maintaining the schema information.

```
>>> peopleDF.write.parquet("file:///tmp/people.parquet")
```

You may view the generated Parquet file:

```
$ ls -l /tmp/people.parquet/
-rw-r--r-- ... 0 Apr 30 15:06 _SUCCESS
-rw-r--r-- ... 634 Apr 30 15:06 part-00000-...-c000.snappy.parquet
```

For testing and debugging purposes, you may create Parquet files from Python collections:

```
>>> tuples = [("alex", "Math", 97),
              ("jane", "Econ", 82),
              ("jane", "Math", 99)]
>>> column_names = ["name", "subject", "grade"]
>>> df = spark.createDataFrame(tuples, column_names) ❶
>>> df.show()
+-----+-----+
|name|subject|grade|
+-----+-----+
|alex|    Math|    97|
|jane|     Econ|    82|
|jane|    Math|    99|
+-----+-----+
>>> df.write.parquet("file:///tmp/parquet") ❷
```

- ❶ Convert your Python collection into a DataFrame
- ❷ Save your DataFrame as a set of Parquet files

Next you can view the created Parquet files:

```
$ ls -1 /tmp/parquet
_SUCCESS
part-00000-...-c000.snappy.parquet
part-00002-...-c000.snappy.parquet
part-00005-...-c000.snappy.parquet
part-00007-...-c000.snappy.parquet
```

Read Parquet Files

In this section, using PySpark, we read in the Parquet file created above. Note that, the Parquet files are self-describing so the schema is preserved. The result of loading a parquet file is also a DataFrame.

```

>>> input_path = "file:///tmp/people.parquet"
>>> parquetFile = spark.read.parquet(input_path)
>>> parquetFile.show()
+-----+-----+
|   name|salary|
+-----+-----+
|Michael|  3000|
|  Andy|  4500|
| Justin|  3500|
|  Berta|  4000|
+-----+-----+

>>> parquetFile.printSchema()
root
| -- name: string (nullable = true)
| -- salary: long (nullable = true)
>>>

```

Parquet files can also be used to create a temporary view and then used in SQL statements:

```

>>> parquetFile.createOrReplaceTempView("parquet_table") ❶
>>> query = "SELECT name, salary FROM parquet_table WHERE salary > 3800"
>>> filtered = spark.sql(query)
>>> filtered.show()
+-----+-----+
|   name|salary|
+-----+-----+
|  Andy|  4500|
|Berta|  4000|
+-----+-----+

```

❶ `parquet_table` acts as a relational table

Parquet supports collection data types (including an array data type), the following example reads a Parquet file, which uses arrays:

```

>>> parquet_file = "examples/src/main/resources/users.parquet"
>>> usersDF = spark.read.parquet(parquet_file)
>>> users.show()
+-----+-----+-----+
|   name|favorite_color|favorite_numbers|
+-----+-----+-----+

```

```

| Alyssa | null | [3, 9, 15, 20] |
|   Ben  | red  | [] |
+-----+-----+
>>> usersDF.printSchema()
root
|-- name: string (nullable = true)
|-- favorite_color: string (nullable = true)
|-- favorite_numbers: array (nullable = true)
|   |-- element: integer (containsNull = true)
>>>

```

Handling Avro Files

Read Avro Files

In this section, using PySpark, we read an Avro file and create a Dataframe from a given file. **Avro** is a data serialization system. Avro stores the data definition in JSON format making it easy to read and interpret, the data itself is stored in binary format making it compact and efficient. Avro files include markers that can be used to splitting large data sets into subsets suitable for MapReduce processing. Avro is a very fast serialization format.

Below we read an Avro file and create an associated DataFrame:

```

$ pyspark --packages org.apache.spark:spark-avro_2.11:2.4.0 ①
SparkSession available as 'spark'.
>>> path = "/pyspark_book/code/chap08/twitter.avro"
>>> df = spark.read.format("avro").load(path) ②
>>> df.show(truncate=False)
+-----+-----+-----+
|username | tweet | timestamp |
+-----+-----+-----+
|miguno | Rock: Nerf paper, scissors is fine.|1366150681|
|BlizzardCS| Works as intended. Terran is IMBA.|1366154481|
+-----+-----+-----+

```

- ① To read/write Avro data file, you have to import the required Avro libraries; the `spark-avro` module is external and not included in `spark-submit` or `pyspark` by default

- ② Read an Avro file and create a DataFrame

Write Avro Files

In this section, using PySpark, we create an Avro file from a given DataFrame (we will use a DataFrame created in previous section):

```
$ pyspark --packages org.apache.spark:spark-avro_2.11:2.4.0 ①
SparkSession available as 'spark'.

>>># df : DataFrame (created in previous section)
>>> output_path = "/tmp/avro/mytweets.avro"
>>> df.select("username", "tweet")
     .write.format("avro")
     .save(output_path) ②
>>> df2 = spark.read.format("avro").load(outputPath) ③
>>> df2.show(truncate=False)
+-----+-----+
|username | tweet
+-----+-----+
|miguno   |Rock: Nerf paper, scissors is fine.
|BlizzardCS|Works as intended. Terran is IMBA.
+-----+-----+
```

- ① To read/write Avro data file, you have to import the required Avro libraries
- ② Create an Avro file
- ③ Create a DataFrame from a created Avro file

Image Data Sources

Spark 2.4.+ enable us to read binary data (such as .jpg, png, ...), which can be useful in many machine learning algorithms such as face recognition and logistic regression. According to the Spark documentation: the image data source is used to load image files from a directory, it can load compressed image (jpeg, png, etc.) into raw image representation via ImageIO in Java

library. The loaded DataFrame has one StructType column: "image", containing image data stored as image schema.

Image Directory

Consider the following images in a directory:

```
$ ls -l chap08/images
-rw-r--r--@ ... 27295 Feb  3 10:55 cat1.jpg
-rw-r--r--@ ... 35914 Feb  3 10:55 cat2.jpg
-rw-r--r--@ ... 26354 Feb  3 10:55 cat3.jpg
-rw-r--r--@ ... 30432 Feb  3 10:55 cat4.jpg
-rw-r--r--@ ... 6641 Feb  3 10:53 duck1.jpg
-rw-r--r--@ ... 11621 Feb  3 10:54 duck2.jpg
-rw-r--r--@ ...      13 Feb  3 10:55 not-image.txt ❶
```

- ❶ Not an image

Creating a DataFrame from Images

Next, we will load all images into a DataFrame and ignore the files, which are not an image (binary type).

```
>>> images_path = '/pyspark_book/code/chap08/images'
>>> df = spark.read
       .format("image") ❶
       .option("dropInvalid", "true") ❷
       .load(images_path) ❸
>>> df.count()
6
```

- ❶ The format has to be "image"
- ❷ Drop/ignore non-image files
- ❸ Load images and create a DataFrame

Next, we examine the DataFrame's schema:

```
>>> df.printSchema()
root
 |-- image: struct (nullable = true)
 |   |-- origin: string (nullable = true) ①
 |   |-- height: integer (nullable = true) ②
 |   |-- width: integer (nullable = true) ③
 |   |-- nChannels: integer (nullable = true) ④
 |   |-- mode: integer (nullable = true) ⑤
 |   |-- data: binary (nullable = true) ⑥
```

- ① Represents the file path of the image
- ② Height of the image
- ③ Width of the image
- ④ Number of image channels
- ⑤ OpenCV-compatible type
- ⑥ Image bytes

Now, let's examine some of the columns for the created image DataFrame:

```
>>> df.select("image.origin", "image.width", "image.height")
      .show(truncate=False)
+-----+-----+-----+
|origin          |width|height|
+-----+-----+-----+
|file:///pyspark_book/.../cat2.jpg |300  |311   |
|file:///pyspark_book/.../cat4.jpg |199  |313   |
|file:///pyspark_book/.../cat1.jpg |300  |200   |
|file:///pyspark_book/.../cat3.jpg |300  |296   |
|file:///pyspark_book/.../duck2.jpg|275  |183   |
|file:///pyspark_book/.../duck1.jpg|227  |222   |
+-----+-----+-----+
```

Read/Write MS SQL Server

The **MS SQL Server** is a relational database management system from Microsoft. The system is designed and built to manage and store information as records in relational tables.

Write MS SQL Server

To write into a SQL Server table, you prepare your desired DataFrame and then write the content of your DataFrame into a table.

The following example write a DataFrame (`df`) into a new SQL table:

```
# define database URL:  
server_name = "jdbc:sqlserver://{{SERVER_ADDRESS}}"  
database_name = "my_database_name"  
url = server_name + ";" + "databaseName=" + database_name + ";"  
# define table name and username/password  
table_name = "my_table_name"  
username = "my_username"  
password = "my_password"  
  
try:  
    df.write \  
        .format("com.microsoft.sqlserver.jdbc.spark") \  
            .mode("overwrite") \  
            .option("url", url) \  
            .option("dbtable", table_name) \  
            .option("user", username) \  
            .option("password", password) \  
            .save()  
except ValueError as error :  
    print("Connector write failed", error)
```

- ❶ The JAR containing this class must be in your CLASSPATH
- ❷ The `overwrite` mode first drops the table if it already exists in the database by default

To append your DataFrame rows to an existing table, you just need to replace `mode("overwrite")` with the `mode("append")`.

Note that Spark's MS SQL connector by default uses READ_COMMITTED isolation level when performing the bulk insert into the database. If you wish to override the isolation level, use the `mssqlIsolationLevel` option as shown below.

```
.option("mssqlIsolationLevel", "READ_UNCOMMITTED")
```

Read MS SQL Server

To read from an existing MS SQL table, we may use the following code snippet:

```
jdbc_df = spark.read \
    .format("com.microsoft.sqlserver.jdbc.spark") \
    .option("url", url) \
    .option("dbtable", table_name) \
    .option("user", username) \
    .option("password", password) \
    .load()
```

Summary

- The Spark Data Sources API provides a pluggable mechanism for accessing structured data through Spark SQL. Data sources can be more than just simple pipes that convert data and pull it into Spark.
- Spark SQL's support for reading data from existing relational database tables, Apache Hive tables as well as from the popular Parquet columnar format.
- Spark provides simple API to integrate with the following data sources (partial list):
 - Hive
 - Avro
 - Parquet (columnar format)

- ORC (columnar format)
- S3
- Linux File System
- Hadoop Distributed File System
- JSON
- HBase
- CSV Files
- MongoDB
- ElasticSearch
- JDBC
- Redis
- Cassandra

Chapter 9. Ranking Algorithms

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at mpotter@oreilly.com.

This chapter provides the following two ranking algorithms and provide their associated solutions in PySpark.

- **Rank Product:** Finds ranks of items (such genes) among all items; this algorithm is a biologically motivated test for the detection of differentially expressed genes in replicated microarray experiments. Spark does not provide any API for Rank Product, so I will present a custom solution.
- **Page Rank:** PageRank is an iterative algorithm for measuring the importance of nodes in a given graph. This algorithm is used by search engines (such as Google) to find the importance of each web page. Spark API offers multiple solutions for Page Rank algorithm and one in particular by GraphFrames API—sample solution will be provided. Then, I will present two custom solutions for the Page Rank algorithm.

SOURCE CODE

NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 9](#).

Rank Product

Rank Product as an algorithm used in bioinformatics field — also known as computational biology. According to Wikipedia, “the rank product is a biologically motivated test for the detection of differentially expressed genes in replicated microarray experiments. It is a simple non-parametric statistical method based on ranks of fold changes.” Also, Rank Product is used as a geometric mean of rank of a gene.

For Rank Product, first I will explain what it is, and then I will provide a PySpark solution for it.

Introduction

To understand the Rank Product algorithm, I will provide a concrete example. Let $\{A_1, \dots, A_k\}$ be a set of (key-value) pairs where keys are unique per dataset. For example, a key can be an item, user, or a gene and a value can be number of items sold, number of friends for the user, and gene value (such as fold change or test expression). Then the **ranked product** of $\{A_1, \dots, A_k\}$ is computed based on the ranks r_i for key i across all k datasets. Typically ranks are assigned based on the sorted values of datasets.

I demonstrate the ranked product with a very simple example by using three data sets A_1, A_2, A_3 :

- $A_{\sim 1 \sim} = \{(K_{\sim 1 \sim}, 30), (K_{\sim 2 \sim}, 60), (K_{\sim 3 \sim}, 10), (K_{\sim 4 \sim}, 80)\}$

we assign the ranks based on the descending sorted values of keys, then

$\text{Rank}(A_{\sim 1 \sim}) = \{ (K_{\sim 1 \sim}, 3), (K_{\sim 2 \sim}, 2), (K_{\sim 3 \sim}, 4), (K_{\sim 4 \sim}, 1) \}$
since $80 > 60 > 30 > 10$. Note that 1 is the highest rank (assigned to the largest value).

- $A_{\sim 2 \sim} = \{(K_{\sim 1 \sim}, 90), (K_{\sim 2 \sim}, 70), (K_{\sim 3 \sim}, 40), (K_{\sim 4 \sim}, 50)\}$

we assign the ranks based on the descending sorted values of keys, then

$\text{Rank}(A_{\sim 2 \sim}) = \{ (K_{\sim 1 \sim}, 1), (K_{\sim 2 \sim}, 2), (K_{\sim 3 \sim}, 4), (K_{\sim 4 \sim}, 3) \}$
since $90 > 70 > 50 > 40$.

- $A_{\sim 3 \sim} = \{(K_{\sim 1 \sim}, 4), (K_{\sim 2 \sim}, 8)\}$

we assign the ranks based on the descending sorted values of keys, then

$\text{Rank}(A_{\sim 3 \sim}) = \{ (K_{\sim 1 \sim}, 2), (K_{\sim 2 \sim}, 1) \}$ since $8 > 4$.

Finally, therefore, the rank product of $\{A_{\sim 1 \sim}, A_{\sim 2 \sim}, A_{\sim 3 \sim}\}$ is expressed as:

$\{(K_1, \sqrt[3]{3 \times 1 \times 2}), (K_2, \sqrt[3]{2 \times 2 \times 1}), (K_3, \sqrt[2]{4 \times 4}), (K_4, \sqrt[2]{1 \times 3})\}$

Application of Rank Product

The rank product (RP) is a biologically motivated test for the detection of differentially expressed genes in replicated micro-array experiments. It is a simple non-parametric statistical method based on ranks of fold changes. In addition to its use in expression profiling, it can be used to combine ranked lists in various application domains, including proteomics, metabolomics, statistical meta-analysis, and general feature selection (Source: Wikipedia). It is shown that Rank Products (RP) a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments.

According to [4], “Rank Product, is relatively intuitive method that can be used to find differentially expressed genes as well as being used as a meta analysis method. It does not use any statistics, but rather scores genes on the bases of their ranks in multiple comparisons. The method is useful if you have very few replicates, or if you want to analyze how well the results from two studies agree.”

According to [7], “Rank Product is a non parametric statistic that detects items that are consistently highly ranked in a number of lists, for example genes that are consistently found among the most strongly upregulated genes in a number of replicate experiments. It is based on the assumption that under the null hypothesis that the order of all items is random the probability of finding a specific item among the top r of n items in a list is:

$$p = \frac{r}{n}$$

Multiplying these probabilities leads to the definition of the rank product

$$RP = (\prod_{i=1}^k \frac{r_i}{n_i})$$

where r_i is the rank of the item in the i -th list and n_i is the total number of items in the i -th list. The smaller the RP value, the smaller the probability that the observed placement of the item at the top of the lists is due to chance. The rank product is equivalent to calculating the geometric mean rank; replacing the product by the sum leads to a statistics (average rank) that is slightly more sensitive to outlier data and puts a higher premium on consistency between the ranks in various lists.”

Is This a Big Data Problem?

Consider 100 studies each with 1000,000 assays and each assay with 60,000 records. This translates to $100 \times 1000,000 \times 60,000 = 6000,000,000$ records, which is a big data.

Calculation of the Rank Product

Given n genes and k replicates, let $e_{g,i}$ be the fold change and $r_{g,i}$ the rank of gene g in the i 'th replicate.

Compute the rank product (RP) via the geometric mean:

$$RP(g) = \sqrt[k]{\prod_{i=1}^k r_{g,i}}$$

or

$$RP(g) = \sqrt[k]{\prod_{i=1}^k r_{g,i}}$$

Formalizing Rank Product

To formalize Rank Product, let's define what we need:

- Let $S = \{S_1, S_2, \dots, S_k\}$ be a set of k studies, where $k > 0$ and each study represent a micro-array experiment
- Let S_i ($i=1, 2, \dots, k$) be a study, which has an arbitrary number of assays identified by $\{A_{i1}, A_{i2}, \dots\}$
- Let each assay (can be represented as a text file) be a set of arbitrary number of records in the following format:

<gene_id><,><gene_value_as_double_data-type>

- Let $gene_id$ be in $\{g_1, g_2, \dots, g_n\}$ (we have n genes).

To find rank product of all studies $S = \{S_1, S_2, \dots, S_k\}$, first, find the mean of values per gene per study, then sort each study by their gene value and then assign a rank for each gene. For example, Let a single study to have 3 assays with the following values:

T
a
b
l
e

g
-
l
. *T*
a
b
l
e

S
i
n
g
l
e

S
t
u
d
y

A
s
s
a

y

s

Assay-1	Assay-2	Assay-3
g1,1.0	g1,2.0	g1,12.0
g2,3.0	g2,5.0	null
g3,4.0	null	g3,2.0
g4,1.0	g4,3.0	g4,15.0
...

The first step is to find the mean of values for each gene (per study):

g1, 5.0
g2, 4.0
g3, 2.0
g4, 8.0

Sorting by value will generate:

g4, 8.0
g1, 5.0
g2, 4.0
g3, 2.0

Next, we do assign a rank based on sorted values (means) of genes: the result will be (the last column is the ranked value):

g4, 8.0, 1
g1, 5.0, 2
g2, 4.0, 3
g3, 2.0, 4

The last step will be to find the rank product for each gene per study:

$S_1 = \{(g_1, r_{11}), (g_2, r_{12}), \dots\}$

$S_2 = \{(g_1, r_{21}), (g_2, r_{22}), \dots\}$

...

$S_k = \{(g_1, r_{k1}), (g_2, r_{k2}), \dots\}$

then Ranked Product of g_j =

$RP(g_j) = \sqrt[k]{\prod_{i=1}^k r_{ij}}$

or

$RP(g_j) = \sqrt[k]{\prod_{i=1}^k r_{ij}}$

PySpark Solution

Spark Solution using Java API for Rank Product is given by me at [here](#) and associated Java Spark code are given at [here](#).

The Spark solution will accept k input paths (each path represents a study, which may have any number of assay files). We use the following steps to implement the Rank Product for all genes used in these studies.

PySpark's high-level solution is presented:

Step-1:

Find the mean per gene per study (in some situations , you may apply other functions such as median or COPA score)

Step-2:

Sort the genes by value per study and then assign rank values (rank values will be 1, 2, ..., N where 1 is assigned to the highest value and N is assigned to the lowest). To sort the dataset by value, we will swap the key with value and then perform the sort. Next we use `RDD.zipWithIndex()`, which zips the RDD with its element indices

(these indices will be the ranks). Since Spark indices will start from zero, so we will add 1 when computing the ranked product.

Step-3:

Finally compute the Rank Product per gene for all studies. This can be accomplished by grouping all ranks by the key. To implement this step, we may use `RDD.groupByKey()` or `RDD.combineByKey()`. Note that, in general, `RDD.combineByKey()` is more efficient and scalable than `RDD.groupByKey()`. This has been discussed at least couple of times in other chapters). To understand `groupByKey()` and `combineByKey()` transformations in details, visit Spark's documentation. The solutions will be labeled as:

- `rank_product_using_groupbykey.py`
- `rank_product_using_combinebykey.py`

Note that the PySpark solution using `combineByKey()` is more efficient than `groupByKey()` solution. When using `combineByKey()`, intermediate values are reduced by local workers (this is called combining results at worker nodes) before being sent for the final reduction, but by using `groupByKey()`, there is no local reduction—combiners are not utilized—at all: all values are sent to one location for further processing.

Input Data Format

Each assay (can be represented as a text file) is a set of arbitrary number of records in the following format:

```
<gene_id><,><gene_value_as_double_data-type>
```

where `gene_id` is a key, which has an associated double data type value.

For demo, where K=3 (number of studies), I will use the following sample input:

```
cat /tmp/rankproduct/input/rp1.txt
K_1,30.0
K_2,60.0
K_3,10.0
K_4,80.0
```

```
cat /tmp/rankproduct/input/rp2.txt
K_1,90.0
K_2,70.0
K_3,40.0
K_4,50.0
```

```
cat /tmp/rankproduct/input/rp3.txt
K_1,4.0
K_2,8.0
```

Output Data Format

We generate output in the following format:

<gene_id><,><R><,><N>

where

R = ranked-product-among-all-input-data

N = number-of-values-participated-in-computing-rank-product

Rank Product Solution by `combineByKey()`

The solution is presented by program

`rank_product_using_combinebykey.py`, which requires the following input/output parameters:

```
# define input/output parameters:
#   sys.argv[1] = output path
#   sys.argv[2] = number of studies (K)
#   sys.argv[3] =   input path for study 1
#   sys.argv[4] =   input path for study 2
# ...
#   sys.argv[K+2] = input path for study K
```

To express Rank Product solution in PySpark using `combineByKey()` transformation, I use a driver code, which calls several Python functions. The

driver program is presented below:

```
# Create an instance of a SparkSession object
spark = SparkSession.builder.getOrCreate()

# Handle input parameters
output_path = sys.argv[1]

# K = number of studies to process
K = int(sys.argv[2])

# Define studies_input_path
studies_input_path = [sys.argv[i+3] for i in range(K)]

# Step-1: Perform Rank Product
means = [compute_mean(studies_input_path[i]) for i in range(K)]

# Step-2: Compute Rank
ranks = [assign_rank(means[i]) for i in range(K)]

# Step-3: Calculate Ranked Products
# ranked_products : RDD[(gene_id, (ranked_product, N))]
ranked_products = compute_ranked_products(ranks)

# Step-4: save the result
ranked_products.saveAsTextFile(output_path)
```

The three main steps (Step-1, Step-2, and Step-3) are discussed below:

Step-1: Perform Rank Product

To find the ranked product, we need to find the mean of gene values per study and this is accomplished by the `compute_mean()` function. To calculate the mean of values by `combineByKey()` transformation per key (as `gene_id`), I create a combiner data type as (`Double, Integer`) which denotes (`sum-of-values, count-of-values`), and finally to find the mean, we divide `sum-of-values` by `count-of-values`.

```
# Compute mean per gene for a single study = set of assays
# @param input_Path set of assay paths separated by ","
# @RETURN RDD[(String, Double)]
def compute_mean(input_path):
    # genes as string records: RDD[String]
```

```

raw_genes = spark.sparkContext.textFile(input_path)

# create RDD[(String, Double)]=RDD[(gene_id, test_expression)]
genes = raw_genes.map(create_pair)

# create RDD[(gene_id, (sum, count))]
genes_combined = genes.combineByKey(
    lambda v: (v, 1), # createCombiner
    lambda C, v: (C[0]+v, C[1]+1), # addAndCount,
    lambda C, D: (C[0]+D[0], C[1]+D[1]) # mergeCombiners
)

# now compute the mean per gene
genes_mean = genes_combined.mapValues(lambda p: float(p[0])/float(p[1]))
return genes_mean
#endif

```

Step-2: Compute Rank

To compute rank per `gene_id`, we perform the following 3 substeps:

1. Sort values based on absolute value of copa scores: to sort by COPA score, we will swap KEY with VALUE and then sort by KEY
2. Assign rank from 1 (to highest copa score) to n (to the lowest COPA score)
3. Calculate rank for each `gene_id` as `Math.power(R~1~ * R~2~ * ... * R~n~, 1/n)`

All of this step is accomplished by the `assign_rank()` function, where ranks are assigned by using `RDD.zipWithIndex()`, which zips this RDD with its element indices.

```

# @param rdd : RDD[(String, Double)] : (gene_id, mean)
# @returns: RDD[(String, Long)] : (gene_id, rank)
def assign_rank(rdd):
    # swap key and value (will be used for sorting by key)
    # convert value to abs(value)
    swapped_rdd = rdd.map(lambda v: (abs(v[1]), v[0]))

```

```

# sort copa scores descending: we need 1 partition so
# that we can zip numbers into this RDD by zipWithIndex()
# If we do not use 1 partition, then indexes will be meaningless
# sorted_rdd : RDD[(Double, String)]
sorted_rdd = swapped_rdd.sortByKey(False, 1)

# use zipWithIndex(): zip values will be 0, 1, 2, ...
# for ranking, we need 1, 2, 3, ..., therefore,
# we will add 1 when calculating the ranked product
# indexed : RDD[((Double, String), Long)]
indexed = sorted_rdd.zipWithIndex()

# add 1 to index to start with 1 rather than 0
# ranked : RDD[(String, Long)]
ranked = indexed.map(lambda v: (v[0][1], v[1]+1))
return ranked
#endif

```

Step-3: Calculate Ranked Products

Finally, this step calculates ranked products by calling the `compute_ranked_products()`, which combines all ranks into one RDD and then calculates the rank by using the `combineByKey()` transformation.

```

# return RDD[(String, (Double, Integer))] = (gene_id, (ranked_product, N))
# where N is the number of elements for computing the ranked product
# @param ranks: array of RDD[(String, Long)]
def compute_ranked_products(ranks):
    # combine all ranks into one
    union_rdd = spark.sparkContext.union(ranks)

    # next find unique keys, with their associated copa scores
    # we need 3 basic function to be able to use combineByKey()
    # combined_by_gene: RDD[(String, (Double, Integer))]
    combined_by_gene = union_rdd.combineByKey(
        lambda v: (v, 1), # createCombiner as C
        lambda C, v: (C[0]*v, C[1]+1), # multiplyAndCount
        lambda C, D: (C[0]*D[0], C[1]+D[1]) # mergeCombiners
    )

    # next calculate ranked products and the number of elements
    ranked_products = combined_by_gene.mapValues(
        lambda v : (pow(float(v[0]), float(v[1])), v[1])
    )

```

```

    )
    return ranked_products
#end-def

```

Sample Run using combineByKey()

```

INPUT1=/tmp/rankproduct/input/rp1.txt
INPUT2=/tmp/rankproduct/input/rp2.txt
INPUT3=/tmp/rankproduct/input/rp3.txt
OUTPUT=/tmp/rankproduct/ouput
./bin/spark-submit rank_product_using_combinebykey.py $OUTPUT 3 $INPUT1 $INPUT2
$INPUT3

output_path=/tmp/rankproduct/ouput
K=3
studies_input_path ['/tmp/rankproduct/input/rp1.txt',
                    '/tmp/rankproduct/input/rp2.txt',
                    '/tmp/rankproduct/input/rp3.txt']
input_path /tmp/rankproduct/input/rp1.txt
raw_genes ['K_1,30.0', 'K_2,60.0', 'K_3,10.0', 'K_4,80.0']
genes [('K_1', 30.0), ('K_2', 60.0), ('K_3', 10.0), ('K_4', 80.0)]
genes_combined [('K_2', (60.0, 1)), ('K_3', (10.0, 1)), ('K_1', (30.0, 1)), ('K_4', (80.0, 1))]
input_path /tmp/rankproduct/input/rp2.txt
raw_genes ['K_1,90.0', 'K_2,70.0', 'K_3,40.0', 'K_4,50.0']
genes [('K_1', 90.0), ('K_2', 70.0), ('K_3', 40.0), ('K_4', 50.0)]
genes_combined [('K_2', (70.0, 1)), ('K_3', (40.0, 1)), ('K_1', (90.0, 1)), ('K_4', (50.0, 1))]
input_path /tmp/rankproduct/input/rp3.txt
raw_genes ['K_1,4.0', 'K_2,8.0']
genes [('K_1', 4.0), ('K_2', 8.0)]
genes_combined [('K_2', (8.0, 1)), ('K_1', (4.0, 1))]

sorted_rdd [(80.0, 'K_4'), (60.0, 'K_2'), (30.0, 'K_1'), (10.0, 'K_3')]
indexed [((80.0, 'K_4'), 0), ((60.0, 'K_2'), 1), ((30.0, 'K_1'), 2), ((10.0, 'K_3'), 3)]
ranked [('K_4', 1), ('K_2', 2), ('K_1', 3), ('K_3', 4)]

sorted_rdd [(90.0, 'K_1'), (70.0, 'K_2'), (50.0, 'K_4'), (40.0, 'K_3')]
indexed [((90.0, 'K_1'), 0), ((70.0, 'K_2'), 1), ((50.0, 'K_4'), 2), ((40.0, 'K_3'), 3)]
ranked [('K_1', 1), ('K_2', 2), ('K_4', 3), ('K_3', 4)]

sorted_rdd [(8.0, 'K_2'), (4.0, 'K_1')]

```

```
indexed [((8.0, 'K_2'), 0), ((4.0, 'K_1'), 1)]  
ranked [('K_2', 1), ('K_1', 2)]
```

To view the final output per key:

```
$ cat /rankproduct/output/part*  
(K_2,(1.5874010519681994, 3))  
(K_1,(1.8171205928321397, 3))  
(K_4,(1.7320508075688772, 2))  
(K_3,(4.0, 2))
```

Rank Product Solution by groupByKey()

The `groupByKey()` solution is presented (as `rank_product_using_groupbykey.py`) by replacing `combineByKey()` transformations by `groupByKey()`. Overall, `combineByKey()` solution is more scalable than `groupByKey()` solution due to shuffling phase of these two transformations.

This section focuses on 2 important ideas:

- Clearly explain how PageRank is calculated
- Use Spark to perform a distributed PageRank algorithm

Page Rank Algorithm

Introduction to PageRank

The purpose of this section is to show how to apply the PageRank algorithm for a given graph (such as web pages of the internet). PageRank is an algorithm often used (as a search engine optimization technique) by search engine companies like Google and Yahoo. PageRank is one of the algorithms that search engines use to determine a page's relevance or importance with respect to other web pages. In a nutshell, PageRank algorithm is expressed as a mathematical formula.

PageRank algorithm's main message is this: a web page is important if it is pointed to by other important web pages. In a nutshell, the PageRank algorithm measures the importance of each node in a graph, assuming an edge from node u to node v represents an endorsement of v 's importance by u . For example, if a Twitter user is followed by many others, then that user will be ranked highly. Any web designer who wants to improve on Google listings, should take the time to fully understand how PageRank really works. Note that, the PageRank algorithm is a link analysis and says nothing about the language, content or size of a page.

The following Figure illustrates the concept of PageRank.

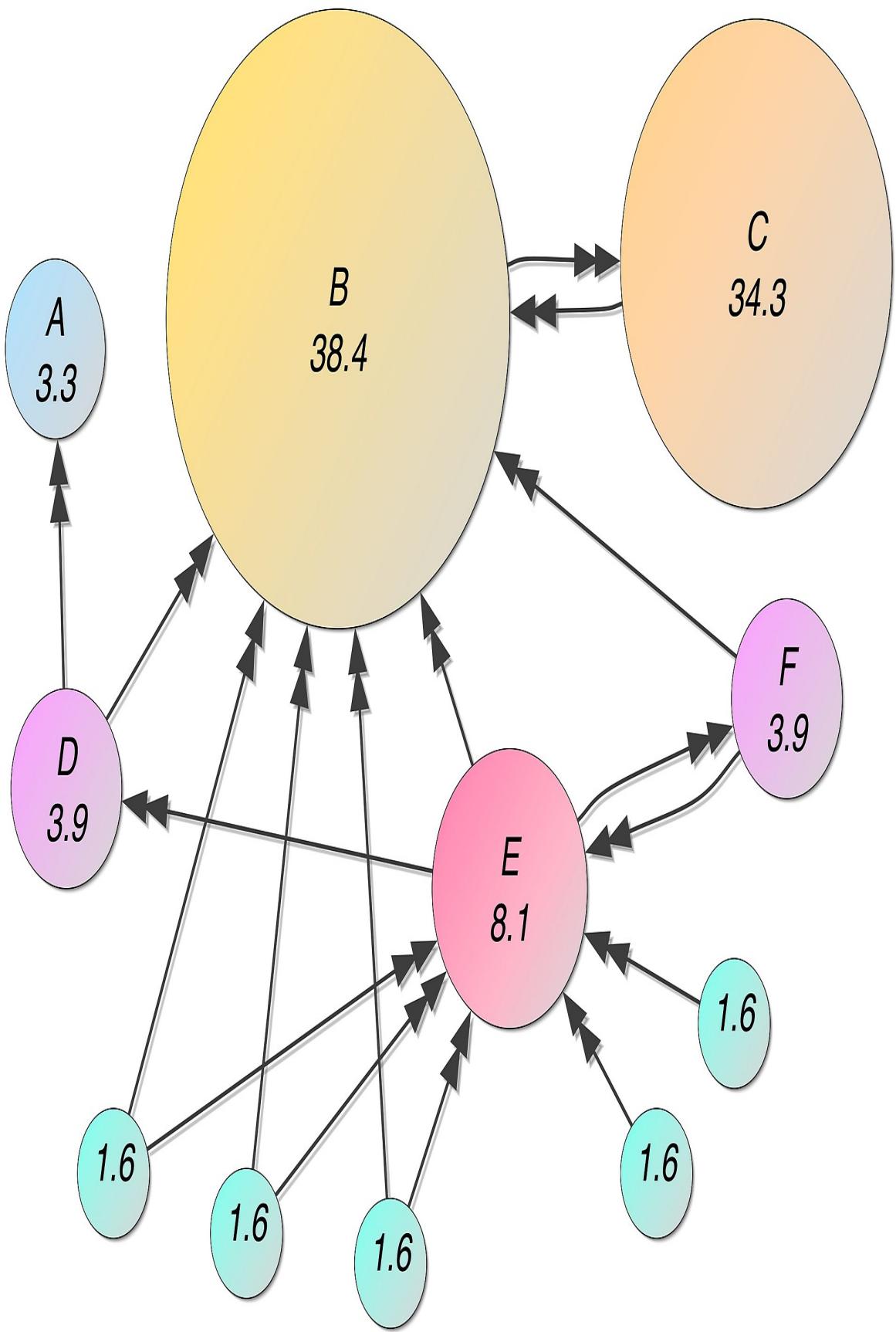


Figure 9-1. PageRanks-Example

According to Wikipedia, “Mathematical PageRanks for a simple network, expressed as percentages. Note that Google uses a logarithmic scale. In the preceding Figure, Page C has a higher PageRank than Page E, even though there are fewer links to C; the one link to C comes from an important page and hence is of high value. If web surfers who start on a random page have an 85% likelihood of choosing a random link from the page they are currently visiting, and a 15% likelihood of jumping to a page chosen at random from the entire web, they will reach Page E 8.1% of the time. The 15% likelihood of jumping to an arbitrary page corresponds to a damping factor of 85%. Without damping, all web surfers would eventually end up on Pages A, B, or C, and all other pages would have PageRank zero. In the presence of damping, Page A effectively links to all pages in the web, even though it has no outgoing links of its own.”

The PageRank algorithm is an iterative algorithm. The algorithm begins at step one with some initial PageRank (as a double data type) assigned to all pages. The PageRank algorithm converges at some point (after some iterations). This means that the PageRank algorithm is applied iteratively until it arrives at a steady state (called convergence point); that is, until a PageRank has been distributed to all pages and a subsequent iteration of the algorithm provides little or no further change (you can check the changes and stop at your desired change) in the distribution of PageRank.

Following the Google paper, PageRank is defined as: We assume page A has the following pages

$$\{T_1, \dots, T_n\}$$

which point to it (i.e., are citations). The parameter d is a damping factor which can be set between 0 and 1. We usually set d to 0.85. There are more details about d in the next section. Also $L(A)$ is defined as the number of links going out of page A. The PageRank (PR) of a page A is given as follows:

$$PR(A) = (1 - d) + d \times \left[\frac{PR(T_1)}{L(T_1)} + \dots + \frac{PR(T_n)}{L(T_n)} \right]$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one. PageRank or $PR(A)$ can be calculated using a simple iterative algorithm, and corresponds to the principal eigenvector of the normalized link matrix of the web.

Note the following definitions:

- $PR(A)$ is the PageRank of page A
- $PR(T_{\sim i})$ is the PageRank of pages $T_{\sim i}$ which link to page A
- $L(T_{\sim i})$ is the number of outbound links on page $T_{\sim i}$
- d is a damping factor which can be set between 0.00 and 1.00

Given a graph of web pages (each node is an HTML document or a page) with incoming and outgoing links, then PageRank algorithm can tell us the importance and relevance of each node. The PR of each page depends on the PR of the pages pointing to it. In short, PageRank is a “vote”, by all the other pages on the Web, about how important a page is.

Next, PageRank is explained in detail. PageRank is one of the many algorithms utilized by the search engines (for example, Google and Yahoo). The goal of PageRank Algorithm is to compute importance of a web page within all other web pages. PageRank algorithm is developed by Larry Page and Sergey Brin in their paper “The Anatomy of a Large-Scale Hypertextual Web Search Engine”. In a nutshell, PageRank algorithm indicates that the ranking or importance of a web page is determined by its estimated importance (determined by link structure) instead of by its content (PageRank is also labeled as a link analysis algorithm). If there is a link from page A to page B then this is a vote, by page A, for page B. We should note that the PageRank is just one of the many factors in overall search engine algorithms. The PageRank algorithm only relies on the in and out links of the pages and

evaluates the importance of web pages without evaluation of the content. In Google, Relevancy is determined by many factors, one of which is the PageRank for a given page.

PageRank is a hyperlink (a reference to another data or web page) analysis algorithm, “that assigns a numerical weighting to each element of a hyperlinked set of documents, such as the World Wide Web, with the purpose of “measuring” its relative importance within the set (for details see [PageRank](#)). Simply put, PageRank indicates the importance of a web page can be judged by the number of hyperlinks (not by its contents) pointing to it from other web pages (this means that if a page has a lot of incoming links then that page is an important one). For sure, Google does not just rely on the PageRank algorithm alone and uses many other factors and algorithms in their search engine.

Following PageRank definition from [Wikipedia](#), in the general case, the PageRank (PR) value for any page u can be expressed as:

$$PR(u) = \sum_{v \in B_u} \frac{PR(v)}{L(v)}$$

where

- B_u is the set containing all pages linking to page u
- $L(v)$ is the number of outbound links from page v

Why is a damping factor added? PageRank is based on the random surfer model. Essentially, the damping factor is a decay factor. What it represents is the chance that a user will stop clicking links and get bored with the current page and then request another random page (as with directly typing in a new URL rather than following a link on the current page). This was originally set to about 85% or 0.85. If the damping factor is 85% then there is assumed to be about a 15% chance that a typical users won't follow any links on the page

and instead navigate to a new random URL (source: [digitalpoint](#)). By adding a damping factor to the PageRank algorithm we get:

$$PR(p_i) = \frac{1-d}{N} + d \sum_{p_j \in B_{p_i}} \frac{PR(p_j)}{L(p_j)}$$

where

- **N** is the total number of pages
- **d** is the damping factor usually set to 0.85
- p_1, p_2, \dots, p_N are the pages under consideration
- B_{p_i} is the set of pages that link to p_i
- $L(p_j)$ is the number of outbound links on page p_j

PageRank's Iterative Computation

Given a set of N web pages, PageRank can be computed either iteratively or algebraically. The iterative method can be viewed as the power iteration method. Iteratively, it can be defined as:

At ($t = 0$), an initial probability distribution is assumed, usually

$$PR(p_i, 0) = \frac{1}{N}$$

At each time step, the computation, as detailed above, yields

$$PR(p_i, t+1) = \frac{1-d}{N} + d \sum_{p_j \in B_{p_i}} \frac{PR(p_j, t)}{L(p_j)}$$

It is believed that Google (and other search engines) recalculates PageRank scores each time it crawls the Web and rebuilds its search index. As the number of documents increased in the collection, the initial approximation of PageRank decreases for all documents. Therefore, PageRank algorithm ranks the websites by the number and quality of incoming links. The quality of an incoming link is defined as a function of the PageRank of the site which link to the other. Please note that this is an extremely simplified form of the original PageRank algorithm. Google (and others who use similar PageRank algorithms) take many other factors (keyword density, traffic, domain age etc.) into consideration for calculating the PageRank of a web page.

Next, we show how the PageRank is calculated. Consider the following simple graph:

```
$ cat simple_graph.txt
A B
B A
A D
D A
```

Therefore we can write:

$$d = 0.85$$

$$PR(A) = (1-d) + d \cdot (PR(B)/L(B) + PR(D)/L(D))$$

$$PR(B) = (1-d) + d \cdot (PR(A)/L(A))$$

$$PR(D) = (1-d) + d \cdot (PR(A)/L(A))$$

where

$$L(A) = 2$$

$$L(B) = 1$$

$$L(D) = 1$$

To calculate these $PR()$ iteratively, we need to initialize $PR(A)$, $PR(B)$, $PR(D)$: let's initialize all to 1.0. We iteratively calculate $PR(A)$, $PR(B)$, $PR(D)$ until these values do not change (i.e., converge)

T

a

b

l

e

g

-
2

.

P

a

g

e

R

a

n

k

I

t

e

r

a

t

i

o

n

S

w

i

t

h

i

n

i
t
i
a
l
v
a
l
u
e
o
f
I
.

0
0

iteration	PR(A)	PR(B)	PR(D)
0	1.0000	1.0000	1.0000
1	1.8500	0.9362	0.9362
2	1.7417	0.8902	0.8902
...
98	1.4595	0.7703	0.7703
99	1.4595	0.7703	0.7703
100	1.4595	0.7703	0.7703

Note that, no matter what values you initialize the PR's, the PageRank algorithm converges and you get the desired results: here we initialized all PR's to 40.00.

T
a
b
l
e
9

-
3

.
P
a
g
e
R
a
n
k

I
t
e
r
a
t
i
o

n
s

w
i
t
h

i
n
i

t
i
a
l
v
a
l
u
e
o
f
4
0
.
0
0

iteration	PR(A)	PR(B)	PR(D)
0	40.0000	40.0000	40.0000
1	68.1500	29.1138	29.1137
2	49.6434	21.2484	21.2484
...
98	1.4595	0.7703	0.7703
99	1.4595	0.7703	0.7703
100	1.4595	0.7703	0.7703

After 100 iterations:

```
PR(A) = 1.4595
PR(B) = 0.7703
PR(D) = 0.7703
```

Custom PageRank in PySpark — 1

For understanding the PageRank algorithm, I will provide a simple custom solution using PySpark, but for production environments, I do recommend that you use the PageRank solution from the GraphFrames or GraphX. The complete custom program and sample input data is provided:

- PySpark program: `pagerank.py`
- Sample input: `pagerank_data.txt`.

This solution uses Spark RDDs to implement the PageRank algorithm. This solution does not use GraphX or GraphFrames. Solutions using GraphX and GraphFrames are presented in the following sections. Algorithm for PageRank using Spark RDDs are presented below.

Input

Let's assume that our input has the following syntax:

```
<source-URL-ID><,><neighbor-URL-ID>
```

Output

The goal of the PageRank algorithm is to generate the following output:

```
<URL-ID> <page-rank-value>
```

Sample Output

Let's run the custom PageRank Algorithm for 15 iterations:

```
spark-submit pagerank.py pagerank_data.txt 15
1 has rank: 0.86013842528.
3 has rank: 0.33174213968.
2 has rank: 0.33174213968.
```

```
5 has rank: 0.473769824736.  
4 has rank: 0.33174213968.
```

PageRank Algorithm

The solution for the PageRank is provided by the following steps:

- **STEP-1:** Read input parameters

This step will read input path and the number of iterations:

```
input_path = sys.argv[1]  
num_of_iterations = int(sys.argv[2])
```

- **STEP-2:** Initialize the SparkSession

Here we create an instance of a `SparkSession` object:

```
spark = SparkSession.builder.getOrCreate()
```

- **STEP-3:** Create `RDD[String]` from input path:

```
records = spark.sparkContext.textFile(input_path)
```

- **STEP-4:** Create initial links

Loads all URLs from input file and initialize their neighbors.

```
def create_pair(record_of_urls):  
    # record_of_urls = "<source-URL>,><neighbor-URL>"  
    tokens = record_of_urls.split(",")  
    source_URL = tokens[0]  
    neighbor_URL = tokens[1]  
    return (source_URL, neighbor_URL)  
#end-def  
  
links = records.map(lambda rec: create_pair(rec)) ❶  
        .distinct() ❷
```

```
.groupByKey() ③  
.cache() ④
```

- ❶ create a pair of (source-URL, neighbor-URL)
 - ❷ make sure there are no duplicate pairs
 - ❸ find all neighbor URLs
 - ❹ cache it since it will be used many times in the iteration
- **STEP-5:** Initialize ranks to 1.0
Loads all URLs with other URLs link to from input file and initialize ranks of them to one.
- ```
ranks = links.map(lambda url_neighbors: (url_neighbors[0], 1.0)) ❶
```
- ❶ `ranks` is an `RDD[(String, Float)]`
- **STEP-6:** Perform iterations...  
Calculates and updates URL ranks continuously using PageRank algorithm. To perform this step, we need two basic functions:

```
def recalculate_rank(rank):
 new_rank = rank * 0.85 + 0.15
 return new_rank
#end-def

def compute_contributions(urls_rank):
 # calculates URL contributions
 # to the rank of other URLs
 urls = urls_rank[1][0]
 rank = urls_rank[1][1]
```

```

#
num_urls = len(urls)
for url in urls:
 yield (url, rank / num_urls)
#end-def

```

Now, let's perform the iterations:

```

for iteration in range(num_of_iterations):
 # calculates URL contributions
 # to the rank of other URLs.
 contributions = links
 .join(ranks) ①
 .flatMap(compute_contributions)

 # re-calculates URL ranks based
 # on neighbor contributions.
 ranks = contributions.reduceByKey(lambda x,y : x+y)
 .mapValues(recalculate_rank)
#end-for

```

- ① Note that `links.join(ranks)` will create elements of the form `[(URL-ID, (ResultIterable, <rank-as-float>)), ...]`
- **STEP-7:** Display the page ranks  
Collects all URL ranks and dump them to console.

```

for (link, rank) in ranks.collect():
 print("%s has rank: %s." % (link, rank))

```

- **STEP-8:** done

```
spark.stop()
```

## Sample Output

Sample output is provided for 20 iterations. You can observe the convergence of the PageRank algorithm in the higher iterations — iteration numbers 16 to 20.

| iteration/node | 1    | 2    | 3    | 4    | 5    |
|----------------|------|------|------|------|------|
| 0              | 1.00 | 1.00 | 1.00 | 1.00 | null |
| 1              | 2.27 | 0.36 | 0.36 | 0.36 | 0.79 |
| 2              | 0.92 | 0.63 | 0.63 | 0.63 | 0.79 |
| ...            |      |      |      |      |      |
| 8              | 0.86 | 0.36 | 0.36 | 0.36 | 0.50 |
| 9              | 0.91 | 0.33 | 0.33 | 0.33 | 0.49 |
| ...            |      |      |      |      |      |
| 16             | 0.85 | 0.33 | 0.33 | 0.33 | 0.47 |
| 17             | 0.86 | 0.33 | 0.33 | 0.33 | 0.47 |
| 18             | 0.85 | 0.33 | 0.33 | 0.33 | 0.47 |
| 19             | 0.86 | 0.33 | 0.33 | 0.33 | 0.47 |
| 20             | 0.85 | 0.33 | 0.33 | 0.33 | 0.47 |

Output is illustrated to show the convergence:

| iterations | node-1 | node-2 | node-3 | node-4 |
|------------|--------|--------|--------|--------|
| 0          | 1.0000 | 1.0000 | 1.0000 | 1.0000 |
| 3          | 2.4832 | 0.5055 | 0.5055 | 0.5055 |
| 4          | 1.4392 | 0.8535 | 0.8535 | 0.8535 |
| ...        |        |        |        |        |
| 11         | 2.0727 | 0.6424 | 0.6424 | 0.6424 |
| 12         | 1.7882 | 0.7373 | 0.7373 | 0.7372 |
| ...        |        |        |        |        |
| 100        | 1.9190 | 0.6937 | 0.6937 | 0.6937 |
| ...        |        |        |        |        |
| 120        | 1.9190 | 0.6937 | 0.6937 | 0.6937 |
| ...        |        |        |        |        |
| 130        | 1.9190 | 0.6937 | 0.6937 | 0.6937 |

## PageRank by GraphFrame

**GraphFrames** is an external package for Spark which provides DataFrame-based Graphs (Spark's GraphX provides RDD-based Graphs). GraphFrames aims to provide both the functionality of GraphX and extended functionality taking advantage of Spark DataFrames. This extended functionality includes

motif finding (finding structural patterns such as triangles), DataFrame-based serialization, and highly expressive graph queries. PageRank solutions are provided below.

The following example in PySpark shows how to find PageRank of a graph using GraphFrames package. There are at least two ways to calculate the page rank of a given graph:

- **Tolerance**: you may compute the page rank by providing the tolerance - the tolerance allowed at convergence. Note that the smaller tolerance means more accuracy of the page rank results:

```
build graph as a GraphFrame instance
graph = GraphFrame(vertices, edges)
#
NOTE: You cannot specify maxIter()
and tol() at the same time.
damping factor = 1 - 0.15 = 0.85
tol - the tolerance allowed at convergence
(smaller => more accurate).
pagerank is computed as a GraphFrame:
pagerank = graph
 .pageRank()
 .resetProbability(0.15)
 .tol(0.0001)
 .run()
```

- **Maximum Iterations**: you may compute the page rank by providing the maximum number of iterations. Note that the higher iterations means more accuracy of the page rank results:

```
build graph as a GraphFrame instance
graph = GraphFrame(vertices, edges)
#
NOTE: You cannot specify maxIter()
and tol() at the same time.
damping factor = 1 - 0.15 = 0.85
maxIter - the max. number of iterations
(higher => more accurate).
pagerank is computed as a GraphFrame:
pagerank = graph
 .pageRank()
 .resetProbability(0.15)
```

```
.maxIter(30)
.run()
```

## Custom PageRank in PySpark — 2

This is another custom solution for PageRank algorithm, which uses adjacency matrix as an input. An adjacency matrix is a matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of nodes are adjacent or not in the graph. For example, if node A links to other 3 nodes (say B, C, and D), then that is presented as an adjacency matrix row:

A B C D

For example, for the following graph (Figure 7.15),



Figure 9-2. Directed Graph with 5 Nodes

adjacency matrix is presented below — note that the first item in each row is the source node and 2nd, 3rd, ... are target nodes.

A B C D  
B C E  
C A D E  
D E  
E B

As the graph indicates, it seems that node E is an important page since many nodes are referencing that node from many nodes. Therefore, page rank of node E will be higher than other nodes. This observation proves that the

importance of a page depends on the number of other pages pointing (votes) to it.

For understanding the PageRank algorithm, I will provide another simple custom solution using PySpark, but for production environments, I do recommend that you use the PageRank solution from the GraphFrames or GraphX. The complete custom program and sample input data is provided:

- PySpark program: `pagerank_2.py`
- Sample input: `pagerank_data_2.txt`.

This solution uses Spark RDDs to implement the PageRank algorithm. This solution does not use GraphX or GraphFrames. Algorithm for PageRank using Spark RDDs are presented below. Before preseting the PageRank algorithm, let's understand a page rank for a given node: assume page A has pages  $\{T_1, \dots, T_n\}$  which point to it (i.e., are citations). The parameter  $d$  is a damping factor which can be set between 0 and 1 (usually set  $d$  to 0.85). Let  $C(A)$  be defined as the number of links going out of page A. Then PageRank of a page A is given as follows:

$$PR(A) = (1-d) + d [PR(T_1)/C(T_1) + \dots + PR(T_n)/C(T_n)]$$

Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one.

Let  $N$  be the number of nodes in a given graph, then the PageRank algorithm can be implemented as following:

1. Initialize each page's (as a node in graph) importance (Rank) as  $1/N$ .
2. Successively update each page's Rank according to the following formulation:

$$PR_i = \frac{1-d}{N} + d \sum_{j \in \{1, \dots, N\}} \frac{PR_j}{C_j}$$

1. Repeat (iterate) the second step until the page ranks converges.

## Input

Let's assume that our input has the following syntax:

```
<source-node><S><target-node-1><S><target-node-2><S>...
```

where S is a single space

## Output

The goal of the PageRank algorithm is to generate the following output:

```
<node> <page-rank-value>
```

## Solution in PySpark

PySpark solution is presented in three main step:

*Map-step:*

For each node  $i$ , calculate vote ( $R_i^{\sim}/D_i^{\sim}$ ) for each out-link of  $i$  and propagate to adjacent nodes.

*Reduce-step:*

For each node  $i$ , sum the upcoming votes and update Rank value ( $R_i^{\sim}$ ).

*Iteration:*

Repeat this Map-Reduce step until Rank values converge (stable or within a margin).

The complete PySpark solution is presented below:

1.. First import required libraries and read input parameters:

```
from __future__ import print_function
import sys
from pyspark.sql import SparkSession

DEFINE your input path
```

```

input_path = sys.argv[1]
print("input_path: ", input_path)
input_path: pagerank_data_2.txt

DEFINE number of iterations
ITERATIONS = int(sys.argv[2])
print("ITERATIONS: ", ITERATIONS)
ITERATIONS: 40

```

2.. Read matrix and create pairs of (K, V), where K is the source node and V is list of target nodes

```

Create an instance of a SparkSession object
spark = SparkSession.builder.getOrCreate()

Read adjacency list and create RDD[String]
matrix = spark.sparkContext.textFile(input_path)
print("matrix=", matrix.collect())
#matrix= ['A B C D', 'B C E', 'C A D E', 'D E', 'E B']
x = "A B C"
returns (A, [B, C])
def create_pair(x):
 tokens = x.split(" ")
 # tokens[0] : source node
 # tokens[1:] : target nodes (links from the source node)
 return (tokens[0], tokens[1:])
#end-def

create links from source node to target nodes
links = matrix.map(create_pair)
print("links=", links.collect())
links= [('A', ['B', 'C', 'D']),
('B', ['C', 'E']),
('C', ['A', 'D', 'E']),
('D', ['E']),
('E', ['B'])]

```

3.. Page Rank algorithm is implemented in 3 steps:

*Step-1: Map*

For each node i, calculate vote ( $R_{\sim i} / D_{\sim i}$ ) for each out-link of i and propagate to adjacent nodes.

### Step-2: Reduce

For each node  $i$ , sum the upcoming votes and update Rank value ( $R_i$ ).

### Step-3: Iteration

Repeat this Map-Reduce step (Step-1 and Step-2) until Rank values converge (stable or within a margin).

```
Find node count
N = links.count()
print("node count N=", N)
node count N= 5

Create and initialize the ranks
ranks = links.map(lambda node: (node[0], 1.0/N))
print("ranks=", ranks.collect())
#ranks= [('A', 0.2), ('B', 0.2), ('C', 0.2), ('D', 0.2), ('E', 0.2)]

for i in range(ITERATIONS):
 # Join graph info with rank info and propagate to
 # all neighbors rank scores (rank/(number of neighbors))
 # And add up ranks from all in-coming edges
 ranks = links.join(ranks) \
 .flatMap(lambda x : [(i, float(x[1][1])/len(x[1][0]))) for i in x[1][0]]) \
 .reduceByKey(lambda x,y: x+y)
 print(ranks.sortByKey().collect())
```

Partial output is given:

```
[('A', 0.0667), ('B', 0.2667), ('C', 0.1667), ('D', 0.1334), ('E', 0.3667)]
[('A', 0.0556), ('B', 0.3889), ('C', 0.1556), ('D', 0.0778), ('E', 0.3223)]
[('A', 0.0518), ('B', 0.3407), ('C', 0.2130), ('D', 0.0704), ('E', 0.3241)]
...
[('A', 0.0638), ('B', 0.3404), ('C', 0.1915), ('D', 0.0851), ('E', 0.3191)]
[('A', 0.0638), ('B', 0.3404), ('C', 0.1915), ('D', 0.0851), ('E', 0.3191)]
[('A', 0.0638), ('B', 0.3404), ('C', 0.1915), ('D', 0.0851), ('E', 0.3191)]
```

The PageRank algorithm clearly indicate the node E is the most important page among all pages, since its page rank (0.3191) is the highest. Finally, Note that the PageRanks form a probability distribution over web pages, so the sum of all web pages' PageRanks will be one:

$\text{PR(A)} + \text{PR(B)} + \text{PR(C)} + \text{PR(D)} + \text{PR(E)} =$   
 $0.0638 + 0.3404 + 0.1915 + 0.0851 + 0.3191 =$   
 $0.9999$

## References

Rank Product

Geometric mean

Rank products: a simple, yet powerful, new method to detect differentially regulated genes in replicated microarray experiments. By Rainer Breitling, Patrick Armengaud, Anna Amtmann, Pawel Herzyk, 11 August 2004

Rank Product

Avoid groupByKey: [http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best\\_practices/prefer\\_reduceByKey\\_over\\_groupByKey.html](http://databricks.gitbooks.io/databricks-spark-knowledge-base/content/best_practices/prefer_reduceByKey_over_groupByKey.html)

Fast Data Processing with Spark - Second Edition, By Krishna Sankar and Holden Karau

Bioconductor RankProd Package Vignette:

<http://www.bioconductor.org/packages/release/bioc/vignettes/RankProd/in/st/doc/RankProd.pdf>

# Chapter 10. Fundamental Data Design Patterns

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

This chapter discusses five fundamental data design patterns:

1. Input-Map-Output
2. Input-Filter-Output
3. Input-Map-Reduce-Output
4. Input-Multiple-Maps-Reduce-Output
5. Input-Map-Combiner-Reduce-Output
6. Input-MapPartitions-Reduce-Output
7. Input-Inverted-Index-Pattern-Output

Even though, I have used all of them in many of my examples, but they deserve an attention since these are foundations of data algorithms and data

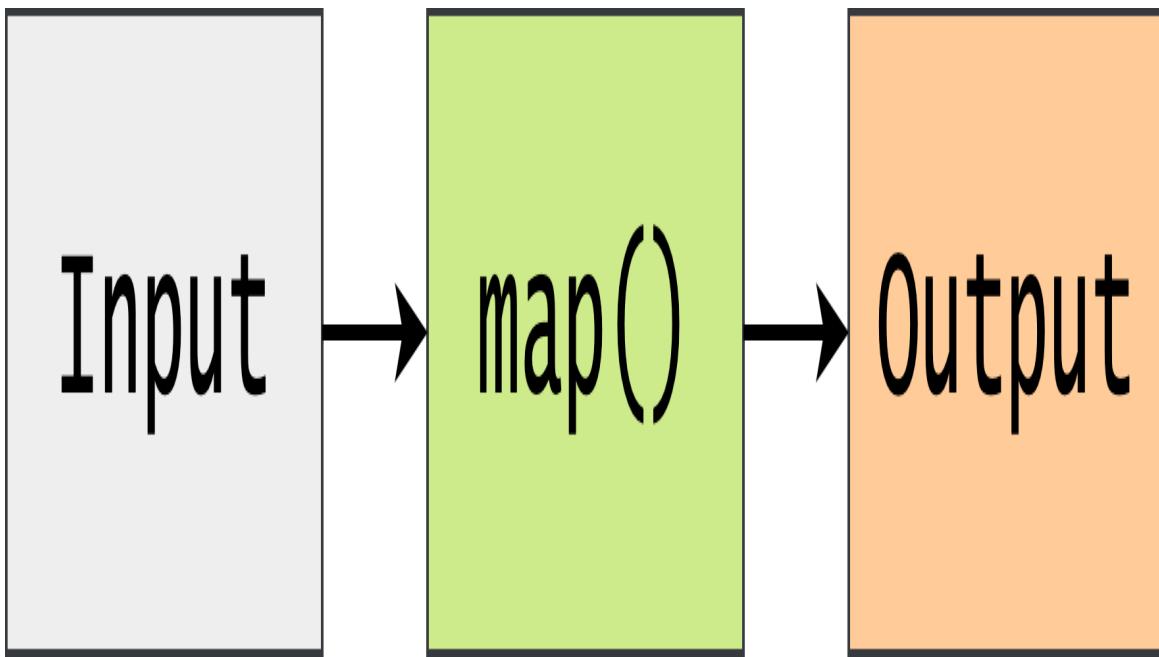
design patterns. Some of fundamental MapReduce design patterns are presented in [MapReduce: Simplified Data Processing on Large Clusters](#). For implementation of design patterns, I will use PySpark rather than MapReduce paradigm— note that MapReduce paradigm has a very limited functionality such as `map()` and `reduce()` functions. But Spark's implementation is not limited just to two functions — Spark is a superset of MapReduce and offers very rich API for solving data problems.

### NOTE

The data design patterns mentioned in this chapter are basic patterns. You can make your own as per requirement of the problem.

## Input-Map-Output

This is the simplest data design pattern: read the input from a set of files, then apply a series of functions to each record, and finally output your desired output. There is no restriction on what a mapper can create from its input: it can create a set of new records or `(key, value)` pairs. No reduction is involved for this design pattern. This data design pattern is illustrated by Figure 18.1.



*Figure 10-1. Input-Map-Output Design Pattern*

Even though this is the simplest design pattern, but it is a very common pattern. The purpose of this design pattern is to change the format of input data and generate output data, which can be used by other mappers and reducers. Some times, the map phase is used to clean and reformat data and then possibly generate (`key`, `value`) pairs to be consumed by reducers.

Consider a scenario where input record has a gender field such as:

```
"0", "f", "F", "Female", "female" (for female)
"1", "m", "M", "Male", "male" (for male)
```

and you want to normalize the gender filed as `{"female", "male", "unknown"}`. Let's assume that each record has the following format:

```
<gender>,<rest-of-record>
```

Then the following function can facilitate the `map` transformation:

```
rec: an input record
def normalize_gender(rec):
 tokens = rec.split(",")
 gender = tokens[0].lower()
```

```

if gender in ['0', 'f', 'female']:
 normalized_rec = "female," + tokens[1]
elif in ['1', 'm', 'male']:
 normalized_rec = "male," + tokens[1]
else:
 normalized_rec = "unknown," + tokens[1]
#
return normalized_rec
#end-def

```

Another scenario can be analyzing movie rating records such as <user\_id><,><movie\_id><,><rating> and your goal is to create a (key, value) pair of (<movie\_id>, (<user\_id>, <rating>)) per record. Further assume that all rating will be converted to integer numbers. Then you may use the following mapper function:

```

rec: <user_id><,><movie_id><,><rating>
def create_pair(rec):
 tokens = rec.split(",")
 user_id = tokens[0]
 movie_id = tokens[1]
 rating = int(tokens[2])
 #
 return (movie_id, (user_id, rating))
#end-def

```

What if you want to map a single input record/element into multiple target elements. How about if you want to drop the entire input record/element — filter out the entire input record. To map a single record into multiple target elements, Spark offers the `flatMap()` transformation, which works on a single element (as `map()`) and produces multiple elements of the target elements. Therefore, if your input is `RDD[V]`, and want to map each `V` into a set of `T` data type elements, then you may use the Spark's `flatMap()` as:

```

source_rdd: RDD[V]
target_rdd: RDD[T]
target_rdd = source_rdd.flatMap(custom_map_function)

v : an element of source_rdd
def custom_map_function(v):
 t = <use-v-to-create-an-iterable-of-T-data-type-elements>

```

```
 return t
#end-def
```

For example if for input record v, you create `t = [t1, t2, t3]`, then v will be mapped to 3 elements of the `target_rdd` as t1, t2, and t3 and if `t= []` — an empty list — then no element will be created in the `target_rdd` — v is filtered out.

If you want to map and filter at the same time — map some records and filter some records at the same time — then you may implement this feature by Spark's `flatMap()` transformation. For example, you have records in the following formats:

```
<word1><,><word2><;><word1><,><word2><;>...<word1><,><word2>
```

Suppose the goal is to keep the records if and only if there are two words — as bigrams — which are separated by comma, otherwise drop — filter out — the record.

The following PySpark snippet shows how to achieve this — therefore, we would like to keep '`w1,w2`', '`w3,w4`', '`w5,w6`', '`w7,w8`', and '`w10,w11`' but to drop '`w0`' and '`w9`'.

```
records = ['w1,w2;w3,w4', 'w0', 'w9', 'w5,w6;w7,w8;w10,w11']
rdd = spark.sparkContext.parallelize(records)

def map_and_filter(rec):
 if ";" in rec:
 bigrams = rec.split(";")
 result = []
 for bigram in bigrams:
 words = bigram.split(",")
 if len(words) == 2: result.append(bigram)
 return result
 else:
 # no semicolon in rec
 words = rec.split ","
 if len(words) == 2: return [rec]
 else: return []
#end-def
```

```
mapped_and_filtered = rdd.flatMap(map_and_filter)
mapped_and_filtered.collect()
>>> mapped_and_filtered.collect()
['w1,w2', 'w3,w4', 'w5,w6', 'w7,w8', 'w10,w11']
```

Therefore, we were able to map proper records into multiple target elements and to filter out improper records at the same time by a single `flatMap()` transformation.

## Input-Filter-Output

This data design pattern keeps your desired records — which satisfy your data requirements — and removes the unwanted records.

This design pattern reads the input from a set of files, then applies a series of filter function(s) to each record, keeps the record which satisfy the boolean predicate, otherwise drops the records.

This data design pattern is illustrated by Figure 18.2.

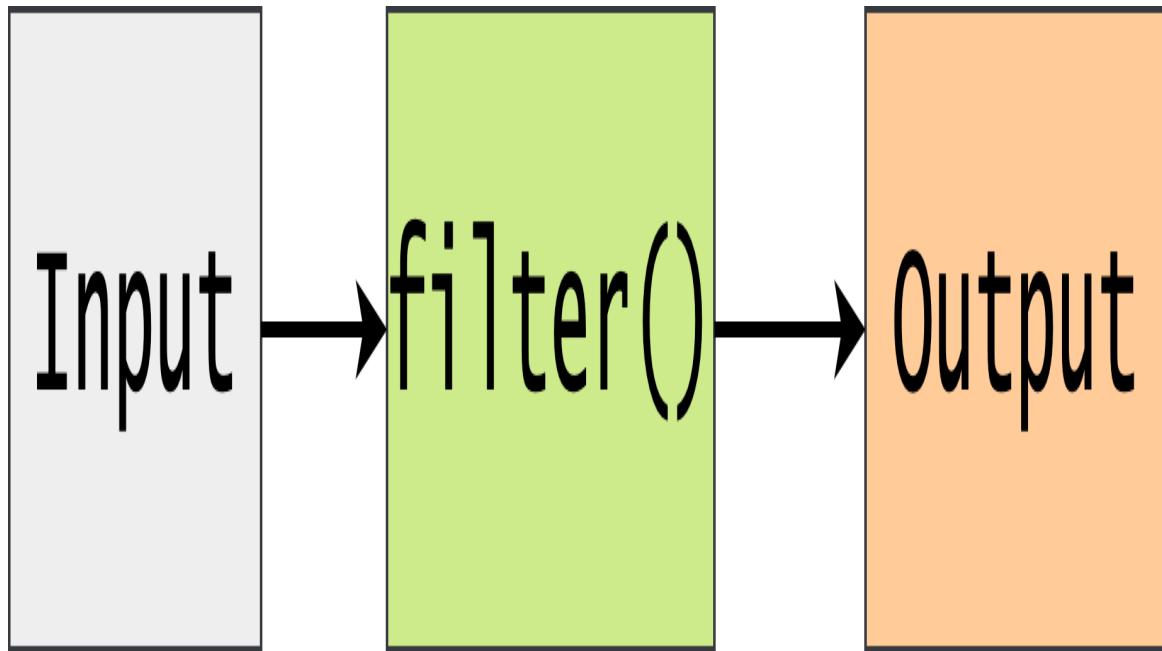


Figure 10-2. Input-Filter-Output Design Pattern

What is the motivation for using the `filter()` design pattern? When your data set is large and you want to take a subset of this data to focus in on it and

may be perform a follow-on analysis.

The simple scenario will be reading input file(s) which denote URLs and then keep the valid URLs and discard non-URLs. This design pattern can be implemented by RDDs and DataFrames.

Imagine input for this scenario has the following record format: each record is a text, supposed to be a URL.

Sample records are listed:

```
http://cnn.com ❶
http://mysite.com ❷
http://www.oreilly.com ❸
https://www.oreilly.com ❹
```

- ❶ keep valid URL
- ❷ drop invalid URL
- ❸ keep valid URL
- ❹ drop invalid URL

## RDD Solution

This design pattern can be easily implemented by `RDD.filter()` function:

```
data = ['http://cnn.com', 'http://mysite.com',
 'http://www.oreilly.com', 'https://www.oreilly.com']

urls = spark.sparkContext.parallelize(data)

return True if a given URL is valid, otherwise return False
def is_valid_URL(url_as_str):
 if (url_as_str is valid):
 return True
 else:
 return False
#end-def
```

```

return a new RDD containing only the
elements that satisfy a predicate.
valid_urls = urls.filter(is_valid_URL)
valid_urls.collect()
['http://cnn.com', 'http://www.oreilly.com']

```

## DataFrame Solution

The solution with DataFrame uses `DataFrame.filter()` function to keep desired records and drop out undesired records.

```

>>> data = [('http://cnn.com' ,), ('http://mysite.com' ,), ('http://www.oreilly.com' ,),
('https://www.oreilly.com',)]

create a single column DataFrame
>>> df = spark.createDataFrame(data, ['url'])

>>> df.show(truncate=False)
+-----+
|url |
+-----+
|http://cnn.com |
|http://mysite.com |
|http://www.oreilly.com|
|https://www.oreilly.com|
+-----+

filter out undesired records
>>> df.filter(df.url.startswith('http://') |
df.url.startswith('https://')).show(truncate=False)
+-----+
|url |
+-----+
|http://cnn.com |
|http://www.oreilly.com|
+-----+

```

## Input-Map-Reduce-Output

If we want to perform an aggregation operation — such as summation or average --, then this pattern is used. This is the most common design pattern for aggregation of values based on some key values. This data design pattern

is illustrated by Figure 18.2. Spark offers the following powerful solutions for implementing this design pattern:

- map phase: `map()`, `flatMap()`, `mapPartitions()`, `filter()`
- reduce phase: `reduceByKey()`, `groupByKey()`,  
`aggregateByKey()`, `combineByKey()`

Many combinations of map and reduce phases can be used to solve data problems.

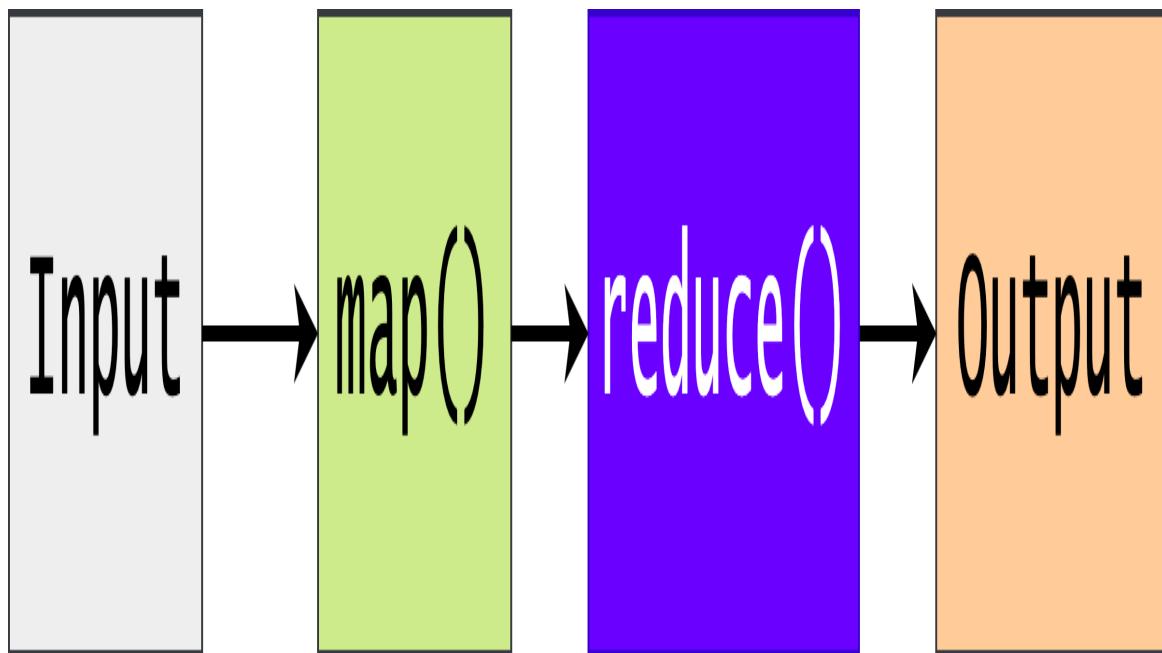


Figure 10-3. Input-Map-Reduce-Output Design Pattern

This is the simplest MapReduce design pattern: read data, perform a map transformation—usually create `(key, value)` pairs—, and then aggregate (such as sum, average, ...) all of the values for the same key; finally save the output.

Consider this scenario: you have records with format of `<name><, ><age>` `<, ><salary>` and you want to compute the average salary per age group and age groups are defined as 0-15, 16-20, 21-25, ..., 96-100+. Therefore, this scenario can be defined as:

- Read input

- `map()`: the mapper processes one record at a time and creates (key, value) pair, where key is an age group and value is the salary. For example, if our record is `alex,22,45000`, then the mapper will create ('21-25', 45000) since age 22 falls into age group of 21-25. The mapper function can be expressed as:

```
rec: <name>,<><age>,<><salary>
def create_key_value_pair(rec):
 tokens = rec.split(",")
 age = int(tokens[1])
 salary = tokens[2]
 if age < 16: return ('0-15', salary)
 if age < 21: return ('16-20', salary)
 ...
 if age < 91: return ('85-90', salary)
 if age < 96: return ('91-95', salary)
 return ('96-100', salary)
#end-def
```

- `reduce()`: the reducer will group keys by age-groups (0-15, 16-20, ...) and then aggregates and find the average salary per group.

To understand this design pattern, consider the following input:

```
alex,22,45000
bob,43,50000,
john,23,65000
jane,41,48000
joe,44,66000
```

Then the mapper will generate the following (key, value) pairs:

```
('21-25', 45000)
('41-45', 54000)
('21-25', 67000)
('41-45', 68000)
('41-45', 70000)
```

Finally, the reducer will group keys by their associated values and find the average salary per age-group:

```
('21-25', [45000, 67000])
('41-45', [54000, 68000, 70000])
```

Grouping by key can be easily implemented by Spark's `groupByKey()` transformation: Let and `rdd` denote  $\text{RDD}[(\text{key}, \text{value})]$ , then using Spark's `groupByKey()`, we may write the reducer as:

```
rdd: RDD[(age-group, salary)]
grouped_by_age_group = rdd.groupByKey()
```

At last, average can be calculated per age group

```
('21-25', 56000)
('41-45', 64000)
```

This can be accomplished by another simple mapper:

```
grouped_by_age_group: RDD[(age-group, [salary-1, salary-2, ...])]
age_group_average = grouped_by_age_group.mapValues(lambda v: sum(v)/len(v))
```

If you desire to use combiners (Spark uses combiners automatically in `reduceByKey()`), then our solution can be revised as:

Then the mapper will generate the following (key, value) pairs, where value is (`sum`, `count`). The reason for creating (`sum`, `count`) as a value is to guarantee that the reducer function is "associative" and "commutative". If your reducer function does not follow these two algebraic rules, then Spark's `reduceByKey()` will not produce the correct semantics using many partitions per input data.

```
('21-25', (45000, 1))
('41-45', (54000, 1))
('21-25', (67000, 1))
('41-45', (68000, 1))
('41-45', (70000, 1))
```

Let and `rdd` denote  $\text{RDD}[(\text{key}, (\text{sum}, \text{count}))]$ , then using Spark's `reduceByKey()` — note the this reducer works on partition-by-partition

basis and uses combiners as well --, we may write the reducer as:

```
(key, (sum, count))
reduced_by_age_group = rdd.reduceByKey(
 lambda x, y: (x[0]+y[0], x[1]+y[1]))
```

Finally, the reducer will reduce keys by their associated values and find the average salary per age-group:

```
('21-25', (112000, 2))
('41-45', (192000, 3))
```

At last, average can be calculated per age group by another simple mapper:

```
('21-25', 56000)
('41-45', 64000)
```

We may solve this design pattern by a combination of Spark's `map()` and `combineByKey()` transformations. The map phase is exactly as we presented before. Therefore the `map()` phase — using `create_key_value_pair()` function — will create the following (`key, value`) pairs:

```
('21-25', 45000)
('41-45', 54000)
('21-25', 67000)
('41-45', 68000)
('41-45', 70000)
```

Let's assume that these (`key, value`) pairs are denoted by `age_group_rdd`. Then we perform the reduction by a pair of `combineByKey()` and `mapValues()` transformations:

```
C denotes (sum-of-salaries, count-of-salaries)
combined = age_group_rdd.combineByKey(
 lambda v : (v, 1), ❶
 lambda C, v: (C[0]+v, C[1]+1), ❷
 lambda C1,C2: (C1[0]+C2[0], C1[1]+C2[1]) ❸
)
```

```
c denotes (sum-of-salaries, count-of-salaries)
avg_per_age_group = combined.mapValues(
 lambda c : c[0]/c[1]
)
```

- ① Create a C as (sum-of-salaries, count-of-salaries)
- ② Merge a salary into C
- ③ Combine two C's (from different partitions) into a single C.

### TIP

Notice that `reduceByKey()` is a special case of `combineByKey()`. For `reduceByKey()`, source and target RDDs must be as `RDD[(K, V)]`. While for `combineByKey()`, the source RDD can be `RDD[(K, V)]` and target RDD can be `RDD[(K, C)]` where V and C may be different data types.

## Input-Multiple-Maps-Reduce-Output

This design pattern involves multiple maps, joins, and reductions. This design pattern is also known as “Reduce-Side-Join” — in MapReduce paradigm, the reduce side join, the reducer is responsible for performing the join operation. To understand this design pattern, let me provide an example: consider the following two inputs:

Input-1: Movies table

| <b>Movie-ID</b> | <b>Movie-Name</b>  |
|-----------------|--------------------|
| 100             | Lion King          |
| 200             | Star Wars          |
| 300             | Fidler on the Roof |
| ...             | ...                |

Input-2: Rating table

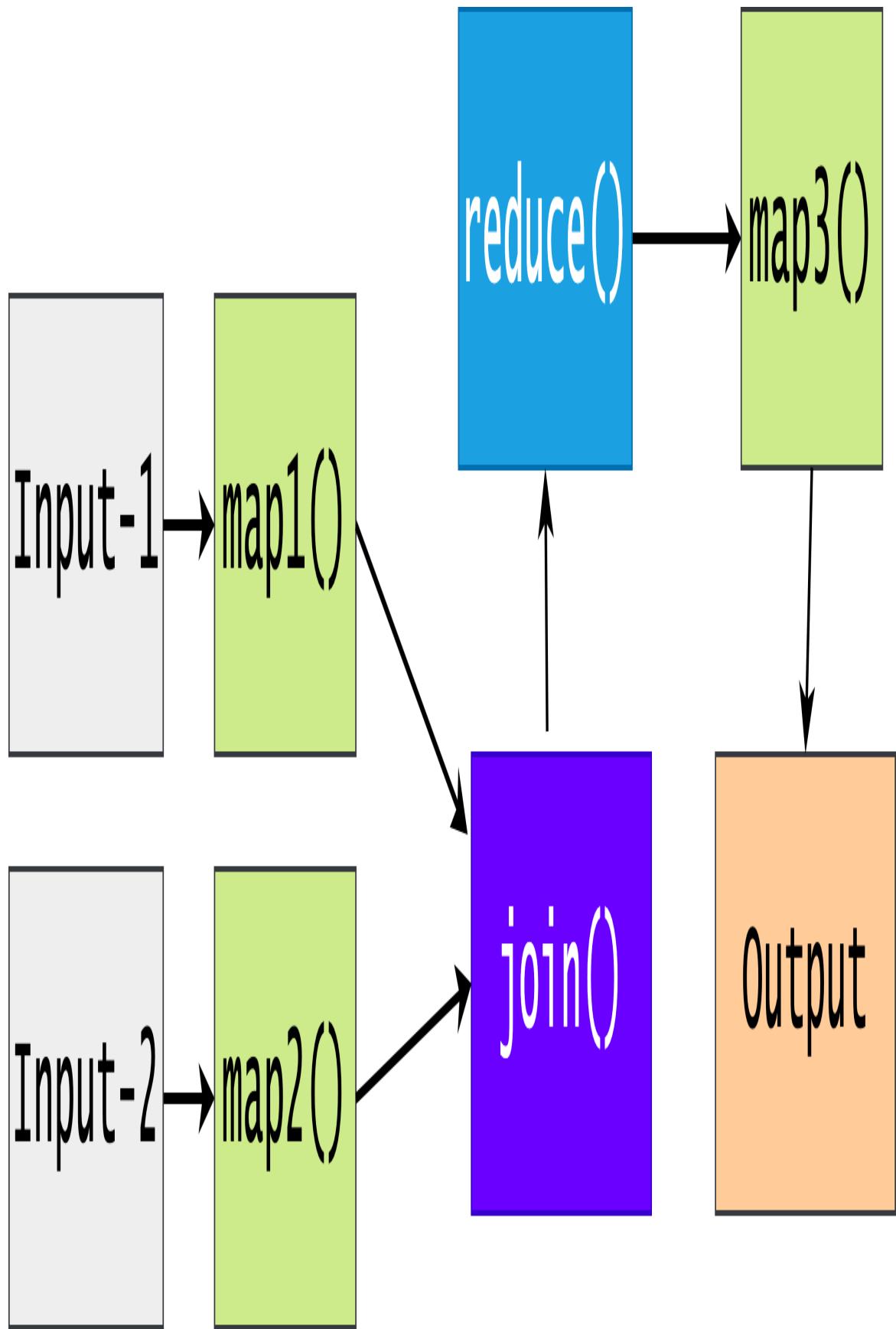
| <b>Movie-ID</b> | <b>Rating</b> | <b>User-ID</b> |
|-----------------|---------------|----------------|
| 100             | 4             | USER-1234      |
| 100             | 5             | USER-3467      |
| 200             | 4             | USER-1234      |
| 200             | 2             | USER-1234      |
| ...             | ...           | ...            |

The final goal is to produce the following output, which is the join of two tables: Movies and Ratings. After join operation is completed, we still need another reduction to find the average rating per Movie-ID.

Output: AVG Rating table

| <b>Movie-ID</b> | <b>Movie-Name</b> | <b>Avg Rating</b> |
|-----------------|-------------------|-------------------|
| 100             | Lion King         | 4.5               |
| 200             | Star Wars         | 3.0               |
| ...             | ...               | ...               |

This data design pattern is illustrated by Figure 18.4.



*Figure 10-4. Input-Map-Reduce-Output Design Pattern*

Let me explain this design pattern step by step:

- Mapper reads the input data which are to be combined based on common column or join key. Read Input-1, then apply `map1()` as a mapper and create (Common-Key, Rest-of-Attributes). Let's apply this to Input-1 — Movies table --, which we will create (`Movie-ID`, `Movie-Name`) pairs, where `Movie-ID` is a key.
- Read Input-2, then apply `map2()` as a mapper and create (Common-Key, Rest-of-Attributes). Let's apply this to Input-2 — Ratings table --, which we will create (`Movie-ID`, (`Movie-Name`, `Rating`)) pairs, where `Movie-ID` is a key and (`Movie-Name`, `Rating`) is a value.
- Next is `join()` operation between outputs of `map1()` and `map2()`. Therefore the goal is to join (`Movie-ID`, `Movie-Name`) pairs with (`Movie-ID`, (`Movie-Name`, `Rating`)) pairs on the common key as `Movie-ID`. The result of this join is (`Movie-ID`, (`Rating`, `Movie-Name`)) pairs.
- The next step is to reduce and aggregate output of the `join()` by using `Movie-ID` as a key: we need all ratings per `Movie-ID` to find the average of ratings.
- The final step is to have a simple mapper (`map3()`) to calculate the average of ratings and depicts the final output

For this design pattern, I provide two PySpark solutions: a solution using RDDs and another one by using DataFrames.

## Solution by RDD

### Prepare Inputs

In this step we create two RDDs to represent our two inputs:

```

spark : SparkSession
movies_by_name = ["100,Lion King", "200,Star Wars",
 "300,Fidler on the Roof", "400,X-Files"]

def create_movie_pair(rec):
 tokens = rec.split(",")
 return (tokens[0], tokens[1])
#end-def

movies = spark.sparkContext.parallelize(movies_by_name)
movies.collect()
['100,Lion King', '200,Star Wars',
 '300,Fidler on the Roof', '400,X-Files']

movies_rdd = movies.map(create_movie_pair)
movies_rdd.collect()
[('100', 'Lion King'), ('200', 'Star Wars'),
 ('300', 'Fidler on the Roof'), ('400', 'X-Files')]

ratings_by_users = ["100,4,USER-1234", "100,5,USER-3467",
 "200,4,USER-1234", "200,2,USER-1234"]
def create_rating_pair(rec):
 tokens = rec.split(",")
 # we drop user_id here (not needed)
 return (tokens[0], int(tokens[1]))
#end-def

ratings = spark.sparkContext.parallelize(ratings_by_users)
ratings.collect()
['100,4,USER-1234', '100,5,USER-3467',
 '200,4,USER-1234', '200,2,USER-1234']

ratings_rdd = ratings.map(create_rating_pair)
ratings_rdd.collect()
[('100', 4), ('100', 5), ('200', 4), ('200', 2)]

```

So far we have created two RDDs:

- `movies_rdd`: `RDD[(Movie-ID, Movie-Name)]`
- `ratings_rdd`: `RDD[(Movie-ID, Rating)]`

Next, we use these two RDDs to perform the join operation on the common key—`Movie-ID`.

```

joined = ratings_kv.join(movies_kv)
joined.collect()
[('200', (4, 'Star Wars')),
 ('200', (2, 'Star Wars')),
 ('100', (4, 'Lion King')),
 ('100', (5, 'Lion King'))]

grouped_by_movieid = joined.groupByKey()
 .mapValues(lambda v: list(v))
grouped_by_movieid.collect()
[('200', [(4, 'Star Wars'), (2, 'Star Wars')]),
 ('100', [(4, 'Lion King'), (5, 'Lion King')])]
```

The last step is to have simple mapper to prepare the final output, which includes the average rating per Movie-ID:

```

def find_avg_rating(values):
 total = 0
 for v in values:
 total += v[0]
 movie_name = v[1]
 return (movie_name, float(total)/len(values))
#end-def

grouped_by_movieid.mapValues(
 lambda values: find_avg_rating(values)).collect()
[
 ('200', ('Star Wars', 3.0)),
 ('100', ('Lion King', 4.5))
]
```

## Solution by DataFrame

Solution by DataFrame is straightforward: create a DataFrame per input and then join them on a common key — Movie-ID.

First let's create DataFrames:

```

movies_by_name = [('100', 'Lion King'), ('200', 'Star Wars'),
 ('300', 'Fidler on the Roof'), ('400', 'X-Files')]
movies_df = spark.createDataFrame(movies_by_name,
 ["movie_id", "movie_name"])
movies_df.show()
```

```

+-----+-----+
| movie_id | movie_name |
+-----+-----+
100	Lion King
200	Star Wars
300	Fidler on the Roof
400	X-Files
+-----+-----+

ratings_by_user = [('100', 4, 'USER-1234'),
 ('100', 5, 'USER-3467'),
 ('200', 4, 'USER-1234'),
 ('200', 2, 'USER-1234')]

ratings_df = spark.createDataFrame(ratings_by_user,
 ["movie_id", "rating", "user_id"]).drop("user_id")
ratings_df.show()
+-----+-----+
| movie_id | rating |
+-----+-----+
100	4
100	5
200	4
200	2
+-----+-----+

```

The join operation is very easy on DataFrames:

```

joined = ratings_df.join(movies_df, "movie_id")
joined.show()
+-----+-----+-----+
| movie_id | rating | movie_name |
+-----+-----+-----+
200	4	Star Wars
200	2	Star Wars
100	4	Lion King
100	5	Lion King
+-----+-----+-----+

output = joined.groupBy("movie_id", "movie_name").avg()
output.show()
+-----+-----+-----+
| movie_id | movie_name | avg(rating) |
+-----+-----+-----+
| 200 | Star Wars | 3.0 |
+-----+-----+-----+

```

## Input-Map-Combiner-Reduce-Output

This design pattern is very similar to the Input-Map-Reduce-Output, but the main difference is that under what conditions combiners can be used as well — to speed up the transformation. In MapReduce paradigm (implemented in Apache Hadoop), a combiner — also known as a semi-reducer — is an optional function that works by accepting the outputs from the mapper function per partition and worker nodes and then aggregates results per key and finally passing the output key-value pairs to the reducer function. In Spark, — according to Spark documentation — combiners are automatically executed per worker nodes and partitions and you do not have to write any special combiner functions. The example of such a transformation is the `reduceByKey()` transformation, which merges the values for each key using an **associative** and **commutative** reduce function.

The main function of a combiner is to summarize and aggregate the mapper output records — as `(key, value)` pairs — with the same key per partition. The purpose of this design pattern is to make sure that combiners — as semi-reducers optimizers per worker nodes — can be used and that your data algorithm will not deliver wrong results. For example, if the goal is to sum up values per key and if we have created two keys — K1 and K2 — as:

`(K1, 30), (K1, 40),  
(K2, 5), (K2, 6), (K2, 7)`

on the same mapper, then the job of a combiner is to summarize these two keys as: `(K1, 70), (K2, 18)`.

This data design pattern is illustrated by Figure 18.5.

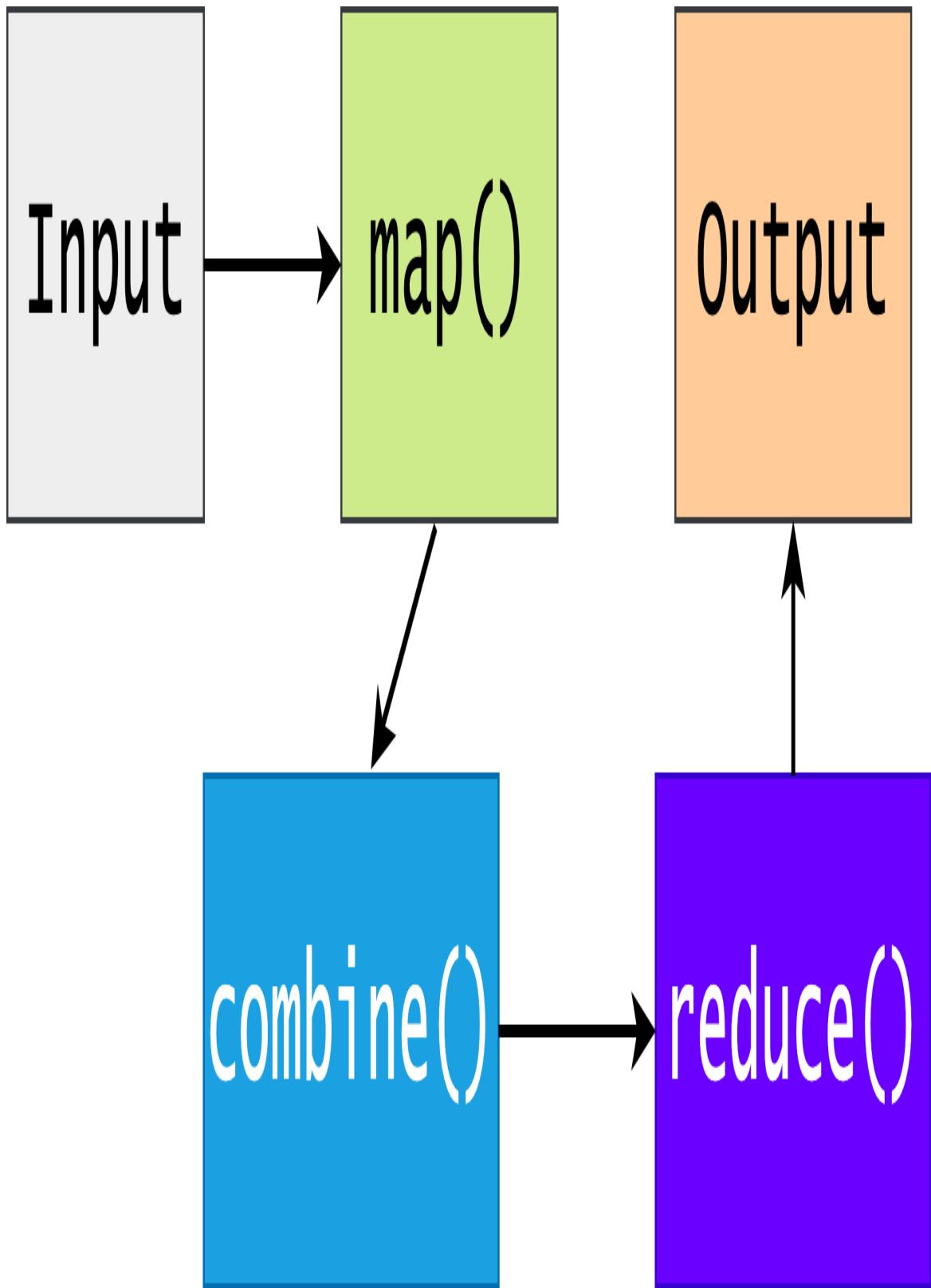


Figure 10-5. Input-Map-Combine-Reduce-Output Design Pattern

To understand this design pattern, let's assume that our input represents cities and their associated temperatures, and the goal is to find average of temperatures per unique city. For example if `source_rdd` represents `RDD[(String, Double)]` for our data, where key is the city name and value is the associated temperature. Then let's find average by city, which you might attempt to write:

```
THIS WILL NOT WORK
let t1, t2 denote temperatures for the same city
avg_per_city = source_rdd.reduceByKey(
 lambda t1, t2: (t1+t2)/2
)
```

But, this is totally a wrong transformation and it will not compute the average per city. As we know the AVG function is not an associative:

```
AVG(1, 2, 3, 4, 5) != AVG(AVG(1, 2), AVG(3, 4, 5))
```

The reason is that average of an average is not an average. Or formally we can say that the average function is not an associative. Why not? The following example shows that:

Partition-1:  
(Paris, 20)  
(Paris, 30)

Partition-2:  
(Paris, 40)  
(Paris, 50)  
(Paris, 60)

Therefore our transformation will create:

Partition-1:  
(Paris, (20+30)/2) = (Paris, 25)

Partition-2:  
(Paris, (40+50)/2) = (Paris, 45)  
(Paris, (45+60)/2) = (Paris, 52.5)

Finally: combining the results of two partitions:

```
(Paris, (25+52.5)/2)) = (Paris, 38.75)
```

Is 38.75 the average of (20, 30, 40, 50, 60)? Of course NOT, the correct average is  $(20+30+40+50+60)/5 = 200/5 = 40$ .

Therefore our reducer function is not correct: the reason is that the average function is not an associative.

Next, with a minor modification, we will make output of mappers as commutative and associative and then we will have the correct averages per every unique city.

The correct solution is provided:

```
sample_cities = [('Paris', 20), ('Paris', 30),
 ('Paris', 40), ('Paris', 50), ('Paris', 60),
 ('Cupertino', 40), ('Cupertino', 60)]

cities_rdd = spark.sparkContext.parallelize(sample_cities)
```

Next, we will create a new RDD from `cities_rdd` to make sure that its values comply with commutative and associative laws:

```
cities_sum_count = cities_rdd.mapValues(lambda v: (v, 1))
```

Therefore, our new RDD, `cities_sum_count`, denotes  $\text{RDD}[(\text{city}, (\text{sum-of-temp}, \text{count-of-temp}))]$ . Since we know that (sum, count) tuple is commutative and associative over an addition (+) operator, therefore, we can write our reduction as:

```
cities_reduced = cities_sum_count.reduceByKey(
 lambda x, y: (x[0]+y[0], x[1]+y[1])
)
```

We do need one final mapper to find the average per city:

```

avg_per_city = cities_reduced.mapValues(
 lambda v: v[0]/v[1]
)

```

Another solution by Spark for this design pattern is to use the `combineByKey()` transformation: Consider `cities_rdd` as our source RDD, then we can find the average per city as:

```

avg_per_city = cities_rdd.combineByKey(
 lambda v: (v, 1), ❶
 lambda C, v: (C[0]+v, C[1]+1) ❷
 lambda C1, C2: (C1[0]+C2[0], C1[1]+C2[1]) ❸
).mapValues(lambda v: v[0]/v[1])

```

- ❶ Denote (`sum`, `count`) as a combined data structure (`C`)
- ❷ Merge values per partition
- ❸ Combine two partitions (combine two `C`'s into one.)

### TIP

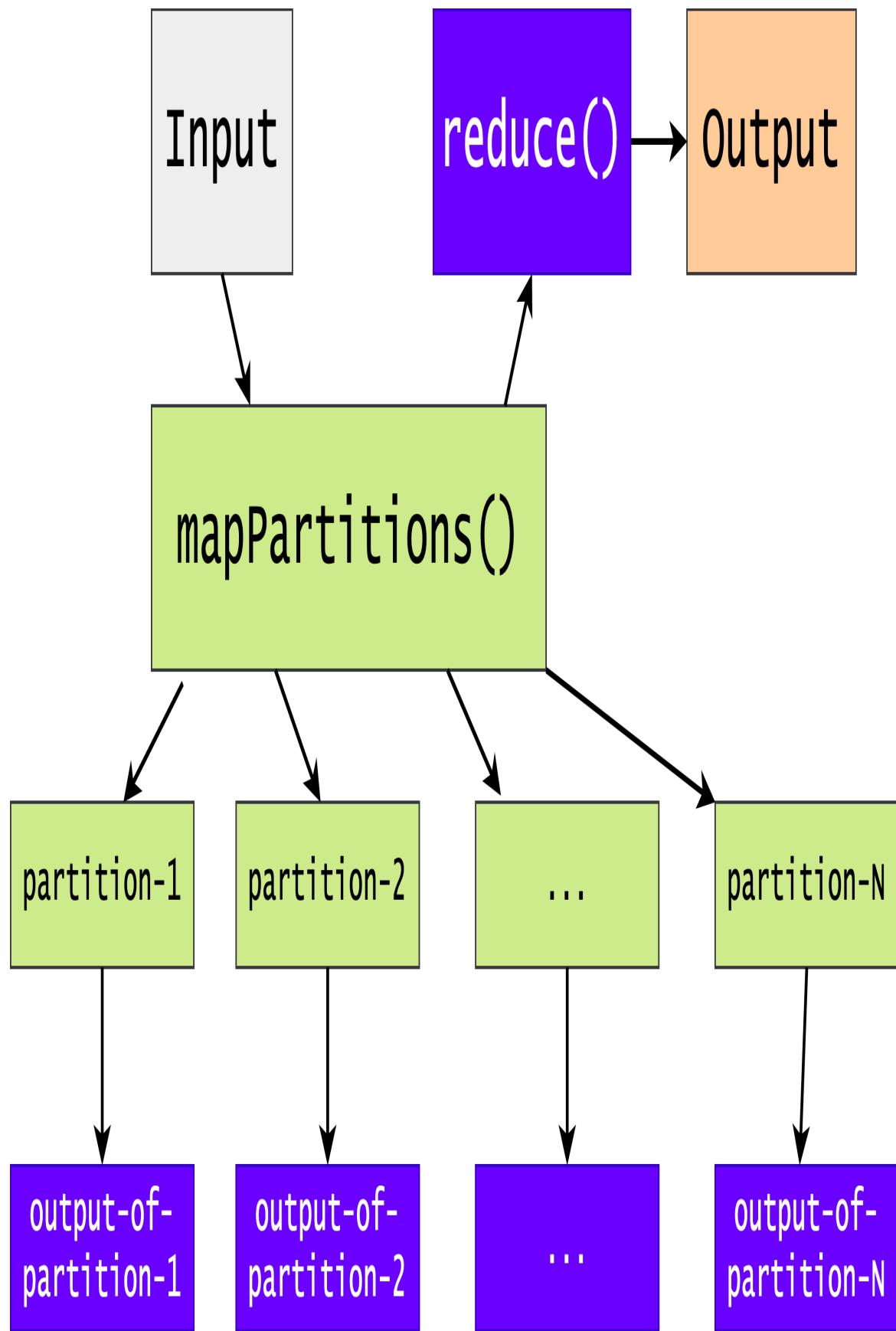
To have a semantically correct combiners — proper combiners — , we must make sure that the output of mappers follow the property of associativity and commutativity.

To learn details on this design pattern, you may reference [Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms](#). Further this paper states that “one principle for designing efficient MapReduce algorithms can be precisely articulated as follows: create a **monoid** out of the intermediate value emitted by the mapper.” Therefore, to have proper and semantically correct combiners, make sure that the output of your mapper is a monoid and it follows **commutative** and **associative** algebraic laws.

## Input-MapPartitions-Reduce-Output

This is a very important data design pattern, which you apply a function to each partition — each partition may have thousands or millions of elements — as opposed to each element. Even though this design pattern has been covered in other chapter, but because of its importance, I will cover it here too. Imagine that you have billions of records and you want to summarize all of these records into a compact data structure — such as a list, array, tuples, or a dictionary.

This data design pattern is illustrated in Figure 18.5.



*Figure 10-6. Input-MapPartitions-Output Design Pattern*

The general scenario can be stated as:

- Input: billions of records
- Processing: Use `mapPartitions()` as a summarization design pattern
  - split input into N partitions
  - analyze/process each partition — by a custom function — independently and concurrently and produce a compact data structure (CDS) such as an array, list, or a dictionary; lets label these outputs as CDS-1, CDS-2, ..., CDS-N
- Reducer: this is a final reducer working on the generated values: CDS-1, CDS-2, ..., CDS-N; the output of this step is a single compact data structure — such as an array, list, or a dictionary.

Spark's `mapPartitions()` is a specialized `map()` that is called only once for each partition. The entire content of the respective partitions is available as a sequential stream of values via the input argument (`Iterator[V]` — V is a data type of source RDD elements). The custom function must return yet another `Iterator[T]` — T is the data type of target RDD elements.

To understand this design pattern, you should understand the difference between `map()` and `mapPartitions()`. What's the difference between an Spark's `map()` and `mapPartitions()` method? The method `map()` converts each element of the source RDD into a single element of the target RDD by applying a function. On the other hand, `mapPartitions()` converts each partition — comprised of thousands and millions of elements — of the source RDD into multiple elements of the result (possibly none).

To understand the general scenario for this design pattern, I am going to present a simple example. Let's have billions of records such as :

```
<name><,><gender><,><salary>
```

and assume that the goal is to sum up salaries of employees based on their gender. Assume the goal is to find only 3 tuples as (key, value) from all input records:

```
("male", (total-number-of-males, sum-of-salaries-for-males))
("female", (total-number-of-females, sum-of-salaries-for-females))
("unknown", (total-number-of-unknowns, sum-of-salaries-for-unknowns))
```

As we can observe from the expected output, there are only three keys: “male”, “female”, and “unknown”.

Below, I present some sample input, which I will show how this design pattern behaves on actual input.

Sample Input:

```
alex,male,22000
david,male,45000
jane,female,38000
mary,female,39000
max,male,55000
nancy,female,67000
ted,x,45000
sam,x,32000
rafa,male,100000
```

By examining input and output, you might think that to generate tons on (key, value) pairs,— where key is a gender and value is a salary— and then aggregate the results by using the `groupByKey()` transformation. This solution is not efficient and has potential problems — below, these problems will be removed by using the Spark’s `mapPartitions()` transformation:

- We have to create billions of (key, value) pairs, which will clutter the cluster network
- Since there are only 3 keys, if you use the `groupByKey()` transformation, then each key will have billions of values, which might cause OOM errors

- Since there are only 3 keys, then the cluster might not be utilized efficiently

Therefore, a naive and inefficient solution is to create (key, value) as (gender, salary): Then there will be only 3 keys for all of the data: “male”, “female”, and “unknown”, where each one will have billions of values and we have to generate so many (key, value) pairs for the final reduction.

The Input-MapPartitions-Reduce-Output design pattern comes to our rescue and offers an efficient solution. Our efficient solution: partition input into N partitions ( $N=200, 400, 1000, 20000, \dots$ )— a partition can be in thousands or millions of records/elements. Value of N can be determined based on input size. The next step is to apply the `mapPartitions()` transformation: map each partition and create a very small dictionary per partition with 3 keys: “male”, “female”, and “unknown”. The final reduction will be to aggregate these N dictionaries.

To understand this design pattern, let's partition our input into 2 partitions:

**Partition-1:**

```
alex,male,22000
david,male,45000
jane,female,38000
mary,female,39000
max,male,55000
```

**Partition-2:**

```
nancy,female,67000
ted,x,45000
sam,x,32000
rafa,male,100000
```

The main idea about this design pattern is to partition input and then process partitions independently and concurrently. For example, if your  $N=1000$  and you have N mappers, then all of them can be executed concurrently. Applying the basic mapping using the `mapPartitions()` we will generate the following dictionaries per partition:

```
Partition-1:
{
 "male": (122000, 3),
 "female": (77000, 2)
}
```

```
Partition-2:
{
 "male": (100000, 1),
 "female": (67000, 1),
 "unknown": (77000, 2)
}
```

Next, we will apply the final reduction: to aggregate output of all partitions into a single dictionary as:

```
final output:
{
 "male": (222000, 4),
 "female": (144000, 3),
 "unknown": (77000, 2)
}
```

When the Input-MapPartitions-Reduce-Output design pattern is used to summarize data, there is no scalability issue since we are creating one simple small data structure (such as a dictionary) per partition. If even we set N to 100,000, still the solution is efficient, since processing 100,000 small dictionaries will not cause any OOM problems.

### TIP

The most important reason to use `mapPartitions()` transformation is performance. By having all the data — as a single partition — needed to calculate on a single server node, we reduce the overhead on the shuffle (the need for serialization and network traffic). To understand partitions, you may refer to [Spark Partitions](#).

Spark's `mapPartitions()` transformation is an ideal tool to implement the Input-MapPartitions-Reduce-Output design pattern since you can have a heavy weight initialization per partition (rather than per element).

The following example shows how to do have heavy weight initialization that should be done once per partition — for many RDD elements — rather than once per RDD element. The `mapPartitions()` provides for the initialization to be done once per worker task/thread/partition instead of once per RDD data element for example:

```
source_rdd: RDD[V]
source_rdd.count() in billions

target_rdd: RDD[T]
target_rdd.count() in thousands

apply transformation
target_rdd = source_rdd.mapPartitions(custom_func)

def custom_func(partition):
 database_connection = <heavy-weight-operation-initialization>
 target_data_structure = <initialize>
 for element in partition
 target_data_structure = update(element,
 target_data_structure,
 database_connection)
 #for-loop
 close(database_connection)
 #
 return target_data_structure
#def
```

## Inverted Index Pattern

In computer science, an inverted index is a database index storing a mapping from content — such as words or numbers — to its locations in a table, or in a document or a set of documents. For example, consider input:

```
doc1: ant, dog
doc2: ant, frog
doc3: dog
doc4: ant
```

Then the goal of inverted index is to create the following index — called Inverted Index.

```
frog: [doc2]
ant: [doc1, doc2, doc4]
dog: [doc1, doc3]
```

Therefore if you want to search for a “dog”, then we exactly know that it is in [doc1, doc3]. Inverted Index design pattern generates an index from a data set to allow for faster searches. Inverted Index is the most popular data structure used in document retrieval systems, used on a large scale in search engines.

Inverted Index has advantages and disadvantages:

#### *Advantage of Inverted Index*

- Inverted index enable us to perform fast full text searches, at a cost of increased processing when a document is added to the database.
- It is easy to develop. Using PySpark, we can implement this design pattern by a series of `map()`, `flatMap()`, and reduction transformations

#### *Disadvantage of Inverted Index*

- Large storage overhead and high maintenance costs on update, delete and insert.

## **Problem Statement**

To implement this design pattern, let me state a problem and then solve it by PySpark. The problem statement can be stated as: we have a dataset, which contains Shakespeare’s works split among many files; the output must contain a list of all words with the file in which it occurs and the number of times it occurs.

## **Input**

A sample input documents for the Inverted Index can be downloaded from the [GitHub](#), which is a series of 35 text files:

```
0ws0110.txt
0ws0210.txt
...
0ws4210.txt
```

## Output

The output will be an inverted index of all documents read in the input phase. Output will have the following format:

```
(word, [(filename1, frequency1), (filename2, frequency2), ...])
```

which indicates that the `word` is in `filename1` (with frequency of `frequency1`) and `word` is in `filename2` (with frequency of `frequency2`), etc.

## PySpark Solution

The Inverted Index high-level solution using PySpark notations is outlined below:

- Step-1: Read input files, filter all stopwords, and apply stemming algorithms if desired. This step creates pairs of (`path, text`).
- Step-2: Create a tuple with count 1. Therefore, the expected output would be `((word, document), 1)`.
- Step-3: Group all `(word, document)` pairs and sum the counts (reduction is required).
- Step-4: Transform tuple of `((word, document), frequency)` into `(word, (document, count))` so that we can count `word(s)` per `document`.
- Step-5: Output sequence of `(document, count)` into a comma separated string
- Step-6: save your Inverted Index

To understand the Inverted Index algorithm, I am going to present 3 small documents and then show the input and output for every step. Consider the following 3 documents:

```
$ ls -l /tmp/documents/
file1.txt
file2.txt
file3.txt

$ cat /tmp/documents/file1.txt
fox jumped
fox jumped high
fox jumped and jumped

$ cat /tmp/documents/file2.txt
fox jumped
fox jumped high
bear ate fox
bear ate honey

$ cat /tmp/documents/file3.txt
fox jumped
bear ate honey
```

### Step-1:

The first step is to read input files and filter all stop words (such as `of`, `the`, ...). You can do more than filtering stop words: for example you may use a stemming algorithm to stem the words too (such as converting `reading` to `read` and so on). Next, we create pairs of (`path`, `text`), where `path` is the full name of input file and `text` is the content of the associated `path`. For example, if a `path` denotes a file such as `/tmp/documents/file1.txt`, then I am using `file1.txt` as a document.

```
docs_path = '/tmp/documents/'
rdd = spark.sparkContext.wholeTextFiles(docs_path)
rdd.collect()
[('file:/tmp/documents/file2.txt',
 'fox jumped\nfox jumped high\nbear ate fox \nbear ate honey\n'),
 ('file:/tmp/documents/file3.txt',
```

```
'fox jumped\nbear ate honey\n'),
('file:/tmp/documents/file1.txt',
 'fox jumped\nfox jumped high\nfox jumped and jumped\n')]
```

Spark's `wholeTextFiles(path)` function read a directory of text files from file system URI. Each file is read as a single record and returned in a (`key`, `value`) pair, where the `key` is the path of each file, the `value` is the content of each file.

*Step-2:*

The second step is to map the given `text` into a set of (`word`, `document`) followed by mapping (`word`, `path`) into ((`word`, `document`), 1), which indicates that the `word` belongs to `document` with frequency of 1.

```
def get_document_name(path):
 tokens = path.split("/")
 return tokens[-1]
#end-def

rdd2 = rdd.map(lambda x : (get_filename(x[0]), x[1]))
rdd2.collect()
[('file2.txt',
 'fox jumped\nfox jumped high\nbear ate fox \nbear ate honey\n'),
 ('file3.txt',
 'fox jumped\nbear ate honey\n'),
 ('file1.txt',
 'fox jumped\nfox jumped high\nfox jumped and jumped\n')]

rdd3 = rdd2.map(lambda x: (x[0], x[1].splitlines()))
rdd3.collect()
[('file2.txt',
 ['fox jumped', 'fox jumped high', 'bear ate fox ', 'bear ate honey']),
 ('file3.txt',
 ['fox jumped', 'bear ate honey']),
 ('file1.txt',
 ['fox jumped', 'fox jumped high', 'fox jumped and jumped'])]
```

Next, we create (`word`, `document`) pairs:

```

def create_pairs(tuple2):
 document = tuple2[0]
 records = tuple2[1]
 pairs = []
 for rec in records:
 for word in rec.split(" "):
 pairs.append((word, document))
 return pairs
#end-def

rdd4 = rdd3.flatMap(create_pairs)
rdd4.collect()
[('fox', 'file2.txt'), ('jumped', 'file2.txt'),
 ('fox', 'file2.txt'), ('jumped', 'file2.txt'), ...]

rdd5 = rdd4.map(lambda x: (x, 1))
rdd5.collect()
[(('fox', 'file2.txt'), 1), (('jumped', 'file2.txt'), 1),
 (('fox', 'file2.txt'), 1), (('jumped', 'file2.txt'), 1), ...]

```

### *Step-3:*

The third step performs a simple reduction to group all (word, document) pairs and sum the counts.

```

frequencies = rdd5.reduceByKey(lambda x, y: x+y)
frequencies.collect()
[((('fox', 'file2.txt'), 3), (('jumped', 'file2.txt'), 2), (('ate', 'file2.txt'), 2),
 ('bear', 'file3.txt'), 1), ...]

```

### *Step-4:*

This step does a very simple transformation and moves the file\_name into the value part:

```

((word, file_name), frequency) => (word, (file_name, frequency))

rdd6 = frequencies.map(lambda v: (v[0][0], (v[0][1], v[1])))
>>> rdd6.collect()
[('fox', ('file2.txt', 3)), ('jumped', ('file2.txt', 2)),
 ('ate', ('file2.txt', 2)), ('bear', ('file3.txt', 1)), ...]

```

### Step-5:

Output sequence of (document, count) into a comma seperated string

```
inverted_index = rdd6.groupByKey()
inverted_index.mapValues(lambda values: list(values)).collect()
[('fox', [('file2.txt', 3), ('file1.txt', 3), ('file3.txt', 1)]),
 ('bear', [('file3.txt', 1), ('file2.txt', 2)]),
 ('honey', [('file3.txt', 1), ('file2.txt', 1)]), ...]
```

For implementing this step, I used the `groupByKey()` transformation. You may use other reduction transformations — such as `reduceByKey()` and `combineByKey()` — to accomplish the same semantics.

For example to implement this step by the `combineByKey()` transformation, you may write it as:

```
convert a tuple into a list
def to_list(a):
 return [a]

append a tuple to a list
def append(a, b):
 a.append(b)
 return a

merge two list from partitions
def extend(a, b):
 a.extend(b)
 return a

inverted_index = rdd6.combineByKey(to_list, append, extend)
```

### Step-6:

Save your created Inverted Index:

```
inverted_index.saveAsTextFile("/tmp/output/")
```

# Chapter 11. Data Design Patterns

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

The goal of this chapter is to introduce some informal but practical data design patterns. I will only discuss the design patterns that are useful in production environments and will not look at the theoretical design patterns (which are just theories and not used in big data solutions), which are not deployed in production environments.

This chapter

- Introduces the basic concepts of design patterns in the context of big data
- Provides some useful and practical design patterns by examples
- Shows how to use Spark’s transformations for implementing data design patterns
- Introduces the concept of “monoids” for better understandings of reduction transformations

## SOURCE CODE

### NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 19](#).

The best design patterns book is the iconic computer science book “Design Patterns: “Elements of Reusable Object-Oriented Software” by four authors: Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known as The “Gang of Four”). I will not present the data design patterns similar to the “Gang of Four” book, but will focus on practical data design patterns. My presentations are informal, practical, and have been used in production environments.

What is a **design pattern**? According to wikipedia: “in software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code.”

Spark, MapReduce, and distributed programming design patterns are common patterns in data related problem solving. MapReduce is a programming model (for example, Hadoop is a concrete implementation of MapReduce) and an associated implementation for processing and generating big data sets with a parallel, distributed algorithm on a cluster. A typical MapReduce job consists of a driver and 3 functions:

- **map()**: a mapper, which supports an equivalent of Spark’s `map()`, `flatMap()`, and `filter()`
- **reduce()**: a reducer , which supports an equivalent of Spark’s `reduceByKey()` and `groupByKey()`
- **combine()** as an optional local reducer

For example, in software engineering, the “singleton” pattern is a design pattern that restricts the instantiation of a class to one object. In MapReduce, **InMapper Combiner** is a design pattern, which does most of the `combine()` functionality (in order to avoid generating too many intermediate (key, value) pairs) inside the

mappers. A Combiner transformation, also known as a semi-reducer, is an optional operation in MapReduce paradigm that operates by accepting the inputs from the mappers and thereafter passing the output key-value pairs to the reducers. The main function of a Combiner is to summarize the mappers outputs with the same key. Another benefit of design patterns is that it makes the objective of code easier to understand and provides known scalability issues (if any), performance profiles and limitations of solutions.

To be practical, I will only focus on some of the practical data design patterns, which can help us on developing big data solutions, which can be deployed to production environments without scalability problems. The data design patterns are patterns, which can help us to write scalable solutions to be deployed on Spark clusters. Overall, there is no silver bullet on adopting and using design patterns and data design patterns, every adopted pattern should be tested (in similar production environments) in detail for performance and scalability by using real data.

For learning design patterns in software engineering, you should look at the “Gang of Four” book (Design Patterns: Elements of Reusable Object-Oriented Software by Erich Gamma, et al). For learning design patterns in MapReduce, you should look at the “MapReduce Design Patterns” book by Donald Miner and Adam Shook and my book “Data Algorithms” (published by O’Reilly).

Below are the partial list of data design patterns and I will cover some of them in this chapter.

- MinMax
- Top-10
- Secondary Sorting
- Stripes
- InMapper Combining
- Summarization Patterns
- Filtering Patterns
- Data Organization Patterns

- Duplicate Detection

Next, I discuss the InMapper Combining data design pattern by some simple examples.

## InMapper Combining

**InMapper Combining** refers to a situation for a mapper to combine and summarize the mappers outputs as much as possible and emit less intermediate (key, value) pairs for the sort and shuffle and reducers (such as `reduceByKey()`, `groupByKey()`, ...). For example, for a classic word count problem, given an input record such as:

```
"fox jumped and fox jumped again fox"
```

Without using InMapper Combiner, we will generate the following (key, value) pairs (before handing it to shufflers and then reducers):

```
(fox, 1)
(jumped, 1)
(and, 1)
(fox, 1)
(jumped, 1)
(again, 1)
(fox, 1)
```

The problem is that for big data, we might generate too many (`word, 1`) pairs, which might be keeping the cluster network too busy and may be less efficient. Using the InMapper Combiner data design pattern, we will combine (summarize the mappers outputs) the (key, value) pairs to generate less (key, value) pairs. For example since there are 3 of (fox, 1), then that will be combined into (fox, 3).

```
(fox, 3)
(jumped, 2)
(and, 1)
(again, 1)
```

The basic concept of the “InMapper Combiner” data design pattern is illustrated by Figure 19.1.

"fox jumped and fox jumped again fox ..."

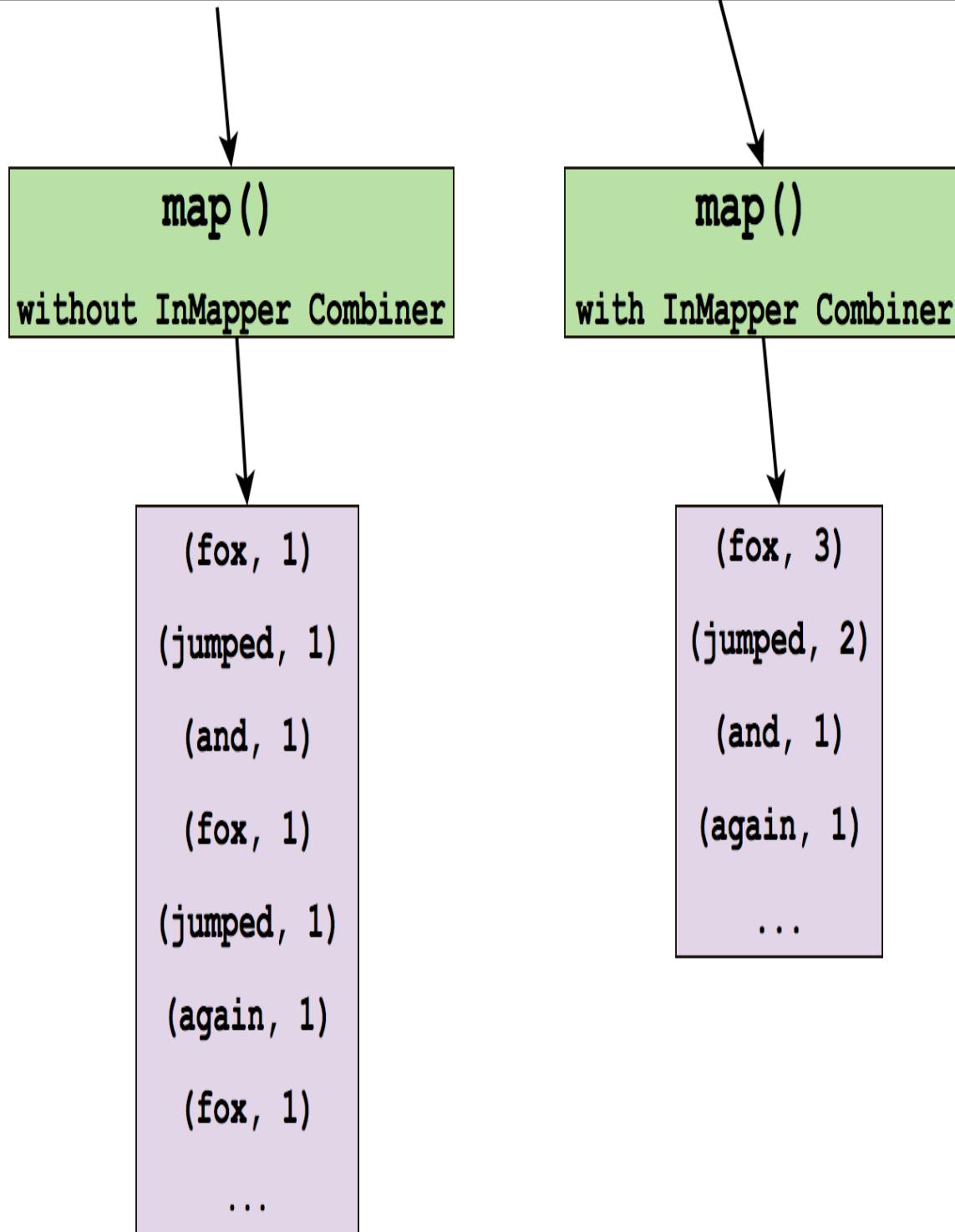


Figure 11-1. InMapper Combiner Concept

Therefore, in the mapper transformation, using InMapper combiner, the 3 pairs of (fox, 1) is replaced by (fox, 3). Without using InMapper combiner, for the word count, we generated seven (key, value) pairs, but by using the InMapper combiner we generated only four (key, value) pairs. If we have a lot of repeated words in our data, then the InMapper Combiner might help us to achieve better performance by generating less intermediate (key, value) pairs.

To demonstrate the InMapper Combining data Design Pattern concepts, we solve counting the frequencies of characters for a set of documents. In simple terms, we want to find out the frequency of each unique character for a given corpus. We will discuss the following solutions:

- Basic MapReduce Design Pattern
- InMapper Combining Per Record
- InMapper Combining Per Partition

## Basic MapReduce Design Pattern

To count the characters, for each record of input, we split it into a set of words, and then we split each word into a character array, and finally we emit a (key, value) pairs, where key is a single character from a character array and value is 1 (frequency count of one character). Since we did not use any custom data types for emitting key-value pairs, we call this the “Basic” MapReduce design pattern. The reducer sums up the frequencies of a single unique character. The problem with this solution is that for large data, we emit too many (key, value) pairs, which can impact the network traffic and hence can impact the performance of the overall solution. Also, because of too many (key, value) emitted pairs, the “sort and shuffle” (grouping values for the same key) phase might take longer time than expected.

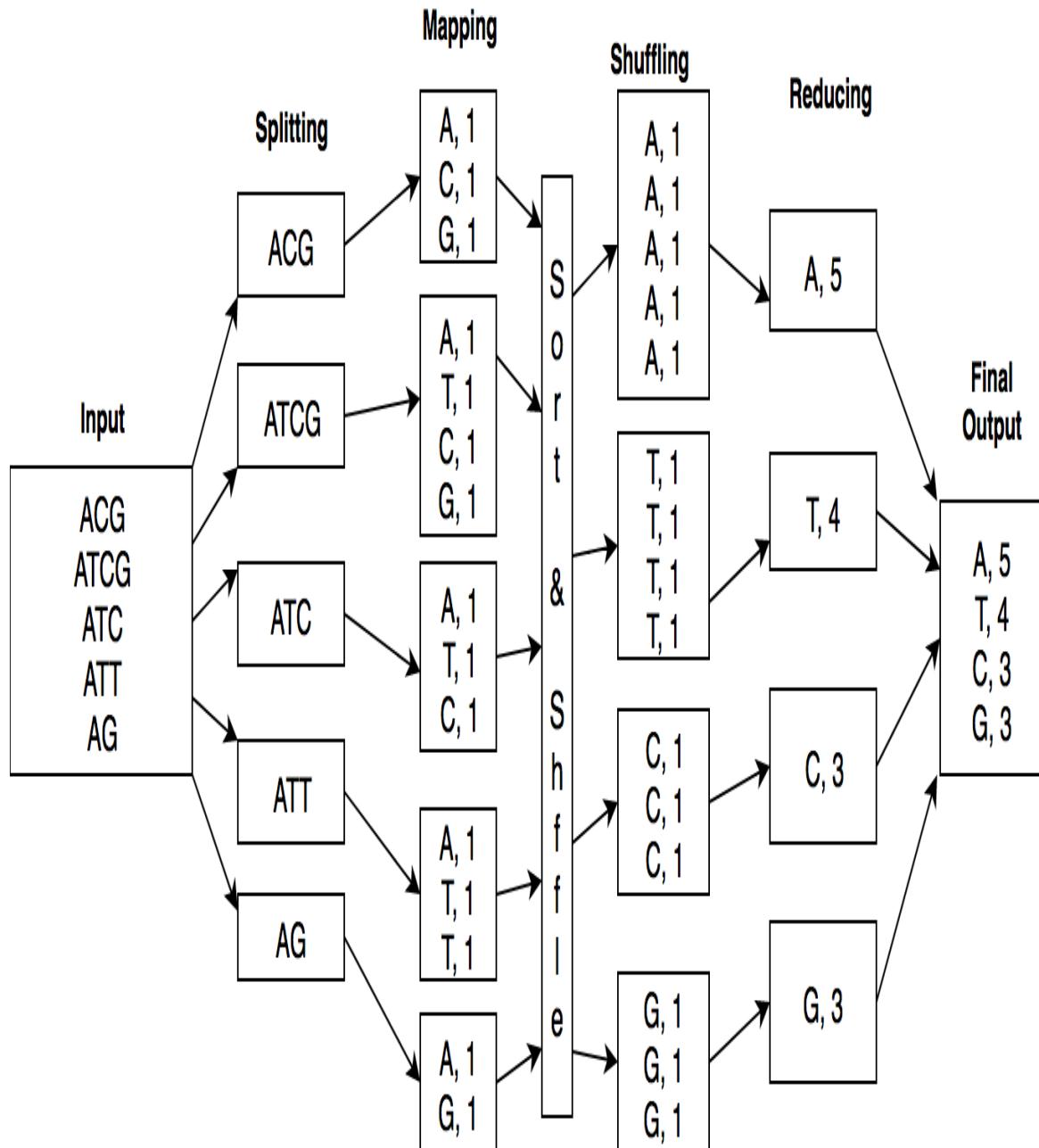


Figure 11-2. Character Count: Basic MapReduce Algorithm

Given an RDD of String (as `RDD[String]`), PySpark solution is provided below:

First we define a simple function, which accepts a single string records and then returns a list of (key, value) pairs, where key is a character and value is 1 (frequency of a character).

```

def mapper(rec):
 words = rec.lower().split() ①
 pairs = [] ②
 for word in words: ③
 for c in word: ④
 pairs.append((c, 1)) ⑤
 #end-for
 #end-for
 return pairs ⑥
#end-def

```

- ① Tokenize a record into an array of words
- ② Create an empty list as pairs
- ③ Iterate over words
- ④ Iterate over a single word
- ⑤ Add each character (c) as (c, 1) to pairs
- ⑥ Return list of (c, 1) for all characters in a given record

The `mapper()` function can be simplified as:

```

def mapper(rec):
 words = rec.lower().split()
 pairs = [(c, 1) for word in words for c in word]
 return pairs
#end-def

```

Next, we use the `mapper()` function to count frequencies of unique characters:

```

spark : an instance of SparkSession
rdd = spark.sparkContext.textFile("/dir/input") ①
pairs = rdd.flatMap(mapper) ②
frequencies = pairs.reduceByKey(lambda a, b: a+b) ③

```

- ① Create an `RDD[String]` from input data
- ②

Map each record into collection of characters and flatten it as a new  
RDD[Character, 1]

- ③ Find frequencies of each unique character

Therefore, InMapper Combining data design pattern reduces the number of (key, value) pairs emitted by mappers, which indeed this reduces the network traffic and improves the performance of execution time.

## InMapper Combining Per Record

This section introduces InMapper Combining Per Record design pattern, which is also known as Local Aggregation Per Record solution. This is very similar to the basic Spark's MapReduce Algorithm with the exception that for a given input record, we aggregate frequencies for a single character before emitting (key, value) pairs. Since the same character may be repeated many times in a given input record, in this solution we emit (key, value) pairs where key is a unique single character within a given record and value is an aggregated frequency for the same character in a given mapper input record. Then we use `reduceByKey()` to aggregate all frequencies for a unique single character. This solution uses “local aggregation” by leveraging the associativity and commutativity of the `reduce()` function to combine values before reducers fetch them across the network. Typically, in MapReduce frameworks (such as Hadoop), combiners are viewed by the runtime as an optional local optimizations. Note that the correctness of the algorithm cannot depend on the combiner (instead we use “force in-mapper combining”). For example, for the following input record:

foxy fox jumped over fence

we will emit the following (key, value) pairs:

(o, 3) (m, 1) (v, 1)  
(x, 2) (p, 1) (r, 1)  
(y, 1) (e, 4) (n, 1)  
(j, 1) (d, 1) (c, 1)

Given an RDD of String (RDD[String]), PySpark solution is provided below:

First we define a simple function, which accepts a single string record (an element of input RDD[String]) and then returns a list of (key, value) pairs, where key is a unique character and value is an aggregated frequency of that character.

```
import collections ①
def local_aggregator(record):
 hashmap = collections.defaultdict(int) ②
 words = record.lower().split() ③
 for word in words: ④
 for c in word: ⑤
 hashmap[c] += 1 ⑥
 #end-for
 #end-for
 print("hashmap=", hashmap)
#
 pairs = [(k, v) for k, v in hashmap.iteritems()] ⑦
 print("pairs=", pairs)
 return pairs ⑧
#end-def
```

- ① The `collections` module provides high-performance container datatypes
- ② Create an empty dictionary[String, Integer], `defaultdict` is a dict subclass that calls a factory function to supply missing values
- ③ Tokenize record into an array of words
- ④ Iterate over words
- ⑤ Iterate over each word
- ⑥ Aggregate characters
- ⑦ Flatten dictionary into a list of (character, frequency)
- ⑧ Return the flattened list of (character, frequency)

Next, we use the `local_aggregator()` function to count frequencies of unique characters:

```
input_path = '/tmp/your_input_data.txt'
rdd = spark.sparkContext.textFile(input_path)
pairs = rdd.flatMap(local_aggregator)
frequencies = pairs.reduceByKey(lambda a, b: a+b)
```

For sure, this solution will emit much less (`key`, `value`) pairs, which is an improvement over the previous solution. The problem with this solution is that we instantiate and use a dictionary per mapper. If we have so many mappers, this might create an OOM problem. But if the number of mappers are not too many, then this solution will scale out. Another improvement of this solution is in the “sort and shuffle” phase: since we do not have too many (`key`, `value`) pairs before reduction, then the sort and shuffle will execute faster than the previous solution.

Next, I present InMapper Combiner Per Partition, which is the most efficient among all InMapper combiners. This is because we use the least amount of memory to summarize and combine (`key`, `value`) pairs per partition (can be comprised of millions of data elements).

## InMapper Combiner Per Partition

InMapper Combiner Per Partition solution emits frequencies of characters per partition (rather than per record) of input data. Each partition of input is a set of input records rather than a single input record. Therefore, there is another way to emit the frequency of characters: build a hash table of `dict[Character, Integer]` from the characters of given “input partition” (as opposed to “input record”—an “input partition” is a partition of input comprised of a set of “input records”) and then emit the (`key`, `value`) pairs, where key is `dict.Entry.getKey()` and value is `dict.Entry.getValue()`. Therefore, the mapper phase will emit a set of (`key`, `value`) pairs, where key is an entry of built hash table.

In this of solution, we use a single hash table per input partition (rather than per input record) to keep track of frequencies of all characters for a given partition.

After the mapper (using the PySpark's `mapPartitions()` transformation) completes the given partition, then we emit all key-value pairs from the frequencies table (the built hash table). Then the reducers will sum up the frequencies and find the final count of characters. Note that the mapper using Hash Table (called the “InMapper Combiner” design pattern) is more efficient than the presented previous solutions because it will not emit too many key-value pairs, which improve efficiency in network by sending less data. The solution will scale out better than the previous two solutions: we use only a single hash table per input partitioner rather than using a hash table per input record. This eliminates possible OOM problems. If even we partition our input into thousands of partitions, this solution scales out very well.

It is clear that the “Basic” MapReduce algorithm generates huge number of key-value pairs compared to the “InMapper Combining” design pattern. The “InMapper Combining” data structure representation is much more compact, since every entry of `dict[Character, Integer]` is equivalent to N basic key-value pairs, where N is equal to `dict.Entry.getValue()`. The “InMapper Combining” approach also generates fewer and shorter intermediate keys, and therefore the execution framework has less sorting to perform. In using the “InMapper Combining” design pattern, you need to be careful on the size of hash table to make sure it will not cause bottlenecks. For Character Count problem, the size of hash table for each mapper (per input partition) will be very small (since we have a limited number of unique characters), therefore there is no performance bottleneck for the presented solution.

For example, for the following “input partition” (as opposed to a single record):

```
foxy fox jumped over fence
foxy fox jumped
foxy fox
```

we will emit the following (key-value) pairs:

```
(f, 7) (u, 2) (v, 1) (j, 2) (y, 3)
(o, 7) (m, 1) (r, 1) (d, 2) (e, 5)
(x, 6) (p, 2) (n, 1) (c, 1)
```

Given an RDD of String (`RDD[String]`), PySpark solution is provided below:

First we define a simple function, which accepts a single input partition (comprised of many input records) and then returns a list of (key, value) pairs, where key is a character and value is an aggregated frequency of that character.

```
def inmapper_combiner(partition_iterator): ①
 hashmap = defaultdict(int) ②
 for record in partition_iterator: ③
 words = record.lower().split() ④
 for word in words: ⑤
 for c in word: ⑥
 hashmap[c] += 1 ⑦
 #end-for
 #end-for
 #end-for
 print("hashmap=", hashmap)
 #
 pairs = [(k, v) for k, v in hashmap.items()] ⑧
 print("pairs=", pairs)
 return pairs ⑨
#end-def
```

- ① partition\_iterator represents a single input partition comprised of a set of records
- ② Create an empty `dictionary[String, Integer]`
- ③ Get a single record from a partition
- ④ Tokenize record into an array of words
- ⑤ Iterate over words
- ⑥ Iterate over each word
- ⑦ Aggregate characters
- ⑧ Flatten dictionary into a list of (`character, frequency`)
- ⑨ Return the flattened list of (`character, frequency`)

Next, we use the `inmapper_combiner()` function to count frequencies of unique characters:

```
rdd = spark.sparkContext.textFile("/.../input") ❶
pairs = rdd.mapPartitions(inmapper_combiner) ❷
frequencies = pairs.reduceByKey(lambda a, b: a+b)
```

- ❶ Create an RDD[String] from input file(s)
- ❷ The `mapPartitions()` transformation returns a new RDD by applying a function to each input partition (as opposed to a single input record) of this RDD

For sure, this solution will emit much less (key, value) pairs, which is an improvement over previous two solutions. This solution is very efficient since we instantiate and use a single dictionary per input partition (rather than per input record). If even we have so many mappers, this will not create an OOM problem. No matter the number of input partitions, this solution will scale out. Another improvement of this solution is in the “sort and shuffle” phase: since we do not have too many (key, value) pairs before reduction, then the sort and shuffle will execute much faster than the previous two solutions. The “InMapper Combining” design pattern reduces the amount of data that needs to be transferred between the mapper and reducer. However, the “InMapper Combining” algorithm may run into a problem if the number of unique keys grows too large for the associative array to fit in memory. If this is the case you will have to revert to the basic Pairs design pattern approach. Note that with “InMapper Combining”, the mappers will generate only those key-value pairs that need to be shuffled across the network to the reducers.

In implementing the “InMapper Combining” design pattern, we used Spark’s powerful `mapPartitions()` transformation to transform each input partition into a single `dict[Character, Integer]` and then we aggregate/merge these into a single final `dict[Character, Integer]`. For “char counting”, this algorithm is efficient and faster than other algorithms. When you are to create or aggregate small amount of information from large chunk of data (such as finding frequencies of characters — since the number of ASCII characters are very

limited), then using “InMapper Combining” design pattern makes sense and a best option to implement this design pattern is the Spark’s `mapPartitions()` transformation.

We should also consider the scalability bottlenecks for the “InMapper Combining” design pattern. Using the “InMapper Combining” design pattern, we make the assumption that, at any point in time, each associative array per mapper partition (i.e., `dic[Character, Integer]`) is small enough to fit into memory—otherwise, memory paging will significantly impact performance. For character count, the size of the associative array (per mapper partition) is bounded by the number of unique characters. Therefore, for character count problem, there is no scalability bottlenecks by using the “InMapper Combining” design pattern.

But what are the advantages and disadvantages of the InMapper Combining design pattern? The InMapper Combining has the following advantages and disadvantages:

- Advantages
  - Create less (`key, value`) pairs by mappers
  - Far less sorting and shuffling of (`key, value`) pairs
  - Can make better use of combiners as optimizers
  - Scale out solution
- Disadvantages
  - More difficult to implement (you need some custom functions for handling each partition)
  - Underlying object (per mapper partition) is more heavyweight
  - Fundamental limitation in terms of size of the underlying object (for the character count problem: an associative array per mapper partition)

The InMapper Combining design pattern enable us to reduce the number of (`key, value`) pairs emitted by mappers. Hence, this can improve the performance of your entire data transformations.

## Top-10

The Top-10 design-pattern is illustrated by the Figure 19.3.

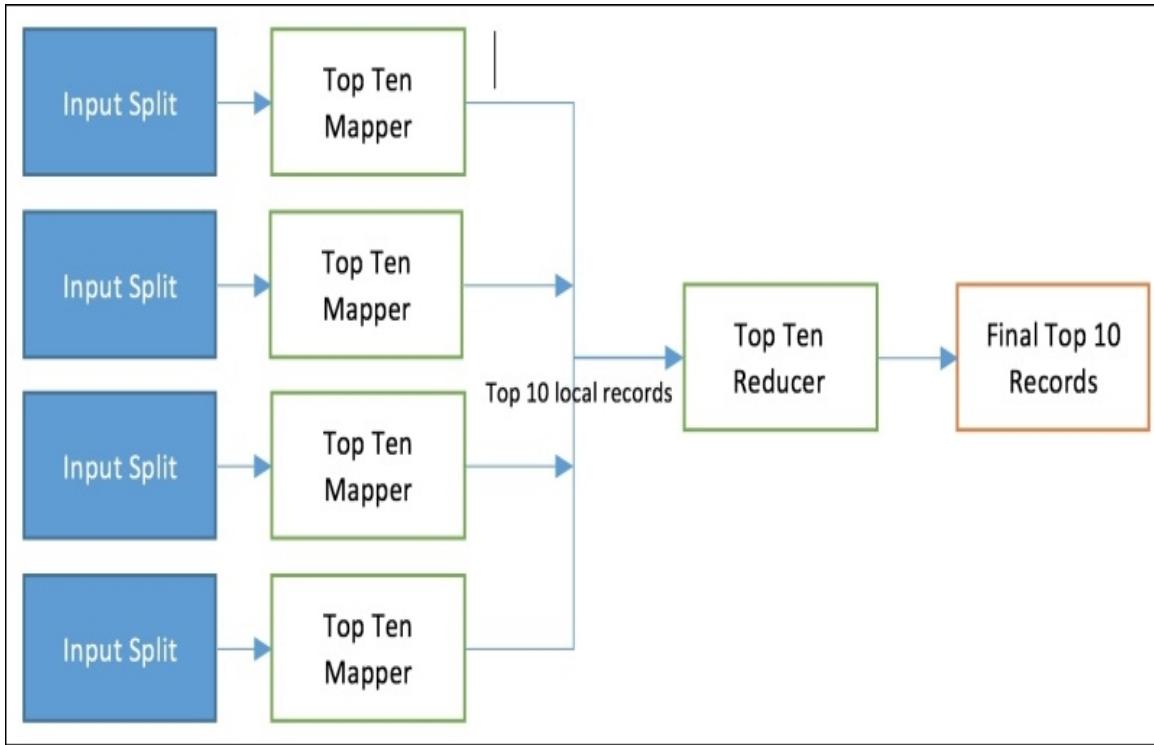


Figure 11-3. Top-10 Design Pattern

Creating a Top-10 list is a common task in many data intensive operations. For example, we might ask the following questions:

- What are the top-10 list of the URLs visited last day/week/month?
- What are the top-10 list of items purchased from Amazon last week/month?
- What are the top-10 list of search queries from Google in the last day/week/month?
- What are the top-10 list of liked items from Facebook yesterday?
- What are the top-10 list of cartoons of all time?

For example, for a search engine (such as Google), it is desirable to know the Top-10 visited URLs per day, week, or month. If we have a table of two

columns: (`url`, `frequency`), then finding Top-10 URLs in SQL is straightforward:

```
SELECT url, frequency
 FROM url_table
 ORDER BY frequency DESC
 LIMIT 10;
```

Also, finding Top-N (where  $N > 0$ ) records in Spark is very easy: let `R` be an RDD of (`String`, `Integer`) (pair of String and Integer), where key is a String (representing a URL) and value is the frequency of URL. Then we may use `RDD.takeOrdered(N)` to find the Top-N lists. The general format of `RDD.takeOrdered()` is given below:

```
takeOrdered(N, key=None)
Description:
 Get the N elements from an RDD ordered in ascending
 order or as specified by the optional key function.
```

Assuming that  $N$  is an integer greater than 0, using `RDD.takeOrdered()` solves the Top-N list as:

```
N = 10

Sort by keys (ascending):
RDD.takeOrdered(N, key = lambda x: x[0])

Sort by keys (descending):
RDD.takeOrdered(N, key = lambda x: -x[0])

Sort by values (ascending):
RDD.takeOrdered(N, key = lambda x: x[1])

Sort by values (descending):
RDD.takeOrdered(N, key = lambda x: -x[1])
```

For sake of argument assume that `takeOrdered()` does not have an optimal performance for a very large data set. Then what other options do we have? We might use the Top-10 design pattern for MapReduce paradigm. How does the Top-10 design pattern work for MapReduce paradigm? We'll discuss that throughout the rest of this section.

Given a large set of { (key-as-string, value-as-integer) } pairs, then finding a Top-N ( where N > 0) list is a “design pattern”, which is an independent reusable solution to a common Top-10 problem, which enable us to produce reusable code. For example, let **key-as-string** be a URL and **value-as-integer** be the number of times that URL is visited, then you might ask: what are the Top-10 URLs for a given data (means find the 10 URLs with the highest frequencies)? This kind of a question is common for this type of (key, value) pairs. Finding “Top-10 list” falls into “Filtering Pattern”: you filter out data and find Top-10 list. Also note that Top-10 function is a function which is **commutative** and **associative** and therefore using partitioners, combiners, and reducers will always produce correct results. Let T be a Top-10 function, and let **a**, **b**, and **c** be a set of values (such as frequencies) for the same key, then we can write:

- Commutative

$$T(a, b) = T(b, a)$$

- Associative

$$T(a, T(b, c)) = T(T(a, b), c)$$

### TOP-10 TIP

For details on the “top-10 list” design pattern refer to MapReduce Design Patterns book by Donald Miner.

This section provides a complete PySpark solution for Top-10 design pattern. Given a pair RDD of (**String**, **Integer**), the goal is to find Top-10 list for the given RDD. In our solution, we assume that all keys are unique. If the keys are not unique, then you may use the `reduceByKey()` (before finding Top-10) to make all keys unique.

Our MapReduce solutions will generalize the “top 10 list” and will be able to find “top-N list” (for N > 0). For example, we will be able to find “top 10 cats”, “top 50 most visited web sites”, or “top 100 search queries of a search engine”.

## Top-N Formalized

Let  $N$  be an integer number and  $N > 0$ . Let  $L$  be a list of pairs of  $(T, \text{ Integer})$ , where  $T$  can be any type (such as String, URL, ...),  $L.size() = s$ ,  $s > N$ , and elements of  $L$  be:

$$\{(K_i, V_i), i = 1, 2, \dots, s\}$$

where  $K_i$  has a data type of  $T$  and  $V_i$  is an Integer type (this is the frequency of  $K_i$ ). Let  $\{\text{sort}(L)\}$  be a sorted list where sort is done by using frequency as a key:

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

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

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

For Top-N solution, we will use Python's `SortedDict`. Before providing Top-N solution, let's understand `SortedDict`. Python's sorted dict (`SortedDict`) is a sorted mutable mapping. Sorted dict keys are maintained in sorted order. The design of sorted dict is simple: sorted dict inherits from dict to store items and maintains a sorted list of keys. Sorted dict keys must be hashable and comparable. The hash and total ordering of keys must not change while they are stored in the sorted dict.

To implement Top-N, we need a hash table data structure such as `sortedcontainers.SortedDict()` that we can have a total order on its keys (keys represent frequencies). The easy way to build a Top-N list in Python is to use `SortedDict()`, a dictionary that further provides a total ordering on its keys. The dictionary is ordered according to the natural ordering of its keys. We will keep adding `(frequency, url)` to the `SortedDict()`, but keep its size at  $N$  (note that `frequency` is the key since `SortedDict()` entries are sorted by its key). When the size is  $N+1$ , we will pop out the smallest frequency by `SortedDict.popitem(0)`.

Example and usage of `SortedDict()` is given below:

```
>>> from sortedcontainers import SortedDict
>>> sd = SortedDict({10: 'a', 2: 'm', 3: 'z', 5: 'b', 6: 't', 100: 'd', 20: 's'})
>>> sd
SortedDict({2: 'm', 3: 'z', 5: 'b', 6: 't', 10: 'a', 20: 's', 100: 'd'})
>>> sd.popitem(0)
(2, 'm')
>>> sd
SortedDict({3: 'z', 5: 'b', 6: 't', 10: 'a', 20: 's', 100: 'd'})
>>> sd[50] = 'g'
>>> sd
SortedDict({3: 'z', 5: 'b', 6: 't', 10: 'a', 20: 's', 50: 'g', 100: 'd'})
>>> sd.popitem(0)
(3, 'z')
>>> sd
SortedDict({5: 'b', 6: 't', 10: 'a', 20: 's', 50: 'g', 100: 'd'})
>>> sd[9] = 'h'
>>> sd
SortedDict({5: 'b', 6: 't', 9: 'h', 10: 'a', 20: 's', 50: 'g', 100: 'd'})
>>> sd.popitem(0)
(5, 'b')
>>> sd
SortedDict({6: 't', 9: 'h', 10: 'a', 20: 's', 50: 'g', 100: 'd'})
>>>
>>> len(sd)
6
```

Next, I present a Top-10 solution using PySpark.

## PySpark Solution

The PySpark solution is pretty straightforward: we partition the RDD into  $M$  partitions (where  $M > 1$ ) by using the `mapPartitions()` transformation. Each mapper will find a local “top-N” list (for  $N > 0$ ) and then will pass it to a single reducer. Then the single reducer will find the final “top-N” list from all local “top-N” list passed from mappers. In general, in most of the MapReduce algorithms, having a single reducer is problematic and will cause a performance bottleneck (the reason for bottleneck is that one reducer in one server receives all data — which can be very big data volume — and all other cluster nodes do nothing, so the entire pressure and load will be on a single node, which can cause performance bottlenecks). Here, our single reducer will not cause a

performance problem. Here is how: let's assume that we will have 5,000 partitions, then each mapper will only generate 10 (key, value) pairs. Therefore, our single reducer will only get 50,000 records (which is not that much data at all to cause performance bottleneck!).

Our high level solution for Top-10 Design Pattern is illustrated by Figure 19.3.

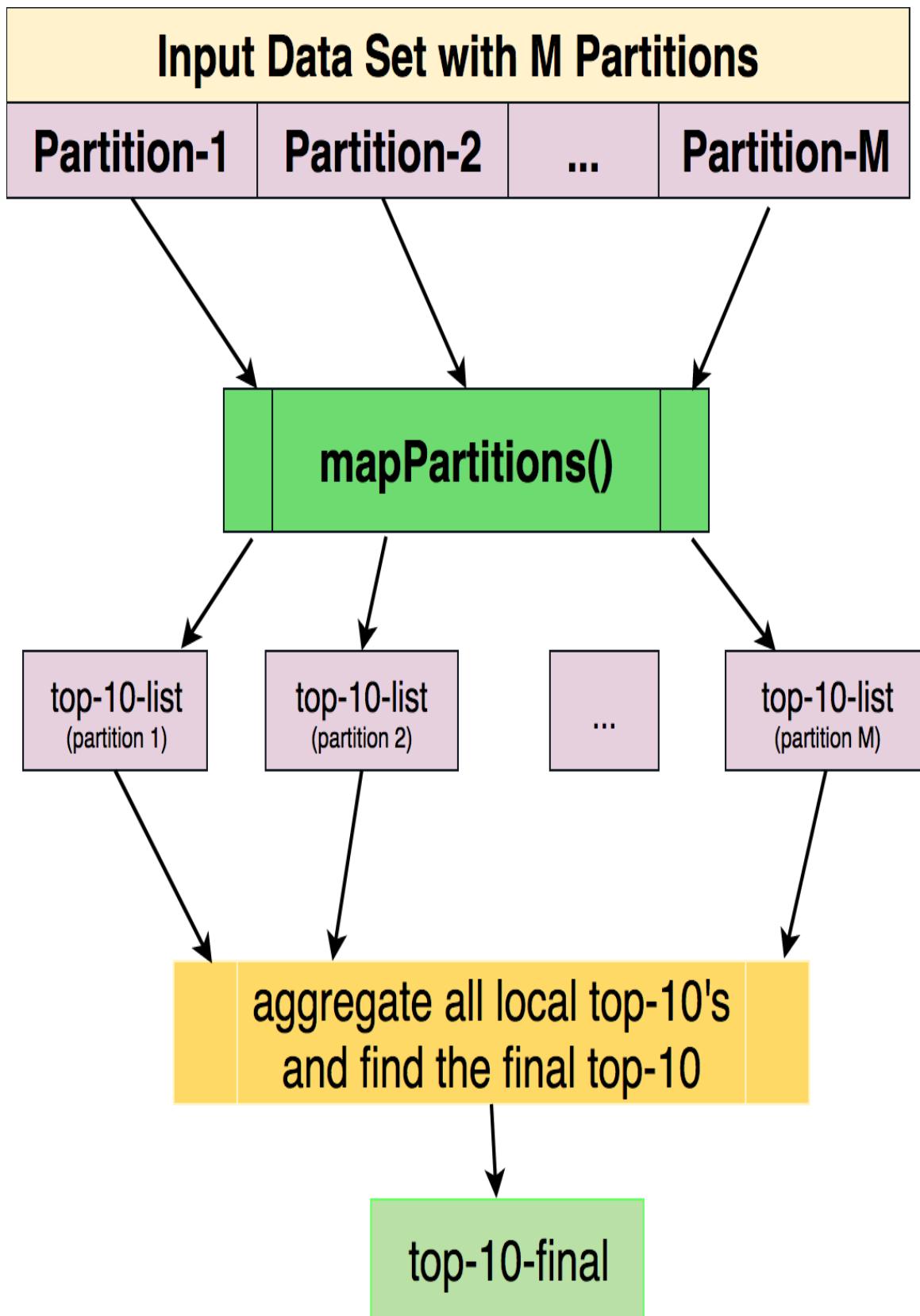


Figure 11-4. High Level Solution for Top-10 Design Pattern

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

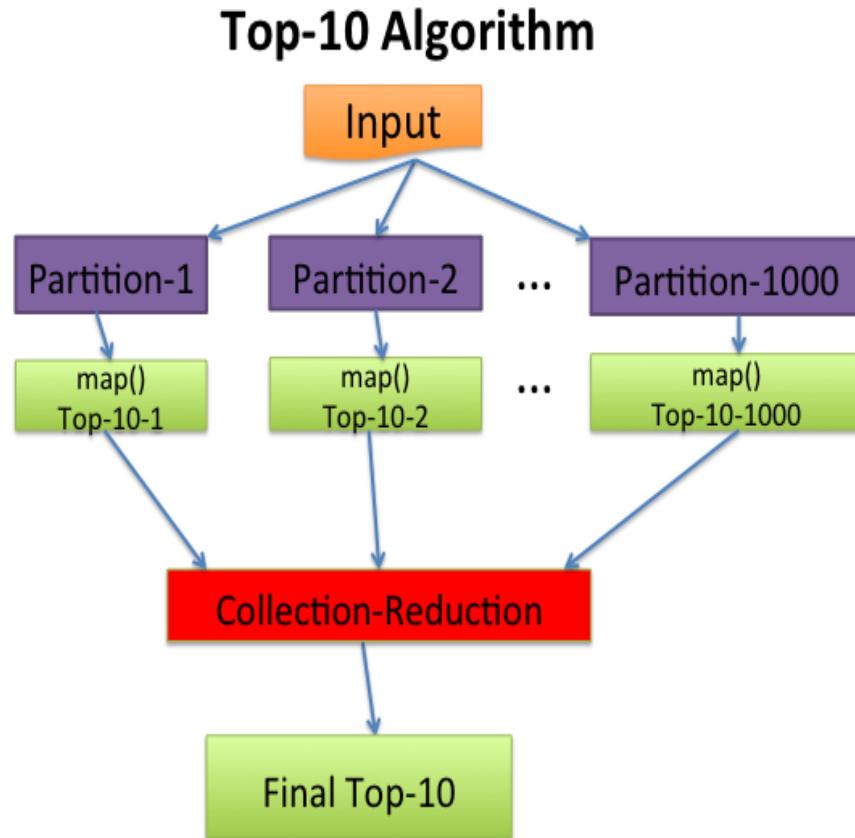


Figure 11-5. Top-10 MapReduce Algorithm: for Unique Keys

Let `spark` be an instance of a `SparkSession`. This is how the Top-10 is solved by using the `mapPartitions()` transformation and a custom Python function named `top10_design_pattern()`:

```

pairs = [('url-1', 10), ('url-2', 8), ('url-3', 4), ...]
rdd = spark.sparkContext.parallelize(pairs)
mapPartitions(f, preservesPartitioning=False)
Return a new RDD by applying a function

```

```
to each partition of this RDD.
top10 = rdd.mapPartitions(top10_design_pattern)
```

To complete the implementation, we present the `top10_design_pattern()` function, which finds top-10 for each partition (each partition is a set of (key, value) pairs):

```
from sortedcontainers import SortedDict
#
def top10_design_pattern(partition_iterator): ①
 sd = SortedDict() ②
 for url, frequency in partition_iterator: ③
 sd[frequency] = url ④
 if (len(sd) > 10): ④
 sd.popitem(0) ⑤
 #end-for
 print("local sd=", sd)
 #
 pairs = [(k, v) for k, v in sd.items()] ⑥
 print("top 10 pairs=", pairs)
 return pairs ⑦
#end-def
```

- ① The `partition_iterator` is an iterator for a single partition; an iterator over a set of (URL, frequency)
- ② Create an empty `SortedDict` of (Integer, String)
- ③ Iterate over a set of (URL, frequency)
- ④ Put (frequency, URL) into `SortedDict`
- ⑤ Keep/limit the size of sorted dictionary to 10 (remove the lowest frequency)
- ⑥ Convert `SortedDict` (which is a local top-10) into a list of (k, v)
- ⑦ Return a local top-10 list for a single partition

This is how it works: input is divided into a number of partitions. The number of partitions can be 8, 20, 100, 4000, ... — depends on the size of cluster and number of available memory, disks, CPU, and other resources. Note that a

programmer can set the number of partitions explicitly as well. Each partition is a set of (`URL`, `frequency`) pair elements. Each mapper accepts a partition of elements where each element is a pair of (`URL`, `frequency`). After mapper finishes creating a top-10 list as `SortedDict[Integer, String]`, then the function returns the local top-10 list. Note that we use a single dictionary (such as a `SortedDict`) per partition (and not per element of source RDD).

## Implementation in PySpark

The PySpark implementation is presented below. Note that this solution assumes that all URLs (i.e., keys) are unique. If The URLs are not unique, then you may use the `reduceByKey()` transformation to make all keys unique.

```
cat /tmp/urls.txt
url-0001,12
url-0002,22
url-0003,1000
url-0004,199
...
...
```

## How to Find Bottom-10

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

Replace the following

```
find top-10
if (len(sd) > 10): ①
 sd.popitem(0) ②
```

With

```
find bottom-10
if (len(sd) > 10): ③
 sd.popitem(-1) ④
```

- ① if size of `SortedDict` is larger than 10

- ② then remove the lowest frequency URL from the dictionary
- ③ if size of `SortedDict` is larger than 10
- ④ then remove the highest frequency URL from the dictionary

Next let's discuss how to partition your input and RDDs. Partitioning RDD is a combination of art and science. What is the right number of partitions for your cluster? There is no magic silver bullet formula for calculating the proper number of partitions. This does depend on the number of cluster nodes, the number of cores per server, and the size of RAM available. My experience indicates that you need to set this by trial and experience. The general rule of thumb is to use the following per executor.

```
2 * num_executors * cores_per_executor
```

Also, when you want to create your first RDD, you can set the number of partitions as:

```
input_path = "/data/my_input_path"
desired_num_of_partitions = 16
rdd = spark.sparkContext.textFile(input_path, desired_num_of_partitions)
```

For example the preceding example creates an `RDD[String]` with 16 partitions. If you do not set the number of partitions explicitly, then the Spark cluster manager will set it to a default number (based on the resources available from the cluster).

If your RDD is already created with some partitions, then you may reset the new number of partitions by using the `coalesce()` function: let `rdd` be an `RDD[T]`:

```
rdd : RDD[T]
desired_number_of_partitions = 40
rdd2 = rdd.coalesce(desired_number_of_partitions)
```

The newly created `rdd2` (as `RDD[T]`) will have 40 partitions. According to Spark documentation, the `coalesce()` function is defined as:

```
pyspark.RDD.coalesce:
coalesce(numPartitions, shuffle=False)
```

Description:

Return a new RDD that is reduced into numPartitions partitions.

Next, I introduce the **MinMax** design pattern, which the goal is to find (**minimum**, **maximum**) of large data set of numbers.

## MinMax

Given a set of billions of numbers, the goal is to find the minimum, maximum, and count of all of the given numbers. MinMax is a “Numerical Summarizations” design pattern. This pattern can be used in the scenarios where the data you are dealing with or you want to aggregate is of numerical type and the data can be grouped by specific fields. To understand the concept of **MinMax** design pattern, I am going to present three different solutions with quite different performances.

### Solution-1: Classic MapReduce

The naive approach will be to emit the following (key, value) pairs:

```
("min", number)
("max", number)
("count", 1)
```

for all of the numbers in the given input data set, then sort and shuffle will group all values by three keys: “min”, “max”, and “count”, and finally, we may use a reducer to iterate all numbers and find “min”, “max”, and “count” for all numbers. The problem with this approach is that we have to move billions of (key, value) pairs in the network and then create three huge `Iterable<Long>` (assuming the data type of numbers is Long data type). This is a possible solution, which might not scale out and might have serious performance problems. In the reduction phase, this solution will not utilize the whole cluster due to having only three unique keys: “min”, “max”, and “count”.

### Solution-2: Sorting

The next solution will sort all numbers and then you get the top (for “max”) and bottom (for “min”) and “count” of the data set. If the performance is acceptable, then it is a good solution. But if you have a lot of numbers, then sorting time might not be acceptable.

### **Solution-3: Spark’s mapPartitions()**

The final solution (the most efficient from performance and scalability point of view) splits data into N chunks (partitions), and then uses Spark’s `mapPartitions()` transformation to emit three (`key, value`) pairs from a single partition:

```
("min", the-minimum-number-in-a-partition)
("max", the-maximum-number-in-a-partition)
("count", count-of-numbers-in-a-partition)
```

Finally, we find the min, max, and count from all partitions. This solution scales out very well. No matter how many partitions we have, this will work and will not create OOM error. For example, if you have 500 billion numbers (assume one or more numbers per record), then partition it by 100,000 (in worst case, each partition will have 5 million records — one number per record), which means that each partition will get about 5 million numbers. Then each partition will emit three pairs as illustrated above. Finally, you find min, max, and count of  $100,000 \times 3 \text{ pairs} = 300,000$  numbers. Finding min, max, and count for 300,000 numbers is very trivial and will not cause any scalability problem at all.

The high-level solution is illustrated by the Figure 19.5.

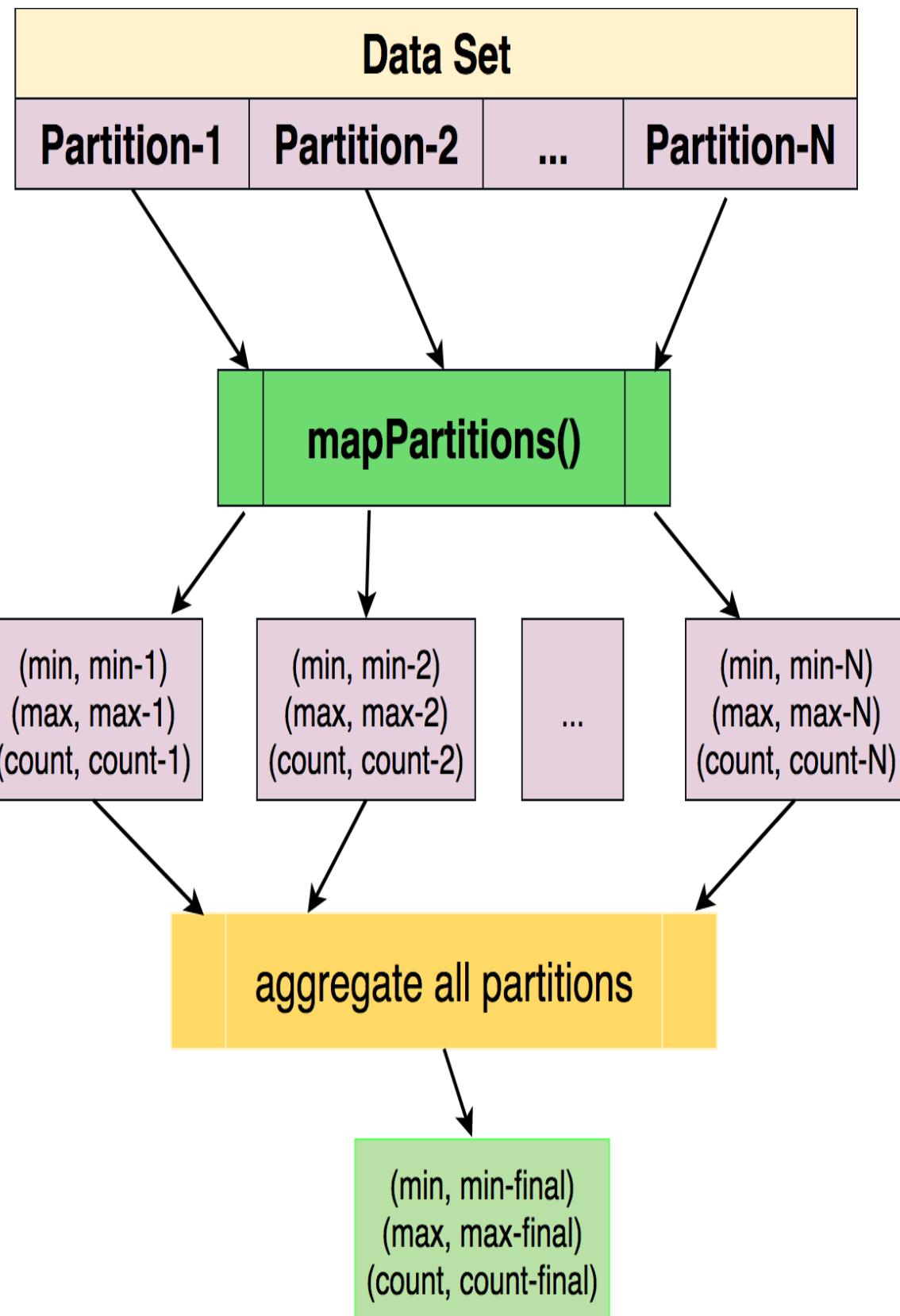


Figure 11-6. High Level Solution for MinMax Design Pattern

## Solution-3 Input

We assume that our input records have the following format:

```
<number>,><number>,><number>...
```

Sample records are presented below:

```
10,345,24567,2,100,345,9000,765
2,34567,23,13,45678,900
...
```

## PySpark Solution

Here, we present the PySpark solution for solving the MinMax problem. For details see the `minmax_use_mappartitions.py` program.

```
input_path = <your-input-path>
rdd = spark.sparkContext.textFile(input_path) ❶
min_max_count = rdd.mapPartitions(minmax) ❷
min_max_count_list = min_max_count.collect() ❸
final_min_max_count = find_min_max_count(min_max_count_list) ❹
```

- ❶ Return a new RDD from the given input
- ❷ Return an RDD of (`min`, `max`, `count`) from each partition by applying the `minmax` function
- ❸ Collect (`min`, `max`, `count`) from all partitions as a list
- ❹ Find the (`final_min`, `final_max`, `final_count`) by calling the `find_min_max_count()`

Next, `minmax()` function is presented:

```
def minmax(iterator): ❶

 print("type(iterator)=", type(iterator)) ❷
 # ('type(iterator)=' , <type 'itertools.chain'>)
 #
```

```

 first_time = False ❸
#
for record in iterator: ❹
 numbers = [int(n) for n in record.split(",")] ❺
 if (first_time == False): ❻
 # initialize count, min, max to the 1st record values
 local_min = min(numbers)
 local_max = max(numbers)
 local_count = len(numbers)
 first_time = True
 else: ❼
 # update count, min, and max
 local_count += len(numbers)
 local_max = max(max(numbers), local_max)
 local_min = min(min(numbers), local_min)
#end-for
return [(local_min, local_max, local_count)] ❽
#endif

```

- ❶ The `iterator` is a type of `itertools.chain`
- ❷ Print the type of iterator (for debugging only)
- ❸ Define some variables for return
- ❹ Iterate the iterator (record holds a single record)
- ❺ Tokenize input and build an array of numbers
- ❻ If its the first record, then find out `min`, `max`, and `count`
- ❼ If not the first record, then update `local_min`, `local_max`, and `local_count`
- ❽ Finally return a triplet from each partition

What if some of the partitions are empty? Is it possible for a partition to be empty? Yes, there are many reasons for that (according to Spark documentation). What is an empty partition? An empty partition is a partition with no data at all. Therefore, we have to handle empty partitions gracefully (in case there are empty partitions).

Here I will show how to handle empty partitions. Error handling in Python is done through the use of exceptions that are caught in `try` blocks and handled in `except` blocks. Try and Except in Python is defined as: If an error is encountered, a `try` block code execution is stopped and transferred down to the `except` block.

```
def minmax(iterator): ①
 #
 print("type(iterator)=", type(iterator)) ②
 # ('type(iterator)=' , <type 'itertools.chain'>)
 #
 try:
 first_record = next(iterator) ③
 except StopIteration: ④
 return [(1, -1, 0)] # WHERE min > max to filter out later
 #
 # initialize count, min, max to the 1st record values
 numbers = [int(n) for n in first_record.split(",")] ⑤
 local_min = min(numbers)
 local_max = max(numbers)
 local_count = len(numbers)
 #
 for record in iterator: ⑥
 numbers = [int(n) for n in record.split(",")]
 # update min, max, count
 local_count += len(numbers)
 local_max = max(max(numbers), local_max)
 local_min = min(min(numbers), local_min)
 # end-for
 return [(local_min, local_max, local_count)] ⑦
```

- ① The `iterator` is a type of `itertools.chain`
- ② Print the type of iterator (for debugging only)
- ③ Try to get the first record from a given iterator, if successful, then the `first_record` is initialized to the first record of a partition
- ④ If you are here, then it means that the partition is empty, return a fake triplet
- ⑤ Set `min`, `max`, and `count` from the first record
- ⑥ Iterate the `iterator` for 2nd, 3rd, ... records (record holds a single record)

- ⑦ Finally return a triplet from each partition

How should we test handling empty partitions? The program `minmax_force_empty_partitions.py` (source code of Chapter 12 in GitHub) forces to create empty partitions and handles empty partitions gracefully. How do you force to create empty partitions? Set the “number of partitions” high enough such as greater than the number of input records. For example, if your input has  $N$  records, then set the number of partitions to  $N+3$ , which the partitioner might create up to 3 empty partitions.

## The Composite Pattern and Monoids

This section explores the concept of the “composite pattern” and Monoids and will show how to use “composite pattern” and Monoids in the context of Spark and PySpark.

### Composite Pattern

What is the “composite pattern”? In a nutshell, the composite pattern is a structural design pattern (also called a partitioning design pattern) that can be used when a group of objects can be treated the same as a single object in that group. You may use the composite pattern to create hierarchies of objects and groups of objects, quickly turning the structure into a tree with leaves (objects) and composites (sub-groups). The composite pattern in UML notation is illustrated in Figure 19.6.

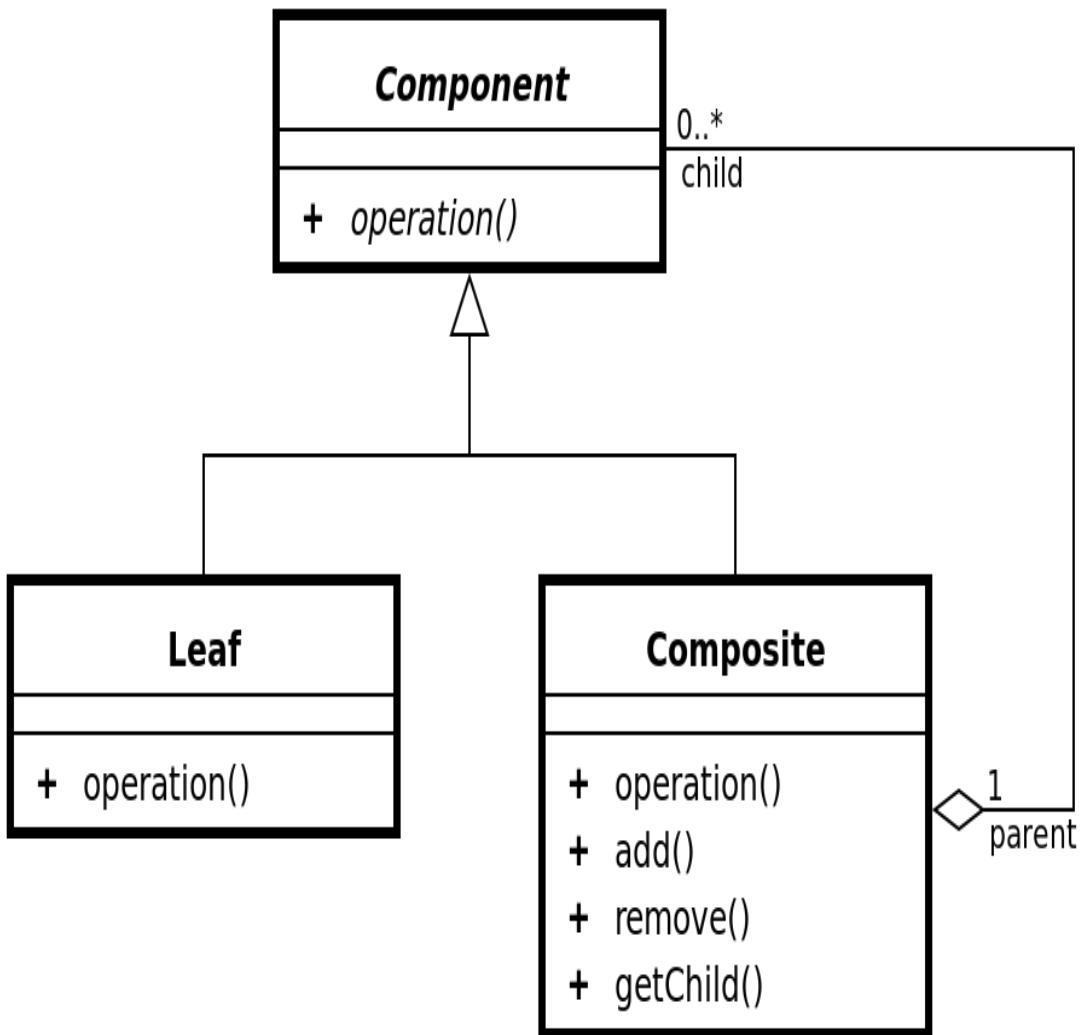


Figure 11-7. Composite Pattern Diagram

According to Wikipedia: “in software engineering, the composite pattern is a partitioning design pattern. The composite pattern describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies. Implementing the composite pattern lets clients treat individual objects and compositions uniformly.”

For example, using a tree data structure, composite pattern is illustrated as Figure 19.7. Another example of composite pattern will be adding a set of

numbers, which is illustrated as Figure 19.7. Here the numbers are the leaves and composites are the addition operator.

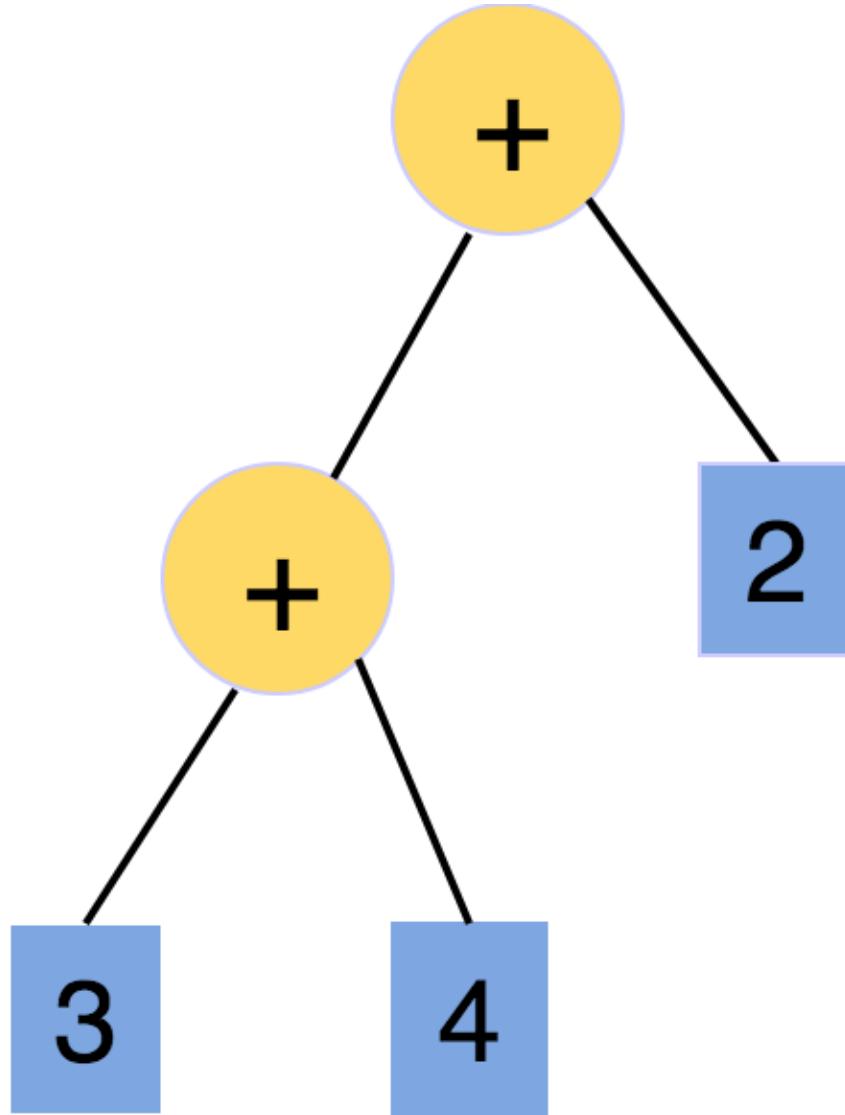


Figure 11-8. Composite Pattern Example: Addition

Next, I discuss the concept of monoids in the context of composite pattern.

## Monoids

We have already seen the use of monoids in using reduction transformations (chapter 4). Here we will look at the monoids in the context of composite pattern. From the composite pattern definition, it should be obvious to see that there is a commonality between **Monoids** and **composite patterns**. The

composite pattern is a design pattern, which is commonly used in programming languages such as Java, Python, and Scala and also used in big data for composing (such as addition and concatenation operators) and aggregating a set of data points.

To observe the commonality between Monoids and composite patterns, let's take a look at the definition of “Monoids” (source: [Wikipedia](#)):

In abstract algebra, a branch of mathematics, a monoid is an algebraic structure with a single associative binary operation and an identity element. Monoids are studied in semigroup theory, because they are semigroups with identity. Monoids occur in several branches of mathematics; for instance, they can be regarded as categories with a single object. Thus, they capture the idea of function composition within a set. In fact, all functions from a set into itself form naturally a monoid with respect to function composition. Monoids are also commonly used in computer science, both in its foundational aspects and in practical programming.

An application of monoids in computer science is so-called the “MapReduce” programming model. The MapReduce programming model, consists of three functions: `map()`, `combine()`, and `reduce()`. These three functions are very similar to the `map()` and `flatMap()` functions (`combine()` is an optional operation) and `reduce()` transformations in Spark. Given a dataset, `map()` consists of mapping arbitrary data to elements of a specific monoid, `combine()` aggregates/folds data in local level (worker nodes in cluster), and finally the `reduce()` consists of aggregating/folding those elements, so that in the end we produce just one element.

So in terms of a programming language semantics, a monoid is just an interface with one abstract value and one abstract method. The abstract method for a Monoid is the append operation (it can be an addition operator on integers and can be a concatenation operator on string objects). The “abstract value” for a Monoid is the identity value, for example, for adding a set of integer numbers, identity value is zero and for concatenating strings, identity value is an empty string (string of length zero). Note that, the identity value is defined as the value you can append to any value that will always result in the original value unmodified. For example, the identity value for collections data structures is the

empty collection because appending a collection to an empty collection will typically produce the same collection unmodified.

In this paper: [Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms](#), Jimmy Lin clearly defines and relates monoids as a design principle for efficient MapReduce algorithms. But what is a monoid, what properties defines it, and how does it aid the MapReduce paradigm. Following abstract algebra, “a monoid is an algebraic structure with a single associative binary operation and an identity element”. Jimmy Lin shows that when your MapReduce operations (`map()`, `reduce()`, and `reduceByKey()` transformations in Spark) are not monoids, then it is very hard (and may be impossible) to use “combiners” efficiently. Next we will briefly review MapReduce’s combiners and abstract algebra’s monoids and see how they are related to each other.

Also, [David Saile](#) states that “recall that a monoid is an algebraic structure with a single associative binary operation and an identity element. For example, the natural numbers  $\mathbb{N}$  (the term “natural numbers” refers either to the set of positive integers  $\{1, 2, 3, \dots\}$  or to the set of non-negative integers  $\{0, 1, 2, 3, \dots\}$ ) form a monoid under addition with identity element zero. In classic MapReduce, the mapper is not constrained, but the reducer is required to be (the iterated application of) an associative operation. Recent research argued that reduction is in fact monoidal in known applications of MapReduce. That is, reduction is indeed the iterated application of an associative operation  $f()$  with a unit  $u$ . In the case of the word-occurrence count example, reduction iterates addition  $+$  with  $0$  as unit. The parallel execution schedule may be more flexible if commutativity is required in addition to associativity.”

In MapReduce framework, the combiner (as an optional plug-in component) is a “local-reduce” process which operates only on data generated by one server. Successful use of combiners reduces the amount of intermediate data generated by the mappers on a given single server (that is why it is so-called a local reducer). Combiners can be used as a MapReduce optimization to reduce network traffic (by decreasing size of the transient data as  $(key, value)$  pairs) between mappers and reducers. Typically, combiners have the same interface as reducers. The combiner must have the following characteristics:

- Combiners receives as input all the data emitted by the mapper instances on a given server (this is called a local aggregation)
- Combiners output is sent to the reducers — some programmers call this as a local server reduction!
- The combiner must be side-effect free; combiners may run an indeterminate number of times.
- The combiner must have the same input and output key types (see example below)
- The combiner must have the same input and output value types (see example below)
- The combiner runs in memory after the map phase

Therefore, a combiner skeleton should be defined as:

```
key: as KeyType
values: as Iterable<ValueType>
def combine(key, values):
 ...
 # use key and values to create new_key and new_value
 new_key = <a-value-of-KeyType>
 new_value = <a-value-of-ValueType>
 ...
 return (new_key, new_value);
...
#end-def
```

This template indicates that `(key, value)` pairs generated by a combiner has to match the `(key, value)` pairs received (as an input) by reducers. For example, if mapper outputs `(T~1~, T~2~)` pairs (`key` is type `T~1~` and value is type `T~2~`) then a combiner has to emit `(T~1~, T~2~)` pairs as well.

The MapReduce/Hadoop does not have an explicit `combine()` function, we just use `reduce()` in Hadoop to implement combiners, but we use a plugin of `Job.setCombinerClass()` to define a combiner class.

Furthermore, Jimmy Lin concludes that “one principle for designing efficient MapReduce algorithms can be precisely articulated as follows: create a monoid

out of the intermediate value emitted by the mapper. Once we “monoidify” the object, proper use of combiners and the in-mapper combining techniques becomes straightforward.” (source: [Monoidify! Monoids as a Design Principle for Efficient MapReduce Algorithms](#))

Some programming languages have direct support for monoids. For example, the Haskell programming language has a direct support for monoids. In Haskell, “a monoid is a type with a rule for how two elements of that type can be combined to make another element of the same type.”

## Definition of Monoid

Monoid is a triplet  $(S, f, e)$ , where

- $S$  is a set
  - $S$  is called underlying set of monoid
- $f: S \times S \rightarrow S$ 
  - $f$  is a mapping called binary operation of monoid
- $e \in S$ 
  - $e$  is the identity operation of monoid

A monoid with binary operation  $+$  (note that, here, the binary operation is denoted by  $+$  and it does not mean a mathematical addition operator) satisfies the following three axioms (note that  $f(a,b) = a + b$ ):

- Closure: for all  $a, b$  in  $S$ , the result of the operation  $(a + b)$  is also in  $S$ .
- Associativity: for all  $a, b$  and  $c$  in  $S$ , the following equation holds:

$$((a + b) + c) = (a + (b + c))$$

- Identity element: there exists an element  $e$  in  $S$ , such that for all elements  $a$  in  $S$ , the following two equations hold:

```
e + a = a
a + e = a
```

And in mathematical notation we can write these as

- Closure:

```
forall a,b in S: a + b in S
```

- Associativity:

```
for all a,b,c in S: ((a + b) + c) = (a + (b + c))
```

- Identity element:

```
{
exists e in S:
 for all a in S:
 e + a = a
 a + e = a
}
```

A monoid might have other properties: The monoid operator might (but isn't required to) obey other properties like:

- idempotency:

```
{ for all a in S: a + a = a }
```

- commutativity:

```
{ for all a, b in S: a + b = b + a }
```

## How to form a Monoid?

To form a monoid first we need a type  $S$ , which can define a set of values such as integers:  $\{0, -1, +1, -2, +2, \dots\}$ . The second component is a binary function:

$+ : S \times S \rightarrow S$

Then we need to make sure that for any two values  $x$  in  $S$  and  $y$  in  $S$  we get a result object, the combination of  $x$  and  $y$ :

$$x + y : S$$

For example, let type  $S$  be a set of integers, then the binary operation may be addition (+), multiplication (\*) or division (/). Finally, as the third and most important ingredient we need binary operation to follow a set of laws. If it does, then  $S$  together with binary operation (+) is called a monoid. We say:  $(S, +, e)$  is a monoid, where  $e$  in  $S$  is the identity element (such as 0 for addition and 1 for multiplication). Also note that the binary division operator ('/') over a set of real numbers is not a monoid:

$$\begin{aligned} ((12 / 4) / 2) &\text{ not equal } (12 / (4 / 2)) \\ ((12 / 4) / 2) &= (3 / 2) = 1.5 \\ (12 / (4 / 2)) &= (12 / 2) = 6.0 \end{aligned}$$

In a nutshell, monoids capture the notion of combining arbitrarily many things into a single thing together with a notion of an empty thing called the identity. One simple example is addition on natural numbers  $\{1, 2, 3, \dots\}$ . The addition function  $+$  allows us to combine arbitrarily many natural numbers into a single natural number, the sum. The identity is the number zero. Another example is string concatenation. The concatenation operator allows us to combine arbitrarily many strings into a single string. The identity value is the empty concatenation, the empty string.

## Monoidic and Non-Monoidic Examples

To understand the concept of monoids, below, I have listed some monoid and non-monoid examples. For example, to use Spark's `reduceByKey()` effectively, you must make sure that the reduction function is a monoid, since Spark uses combiners on the `reduceByKey()` transformation.

## Subtraction over Set of Integers

The set  $S = \{0, 1, 2, \dots\}$  is a commutative monoid for the MAX (maximum) operation, whose identity element is 0 (zero).

$\text{MAX}(a, \text{MAX}(b, c)) = \text{MAX}(\text{MAX}(a, b), c)$   
 $\text{MAX}(a, 0) = a$   
 $\text{MAX}(0, a) = a$   
 $\text{MAX}(a, b) \in S$

## Subtraction over Set of Integers

Subtraction operator (-) over a set of integers does not define a monoid; this operation is not associative:

$$(1 - 2) - 3 = -4$$

$$1 - (2 - 3) = 2$$

## Addition over Set of Integers

Addition operator (+) over a set of integers defines a monoid; this operation is commutative and associative and the identity element is 0:

$$(1 + 2) + 3 = 6$$

$$1 + (2 + 3) = 6$$

$$n + 0 = n$$

$$0 + n = n$$

we can formalize this monoid as (below  $e(+)$  defines an identity element):

$$S = \{0, -1, +1, -2, +2, -3, +3, \dots\}$$

$$e(+) = 0 \quad (\text{note: identity element is } 0)$$

$$f(a, b) = f(b, a) = a + b$$

## Multiplication over Set of Integers

The natural numbers,  $N = \{0, 1, 2, 3, \dots\}$ , form a commutative monoid under multiplication (identity element one).

## Mean over Set of Integers

On the other hand, the natural numbers,  $N = \{0, 1, 2, 3, \dots\}$ , does not form a monoid under the mean (average) function. The following example shows that the mean of means of arbitrary subsets of a set of values is not the same as the mean of the set of values:

```

MEAN(1, 2, 3, 4, 5)
-- NOT EQUAL --
MEAN(MEAN(1,2,3), MEAN(4,5))

MEAN(1, 2, 3, 4, 5) = (1+2+3+4+5)/5
 = 15/5
 = 3

MEAN(MEAN(1,2,3), MEAN(4,5)) = MEAN(2, 4.5)
 = (2 + 4.5)/2
 = 3.25

```

Therefore, if you want to find average of values for a pair RDD (as `RDD[(key, integer)]`), then you may not use the following transformation (which might yeild a wrong value due to aprtitioning):

```

let rdd be a RDD[(key, integer)]
average_per_key = rdd.reduceByKey(lambda x, y: (x+y)/2)

```

The correct way to find average per key is to make it a monoid:

```

let rdd be a RDD[(key, integer)]
create value as (sum, count) pair: this makes a monoid
rdd2 = rdd.mapValues(lambda n: (n, 1))
find (sum, count) per key
sum_count = rdd2.reduceByKey(lambda x, y: (x[0]+y[0], x[1]+y[1]))
now, given (sum, count) per key, find the average per key
average_per_key = sum_count.mapValues(lambda x: x[0]/x[1])

```

## Non-Commutative Example

For a non-commutative example, consider the collection of all binary strings: an element is a finite ordered sequence of 0's and 1's. The binary operation is just concatenation: e.g.  $\text{concat}(1011, 001001) = 1011001001$ . The identity element is the “empty string” (String of size 0). Therefore concatenation of binary strings is a monoid.

## Median over Set of Integers

The natural numbers does not form a monoid under the MEDIAN function:

```

MEDIAN(1, 2, 3, 5, 6, 7, 8, 9)
-- NOT EQUAL --
MEDIAN(MEDIAN(1,2,3), MEDIAN(5,6,7,8,9))

MEDIAN(1, 2, 3, 5, 6, 7, 8, 9)
= (5 + 6) / 2
= 11 / 2
= 5.5

MEDIAN(MEDIAN(1,2,3), MEDIAN(5,6,7,8,9))
= MEDIAN(2, 7) =
= (2 + 7) / 2
= 9 / 2
= 4.5

```

## Concatenation over Lists

A good example of a monoid is list of objects. Lists with concatenation (+) and the empty list (as []) are a monoid: for any list, we can write:

```

Let L be a list
L + [] = L
[] + L = L

```

Also, note that the concatenation function is associative. Given two lists, say [1,2,3] and [7,8], you can join them together using + to get [1,2,3,7,8]. There's also the empty list []. Using + to combine [] with any list gives you back the same list, for example []+[1,2,3] = [1,2,3] and [1,2,3]+[] = [1,2,3].

## Union and Intersection over Integers

Sets under union or intersection over a set of integers forms a monoid.

## Matrix Example

Let  $N = \{1, 2, 3, \dots\}$ . Let  $m, n \in N$ . Then the set of  $m \times n$  matrices with integer entries, written as  $\mathbf{Z}^{m \times n}$  satisfies properties that make it a monoid under addition:

- closure is guaranteed by the definition
- the associative property is guaranteed by the associative property of its elements; and
- the additive identity is  $\mathbf{0}$ , the zero matrix

What have we learned from these examples and monoids concepts? Spark's `reduceByKey()` is an efficient transformation, which merges the values for each key using an associative and commutative reduce function. Therefore, we have to make sure that `reduceByKey()`'s reduce function is a monoid, otherwise you might not get correct reduction results.

## Not a Monoid Example

Given a large number of (key, value) pairs where the keys are strings and the values are integers, the goal is to find the average of all the values by key. In SQL, this is accomplished as (assuming that our table — named as `mytable` — has `key` and `value` columns):

- Select All Data

```
SELECT key, value FROM mytable
```

| key  | value |
|------|-------|
| ---  | ----- |
| key1 | 10    |
| key1 | 20    |
| key1 | 30    |
| key2 | 40    |
| key2 | 60    |

```
key3 20
key3 30
```

- Select All Data and GROUP BY key

```
SELECT key, AVG(value) as avg FROM mytable GROUP BY key
```

```
key avg
--- ---
key1 20
key2 50
key3 25
```

Here is the first version of MapReduce algorithm, where the mapper is not generating monoid outputs for the mean/average function.

- Mapper function:

```
key a string object
value an long associated with key
map(key, value) {
 emit(key, value);
}
```

- Reducer function:

```
#key a string object
values as list of long data type numbers
reduce(key, values) {
 sum = 0
 count = 0
 for (i : list) {
 sum = sum + i
 count++;
 }
 average = sum / count
 emit(key, average)
}
```

Two observations from the first version of MapReduce algorithm:

- the algorithm is not very efficient: since there will be too much work by “sort and shuffle” functions of MapReduce framework

- We can not use the reducer as a combiner: since we know that the mean of means of arbitrary subsets of a set of values is not the same as the mean of the set of values.

Note that using combiners make Spark, MapReduce, and distributed algorithms efficient by reducing network traffic (you need to ensure that the combiner provides sufficient aggregation) and reducing the load of “sort and shuffle” functions of MapReduce framework. Now the question is how can we use our reducer to work as a combiner? The answer is to make output of the mapper to be a monoid: we change the output of a mapper. Once your mapper outputs monoids, then combiners and reducers will behave correctly.

Next, let's look a monoid example.

## Monoid Example

In this section, I am revising the mapper to generate `(key, value)` pairs where `key` is the string and `value` is a pair `(sum, count)`, which has a monoid property. The `(sum, count)` data structure is a monoid and the identity element is `(0, 0)`. The proof is given below:

Monoid type is `(N, N)` where `N = {set of integers}`

Identity element is `(0, 0)`:

$$\begin{aligned} (\text{sum}, \text{count}) + (0, 0) &= (\text{sum}, \text{count}) \\ (0, 0) + (\text{sum}, \text{count}) &= (\text{sum}, \text{count}) \end{aligned}$$

Let `a = (sum1, count1)`, `b = (sum2, count2)`, `c = (sum3, count3)`

Then associativity holds:

$$(a + (b + c)) = ((a + b) + c)$$

`+` is the binary function:

$$a + b = (\text{sum1+sum2}, \text{count1+count2}) \text{ in } (N, N)$$

Now, let's write a mapper (for MapReduce paradigm) for a Monoid data type

```
key a string object
value a long data type associated with key
emits (key, (sum, count))
map(key, value) {
```

```

 emit (key, (value, 1))
}

```

As you can see, the key is the same as before, but the value is a pair of (sum, count). Now, the output of mapper is a monoid where the identity element is (0, 0). The element-wise sum operation can be performed as:

```

element1 = (key, (sum1, count1))
element2 = (key, (sum2, count2))

==> values for the same key are reduced as:

```

$$\begin{aligned}
&\text{element1} + \text{element2} \\
&= (\text{sum1}, \text{count1}) + (\text{sum2}, \text{count2}) \\
&= (\text{sum1+sum2}, \text{count1+count2})
\end{aligned}$$

Now the mean function will be calculated correctly since mappers output monoids: imagine values for a single key are {1, 2, 3, 4, 5} and {1, 2, 3} goes to partition-1 and {4, 5} goes to partition-2:

$$\begin{aligned}
&\text{MEAN}(1, 2, 3, 4, 5) \\
&= \text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5)) \\
&= (1+2+3+4+5) / 5 \\
&= 15 / 5 \\
&= 3
\end{aligned}$$

Partition-1:  
 $\text{MEAN}(1, 2, 3) = \text{MEAN}(6, 3)$

Partition-2:  
 $\text{MEAN}(4, 5) = \text{MEAN}(9, 2)$

Merging partitions:  
 $\text{MEAN}(\text{MEAN}(1, 2, 3), \text{MEAN}(4, 5))$   
 $= \text{MEAN}(\text{MEAN}(6, 3), \text{MEAN}(9, 2))$   
 $= \text{MEAN}(15, 5)$   
 $= 15 / 5$   
 $= 3$

The revised algorithm, where mappers (defined above) outputs are monoids is presented below (for a given pair (sum, count), `pair.1` refers to sum and `pair.2` refers to count):

```

key a string object
values is a list of pairs as: [(s1, c1), (s2, c2), ...]
combine(key, values) {
 sum = 0
 count = 0
 for (pair : values) {
 sum += pair.1
 count += pair.2
 }
 emit (key, (sum, count))
}

```

The `reduce()` transformation is presented below:

```

key a string object
value is a list of pair as [(s1, c1), (s2, c2), ...]
reduce(key, values) {
 sum = 0
 count = 0
 for (pair : values) {
 sum += pair.1
 count += pair.2
 }
 average = sum / count
 emit (key, average)
}

```

Therefore, in order to guarantee that your combiners are executing correctly, you must make sure that your mappers are generating monoidic data types.

## PySpark Implementation of Monodized Mean

The goal of this section is to provide Monodized Mean solution: it means that combiners can be used to assist in aggregating values. To compute mean of numbers for the same key, we can group all values (using the Spark's `groupByKey()` transformation) per key and then find the sum and divide by the count of numbers. This is not an optimal solution due to the fact the `groupByKey()` might create OOM problems.

For Monodized Mean solution, for a given `(key, number)` we will emit `(key, (number, 1))`. The associated value for a key denotes a pair of `(sum, frequency)`. Earlier, I demonstrated that using `(sum, frequency)` for values enable

us to use combiners and reducers for proper calculation of the mean function. With this set up, we can use `reduceByKey()` transformation, which is very efficient reducer. How do we reduce? Given the following two pairs for the same key (in earlier section, I proved that `(sum, count)` is a monoid data type for finding average of values — therefore using `reduceByKey()` will work correctly):

```
(key, value1) = (key, (sum1, count1))
(key, value2) = (key, (sum2, count2))
```

This is how reduction function will work:

```
value1 + value2 =
(sum1, count1) + (sum2, count2) =
(sum1+sum2, count1+count2)
```

Once the reduction (by `reduceByKey()`) is done, then we need an additional mapper to find the average by dividing the sum by the count (which will be shown in the following subsections).

## Input

The input record format will be

Input Record Format:  
`<key-as-string>,<value-as-integer>`

Examples:

```
key1,100
key2,46
key1,300
```

## PySpark Solution

The high-level steps are presented by the following four steps:

- Step-1: Read input and create the first RDD as `RDD[String]`
- Step-2: Apply `map()` to create `RDD[key, (number, 1)]`

- Step-3: Perform reduction by `reduceByKey()`, which will create `RDD[key, (sum, count)]`
- Step-4: Apply `mapValue()` to create average RDD as `RDD[key, (sum / count)]`

The complete PySpark program (`average_monoid_driver.py`) is listed below.

First, we need two simple Python functions to help us in using Spark transformations.

First function: `create_pair` to accept a String object as “key,number” and returns `(key, (number, 1))` pair.

```
record as String of "key,number"
def create_pair(record):
 tokens = record.split(",")
 key = tokens[0]
 number = int(tokens[1])
 return (key, (number, 1))
end-def
```

Second function: `add_pairs` accept two tuples of `(sum1, count1)` and `(sum2, count2)` and returns sum of tuples `(sum1+sum2, count1+count2)`.

```
a = (sum1, count1)
b = (sum2, count2)
def add_pairs(a, b):
 # sum = sum1+sum2
 sum = a[0] + b[0]
 # count = count1+count2
 count = a[1] + b[1]
 return (sum, count)
end-def
```

Here is the complete PySpark solution:

```
from __future__ import print_function ①
import sys ②
from pyspark.sql import SparkSession ③

if len(sys.argv) != 2: ④
 print("Usage: average_monoid_driver.py <file>", file=sys.stderr)
 exit(-1)
```

```

spark = SparkSession.builder.getOrCreate() ⑤

sys.argv[0] is the name of the script.
sys.argv[1] is the first parameter
input_path = sys.argv[1] ⑥
print("input_path: {}".format(input_path))

read input and create an RDD<String>
records = spark.sparkContext.textFile(input_path) ⑦

create a pair of (key, (number, 1)) for "key,number"
key_number_one = records.map(create_pair) ⑧

aggregate the (sum, count) of each unique key
sum_count = key_number_one.reduceByKey(add_pairs) ⑨

create the final RDD as RDD[key, average]
averages = sum_count.mapValues(lambda (sum, count): sum / count) ⑩
print("averages.take(5): ", averages.take(5))

done!
spark.stop()

```

- ➊ Import the print() function
- ➋ Import System-specific parameters and functions
- ➌ Import SparkSession from the `pyspark.sql` module
- ➍ Make sure that we have 2 parameters in the command line
- ➎ Create an instance of a SparkSession object by using the builder pattern `SparkSession.builder` class
- ➏ Define input path (this can be a file or a directory containing any number of files)
- ➐ Read input and create the first RDD as `RDD[String]` where each object has this format: “key,number”
- ➑ Create `key_number_one` RDD as `(key, (number, 1))`

- ⑨ Aggregate (`sum1, count1`) with (`sum2, count2`) and create (`sum1+sum2, count1+count2`) as values
- ⑩ Apply the `mapValues()` transformation to find final average per key

## Conclusion on Using Monoids

We observed that in MapReduce paradigm (which is a foundation of Hadoop, Spark, Tez, ...), if your mapper generates monoids then you can utilize combiners for optimization and efficiency purposes (using combiners reduces network traffic and make MapReduce's "sort and shuffle" functions more efficient by processing less data). Also, we showed that how to monodify MapReduce algorithms. The challenge to us is to monodify our MapReduce algorithms. In general, combiners can be used when the function you want to apply is both commutative and associative (properties of a monoid). For example, classic word count is a monoid over a set of integers with the `+` operation (here you can use a combiner). But the mean function (which is not associative as shown in the counter example above) over a set of integers does not form a monoid. Therefore, if combiners used properly, then it will significantly cuts down the amount of data shuffled from the maps to the reducers.

## Functors and Monoids

Now that we have seen the definition and use of monoids in MapReduce framework, we can even apply higher-order functions (like "functors") to monoids. A functor is an object that is a function (or we can say that it is a function and object at the same time). The Java JDK8 has a direct support for functors and it is named lambdas (for details see [Lambda project](#)). Programming languages JDK6 and JDK7 do not have direct concept of functors, because functions are not first-class objects in Java; it means that you can not pass a function name as an argument to another function. There is a simple way to simulate functors in Java by defining an interface and a method (this is a very simplistic simulation):

```

public interface FunctorSimulation<T1, T2> {
 T2 apply(T1 input);
}

```

First, we present a “functor” on a monoid by a simple example: Let  $\text{MONOID} = (t, e, f)$  be a monoid, where  $T$  is a type (set of values),  $e$  is the identity element, and  $f$  is the “ $+$ ” binary plus function:

```

MONOID = {
 type T
 val e : T
 val plus : T x T -> T
}

```

Then we define a functor  $\text{Prod}$  as

```

functor Prod (M : MONOID) (N : MONOID) = {
 type t = M.T * N.T
 val e = (M.e, N.e)
 fun plus((x1,y1), (x2,y2)) = (M.plus(x1,x2), N.plus(y1,y2))
}

```

Then we can define other functors such as  $\text{Square}$  as:

```
functor Square (M : MONOID) : MONOID = Prod M M
```

Also, a functor between two monoids can be defined as: Let  $(M_1, f_1, e_1)$  and  $(M_2, f_2, e_2)$  be monoids. A functor:

$$F : (M_1, f_1, e_1) \rightarrow (M_2, f_2, e_2)$$

is specified by a object map (monoids are categories with a single object) and an arrow map:  $F : M_1 \rightarrow M_2$  and the following conditions will hold:

$$\forall a,b \in M_1, F(f_1(a,b)) = f_2(F(a), F(b))$$

$$F(e_1) = e_2$$

A functor between two monoids is just a “monoid homomorphism.” For example, for String data type, function `Length()` that counts the number of letters in a word is a monoid homomorphism.

- `Length("") = 0` (length of an empty string is 0)

- If  $\text{Length}(x) = m$  and  $\text{Length}(y) = n$ , then concatenation  $x + y$  of strings has  $m + n$  letters, for example:

```

Length("String" + "ology")
= Length("Stringology")
= 11
= 6 + 5
= Length("String") + Length("ology")

```

Therefore creating monoids by mappers is a guarantee that the reducers can take advantage of using combiners effectively and correctly.

Next, we take a look at a **map-side join** design pattern, which can reduce the join performance time between two large data sets.

## Map-Side Join by RDDs

A join operation is used to combine rows from two (or more) tables, based on a common column (or columns) between them. Join is an expensive operation and since big data engines (such as Hadoop and Spark) do not support indexing of data tables, therefore, join can be an expensive. Here, I introduce a map-side join, which can reduce the cost of join between two tables. Map-side join is a process where two data sets are joined by the mapper rather than using the actual join function (which can happen by combination of a mapper and reducer). What are the advantages of using map-side join? The advantages of using map side join are as follows:

- Map-side join might help in minimizing the cost that is incurred for sorting and merging in the shuffle and reduce stages.
- Map-side join might help in improving the performance of the task by decreasing the time to finish the task.

Using SQL terminology, a JOIN clause is used to combine rows from two (or more) tables, based on a related column between them. Typically, a SQL JOIN is an expensive operation (unless you have built proper indexes). In general, joins on large sets are expensive, but very rarely do you want to join the entire contents of large table A with the entire contents of large table B.

To understand map side join, assume that we have two tables (using MySQL database): EMP and DEPT as defined below. Let's see how a join works between these two tables. Consider the following two tables EMP and DEP:

```
mysql> use testdb;
Database changed

mysql> select * from emp;
+-----+-----+-----+
| emp_id | emp_name | dept_id |
+-----+-----+-----+
1000	alex	10
2000	ted	10
3000	mat	20
4000	max	20
5000	joe	10
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from dept;
+-----+-----+-----+
| dept_id | dept_name | dept_location |
+-----+-----+-----+
10	ACCOUNTING	NEW YORK, NY
20	RESEARCH	DALLAS, TX
30	SALES	CHICAGO, IL
40	OPERATIONS	BOSTON, MA
50	MARKETING	Sunnyvale, CA
60	SOFTWARE	Stanford, CA
+-----+-----+-----+
6 rows in set (0.00 sec)
```

Next we join (as INNER JOIN) these tow tables on the dept\_id key:

```
mysql> select e.emp_id, e.emp_name, e.dept_id, d.dept_name, d.dept_location
 from emp e, dept d
 where e.dept_id = d.dept_id;
+-----+-----+-----+-----+-----+
| emp_id | emp_name | dept_id | dept_name | dept_location |
+-----+-----+-----+-----+-----+
1000	alex	10	ACCOUNTING	NEW YORK, NY
2000	ted	10	ACCOUNTING	NEW YORK, NY
5000	joe	10	ACCOUNTING	NEW YORK, NY
3000	mat	20	RESEARCH	DALLAS, TX
4000	max	20	RESEARCH	DALLAS, TX
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)
```

Map-side Join is similar to a join but all the task will be performed by the mapper alone (note that the result of an INNER JOIN and map side join must be identical).

Given two tables A and B, the map-side join will be mostly suitable when table A is large and table B is a small to medium. Since table B is not very big, then we create a hash table from B and then broadcast it to all nodes. Next, we iterate all elements of table A by a mapper and then accessing a broadcasted hash table (which denotes table B).

Create two RDDs:

First, we create EMP as RDD[(dept\_id, (emp\_id, emp\_name))]:

```
EMP = spark.sparkContext.parallelize(
[
 (10, (1000, 'alex')),
 (10, (2000, 'ted')),
 (20, (3000, 'mat')),
 (20, (4000, 'max')),
 (10, (5000, 'joe'))
])
```

Next, we create DEPT as RDD[(dept\_id, (dept\_name , dept\_location))]:

```
DEPT= spark.sparkContext.parallelize(
[(10, ('ACCOUNTING', 'NEW YORK, NY')),
 (20, ('RESEARCH', 'DALLAS, TX')),
 (30, ('SALES', 'CHICAGO, IL')),
 (40, ('OPERATIONS', 'BOSTON, MA')),
 (50, ('MARKETING', 'Sunnyvale, CA')),
 (60, ('SOFTWARE', 'Stanford, CA'))
])
```

Now that both EMP and DEPT have a common key on the dept\_id, then we join them:

```
>>> sorted(EMP.join(DEPT).collect())
[
 (10, ((1000, 'alex'), ('ACCOUNTING', 'NEW YORK, NY'))),
 (10, ((2000, 'ted'), ('ACCOUNTING', 'NEW YORK, NY'))),
 (10, ((5000, 'joe'), ('ACCOUNTING', 'NEW YORK, NY'))),
 (20, ((3000, 'mat'), ('RESEARCH', 'DALLAS, TX'))),
```

```
(20, ((4000, 'max'), ('RESEARCH', 'DALLAS, TX')))
]
```

How will the map-side join optimize the task? Here, I assume that EMM is a large data set and DEPT is a relatively small dat set. Using map side join, to join EMP with DEPT on dept\_id, we will create a broadcast varibale from a small table (this is a custom hash builder from an RDD):

```
build a dictionary of (key, value),
where key = dept_id
value = (dept_name , dept_location)

def to_hash_table(dept_as_list):
 hash_table = {}
 for d in dept_as_list:
 dept_id = d[0]
 dept_name_location = d[1]
 hash_table[dept_id] = dept_name_location
 return hash_table
#end-def

dept_hash_table = to_hash_table(DEPT.collect())
```

Alternatively, you may build the same hash table by using an Spark action `collectAsMap()`, which returns the key-value pairs in this RDD (DEPT) to the master as a dictionary. Therefore, we can write:

```
dept_hash_table = DEPT.collectAsMap()
```

Now, using `pyspark.SparkContext.broadcast`, you can broadcast a read-only variable (`dept_hash_table`) to the Spark cluster, which will be available to all kinds of transformations (including mappers and reducers).

```
sc = spark.sparkContext
hash_table_broadcasted = sc.broadcast(dept_hash_table)
```

Now, how do we perfor the map side join? In your mapper, you can access your `dept_hash_table` via:

```
dept_hash_table = hash_table_broadcasted.value
```

Next, I show the map side join by using a `map()` transformation:

```
joined = EMP.map(map_side_join)
```

Finally, `map_side_join()` is defined as:

```
e as an element of EMP RDD
def map_side_join(e):
 dept_id = e[0]
 # get hash_table from broadcasted object
 hash_table = hash_table.broadcasted.value
 dept_name_location = hash_table[dept_id]
 return (e, dept_name_location)
#end-def
```

Therefore, with map side join, you can avoid shuffling which can cost a lot and you avoid significant network I/O.

With a map-side join, we just used a `map()` function for each row of the EMP table, and retrieved dimension values (such as `dept_name` and `dept_location`) from the broadcasted hash table. If EMP table is very large, the `map()` function will be executed concurrently for each partition that has own copy of hash table. The map side join approach allows us not to shuffle the fact table (i.e., DEPT) and to get quite good join performance.

Below, I have listed advantages and disadvantages of using a map-side join.

#### *Advantages of using map-side join*

- The true join cost can be reduced since we are minimizing the cost that is incurred for sorting and merging in the shuffle and reduce stages.
- Map-side join improves the performance of the join task by decreasing the time to finish the join operation.

#### *Disadvantages of map-side join*

- Map-side join data pattern is proper when one of the RDDs/tables on which you perform map-side join operation is small enough to fit into the memory.

- If both RDDs/tables are not small at all, then map-side join is not suitable to perform map-side join on the RDDs/tables.

Next, I discuss another design pattern (Join using Bloom filters), which can be used in efficient joining of two tables.

## Map-Side Join by DataFrames

As I discussed in the preceding section, Map-side join makes sense when one of the tables — so called a fact table — is huge and the other table — so called a dimension table — is small enough to be broadcasted in memory.

In the following example, I will show how to use DataFrames along with Broadcast variables to solve the Map-side join operation

Consider the following tables (one fact and two dimensional tables). The idea for this example is inspired from [Map-Side Join in Spark](#).

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*l*  
-  
*l*  
. *F*  
*l*  
*i*  
*g*  
*h*  
*t*  
*s*

(  
*f*  
*a*  
*c*  
*t*  
*t*  
*a*  
*b*  
*l*  
*e*  
)

| from | to  | airline | flight number | departure |
|------|-----|---------|---------------|-----------|
| DTW  | ORD | SW      | 225           | 17:10     |
|      |     |         |               |           |

|     |     |    |      |      |
|-----|-----|----|------|------|
| DTW | JFK | SW | 355  | 8:20 |
| SEA | JFK | DL | 418  | 7:00 |
| SFO | LAX | AA | 1250 | 7:05 |
| SFO | JFK | VX | 12   | 7:05 |
| JFK | LAX | DL | 424  | 7:10 |
| LAX | SEA | DL | 5737 | 7:10 |

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*l*  
-  
*2*  
. *A*  
*i*  
*r*  
*p*  
*o*  
*r*  
*t*  
*s*

(  
*d*  
*i*  
*m*  
*e*  
*n*  
*s*  
*i*  
*o*  
*n*

*t*  
*a*  
*b*  
*l*

*e*  
)

| <b>code</b> | <b>name</b>             | <b>city</b>   | <b>state</b> |
|-------------|-------------------------|---------------|--------------|
| DTW         | Detroit Airport         | Detroit       | MI           |
| ORD         | Chicago O'Hare          | Chicago       | IL           |
| JFK         | John F. Kennedy Airport | New York      | NY           |
| LAX         | Los Angeles Airport     | Los Angeles   | CA           |
| SEA         | Seattle-Tacoma Airport  | Seattle       | WA           |
| SFO         | San Francisco Airport   | San Francisco | CA           |

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*l*  
-  
*3*  
. *A*  
*i*  
*r*  
*l*  
*i*  
*n*  
*e*  
*s*

(  
*d*  
*i*  
*m*  
*e*  
*n*  
*s*  
*i*  
*o*  
*n*

*t*  
*a*  
*b*  
*l*

*e*  
)

| <b>code</b> | <b>airline name</b> |
|-------------|---------------------|
| SW          | Southwest Airlines  |
| AA          | American Airlines   |
| DL          | Delta Airlines      |
| VX          | Virgin America      |

Suppose the goal to to expand the `Flights` table and replace airline codes with the actual airline names and replace airport codes with the actual airport names. This is join operation of `flights` — facts table — with two dimensional tables (`Airports` and `Airlines`). Since the dimensional tables are small enough, therefore we can broadcast these to all mappers in all worker nodes. Our desired joined output will look like:

T  
a  
b  
l  
e

l  
l  
-  
4  
.J  
o  
i  
n  
e  
d

t  
a  
b  
l  
e

| <b>from city</b> | <b>to city</b> | <b>airline</b>     | <b>flight number</b> | <b>departure</b> |
|------------------|----------------|--------------------|----------------------|------------------|
| Detroit          | Chicago        | Southwest Airlines | 225                  | 17:10            |
| Detroit          | New York       | Southwest Airlines | 355                  | 8:20             |
| Seattle          | New York       | Delta Airlines     | 418                  | 7:00             |
| San Francisco    | Los Angeles    | American Airlines  | 1250                 | 7:05             |
| San Francisco    | New York       | Virgin America     | 12                   | 7:05             |
| New York         | Los Angeles    | Delta Airlines     | 424                  | 7:10             |

To achieve the **Joined** table result, we need to do the following:

*Step-1: Create Cache for Airports*

Create a broadcast variable for airports: first we create an RDD and then save it a `dictionary[(key, value)]` where `key` is an airport `code`, and `value` is associated values.

*Step-2: Create Cache for Airlines*

Create a broadcast variable for airlines: first we create an RDD and then save it a `dictionary[(key, value)]` where `key` is an airline `code`, and `value` is the name of airline.

*Step-3: Create Facts Table — Flights*

Map each record of the `Flights` table and perform a simple join by lookup dictionaries created in Steps 1 & 2.

*Step-4: Map-Side Join:*

Map each record of the `Flights` table — known as a facts table — and perform a simple join by lookup cache dictionaries created in Steps 1 & 2.

## **Step-1: Create Cache for Airports**

This step creates a cache — as a Broadcast data to be available in all worker nodes — for Airports table:

```
>>> airports_data = [
... ("DTW", "Detroit Airport", "Detroit", "MI"),
... ("ORD", "Chicago O'Hare", "Chicago", "IL"),
... ("JFK", "John F. Kennedy Int. Airport", "New York", "NY"),
... ("LAX", "Los Angeles Int. Airport", "Los Angeles", "CA"),
... ("SEA", "Seattle-Tacoma Int. Airport", "Seattle", "WA"),
... ("SFO", "San Francisco Int. Airport", "San Francisco", "CA")
...]
>>>
>>> airports_rdd = spark.sparkContext.parallelize(airports_data)\
```

```

... .map(lambda tuple4: (tuple4[0], (tuple4[1],tuple4[2],tuple4[3])))

>>> airports_dict = airports_rdd.collectAsMap()
>>>
>>> airports_cache = spark.sparkContext.broadcast(airports_dict)
>>> airports_cache.value
{'DTW': ('Detroit Airport', 'Detroit', 'MI'),
 'ORD': ("Chicago O'Hare", 'Chicago', 'IL'),
 'JFK': ('John F. Kennedy Int. Airport', 'New York', 'NY'),
 'LAX': ('Los Angeles Int. Airport', 'Los Angeles', 'CA'),
 'SEA': ('Seattle-Tacoma Int. Airport', 'Seattle', 'WA'),
 'SFO': ('San Francisco Int. Airport', 'San Francisco', 'CA')}

```

## Step-2: Create Cache for Airlines

This step creates a cache — as a Broadcast data to be available in all worker nodes — for Airlines table:

```

>>> airlines_data = [
... ("SW", "Southwest Airlines"),
... ("AA", "American Airlines"),
... ("DL", "Delta Airlines"),
... ("VX", "Virgin America")
...]

>>> airlines_rdd = spark.sparkContext.parallelize(airlines_data) \
... .map(lambda tuple2: (tuple2[0], tuple2[1]))

>>> airlines_dict = airlines_rdd.collectAsMap()
>>> airlines_cache = spark.sparkContext.broadcast(airlines_dict)
>>> airlines_cache
>>> airlines_cache.value
{'SW': 'Southwest Airlines',
 'AA': 'American Airlines',
 'DL': 'Delta Airlines',
 'VX': 'Virgin America'}

```

## Step-3: Create Facts Table

This step creates a DataFrame — as a fact table — to be joined with caches created in Steps 1 & 2.

```

>>> flights_data = [
... ("DTW", "ORD", "SW", "225", "17:10"),
... ("DTW", "JFK", "SW", "355", "8:20"),
... ("SEA", "JFK", "DL", "418", "7:00"),
...

```

```

... ("SFO", "LAX", "AA", "1250", "7:05"),
... ("SFO", "JFK", "VX", "12", "7:05"),
... ("JFK", "LAX", "DL", "424", "7:10"),
... ("LAX", "SEA", "DL", "5737", "7:10")
...]
>>> flight_columns = ["from", "to", "airline", "flight_number", "departure"]
>>> flights = spark.createDataFrame(flights_data, flight_columns)
>>> flights.show(truncate=False)
+-----+-----+-----+-----+
|from|to |airline|flight_number|departure|
+-----+-----+-----+-----+
|DTW |ORD|SW |225 |17:10 |
|DTW |JFK|SW |355 |8:20 |
|SEA |JFK|DL |418 |7:00 |
|SFO |LAX|AA |1250 |7:05 |
|SFO |JFK|VX |12 |7:05 |
|JFK |LAX|DL |424 |7:10 |
|LAX |SEA|DL |5737 |7:10 |
+-----+-----+-----+-----+

```

## Step-4: Apply Map-Side Join

This step iterates the fact table — flights data — and performs the map-side join:

```

>>> from pyspark.sql.functions import udf
>>> from pyspark.sql.types import StringType
>>>
>>> def get_airport(code):
... return airports_cache.value[code][1]
...
>>> def get_airline(code):
... return airlines_cache.value[code]

>>> airport_udf = udf(get_airport, StringType())
...
>>> airport_udf = udf(get_airport, StringType())
>>>
>>> flights.select(
... airport_udf("from").alias("from_city"), ❶
... airport_udf("to").alias("to_city"), ❷
... airline_udf("airline").alias("airline_name"), ❸
... "flight_number", "departure").show(truncate=False)
+-----+-----+-----+-----+
|from_city|to_city|airline_name|flight_number|departure|
+-----+-----+-----+-----+
|Detroit |Chicago|Southwest Airlines|225 |17:10 |
|Detroit |New York|Southwest Airlines|355 |8:20 |
|Seattle |New York|Delta Airlines |418 |7:00 |
+-----+-----+-----+-----+

```

|               |             |                   |      |      |  |
|---------------|-------------|-------------------|------|------|--|
| San Francisco | Los Angeles | American Airlines | 1250 | 7:05 |  |
| San Francisco | New York    | Virgin America    | 12   | 7:05 |  |
| New York      | Los Angeles | Delta Airlines    | 424  | 7:10 |  |
| Los Angeles   | Seattle     | Delta Airlines    | 5737 | 7:10 |  |

- ❶ Map-Side join for Airport
- ❷ Map-Side join for Airport
- ❸ Map-Side join for Airline

## Efficient Joins using Bloom filters

Given two RDDs as  $A = \text{RDD}[(K, V)]$  (as a bigger RDD) and  $B = \text{RDD}[(K, W)]$  (as a smaller RDD), Spark enable us to perform join operation on key K. Joining two RDDs is a common operation when working with Spark. In some of the cases, a join is used as a form of filtering. For example, you want to perform an operation on a subset of the records in the  $\text{RDD}[(K, V)]$ , represented by entities in another  $\text{RDD}[(K, W)]$ . For this you can use an inner join to achieve that effect. But, you want to avoid the shuffle that the join operation introduces, especially if the  $\text{RDD}[(K, W)]$  you want to use for filtering is significantly smaller than the main  $\text{RDD}[(K, V)]$  on which you will perform your further computation.

You may do a broadcast join using a set (as a Bloom filter) constructed by collecting the smaller RDD you wish to filter by. But, this means collecting the whole smaller  $\text{RDD}[(K, W)]$  in driver memory, and even if it is relatively small (several thousand or million records), that can still lead to some undesirable memory pressure. If you want to avoid the shuffle that the join operation introduces, then you may use the **Bloom filter**. So the problem of joining  $\text{RDD}[(K, V)]$  with  $\text{RDD}[(K, W)]$  is reduced into a simple `map()` transformation, which we will check the key K against the Bloom filter constructed from the smaller  $\text{RDD}[(K, W)]$ .

## Bloom filter

A Bloom filter is a space-efficient probabilistic data structure, conceived by B. H. Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not; i.e. a query returns either **possibly in set** or **definitely not in set**. Elements can be added to the set, but not removed (though this can be addressed with a “counting” filter). The more elements that are added to the set, the larger the probability of false positives. The Bloom filter data structure may return true for elements that are not actually members of the set (this is called false-positives errors), but it will never return false for elements that are in the set; for each element in the set, the Bloom filter must return true.

Therefore, in a nutshell, we can summarize Bloom filter properties as:

- Given a big set  $S = \{x_1, x_2, \dots, x_n\}$ , Bloom filter is a probabilistic, fast, and space efficient cache builder for a big data set; this is basically approximating set membership operation:

is  $x$  in  $S$

- It tries to answer lookup questions:

does item  $x$  exist in a set  $S$ ?

- It allows false positive errors as they only cost us an extra data set access. This means that for some  $x$ , which is not in the set, Bloom filter might indicate that  $x$  is in the set.
- It does not allow false negative errors, because they result in wrong answers. This means that if  $x$  is not in the set, then for sure it will indicate that  $x$  is not in the set. Thus, the Bloom filter does not allow false negatives, but can allow false positives.

Let's focus on a simple join example between two relations or tables: let's say that we want to join  $R = \text{RDD}(K, V)$  and  $S = \text{RDD}(K, W)$  on a common key  $K$ . Further assume that

$\text{count}(R) = 1,000,000,000$  (larger data set)  
 $\text{count}(S) = 10,000,000$  (smaller data set)

To do basic join, we need to check 10 trillion ( $10^{12}$ ) records, which is a huge and time consuming process. One idea to reduce time and complexity of join operation between R and S is to use a Bloom filter on relation S (smaller data set) and then use the built Bloom filter data structure on relation R. This can eliminate the non-needed records from R (it might reduce it to 20,000,000) and hence the join becomes fast and efficient.

Next we semi-formalize Bloom filter data structure: what is involved in Bloom filter probabilistic data structures? How do we construct one? what is the probability of **false positive errors**, and how we can decrease the probability of **false positive errors**. This is how it works:

- Given a set  $S = \{x_1, x_2, \dots, x_n\}$
- Let  $B$  be an  $m$  ( $m > 1$ ) bit array, initialized with 0s.  $B$ 's elements are  $B[0], B[1], B[2], \dots, B[m-1]$ . The memory required for array  $B$  is only a fraction of the one needed for storing the whole set  $S$ . By selecting the bigger bit vector (array  $B$ ), the probability of false positive rate will decrease.
- Let  $\{H_1, H_2, \dots, H_k\}$  be a set of  $k$  hash functions, If  $H_i(x_j) = a$  then set  $B[a] = 1$ . You may use SHA1, MD5, and Murmur as hash functions. For example, you may use the following as hash functions:
  - $H_i(x) = MD5(x+i)$
  - $H_i(x) = MD5(x || i)$
- To check if  $x$  in  $S$ , check  $B$  at  $H_i(x)$ . All  $k$  values must be 1.
- It is possible to have a false positive; all  $k$  values are 1, but  $x$  is not in  $S$ .
- The probability of false positives is (for details, see Wikipedia).

$$\Pr(1 - \Pr[1 - \frac{1}{m}]^k)^k \approx \Pr(1 - e^{-\frac{k}{m}})^k$$

- What are the optimal hash functions? What is the optimal number of hash functions? For a given  $m$  (number of bits selected for Bloom filter) and  $n$  (size of big data set), the value of  $k$  (the number of hash functions) that minimizes the probability of false positives is ( $\ln$  stands for “natural logarithm”)

$$k = \lceil m / n \rceil \ln(2)$$

$$m = -\frac{n}{\ln(p)} \approx \frac{n}{(\ln(2))^2}$$

Therefore, the probability that a specific bit has been flipped to 1 is:

$$1 - \left(1 - \frac{1}{m}\right)^k \approx 1 - e^{-kn/m}$$

Next, let's take a look at a Bloom filter example.

## A Simple Bloom Filter Example

This example shows how to insert and query on a bloom filter of size 10 ( $m = 10$ ) with three hash functions  $H = \{H_1, H_2, H_3\}$  and let  $H(x)$  denote the result of these three hash functions. We start with a 10-bit long array  $B$  initialized to 0:

```

Array B:
initialized:
 index 0 1 2 3 4 5 6 7 8 9
 value 0 0 0 0 0 0 0 0 0 0

insert element a, H(a) = (2, 5, 6)
 index 0 1 2 3 4 5 6 7 8 9
 value 0 0 1 0 0 1 1 0 0 0

insert element b, H(b) = (1, 5, 8)
 index 0 1 2 3 4 5 6 7 8 9
 value 0 1 1 0 0 1 1 0 1 0

query element c
H(c) = (5, 8, 9) => c is not a member (since B[9]=0)

query element d
H(d) = (2, 5, 8) => d is a member (False Positive)

query element e
H(e) = (1, 2, 6) => e is a member (False Positive)

```

```
query element f
H(f) = (2, 5, 6) => f is a member (Positive)
```

## Bloom Filter in Python

The following code segment shows how to create and use a Bloom filter in Python (you may roll your own Bloom filter library, but if you find a proper library, then use it).

```
instantiate BloomFilter with custom settings,
>>> from bloom_filter import BloomFilter
>>> bloom = BloomFilter(max_elements=100000, error_rate=0.01)

Test whether the bloom-filter has seen a key:
>>> "test-key" in bloom
False

Mark the key as seen
>>> bloom.add("test-key")

Now check again
>>> "test-key" in bloom
True
```

## Using Bloom Filter in PySpark

Bloom filter is a small, compact, and fast data structure for set membership. It can be used in the join of two RDDs/relations/tables such as R(K, V) and S(K, W) where one of the relations has huge number of records (for example, R to have 1000,000,000 records) and the other relation has small number of records (for example, S to have 10,000,000 records). To do a join on key field “K” between R and S, it will take a long time and it is inefficient.

We can use Bloom filter data structure in the following way: build a Bloom filter out of relation S(K, W), and then test values R(K, V) for membership using the built Bloom filter (using Sparks broadcast mechanism). Note that, for reduce-side join optimization, we use a Bloom filter on the map tasks, which will force an I/O cost reduction for the PySpark job. How do we use the Bloom filter concept in mappers? The following steps show how to use a Bloom filter

(representation of S) in mappers (R), which will be substitute for the join operation between R and S.

#### *STEP-1: Build Bloom filter*

Construction of the bloom filter; this a small PySpark code segment, which uses the smaller of the two relations/tables for constructing the bloom filter data structure. First, initialize the Bloom filter (create an instance of `BloomFilter`), then you may use `BloomFilter.add()` to build the Bloom filter. Let's call the built Bloom filter as `the_bloom_filter`

#### *STEP-2: Broadcast the Built Bloom filter*

Use `SparkContext.broadcast()` to broadcast the built Bloom filter (`the_bloom_filter`) to all worker nodes (so that it can be available to all Spark transformations including mappers)

```
to broadcast it to all worker nodes for read only purposes
sc = spark.sparkContext
broadcasted_bloom_filter = sc.broadcast(the_bloom_filter)
```

#### *STEP-3: Use the broadcasted object in Mappers*

Now, we can user the Bloom filter to get rid of the non-needed elements.

```
e is an element of R(k, b)
def bloom_filter_function(e):
 # get a copy of Bloom filter
 the_bloom_filter = broadcasted_bloom_filter.value()
 # use the_bloom_filter for element e
 key = e[0]
 if key in the_bloom_filter:
 return True
 else:
 return False
#end-def
```

We use the `filter_function()` for `R=RDD[(K, V)]` to keep the elements if and only if the key is in `S=RDD[(K, W)]`.

```
R=RDD[(K, V)]
joined = RDD[(K, V)] where K is in S=RDD[(K, W)]
```

```
joined = R.filter(bloom_filter_function)
```

## Binning: Data Organization Pattern

What is binning? Binning is a way to group a number of more or less continuous values into a smaller number of “bins” — also called “buckets”. For example, if you have a cesus data about a group of people, you might want to map their ages into a smaller number of age intervals. One way is to define age groups as 0-5, 6-10, 11-15, ..., 96-100+. One important aspect of binning is if some is 14 years old, then

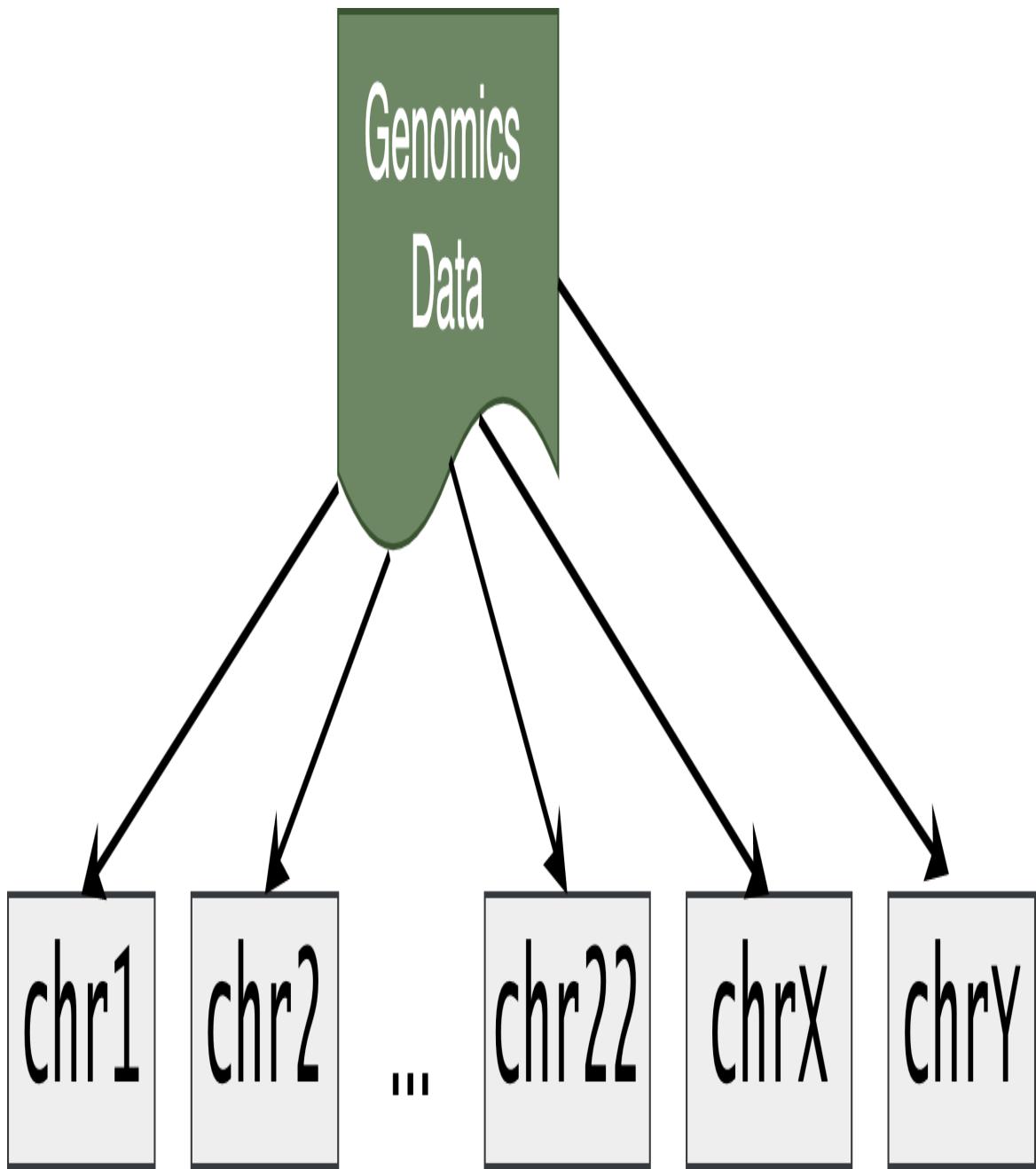
*Example 11-1.*

---

==== Binning: Data Organization Pattern What is binning? Binning is a way to group a number of more or less continuous values into a smaller number of “bins” — also called “buckets”. For example, if you have a cesus data about a group of people, you might want to map their ages into a smaller number of age intervals. One way is to define age groups as 0-5, 6-10, 11-15, ..., 96-100+. One important aspect of binning is if some is 14 years old, then you will only search a bin labeled as 11-15, because 14 falls into a bin labeled 11-15.

Therefore binning can help us to do faster queries by examining a slide of a data rather than the whole data. The binning design pattern moves the records into categories (called bins) irrespective of the order of records.

In genomics data, chromosomes are labeled as `chr1`, `chr2`, ..., `chr22`, `chrX`, `chrY`, and `chrMT`. For a human being sample, we have 3 billion pairs of chromosomes, where `chr1` has about 250 million positions, `chr7` has 160 million positions, and so on. If you want to search for a “variant\_key” of `10:100221486:100221486:G`, then you have to search billions of records to find that variant. This type of search is very inefficient, since for every search you have to scan all records to find your desired record. The most simple bin is to group data by chromosomes. Therefore, to search for `10:100221486:100221486:G`, we will look into a bin labeled `chr10` rather than searching all of the data. This binning can be illustrated by the Figure 19.7.



*Figure 11-9. Binning by Chromosome*

---

To implement this binning algorithm, first we read input and create a DataFrame with proper columns, and then create an additional column—called `chr_id`—, which will denote a bin for a chromosome. Therefore, `chr_id` column will have values in `{chr1, chr2, ..., chr22, chrX, chrY, chrMT}`. Therefore, to search for variant of `10:100221486:100221486:G`, we may be just look for a bin called `chr10`.

It is possible to perform the binning algorithm in several layers (first by chromosome and then by modulo of the start position) as illustrated by the Figure 19.8

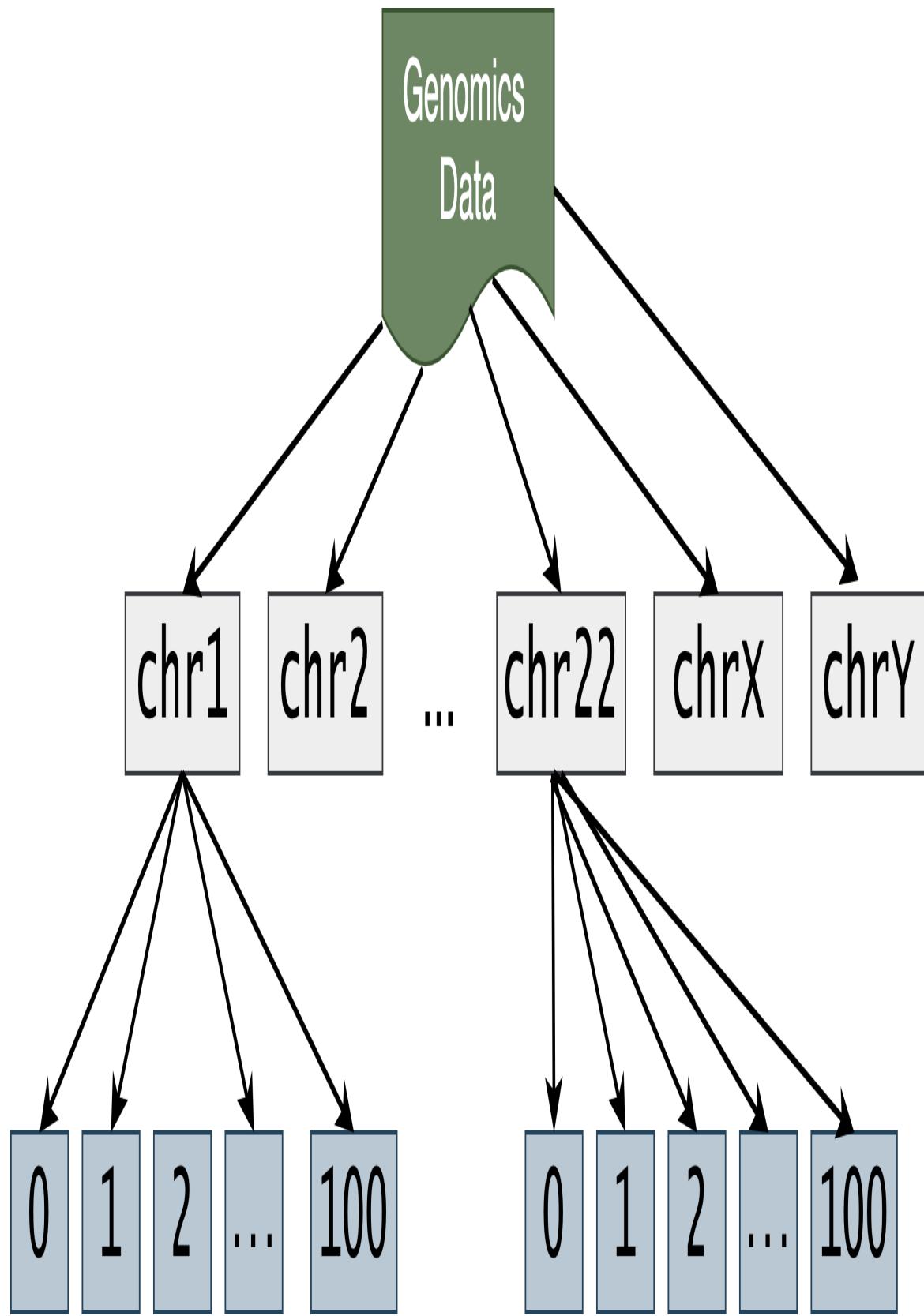


Figure 11-10. Binning by Chromosome

---

After binning variants by chromosomes, we can easily bin the variants by the start position of variant as well. This will be quite helpful because we might have millions of variants per chromosomes; and this can help us to reduce the query time by examining a very thin slice of data rather than the whole data. Before defining binning for variants by start positions, let's understand the variant structure:

```
<chromosome><:><start_position><:><stop_position><:><allele>
```

A simple binning algorithm will be to partition the `start_position` into 101 (as a prime number, depending on the volume of data you may select a proper prime number) bins. Therefore given a variant, we will create another new column called a `bin` and its value will be defined as:

```
modulo = start_position % 101
```

For example, for a variant of `10:100221486:100221486:G`, the `bin` value will be `95` (`100221486 % 101 = 95`). Therefore our bin values will be in  $\{0, 1, 2, \dots, 100\}$ .

By an example, I will show how the binnings can be done for a genomics data.

First, we create a DataFrame from genomics data (I have only included some relevant columns and many columns are omitted for clarity of presentation).

```
variants = [('S-100', 'Prostate-1', '5:163697197:163697197:T', 2),
 ('S-200', 'Prostate-1', '5:3420488:3420488:C', 1),
 ('S-100', 'Genome-1000', '3:107988242:107988242:T', 1),
 ('S-200', 'Genome-1000', '3:54969706:54969706:T', 3)]
```

```
columns = ['SAMPLE_ID', 'STUDY_ID', 'VARIANT_KEY', 'ZYGOSITY']
```

```
df = spark.createDataFrame(variants, columns)
df.show(truncate=False)
```

| SAMPLE_ID | STUDY_ID    | VARIANT_KEY             | ZYGOSITY |
|-----------|-------------|-------------------------|----------|
| S-100     | Prostate-1  | 5:163697197:163697197:T | 2        |
| S-200     | Prostate-1  | 5:3420488:3420488:C     | 1        |
| S-100     | Genome-1000 | 3:107988242:107988242:T | 1        |
| S-200     | Genome-1000 | 3:54969706:54969706:T   | 3        |

Next, we create a binning function for a `chr_id` — to be extracted from a given `variant_key`.

```
def extract_chr(variant_key):
 tokens = variant_key.split(":")
```

```

 return "chr" + tokens[0]
#end-def

```

To use the `extract_chr()` function, first, we have to create a UDF:

```

from pyspark.sql.functions import udf
from pyspark.sql.types import StringType
extract_chr_udf = udf(extract_chr, StringType())

binned_by_chr = df.select("SAMPLE_ID", "STUDY_ID", "VARIANT_KEY", "ZYGOSITY",
extract_chr_udf("VARIANT_KEY").alias("CHR_ID"))

```

```

binned_by_chr.show(truncate=False)
+-----+-----+-----+-----+
|SAMPLE_ID|STUDY_ID |VARIANT_KEY |ZYGOSITY|CHR_ID|
+-----+-----+-----+-----+
S-100	Prostate-1	5:163697197:163697197:T	2	chr5
S-200	Prostate-1	5:3420488:3420488:C	1	chr5
S-100	Genome-1000	3:107988242:107988242:T	1	chr3
S-200	Genome-1000	3:54969706:54969706:T	3	chr3
+-----+-----+-----+-----+

```

To create a second level of binning, we need another Python function to find `start_position % 101`.

```

101 is the number of bins per chromosome
def create_modulo(variant_key):
 tokens = variant_key.split(":")
 start_position = int(tokens[1])
 return start_position % 101
#end-def

```

Next we create a new modulo bin by using the start position of a given variant:

```

from pyspark.sql.functions import udf
from pyspark.sql.types import IntegerType
create_modulo_udf = udf(create_modulo, IntegerType())

binned_by_chr_and_position = df.select("SAMPLE_ID", "STUDY_ID", "VARIANT_KEY",
"ZYGOSITY", extract_chr_udf("VARIANT_KEY").alias("CHR_ID"),
create_modulo_udf("VARIANT_KEY").alias("modulo"))

binned_by_chr_and_position.show(truncate=False)
+-----+-----+-----+-----+-----+
|SAMPLE_ID|STUDY_ID |VARIANT_KEY |ZYGOSITY|CHR_ID|modulo|
+-----+-----+-----+-----+-----+
S-100	Prostate-1	5:163697197:163697197:T	2	chr5	33
S-200	Prostate-1	5:3420488:3420488:C	1	chr5	22
S-100	Genome-1000	3:107988242:107988242:T	1	chr3	52
S-200	Genome-1000	3:54969706:54969706:T	3	chr3	52
+-----+-----+-----+-----+-----+

```

You may save your DataFrame as a Parquet format without binning:

```
binned_by_chr_and_position.write.mode("append")\
 .parquet('/tmp/genome1/')

$ ls -l /tmp/genome1/
-rw-r--r-- ... 0 Jan 18 14:34 _SUCCESS
-rw-r--r-- ... 575 Jan 18 14:34 part-00000-....snappy.parquet
-rw-r--r-- ... 1391 Jan 18 14:34 part-00001-....snappy.parquet
-rw-r--r-- ... 1355 Jan 18 14:34 part-00003-....snappy.parquet
-rw-r--r-- ... 1400 Jan 18 14:34 part-00005-....snappy.parquet
-rw-r--r-- ... 1382 Jan 18 14:34 part-00007-....snappy.parquet
```

To save data with binning information, you may use `partitionBy()` function:

```
binned_by_chr_and_position.write.mode("append")\
 .partitionBy("CHR_ID", "modulo")\
 .parquet('/tmp/genome2/')
```

```
$ ls -R /tmp/genome2/
CHR_ID=chr3 CHR_ID=chr5 _SUCCESS
```

```
/tmp/genome2//CHR_ID=chr3:
modulo=52
```

```
/tmp/genome2//CHR_ID=chr3/modulo=52:
part-00005-....snappy.parquet
part-00007-....snappy.parquet
```

```
/tmp/genome2//CHR_ID=chr5:
modulo=22 modulo=33
```

```
/tmp/genome2//CHR_ID=chr5/modulo=22:
part-00003-....snappy.parquet
```

```
/tmp/genome2//CHR_ID=chr5/modulo=33:
part-00001-....snappy.parquet
```

### ==== Sorting: Data Organization Pattern

Sorting of data records — order of data from record to record — is a common task in many programming languages such as Python and Java. Sorting on data records can be done Ascending (from smallest to largest) or Descending (from largest to smallest). Sorting is any process of arranging records systematically, and has two common, yet distinct meanings: ordering: arranging records in a sequence ordered by some criterion; categorizing: grouping items with similar properties. There are many well-known sorting algorithms — such as Quick Sort, Bubble Sort, Heap Sort, ... — with different performance complexities.

In 2014, Databricks **participated in the Sort benchmark** and set a new world record for sorting for 100 TB of data (1 trillion 100-byte records) using Spark with using 206 nodes cluster.

Spark offers several — listed below — functions for sorting RDDs and DataFrames in PySpark. Use of these sorting functions are straightforward.

```
pyspark.RDD.repartitionAndSortWithinPartitions (Python method)
pyspark.RDD.sortBy (Python method)
pyspark.RDD.sortByKey (Python method)
pyspark.sql.DataFrame.sort (Python method)
pyspark.sql.DataFrame.sortWithinPartitions (Python method)
pyspark.sql.DataFrameWriter.sortBy (Python method)
pyspark.sql.functions.array_sort (Python function)
pyspark.sql.functions.sort_array (Python function)
```

### == Summary

MapReduce design patterns are common patterns in data-related problem solving. These design patterns enable us to solve similar data problems in an efficient manner

Some of MapReduce design patterns are:

- Summarization patterns: get a top-level view by summarizing and grouping data
- Filtering patterns: view data subsets by using predicates
- Data organization patterns: reorganize data to work with other systems, or to make MapReduce analysis easier
- Join patterns: analyze different datasets together to discover interesting relationships
- Meta patterns: piece together several patterns to solve multi-stage problems, or to perform several analytics in the same job
- Input and output patterns: customize the way you use persistent store (such as HDFS and S3) to load or store data

Some of Summarization Patterns are listed:

- Word Count

- MinMax Count
- Summarization
- Map-side Join
- Join with Bloom filter

# Chapter 12. Join Design Patterns

---

## A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author’s raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 12th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at [mpotter@oreilly.com](mailto:mpotter@oreilly.com).

The goal of this chapter is to introduce some informal but practical join data design patterns. I will only discuss the join design patterns that are practical and useful in real world environments and will not look at the theoretical join design patterns (which are just theories and not used in big data solutions), which are not deployed or used in production environments.

This chapter

- Introduces the basic concepts of join of two data sets
- Provides some useful and practical join design patterns by examples
- Shows how to use Spark’s transformations for implementing join data design patterns
- Below are the partial list of join design patterns and I will cover them in this chapter:
  - InMapper Combining
  - Join patterns (inner-join, left-join, ...)

## SOURCE CODE

### NOTE

Complete PySpark programs for this chapter are presented in [GitHub Chapter 12](#).

## Introduction

In relational database world, the join operation is a common function between two tables where they have a common key.

Consider the following two tables T1 and T2:

```
T1 = {(k1, v1)}
T2 = {(k2, v2)}
```

where

k1 is the key for T1 and v1 are the associated attributes  
k2 is the key for T2 and v2 are the associated attributes

Then a simple inner join (an inner join in relational databases creates a new table by combining rows that have matching values in two or more tables) can be defined as:

```
T1.join(T2) =
T2.join(T1) =
{(k, v1, v2)}
```

where

k = k1 = k2,  
(k, v1) in T1,  
(k, v2) in T2

To understand the join operation, let's create two tables T1 and T2 and then join them.

First, let's create tables:

```

>>> d1 = [('a', 10), ('a', 11), ('a', 12), ('b', 100), ('b', 200), ('c', 80)]
>>> T1 = spark.createDataFrame(d1, ['id', 'v1'])
>>> T1.show()
+---+---+
| id| v1|
+---+---+
| a | 10|
| a | 11|
| a | 12|
| b |100|
| b |200|
| c | 80|
+---+---+

>>> d2 = [('a', 40), ('a', 50), ('b', 300), ('b', 400), ('d', 90)]
>>> T2 = spark.createDataFrame(d2, ['id', 'v2'])
>>> T2.show()
+---+---+
| id| v2|
+---+---+
| a | 40|
| a | 50|
| b |300|
| b |400|
| d | 90|
+---+---+

```

Next, let's join tables T1 and T2 as an inner join (the default join in Spark is an inner join). As we notice from the output of join: the `id = c` (from T1) and `id = d` (from T2) are dropped since they are not common between tables T1 and T2.

```

>>> joined = T1.join(T2, (T1.id == T2.id))
>>> joined.show(100, truncate=False)
+---+---+---+---+
| id |v1 |id |v2 |
+---+---+---+---+
| a | 10 |a | 50 |
| a | 10 |a | 40 |
| a | 11 |a | 50 |
| a | 11 |a | 40 |
| a | 12 |a | 50 |
| a | 12 |a | 40 |
| b |100|b |400|
| b |100|b |300|
| b |200|b |400|
| b |200|b |300|
+---+---+---+---+

```

In theory there are many join functions defined between two tables with a common key found in both tables. But, in practice, three types of join are common:

- inner join
- left join
- right join

All these joins are supported in PySpark. To understand the concept of join, you may refer to [Joins in SQL](#).

In general, the join operation of two tables is an expensive operation since to find a join, a cartesian product of two tables are required. For example, if table T1 has 3 billion rows and table T2 has one million rows, then the cartesian product of these two tables will have 3 Quadrillion (1 followed with 15 zeros) data points. In this chapter, I provide some basic join design patterns which can help to simplify the join and reduce the cost of join operation. There is no silver bullet on adopting and using join design patterns, every adopted join pattern should be tested with for performance and scalability by using real data sets.

## Join in MapReduce

In this section, I am assuming that we are using PySpark (a MapReduce implementation instance) without using the `join` function. The purpose of this section is to understand how the `join` implementation works in big data.

Assume to have two relations: R(K, B) and S(K, C) where K is a common key between relations R and S. Also assume that B and C represents attributes of R and S respectively. How do we find the join of R and S? The goal of join operation is find tuples that agree on their key K. A MapReduce implementation of Natural Join for R and S can implemented as:

*Map:*

- For a tuple (k, b) in R emit a (key, value) pair as (k, ("R", b))
- For a tuple (k, c) in S, emit a (key, value) pair as (k, ("S", c))

Once mappers are done, then we can execute the reducer as:

*Reduce:*

If a reducer key  $k$  has value list  $[("R", v), ("S", w)]$ , then emit a single (key, value) pair as  $(k, (v, w))$ . Note that  $\text{join}(R, S)$  will produce  $(k, (v, w))$ , while  $\text{join}(S, R)$  will produce  $(k, (w, v))$ .

If a reducer key  $k$  has value list  $[("R", v1), ("R", v2), ("S", w1), ("S", w2)]$ , then we will emit four (key, value) pairs as

```
(k, (v1, w1))
(k, (v1, w2))
(k, (v2, w1))
(k, (v2, w2))
```

Therefore, to perform a natural join between two relations  $R$  and  $S$ , we need to perform 2 map functions and one reducer.

## Map Phase

The map phase has two steps:

Step-1: map relation  $R$

```
key: relation R
value: (k, b) tuple in R
map(key, value) {
 emit(k, ("R", b))
}
```

Step-2: map relation  $S$

```
key: relation S
value: (k, c) tuple in S
map(key, value) {
 emit(k, ("S", c))
}
```

The output of mappers (as input to “sort and shuffle” phase) will be:

```

(k1, "R", r1)
(k1, "R", r2)
...
(k1, "S", s1)
(k1, "S", s2)
...
(k2, "R", r3)
(k2, "R", r4)
...
(k2, "S", s3)
(k2, "S", s4)
...

```

## Reducer Phase

Before, we write a reducer function, we need to understand the magic of MapReduce, which is called a “sort and shuffle” phase (similar to SQL’s GROUP BY function). Once all mappers are done, the “sort and shuffle” phase will be applied to the output of mappers and it will produce input for reducers.

The output of “sort and shuffle” phase will be:

```

(k1, [("R", r1), ("R", r2), ..., ("S", s1), ("S", s2), ...]
(k2, [("R", r3), ("R", r4), ..., ("S", s3), ("S", s4), ...]
...

```

The reducer function is presented below: per key k, we build 2 lists: `list_R` (which will hold R values/attributes) and `list_S` (which will hold S values/attributes), then we perform a cartesian product of `list_R` and `list_S` to find the join tuples.

```

key: a unique key
values: [(relation, attrs)] where relation in {"R", "S"}
and attrs are the relation attributes
map(key, values) {
 list_R = []
 list_S = []
 for (tuple in values) {
 relation = tuple[0]
 attributes = tuple[1]
 if (relation == "R") {
 list_R.append(attributes)
 }
 else {

```

```

 list_S.append(attributes)
 }
}

if (len(list_R) == 0) OR (len(list_S) == 0) {
 # no common key
 return
}

len(list_R) > 0 AND len(list_S) > 0
perform cartesian product of list_R and list_S
for (r in list_R) {
 for (s in list_S) {
 emit(key, (r, s))
 }
}
}

```

## Implementation in PySpark

This section shows how to implement the natural join of two data sets (with some common keys) in PySpark without using the `join` function (defined as `pyspark.RDD.join` and `pyspark.sql.DataFrame.join`) provided in PySpark. I present this solution to show that Spark is so powerful and it can implement custom joins if desired.

To solve this problem, consider the following data sets T1 and T2:

```

d1 = [('a', 10), ('a', 11), ('a', 12), ('b', 100), ('b', 200), ('c', 80)]
d2 = [('a', 40), ('a', 50), ('b', 300), ('b', 400), ('d', 90)]
T1 = spark.sparkContext.parallelize(d1)
T2 = spark.sparkContext.parallelize(d2)

```

Next, we map these RDDs to include the name of the relation:

```

t1_mapped = T1.map(lambda x: (x[0], ("T1", x[1])))
t2_mapped = T2.map(lambda x: (x[0], ("T2", x[1])))

```

In order to perform a reduction of the same key, we combine these two data sets into a single data set:

```
combined = t1_mapped.union(t2_mapped)
```

Then we perform the `groupByKey()` transformation on a single combined data set:

```
grouped = combined.groupByKey()
```

finally we perform a cartesian product for the values of each grouped entry:

```
entry[0]: key
entry[1]: values as:
[("T1", t11), ("T1", t12), ..., ("T2", t21), ("T2", t22), ...]
def cartesian_product(entry):
 T1 = []
 T2 = []
 key = entry[0]
 values = entry[1]
 for tuple in values:
 relation = tuple[0]
 attributes = tuple[1]
 if (relation == "T1"): T1.append(attributes)
 else: T2.append(attributes)
 #end-for

 if (len(T1) == 0) or (len(T2) == 0):
 # no common key
 return []

 # len(T1) > 0 AND len(T2) > 0
 joined_elements = []
 # perform cartesian product of T1 and T2
 for v in T1:
 for w in T2:
 joined_elements.append((key, (v, w)))

 #end-for
 #end-for
 return joined_elements
#end-def

joined = grouped.flatMap(cartesian_product)
```

## Map-Side Join by RDDs

A join operation is used to combine rows from two (or more) tables, based on a common column (or columns) between them. Join is an expensive operation and

since big data engines (such as Hadoop and Spark) do not support indexing of data tables, therefore, join can be an expensive. Here, I introduce a map-side join, which can reduce the cost of join between two tables. Map-side join is a process where two data sets are joined by the mapper rather than using the actual join function (which can happen by combination of a mapper and reducer). What are the advantages of using map-side join? The advantages of using map side join are as follows:

- Map-side join might help in minimizing the cost that is incurred for sorting and merging in the shuffle and reduce stages.
- Map-side join might help in improving the performance of the task by decreasing the time to finish the task.

Using SQL terminology, a JOIN clause is used to combine rows from two (or more) tables, based on a related column between them. Typically, a SQL JOIN is an expensive operation (unless you have built proper indexes). In general, joins on large sets are expensive, but very rarely do you want to join the entire contents of large table A with the entire contents of large table B.

To understand map side join, assume that we have two tables (using MySQL database): EMP and DEPT as defined below. Let's see how a join works between these two tables. Consider the following two tables EMP and DEP:

```
mysql> use testdb;
Database changed

mysql> select * from emp;
+-----+-----+-----+
| emp_id | emp_name | dept_id |
+-----+-----+-----+
1000	alex	10
2000	ted	10
3000	mat	20
4000	max	20
5000	joe	10
+-----+-----+-----+
5 rows in set (0.00 sec)

mysql> select * from dept;
+-----+-----+-----+
| dept_id | dept_name | dept_location |
+-----+-----+-----+
```

```

10	ACCOUNTING	NEW YORK, NY
20	RESEARCH	DALLAS, TX
30	SALES	CHICAGO, IL
40	OPERATIONS	BOSTON, MA
50	MARKETING	Sunnyvale, CA
60	SOFTWARE	Stanford, CA
+-----+-----+-----+
6 rows in set (0.00 sec)

```

Next we join (as INNER JOIN) these tow tables on the dept\_id key:

```

mysql> select e.emp_id, e.emp_name, e.dept_id, d.dept_name, d.dept_location
 from emp e, dept d
 where e.dept_id = d.dept_id;
+-----+-----+-----+-----+-----+
| emp_id | emp_name | dept_id | dept_name | dept_location |
+-----+-----+-----+-----+-----+
1000	alex	10	ACCOUNTING	NEW YORK, NY
2000	ted	10	ACCOUNTING	NEW YORK, NY
5000	joe	10	ACCOUNTING	NEW YORK, NY
3000	mat	20	RESEARCH	DALLAS, TX
4000	max	20	RESEARCH	DALLAS, TX
+-----+-----+-----+-----+-----+
5 rows in set (0.00 sec)

```

Map-side Join is similar to a join but all the task will be performed by the mapper alone (note that the result of an INNER JOIN and map side join must be identical).

Given two tables A and B, the map-side join will be mostly suitable when table A is large and table B is a small to medium. Since table B is not very big, then we create a hash table from B and then broadcast it to all nodes. Next, we iterate all elements of table A by a mapper and then accessing a broadcasted hash table (which denotes table B).

Create two RDDs:

First, we create EMP as RDD[(dept\_id, (emp\_id, emp\_name))]:

```

EMP = spark.sparkContext.parallelize(
[
 (10, (1000, 'alex')),
 (10, (2000, 'ted')),
 (20, (3000, 'mat')),
 (20, (4000, 'max')),
]

```

```

 (10, (5000, 'joe'))
])

```

Next, we create DEPT as RDD[(dept\_id, (dept\_name , dept\_location))]:

```

DEPT= spark.sparkContext.parallelize(
[(10, ('ACCOUNTING', 'NEW YORK, NY')),
(20, ('RESEARCH', 'DALLAS, TX')),
(30, ('SALES', 'CHICAGO, IL')),
(40, ('OPERATIONS', 'BOSTON, MA')),
(50, ('MARKETING', 'Sunnyvale, CA')),
(60, ('SOFTWARE', 'Stanford, CA'))
])

```

Now that both EMP and DEPT have a common key on the dept\_id, then we join them:

```

>>> sorted(EMP.join(DEPT).collect())
[
(10, ((1000, 'alex'), ('ACCOUNTING', 'NEW YORK, NY'))),
(10, ((2000, 'ted'), ('ACCOUNTING', 'NEW YORK, NY'))),
(10, ((5000, 'joe'), ('ACCOUNTING', 'NEW YORK, NY'))),
(20, ((3000, 'mat'), ('RESEARCH', 'DALLAS, TX'))),
(20, ((4000, 'max'), ('RESEARCH', 'DALLAS, TX')))
]

```

How will the map-side join optimize the task? Here, I assume that EMM is a large data set and DEPT is a relatively small dat set. Using map side join, to join EMP with DEPT on dept\_id, we will create a broadcast varibale from a small table (this is a custom hash builder from an RDD):

```

build a dictionary of (key, value),
where key = dept_id
value = (dept_name , dept_location)
#
def to_hash_table(dept_as_list):
 hast_table = {}
 for d in dept_as_list:
 dept_id = d[0]
 dept_name_location = d[1]
 hast_table[dept_id] = dept_name_location
 return hast_table
#endif

```

```
dept_hash_table = to_hash_table(DEPT.collect())
```

Alternatively, you may build the same hash table by using an Spark action `collectAsMap()`, which returns the key-value pairs in this RDD (DEPT) to the master as a dictionary. Therefore, we can write:

```
dept_hash_table = DEPT.collectAsMap()
```

Now, using `pyspark.SparkContext.broadcast`, you can broadcast a read-only variable (`dept_hash_table`) to the Spark cluster, which will be available to all kinds of transformations (including mappers and reducers).

```
sc = spark.sparkContext
hash_table_broadcasted = sc.broadcast(dept_hash_table)
```

Now, how do we perform the map side join? In your mapper, you can access your `dept_hash_table` via:

```
dept_hash_table = hash_table_broadcasted.value
```

Next, I show the map side join by using a `map()` transformation:

```
joined = EMP.map(map_side_join)
```

Finally, `map_side_join()` is defined as:

```
e as an element of EMP RDD
def map_side_join(e):
 dept_id = e[0]
 # get hash_table from broadcasted object
 hash_table = hash_table_broadcasted.value
 dept_name_location = hash_table[dept_id]
 return (e, dept_name_location)
#end-def
```

Therefore, with map side join, you can avoid shuffling which can cost a lot and you avoid significant network I/O.

With a map-side join, we just used a `map()` function for each row of the EMP table, and retrieved dimension values (such as `dept_name` and `dept_location`) from the broadcasted hash table. If EMP table is very large, the `map()` function will be executed concurrently for each partition that has own copy of hash table. The map side join approach allows us not to shuffle the fact table (i.e., DEPT) and to get quite good join performance.

Below, I have listed advantages and disadvantages of using a map-side join.

#### *Advantages of using map-side join*

- The true join cost can be reduced since we are minimizing the cost that is incurred for sorting and merging in the shuffle and reduce stages.
- Map-side join improves the performance of the join task by decreasing the time to finish the join operation.

#### *Disadvantages of map-side join*

- Map-side join data pattern is proper when one of the RDDs/tables on which you perform map-side join operation is small enough to fit into the memory.
- If both RDDs/tables are not small at all, then map-side join is not suitable to perform map-side join on the RDDs/tables.

Next, I discuss another design pattern (Join using Bloom filters), which can be used in efficient joining of two tables.

## **Map-Side Join by DataFrames**

As I discussed in the preceding section, Map-side join makes sense when one of the tables — so called a fact table — is huge and the other table — so called a dimension table — is small enough to be broadcasted in memory.

In the following example, I will show how to use DataFrames along with Broadcast variables to solve the Map-side join operation

Consider the following tables (one fact and two dimensional tables). The idea for this example is inspired from [Map-Side Join in Spark](#).

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*2*  
-  
*l*  
. *F*  
*l*  
*i*  
*g*  
*h*  
*t*  
*s*

(  
*f*  
*a*  
*c*  
*t*  
*t*  
*a*  
*b*  
*l*  
*e*  
)

| from | to  | airline | flight number | departure |
|------|-----|---------|---------------|-----------|
| DTW  | ORD | SW      | 225           | 17:10     |
|      |     |         |               |           |

|     |     |    |      |      |
|-----|-----|----|------|------|
| DTW | JFK | SW | 355  | 8:20 |
| SEA | JFK | DL | 418  | 7:00 |
| SFO | LAX | AA | 1250 | 7:05 |
| SFO | JFK | VX | 12   | 7:05 |
| JFK | LAX | DL | 424  | 7:10 |
| LAX | SEA | DL | 5737 | 7:10 |

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*2*  
-  
*2*  
. *A*  
*i*  
*r*  
*p*  
*o*  
*r*  
*t*  
*s*

(  
*d*  
*i*  
*m*  
*e*  
*n*  
*s*  
*i*  
*o*  
*n*

*t*  
*a*  
*b*  
*l*

*e*  
)

| <b>code</b> | <b>name</b>             | <b>city</b>   | <b>state</b> |
|-------------|-------------------------|---------------|--------------|
| DTW         | Detroit Airport         | Detroit       | MI           |
| ORD         | Chicago O'Hare          | Chicago       | IL           |
| JFK         | John F. Kennedy Airport | New York      | NY           |
| LAX         | Los Angeles Airport     | Los Angeles   | CA           |
| SEA         | Seattle-Tacoma Airport  | Seattle       | WA           |
| SFO         | San Francisco Airport   | San Francisco | CA           |

*T*  
*a*  
*b*  
*l*  
*e*

*l*  
*2*  
-  
*3*  
. *A*  
*i*  
*r*  
*l*  
*i*  
*n*  
*e*  
*s*

(  
*d*  
*i*  
*m*  
*e*  
*n*  
*s*  
*i*  
*o*  
*n*

*t*  
*a*  
*b*  
*l*

*e*  
)

| <b>code</b> | <b>airline name</b> |
|-------------|---------------------|
| SW          | Southwest Airlines  |
| AA          | American Airlines   |
| DL          | Delta Airlines      |
| VX          | Virgin America      |

Suppose the goal to to expand the `Flights` table and replace airline codes with the actual airline names and replace airport codes with the actual airport names. This is join operation of `flights` — facts table — with two dimensional tables (`Airports` and `Airlines`). Since the dimensional tables are small enough, therefore we can broadcast these to all mappers in all worker nodes. Our desired joined output will look like:

T  
a  
b  
l  
e

l  
2  
-  
4  
.J  
o  
i  
n  
e  
d

t  
a  
b  
l  
e

| <b>from city</b> | <b>to city</b> | <b>airline</b>     | <b>flight number</b> | <b>departure</b> |
|------------------|----------------|--------------------|----------------------|------------------|
| Detroit          | Chicago        | Southwest Airlines | 225                  | 17:10            |
| Detroit          | New York       | Southwest Airlines | 355                  | 8:20             |
| Seattle          | New York       | Delta Airlines     | 418                  | 7:00             |
| San Francisco    | Los Angeles    | American Airlines  | 1250                 | 7:05             |
| San Francisco    | New York       | Virgin America     | 12                   | 7:05             |
| New York         | Los Angeles    | Delta Airlines     | 424                  | 7:10             |

To achieve the **Joined** table result, we need to do the following:

*Step-1: Create Cache for Airports*

Create a broadcast variable for airports: first we create an RDD and then save it a `dictionary[(key, value)]` where `key` is an airport `code`, and `value` is associated values.

*Step-2: Create Cache for Airlines*

Create a broadcast variable for airlines: first we create an RDD and then save it a `dictionary[(key, value)]` where `key` is an airline `code`, and `value` is the name of airline.

*Step-3: Create Facts Table — Flights*

Map each record of the `Flights` table and perform a simple join by lookup dictionaries created in Steps 1 & 2.

*Step-4: Map-Side Join:*

Map each record of the `Flights` table — known as a facts table — and perform a simple join by lookup cache dictionaries created in Steps 1 & 2.

## **Step-1: Create Cache for Airports**

This step creates a cache — as a Broadcast data to be available in all worker nodes — for Airports table:

```
>>> airports_data = [
... ("DTW", "Detroit Airport", "Detroit", "MI"),
... ("ORD", "Chicago O'Hare", "Chicago", "IL"),
... ("JFK", "John F. Kennedy Int. Airport", "New York", "NY"),
... ("LAX", "Los Angeles Int. Airport", "Los Angeles", "CA"),
... ("SEA", "Seattle-Tacoma Int. Airport", "Seattle", "WA"),
... ("SFO", "San Francisco Int. Airport", "San Francisco", "CA")
...]
>>>
>>> airports_rdd = spark.sparkContext.parallelize(airports_data)\
```

```

... .map(lambda tuple4: (tuple4[0], (tuple4[1],tuple4[2],tuple4[3])))

>>> airports_dict = airports_rdd.collectAsMap()
>>>
>>> airports_cache = spark.sparkContext.broadcast(airports_dict)
>>> airports_cache.value
{'DTW': ('Detroit Airport', 'Detroit', 'MI'),
 'ORD': ("Chicago O'Hare", 'Chicago', 'IL'),
 'JFK': ('John F. Kennedy Int. Airport', 'New York', 'NY'),
 'LAX': ('Los Angeles Int. Airport', 'Los Angeles', 'CA'),
 'SEA': ('Seattle-Tacoma Int. Airport', 'Seattle', 'WA'),
 'SFO': ('San Francisco Int. Airport', 'San Francisco', 'CA')}

```

## Step-2: Create Cache for Airlines

This step creates a cache — as a Broadcast data to be available in all worker nodes — for Airlines table:

```

>>> airlines_data = [
... ("SW", "Southwest Airlines"),
... ("AA", "American Airlines"),
... ("DL", "Delta Airlines"),
... ("VX", "Virgin America")
...]

>>> airlines_rdd = spark.sparkContext.parallelize(airlines_data) \
... .map(lambda tuple2: (tuple2[0], tuple2[1]))

>>> airlines_dict = airlines_rdd.collectAsMap()
>>> airlines_cache = spark.sparkContext.broadcast(airlines_dict)
>>> airlines_cache
>>> airlines_cache.value
{'SW': 'Southwest Airlines',
 'AA': 'American Airlines',
 'DL': 'Delta Airlines',
 'VX': 'Virgin America'}

```

## Step-3: Create Facts Table

This step creates a DataFrame — as a fact table — to be joined with caches created in Steps 1 & 2.

```

>>> flights_data = [
... ("DTW", "ORD", "SW", "225", "17:10"),
... ("DTW", "JFK", "SW", "355", "8:20"),
... ("SEA", "JFK", "DL", "418", "7:00"),
...

```

```

... ("SFO", "LAX", "AA", "1250", "7:05"),
... ("SFO", "JFK", "VX", "12", "7:05"),
... ("JFK", "LAX", "DL", "424", "7:10"),
... ("LAX", "SEA", "DL", "5737", "7:10")
...]
>>> flight_columns = ["from", "to", "airline", "flight_number", "departure"]
>>> flights = spark.createDataFrame(flights_data, flight_columns)
>>> flights.show(truncate=False)
+-----+-----+-----+-----+
|from|to |airline|flight_number|departure|
+-----+-----+-----+-----+
|DTW |ORD|SW |225 |17:10 |
|DTW |JFK|SW |355 |8:20 |
|SEA |JFK|DL |418 |7:00 |
|SFO |LAX|AA |1250 |7:05 |
|SFO |JFK|VX |12 |7:05 |
|JFK |LAX|DL |424 |7:10 |
|LAX |SEA|DL |5737 |7:10 |
+-----+-----+-----+-----+

```

## Step-4: Apply Map-Side Join

This step iterates the fact table — flights data — and performs the map-side join:

```

>>> from pyspark.sql.functions import udf
>>> from pyspark.sql.types import StringType
>>>
>>> def get_airport(code):
... return airports_cache.value[code][1]
...
>>> def get_airline(code):
... return airlines_cache.value[code]

>>> airport_udf = udf(get_airport, StringType())
...
>>> airport_udf = udf(get_airport, StringType())
>>>
>>> flights.select(
 airport_udf("from").alias("from_city"), ❶
 airport_udf("to").alias("to_city"), ❷
 airline_udf("airline").alias("airline_name"), ❸
 "flight_number", "departure").show(truncate=False)
+-----+-----+-----+-----+
|from_city|to_city|airline_name|flight_number|departure|
+-----+-----+-----+-----+
|Detroit |Chicago|Southwest Airlines|225 |17:10 |
|Detroit |New York|Southwest Airlines|355 |8:20 |
|Seattle |New York|Delta Airlines |418 |7:00 |
+-----+-----+-----+-----+

```

|               |             |                   |      |      |  |
|---------------|-------------|-------------------|------|------|--|
| San Francisco | Los Angeles | American Airlines | 1250 | 7:05 |  |
| San Francisco | New York    | Virgin America    | 12   | 7:05 |  |
| New York      | Los Angeles | Delta Airlines    | 424  | 7:10 |  |
| Los Angeles   | Seattle     | Delta Airlines    | 5737 | 7:10 |  |

- ❶ Map-Side join for Airport
- ❷ Map-Side join for Airport
- ❸ Map-Side join for Airline

## Efficient Joins using Bloom filters

Given two RDDs as  $A = \text{RDD}[(K, V)]$  (as a bigger RDD) and  $B = \text{RDD}[(K, W)]$  (as a smaller RDD), Spark enable us to perform join operation on key K. Joining two RDDs is a common operation when working with Spark. In some of the cases, a join is used as a form of filtering. For example, you want to perform an operation on a subset of the records in the  $\text{RDD}[(K, V)]$ , represented by entities in another  $\text{RDD}[(K, W)]$ . For this you can use an inner join to achieve that effect. But, you want to avoid the shuffle that the join operation introduces, especially if the  $\text{RDD}[(K, W)]$  you want to use for filtering is significantly smaller than the main  $\text{RDD}[(K, V)]$  on which you will perform your further computation.

You may do a broadcast join using a set (as a Bloom filter) constructed by collecting the smaller RDD you wish to filter by. But, this means collecting the whole smaller  $\text{RDD}[(K, W)]$  in driver memory, and even if it is relatively small (several thousand or million records), that can still lead to some undesirable memory pressure. If you want to avoid the shuffle that the join operation introduces, then you may use the Bloom filter. So the problem of joining  $\text{RDD}[(K, V)]$  with  $\text{RDD}[(K, W)]$  is reduced into a simple `map()` transformation, which we will check the key K against the Bloom filter constructed from the smaller  $\text{RDD}[(K, W)]$ .

## Bloom filter

A Bloom filter is a space-efficient probabilistic data structure, conceived by B. H. Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not; i.e. a query returns either **possibly in set** or **definitely not in set**. Elements can be added to the set, but not removed (though this can be addressed with a “counting” filter). The more elements that are added to the set, the larger the probability of false positives. The Bloom filter data structure may return true for elements that are not actually members of the set (this is called false-positives errors), but it will never return false for elements that are in the set; for each element in the set, the Bloom filter must return true.

Therefore, in a nutshell, we can summarize Bloom filter properties as:

- Given a big set  $S = \{x_1, x_2, \dots, x_n\}$ , Bloom filter is a probabilistic, fast, and space efficient cache builder for a big data set; this is basically approximating set membership operation:

is  $x$  in  $S$

- It tries to answer lookup questions:

does item  $x$  exist in a set  $S$ ?

- It allows false positive errors as they only cost us an extra data set access. This means that for some  $x$ , which is not in the set, Bloom filter might indicate that  $x$  is in the set.
- It does not allow false negative errors, because they result in wrong answers. This means that if  $x$  is not in the set, then for sure it will indicate that  $x$  is not in the set. Thus, the Bloom filter does not allow false negatives, but can allow false positives.

Let's focus on a simple join example between two relations or tables: let's say that we want to join  $R = \text{RDD}(K, V)$  and  $S = \text{RDD}(K, W)$  on a common key  $K$ . Further assume that

$\text{count}(R) = 1,000,000,000$  (larger data set)  
 $\text{count}(S) = 10,000,000$  (smaller data set)

To do basic join, we need to check 10 trillion ( $10^{12}$ ) records, which is a huge and time consuming process. One idea to reduce time and complexity of join operation between R and S is to use a Bloom filter on relation S (smaller data set) and then use the built Bloom filter data structure on relation R. This can eliminate the non-needed records from R (it might reduce it to 20,000,000) and hence the join becomes fast and efficient.

Next we semi-formalize Bloom filter data structure: what is involved in Bloom filter probabilistic data structures? How do we construct one? what is the probability of **false positive errors**, and how we can decrease the probability of **false positive errors**. This is how it works:

- Given a set  $S = \{x_1, x_2, \dots, x_n\}$
- Let  $B$  be an  $m$  ( $m > 1$ ) bit array, initialized with 0s.  $B$ 's elements are  $B[0], B[1], B[2], \dots, B[m-1]$ . The memory required for array  $B$  is only a fraction of the one needed for storing the whole set  $S$ . By selecting the bigger bit vector (array  $B$ ), the probability of false positive rate will decrease.
- Let  $\{H_1, H_2, \dots, H_k\}$  be a set of  $k$  hash functions, If  $H_i(x_j) = a$  then set  $B[a] = 1$ . You may use SHA1, MD5, and Murmur as hash functions. For example, you may use the following as hash functions:
  - $H_i(x) = MD5(x+i)$
  - $H_i(x) = MD5(x || i)$
- To check if  $x$  in  $S$ , check  $B$  at  $H_i(x)$ . All  $k$  values must be 1.
- It is possible to have a false positive; all  $k$  values are 1, but  $x$  is not in  $S$ .
- The probability of false positives is (for details, see Wikipedia).

$$\Pr(1 - \Pr[1 - \frac{1}{m}]^k)^k \approx \Pr(1 - e^{-\frac{k}{m}})^k$$

- What are the optimal hash functions? What is the optimal number of hash functions? For a given  $m$  (number of bits selected for Bloom filter) and  $n$  (size of big data set), the value of  $k$  (the number of hash functions) that minimizes the probability of false positives is ( $\ln$  stands for “natural logarithm”)

$$k = \lceil m / n \rceil \ln(2)$$

$$m = -\frac{n}{\ln(p)} \approx \frac{n}{(\ln(2))^2}$$

Therefore, the probability that a specific bit has been flipped to 1 is:

$$1 - (1 - \frac{1}{m})^k \approx 1 - e^{-kn/m}$$

Next, let's take a look at a Bloom filter example.

## A Simple Bloom Filter Example

This example shows how to insert and query on a bloom filter of size 10 ( $m = 10$ ) with three hash functions  $H = \{H_1, H_2, H_3\}$  and let  $H(x)$  denote the result of these three hash functions. We start with a 10-bit long array  $B$  initialized to 0:

```

Array B:
initialized:
 index 0 1 2 3 4 5 6 7 8 9
 value 0 0 0 0 0 0 0 0 0 0

insert element a, H(a) = (2, 5, 6)
 index 0 1 2 3 4 5 6 7 8 9
 value 0 0 1 0 0 1 1 0 0 0

insert element b, H(b) = (1, 5, 8)
 index 0 1 2 3 4 5 6 7 8 9
 value 0 1 1 0 0 1 1 0 1 0

query element c
H(c) = (5, 8, 9) => c is not a member (since B[9]=0)

query element d
H(d) = (2, 5, 8) => d is a member (False Positive)

query element e
H(e) = (1, 2, 6) => e is a member (False Positive)

```

```
query element f
H(f) = (2, 5, 6) => f is a member (Positive)
```

## Bloom Filter in Python

The following code segment shows how to create and use a Bloom filter in Python (you may roll your own Bloom filter library, but if you find a proper library, then use it).

```
instantiate BloomFilter with custom settings,
>>> from bloom_filter import BloomFilter
>>> bloom = BloomFilter(max_elements=100000, error_rate=0.01)

Test whether the bloom-filter has seen a key:
>>> "test-key" in bloom
False

Mark the key as seen
>>> bloom.add("test-key")

Now check again
>>> "test-key" in bloom
True
```

## Using Bloom Filter in PySpark

Bloom filter is a small, compact, and fast data structure for set membership. It can be used in the join of two RDDs/relations/tables such as R(K, V) and S(K, W) where one of the relations has huge number of records (for example, R to have 1000,000,000 records) and the other relation has small number of records (for example, S to have 10,000,000 records). To do a join on key field “K” between R and S, it will take a long time and it is inefficient.

We can use Bloom filter data structure in the following way: build a Bloom filter out of relation S(K, W), and then test values R(K, V) for membership using the built Bloom filter (using Sparks broadcast mechanism). Note that, for reduce-side join optimization, we use a Bloom filter on the map tasks, which will force an I/O cost reduction for the PySpark job. How do we use the Bloom filter concept in mappers? The following steps show how to use a Bloom filter

(representation of S) in mappers (R), which will be substitute for the join operation between R and S.

### *STEP-1: Build Bloom filter*

Construction of the bloom filter; this a small PySpark code segment, which uses the smaller of the two relations/tables for constructing the bloom filter data structure. First, initialize the Bloom filter (create an instance of `BloomFilter`), then you may use `BloomFilter.add()` to build the Bloom filter. Let's call the built Bloom filter as `the_bloom_filter`

### *STEP-2: Broadcast the Built Bloom filter*

Use `SparkContext.broadcast()` to broadcast the built Bloom filter (`the_bloom_filter`) to all worker nodes (so that it can be available to all Spark transformations including mappers)

```
to broadcast it to all worker nodes for read only purposes
sc = spark.sparkContext
broadcasted_bloom_filter = sc.broadcast(the_bloom_filter)
```

### *STEP-3: Use the broadcasted object in Mappers*

Now, we can user the Bloom filter to get rid of the non-needed elements.

```
e is an element of R(k, b)
def bloom_filter_function(e):
 # get a copy of Bloom filter
 the_bloom_filter = broadcasted_bloom_filter.value()
 # use the_bloom_filter for element e
 key = e[0]
 if key in the_bloom_filter:
 return True
 else:
 return False
#end-def
```

We use the `filter_function()` for `R=RDD[(K, V)]` to keep the elements if and only if the key is in `S=RDD[(K, W)]`.

```
R=RDD[(K, V)]
joined = RDD[(K, V)] where K is in S=RDD[(K, W)]
```

```
joined = R.filter(bloom_filter_function)
```

## About the Author

**Dr. Mahmoud Parsian** is a software architect and author. He leads Illumina's Big Data team focused on large-scale genome analytics. He is a practicing software professional with 30+ years of experience as a developer, designer, architect, and author. For the past 15 years, he has been involved in Python, Java server-side, databases, MapReduce, Spark, Machine Learning, and distributed computing.

Dr. Parsian is leading and developing scalable distributed algorithms for genomics data using Python, Java, MapReduce, Spark, and open source tools. Dr. Parsian is an adjunct faculty at Santa Clara University teaching Big Data and Machine Learning.

Dr. Parsian has published four books:

- PySpark Algorithms (Amazon, 2019)
- Data Algorithms (O'Reilly, 2015)
- JDBC Recipes (Apress, 2005)
- JDBC Metadata Recipes (Apress, 2006)

Dr. Parsian earned his M.S. and Ph.D in Computer Science from Iowa State University, Ames, Iowa.