

Lesson Proper for Week 1

What is Big Data Analytics?

BigData analytics is a process used to **extract meaningful insights**, such as hidden patterns, unknown correlations, market trends, and customer preferences. Big Data analytics provides various advantages—it can be used for better **decision making**, **preventing fraudulent activities**, among other things.

What is Big Data?

Big Data is a **massive amount of data** sets that cannot be stored, processed, or analyzed using traditional tools.

Today, there are millions of data sources that generate data at a very rapid rate. These data sources are present across the world. Some of the largest sources of data are social media platforms and networks. Let's use Facebook as an example—it generates more than 500 terabytes of data every day. This data includes pictures, videos, messages, and more.

Data also exists in different formats, like **structured data**, **semi-structured data**, and **unstructured data**. For example, in a regular Excel sheet, data is classified as structured data—with a definite format. In contrast, **emails** fall under semi-structured, and your pictures and videos fall under unstructured data. All this data combined makes up Big Data.

But, Big Data in its raw form is of no use. So, now let us understand Big Data Analytics.

Let's look into the four advantages of Big Data analytics.

Benefits & Advantages of Big Data Analytics

1. Risk Management

Use Case: Banco de Oro, a Philippine banking company, uses Big Data analytics to identify fraudulent activities and discrepancies. The organization leverages it to narrow down a list of suspects or root causes of problems.

2. Product Development and Innovations

Use Case: Rolls-Royce, one of the largest manufacturers of jet engines for airlines and armed forces across the globe, uses Big Data analytics to analyze how efficient the engine designs are and if there is any need for improvements.

3. Quicker and Better Decision Making Within Organizations

Use Case: Starbucks uses Big Data analytics to make strategic decisions. For example, the company leverages it to decide if a particular location would be suitable for a new outlet or not. They will analyze several different factors, such as population, demographics, accessibility of the location, and more.

4. Improve Customer Experience

Use Case: Delta Air Lines uses Big Data analysis to improve customer experiences. They monitor tweets to find out their customers' experience regarding their journeys, delays, and so on. The airline identifies negative tweets and does what's necessary to remedy the situation. By publicly addressing these issues and offering solutions, it helps the airline build good customer relations.

What is Big Data?

According to **Gartner**, the definition of Big Data –

“Big data” is high-volume, velocity, and variety information assets that demand cost-effective, innovative forms of information processing for enhanced insight and decision making. This definition clearly answers the “What is Big Data?” question – Big Data **refers to complex and large data sets that have to be processed and analyzed to uncover valuable information that can benefit businesses and organizations**. However, there are certain basic tenets of Big Data that will make it even simpler to answer what is Big Data:

- It refers to a massive amount of data that keeps on growing exponentially with time.
- It is so voluminous that it cannot be processed or analyzed using conventional data processing techniques.
- It includes data mining, data storage, data analysis, data sharing, and data visualization.

- The term is an all-comprehensive one including data, data frameworks, along with the tools and techniques used to process and analyze the data.

The History of Big Data

Although the concept of big data itself is relatively new, the origins of large data sets go back to the 1960s and '70s when the world of data was just getting started with the first data centers and the development of the relational database.

Around 2005, people began to realize just how much data users generated through Facebook, YouTube, and other online services. Hadoop (an open-source framework created specifically to store and analyze big data sets) was developed that same year. NoSQL also began to gain popularity during this time.

The development of open-source frameworks, such as Hadoop (and more recently, Spark) was essential for the growth of big data because they make big data easier to work with and cheaper to store. In the years since then, the volume of big data has skyrocketed. Users are still generating huge amounts of data—but it's not just humans who are doing it.

With the advent of the Internet of Things (IoT), more objects and devices are connected to the internet, gathering data on customer usage patterns and product performance. The emergence of machine learning has produced still more data.

While big data has come far, its usefulness is only just beginning. Cloud computing has expanded big data possibilities even further. The cloud offers truly elastic scalability, where developers can simply spin up ad hoc clusters to test a subset of data.

Benefits of Big Data and Data Analytics

- Big data makes it possible for you to gain more complete answers because you have more information.
- More complete answers mean more confidence in the data—which means a completely different approach to tackling problems.

Types of Big Data

Now that we are on track with what is big data, let's have a look at the types of big data:

a) Structured

Structured is one of the types of big data and By structured data, we mean data that can be processed, stored, and **retrieved in a fixed format**. It refers to highly organized information that can be readily and seamlessly stored and accessed from a database by simple search engine algorithms. For instance, the employee table in a company database will be structured as the employee details, their job positions, their salaries, etc., will be present in an organized manner.

b) Unstructured

Unstructured data refers to the data that **lacks any specific** form or structure whatsoever. This makes it very difficult and time-consuming to process and analyze unstructured data. Email is an example of unstructured data. Structured and unstructured are two important types of big data.

c) Semi-structured

Semi-structured is the third type of big data. Semi-structured data pertains to the data containing both the formats mentioned above, that is, structured and unstructured data. To be precise, it refers to the data that, although has not been classified under a particular repository (database), yet contains vital information or tags that segregate individual elements within the data. Thus we come to the end of types of data.

CHARACTERISTICS OF BIG DATA

Back in 2001, **Gartner analyst Doug Laney** listed the 3 'V's of Big Data – **Variety, Velocity, and Volume**. Let's discuss the characteristics of big data. These characteristics, isolated, are enough to know what big data is. Let's look at them in depth:

a) Variety

Variety of Big Data refers to **structured, unstructured, and semi-structured data** that is gathered from multiple sources. While in the past, data could only be collected from spreadsheets and databases, today data comes in an array of forms such as emails, PDFs, photos, videos, audios, SM posts, and so much more. Variety is one of the important characteristics of big data.

b) Velocity

Velocity essentially refers to the **speed** at which data is being created in real-time. In a broader prospect, it comprises the rate of change, linking of incoming data sets at varying speeds, and activity bursts.

c) Volume

Volume is one of the characteristics of **big data**. We already know that Big Data indicates huge 'volumes' of data that is being generated on a daily basis from various sources like social media platforms, business processes, machines, networks, human interactions, etc. Such a large amount of data is stored in data warehouses. Thus comes to the end of characteristics of big data.

Why is Big Data Important?

The importance of big data does not revolve around how much data a company has but how a company utilizes the collected data. Every company uses data in its own way; the more efficiently a company uses its data, the more potential it has to grow. The company can take data from any source and analyze it to find answers which will enable:

- 1. Cost Savings:** **Some tools of Big Data like Hadoop and Cloud-Based Analytics** can bring cost advantages to business when large amounts of data are to be stored and these tools also help in identifying more efficient ways of doing business.
- 2. Time Reductions:** The **high speed of tools** like Hadoop and in-memory analytics can easily identify new sources of data which helps businesses analyze data immediately and make quick decisions based on the learning.
- 3. Understand the market conditions:** **By analyzing big data you can get a better understanding of current market conditions**. For example, by analyzing customers' purchasing behaviors, a company can find out the products that are sold the most and produce products according to this trend. By this, it can get ahead of its competitors.
- 4. Control online reputation:** Big data tools can do **sentiment analysis**. Therefore, you can get feedback about **who is saying what about your company**. If you want to monitor and improve the online presence of your business, then, big data tools can help in all this.

5. Using Big Data Analytics to Boost Customer Acquisition and Retention

The customer is the most important asset any business depends on. There is no single business that can claim success without first having to establish a solid customer base.

However, even with a customer base, a business cannot afford to disregard the high competition it faces. If a business is slow to learn what customers are looking for, then it is very easy to begin offering poor quality products. In the end, loss of clientele will result, and this creates an adverse overall effect on business success. The use of big data allows businesses to observe various customer related patterns and trends. Observing customer behavior is important to trigger loyalty.

6. Using Big Data Analytics to Solve Advertisers Problem and Offer Marketing Insights

Big data analytics can help change all business operations. This includes the ability to match customer expectations, changing a company's product line and of course ensuring that the marketing campaigns are powerful.

7. Big Data Analytics As a Driver of Innovations and Product Development

Another huge advantage of big data is the ability to help companies innovate and redevelop their products.

The Lifecycle Phases of Big Data Analytics

Now, let's review how Big Data analytics works:

- **Stage 1 - Business case evaluation** - The Big Data analytics lifecycle begins with a business case, which defines the reason and goal behind the analysis.
- **Stage 2 - Identification of data** - Here, a broad variety of data sources are identified.
- **Stage 3 - Data filtering** - All of the identified data from the previous stage is filtered here to remove corrupt data.
- **Stage 4 - Data extraction** - Data that is not compatible with the tool is extracted and then transformed into a compatible form.
- **Stage 5 - Data aggregation** - In this stage, data with the same fields across different datasets are integrated.
- **Stage 6 - Data analysis** - Data is evaluated using analytical and statistical tools to discover useful information.
- **Stage 7 - Visualization of data** - With tools like Tableau, Power BI, and QlikView, Big Data analysts can produce graphic visualizations of the analysis.
- **Stage 8 - Final analysis result** - This is the last step of the Big Data analytics lifecycle, where the final results of the analysis are made available to business stakeholders who will take action.

Different Types of Big Data Analytics

Here are the four types of Big Data analytics:

1. Descriptive Analytics

This summarizes past data into a form that people can easily read. This helps in creating reports, like a company's revenue, profit, sales, and so on. Also, it helps in the tabulation of social media metrics.

Use Case: The Dow Chemical Company analyzed its past data to increase facility utilization across its office and lab space. Using descriptive analytics, Dow was able to identify underutilized space. This space consolidation helped the company save nearly US \$4 million annually.

2. Diagnostic Analytics

This is done to understand what caused a problem in the first place. Techniques like drill-down, data mining, and data recovery are all examples. Organizations use diagnostic analytics because they provide an in-depth insight into a particular problem.

Use Case: An e-commerce company's report shows that their sales have gone down, although customers are adding products to their carts. This can be due to various reasons like the form didn't load correctly, the shipping fee is too high, or there are not enough payment options available. This is where you can use diagnostic analytics to find the reason.

3. Predictive Analytics

This type of analytics looks into the historical and present data to make predictions of the future. Predictive analytics uses data mining, AI, and machine learning to analyze current data and make predictions about the future. It works on predicting customer trends, market trends, and so on.

Use Case: PayPal determines what kind of precautions they have to take to protect their clients against fraudulent transactions. Using predictive analytics, the company uses all the historical payment data and user behavior data and builds an algorithm that predicts fraudulent activities.

4. Prescriptive Analytics

This type of analytics prescribes the solution to a particular problem. **Perspective analytics works with both descriptive and predictive analytics.** Most of the time, it relies on AI and machine learning.

Use Case: Prescriptive analytics can be used to maximize an airline's profit. This type of analytics is used to build an algorithm that will automatically adjust the flight fares based on numerous factors, including customer demand, weather, destination, holiday seasons, and oil prices.

Big Data Analytics Tools

Here are some of the key big data analytics tools :

- Hadoop - **helps in storing and analyzing data**
- MongoDB - **used on datasets that change frequently**
- Talend - used for data **integration and management**
- Cassandra - a distributed database **used to handle chunks of data**
- Spark - used **for real-time processing** and analyzing large amounts of data
- STORM - an **open-source real-time computational system**
- Kafka - a distributed streaming platform that is used for **fault-tolerant storage**

Big Data Industry Applications

Here are some of the sectors where Big Data is actively used:

Ecommerce - Predicting customer trends and optimizing prices are a few of the ways e-commerce uses Big Data analytics

Marketing - Big Data analytics helps to drive high ROI marketing campaigns, which result in improved sales

Education - Used to develop new and **improve existing courses** based on market requirements

Healthcare - With the help of a patient's medical history, Big Data analytics is used to predict how likely they are to have health issues

Media and entertainment - Used to understand the demand of shows, movies, songs, and more to deliver a personalized recommendation list to its users

Banking - Customer income and spending patterns help to predict the likelihood of choosing various banking offers, like loans and credit cards

Telecommunications - Used to forecast network capacity and improve customer experience

Government - Big Data analytics helps governments in law enforcement, among other things

Lesson Proper for Week 2

What is Spark?

Apache Spark is an open-source, distributed processing system used for big data workloads. It utilizes in-memory caching and optimized query execution for fast queries against data of any size. Simply put, Spark is a fast and general engine for large-scale data processing.

The fast part means that it's faster than previous approaches to work with Big Data like classical MapReduce. The secret for being faster is that Spark runs on memory (RAM), and that makes the processing much faster than on disk drives.

The general part means that it can be used for multiple things like running distributed SQL, creating data pipelines, ingesting data into a database, running Machine Learning algorithms, working with graphs or data streams, and much more.

Components

Apache Spark Core – Spark Core is the underlying general execution engine for the Spark platform that all other functionality is built upon. It provides in-memory computing and referencing datasets in external storage systems.

Spark SQL – Spark SQL is Apache Spark's module for working with structured data. The interfaces offered by Spark SQL provides Spark with more information about the structure of both the data and the computation being performed.

Spark Streaming – This component allows Spark to process real-time streaming data. Data can be ingested from many sources like Kafka, Flume, and HDFS (Hadoop

Distributed File System). Then the data can be processed using complex algorithms and pushed out to file systems, databases, and live dashboards.

MLlib (Machine Learning Library) – Apache Spark is equipped with a rich library known as MLlib. This library contains a wide array of machine learning algorithms- classification, regression, clustering, and collaborative filtering. It also includes other tools for constructing, evaluating, and tuning ML Pipelines. All these functionalities help Spark scale out across a cluster.

GraphX – Spark also comes with a **library to manipulate graph databases and perform computations called GraphX**. GraphX unifies ETL (Extract, Transform, and Load) process, exploratory analysis, and iterative graph computation within a single system.

Why Spark?

I think the following four main reasons from Apache Spark™ official website are good enough to convince you to use Spark.

1. Speed

Run programs up to **100x faster than Hadoop MapReduce in memory**, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.

2. Ease of Use

Write applications quickly in Java, Scala, Python, R.

Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it interactively from the Scala, Python and R shells.

3. Generality

Combine SQL, streaming, and complex analytics.

Spark powers a stack of libraries including SQL and DataFrames, MLlib for machine learning,

GraphX, and Spark Streaming. You can combine these libraries seamlessly in the same application.

4. Runs Everywhere

Spark runs on Hadoop, Mesos, standalone, or in the cloud. It can access diverse data sources including

HDFS, Cassandra, HBase, and S3

Modularity

Spark operations can be applied across many types of workloads and expressed in any of the supported programming languages: Scala, Java, Python, SQL, and R. Spark offers unified libraries with well-documented APIs that include the following modules as core components: Spark SQL, Spark Structured Streaming, Spark MLlib, and GraphX, combining all the workloads running under one engine. We'll take a closer look at all of these in the next section.

You can write a single Spark application that can do it all—no need for distinct engines for disparate workloads, no need to learn separate APIs. With Spark, you get a unified processing engine for your workloads.

Extensibility

Spark focuses on its fast, parallel computation engine rather than on storage. Unlike Apache Hadoop, which included both storage and compute, Spark decouples the two. That means you can use Spark to read data stored in myriad sources—Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMSs, and more—and process it all in memory. Spark's DataFrameReaders and DataFrame Writers can also be extended to read data from other sources, such as Apache Kafka, Kinesis, Azure Storage, and Amazon S3, into its logical data abstraction, on which it can operate.

The community of Spark developers maintains a list of third-party Spark packages as part of the growing ecosystem (see Figure). This rich ecosystem of packages includes Spark connectors for a variety of external data sources, performance monitors, and more.

Hadoop and Spark

Hadoop as a big data processing technology has been around for 10 years and has proven to be the solution of choice for processing large data sets. MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multi-pass computations and algorithms. Each step in the data processing workflow has one Map phase and one Reduce phase and you'll need to convert any use case into MapReduce pattern to leverage this solution.

The Job output data between each step has to be stored in the distributed file system before the next step can begin. Hence, this approach tends to be slow due to replication & disk storage. Also, Hadoop solutions typically include clusters that are hard to set up and manage. It also requires the integration of several tools for different big data use cases (like Mahout for Machine Learning and Storm for streaming data processing).

If you wanted to do something complicated, you would have to string together a series of MapReduce jobs and execute them in sequence. Each of those jobs was high-latency, and none could start until the previous job had finished completely.

Spark allows programmers to develop complex, multi-step data pipelines using directed acyclic graph (DAG) pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data.

Spark runs on top of existing Hadoop Distributed File System (HDFS) infrastructure to provide enhanced and additional functionality. It provides support for deploying Spark applications in an existing Hadoop v1 cluster (with SIMR – Spark-Inside-MapReduce) or Hadoop v2 YARN cluster or even Apache Mesos.

We should look at Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop. It's not intended to replace Hadoop but to provide a comprehensive and unified solution to manage different big data use cases and requirements.

Spark SQL. Spark SQL is a component on top of Spark Core that introduces a new set of data abstraction called SchemaRDD.

Spark Features

Spark takes MapReduce to the next level with less expensive shuffles in the data processing. With capabilities like in-memory data storage and near real-time processing, the performance can be several times faster than other big data technologies.

Spark also supports lazy evaluation of big data queries, which helps with optimization of the steps in data processing workflows. It provides a higher level API to improve developer productivity and a consistent architect model for big data solutions.

Spark holds intermediate results in memory rather than writing them to disk which is very useful especially when you need to work on the same dataset multiple times. It's designed to be an execution engine that works both in-memory and on-disk. Spark operators perform external operations when data does not fit in memory. Spark can be used for processing datasets that are larger than the aggregate memory in a cluster.

Spark will attempt to store as much as data in memory and then will spill to disk. It can store part of a data set in memory and the remaining data on the disk. You have to look at your data and use cases to assess the memory requirements. With this in-memory data storage, Spark comes with performance advantage.

Other Spark features include:

- Supports more than just Map and Reduce functions.
- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
- Provides concise and consistent APIs in Scala, Java and Python.
- Offers interactive shells for Scala and Python. This is not available in Java yet.

Spark is written in **Scala Programming Language** and runs on **Java Virtual Machine (JVM)** environment. It currently supports the following languages for developing applications using Spark:

- Scala
- Java
- Python
- Clojure
- R

Lesson Proper for Week 3

Spark Ecosystem

Other than Spark Core API, there are additional libraries that are part of the Spark ecosystem and provide additional capabilities in Big Data analytics and Machine Learning areas.

These libraries include:

-

Spark Streaming:

- Spark Streaming can be used for processing the real-time streaming data. **This is based on micro batch style of computing and processing.** It uses the DStream which is basically a series of RDDs, to process the real-time data.

-

Spark SQL:

- Spark SQL provides the capability to expose the Spark datasets over JDBC API and allow running the SQL like queries on Spark data using traditional BI and visualization tools. **Spark SQL** allows the users to ETL their data from different formats it's currently in (like JSON, Parquet, a Database), transform it, and expose it for ad-hoc querying.

-

Spark MLlib:

- MLlib is Spark's scalable machine learning library consisting of common learning **algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives.**

-

Spark GraphX:

- GraphX is the new (alpha) Spark API for graphs and graph-parallel computation. At a high level, GraphX extends the Spark RDD by introducing the Resilient Distributed Property Graph: a directed multi-graph with properties attached to each vertex and edge. To support graph computation, GraphX exposes a set of fundamental operators (e.g., subgraph, joinVertices, and aggregateMessages) as well as an optimized variant of the Pregel API. In addition, **GraphX includes** a growing collection of graph algorithms and builders to simplify graph analytics tasks.

Outside of these libraries, there are others like BlinkDB and Tachyon.

BlinkDB is an approximate query engine and can be used for running interactive SQL queries on large volumes of data. It allows users to trade-off query accuracy for response time. It works on large data sets by running queries on data samples and presenting results annotated with meaningful error bars.

Tachyon is a memory-centric distributed file system enabling reliable file sharing at memory-speed across cluster frameworks, such as Spark and MapReduce. It caches working set files in memory, thereby avoiding going to disk to load datasets that are frequently read. This enables different jobs/queries and frameworks to access cached files at memory speed.

And there are also integration adapters with other products like Cassandra (Spark Cassandra Connector) and R (SparkR). With Cassandra Connector, you can use Spark to access data stored in a Cassandra database and perform data analytics on that data. Following diagram (Figure 1) shows how these different libraries in Spark ecosystem are related to each other.

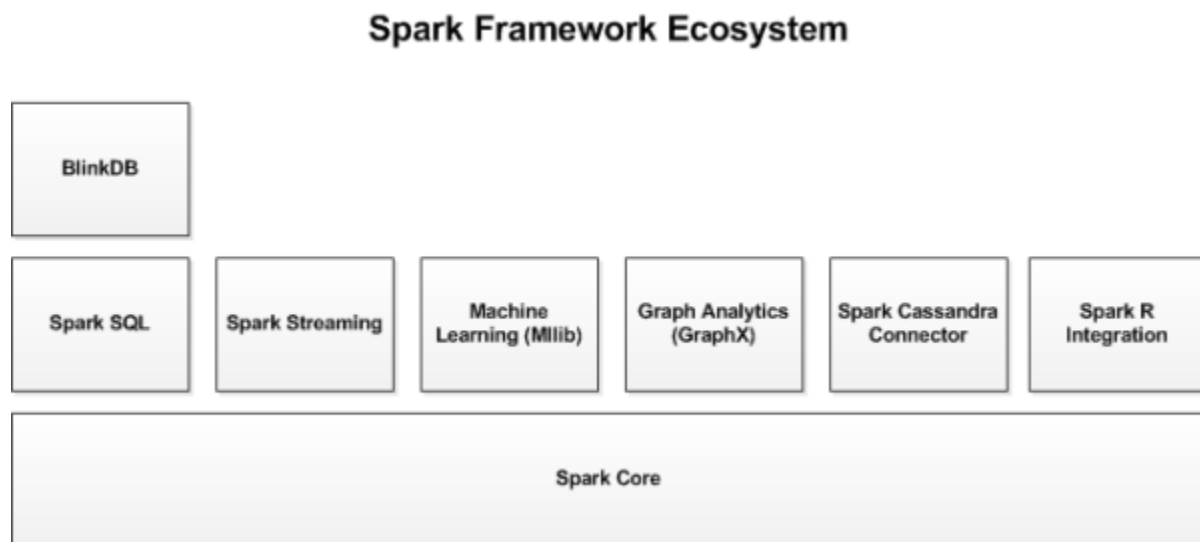


Figure 1. Spark Framework Libraries

We'll explore these libraries in future articles in this series.

Spark Architecture

Spark Architecture includes following **three main components:**

- Data Storage
- API
- Management Framework

Let's look at each of these components in more detail.

Data Storage:

Spark uses the HDFS file system for **data** storage purposes. It works with any Hadoop compatible data source including HDFS, HBase, Cassandra, etc.

API:

The API provides the application developers to create Spark based applications using a standard API interface. Spark provides API for Scala, Java, and Python programming languages.

Following are the website links for the Spark API for each of these languages.

- **Scala API**
- **Java**
- **Python**

Resource Management:

Spark can be deployed as a Stand-alone server or it can be on a distributed computing framework like Mesos or YARN.

Figure 2 below shows these components of Spark architecture model.

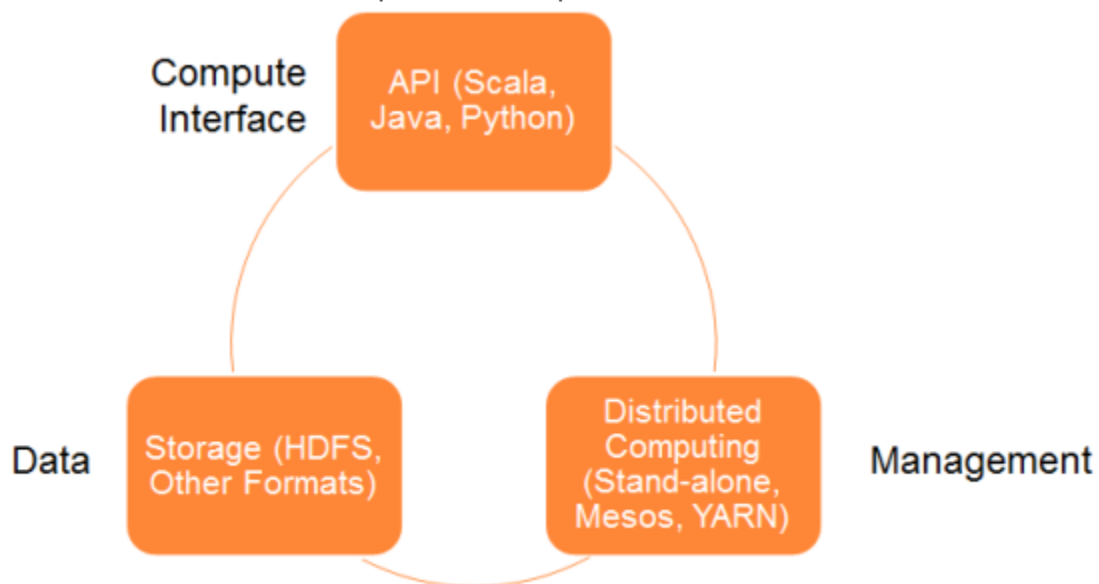


Figure 2. Spark Architecture

Resilient Distributed Datasets

Resilient Distributed Dataset (based on Matei's research paper) or **RDD** is the **core concept in Spark framework**. Think about RDD as a table in a database. It can hold any type of data. Spark stores data in **RDD on different partitions**.

They help with rearranging the computations and optimizing the data processing.

They are also fault tolerant because an RDD knows how to recreate and recompute the datasets.

RDDs are immutable. You can modify an RDD with a transformation but the transformation returns you a new RDD whereas the original RDD remains the same.

RDD supports two types of operations:

- Transformation
- Action

Transformation: Transformations **don't return a single value, they return a new RDD. Nothing gets evaluated when you call a Transformation function, it just takes an RDD and returns a new RDD.**

Some of the Transformation functions are map, filter, flatMap, groupByKey, reduceByKey, aggregateByKey, pipe, and coalesce.

Action: **Action** operation evaluates and returns a new value. When an Action function is called on a RDD object, all the data processing queries are computed at that time and the result value is returned.

Some of the Action operations are reduce, collect, count, first, take, countByKey, and foreach.

How to Install Spark

There are few different to install and use Spark. You can install it on your machine as a stand-alone framework or use one of Spark Virtual Machine (VM) images available from vendors like Cloudera, HortonWorks, or MapR. Or you can also use Spark installed and configured in the cloud (like Databricks Cloud).

In this article, we'll install Spark as a stand-alone framework and launch it locally. Spark 1.2.0 version was released recently. We'll use this version for sample application code demonstration.

How to Run Spark

When you install Spark on the local machine or use a Cloud based installation, there are few different modes you can connect to Spark engine.

The following table shows the Master URL parameter for the different modes of running Spark.

Master URL	Description
Local	Run Spark locally with one worker thread (i.e. no parallelism at all).
local[K]	Run Spark locally with K worker threads (ideally, set this to the number of cores on your machine).
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
spark://HOST:PORT	Connect to the given Spark standalone cluster master. The port must be whichever one your master is configured to use, which is 7077 by default.
mesos://HOST:PORT	Connect to the given Mesos cluster. The port must be whichever one your is configured to use, which is 5050 by default. Or, for a Mesos cluster using ZooKeeper, use mesos://zk://....
yarn-client	Connect to a YARN cluster in client mode. The cluster location will be found based on the HADOOP_CONF_DIR variable.
yarn-cluster	Connect to a YARN cluster in cluster mode. The cluster location will be found based on HADOOP_CONF_DIR.

How to Interact with Spark

Once Spark is up and running, you can connect to it using the Spark shell for interactive **data analysis**. Spark Shell is available in both Scala and Python languages. Java doesn't support an interactive shell yet, so this feature is currently not available in Java. You use the commands `spark-shell.cmd` and `pyspark.cmd` to run Spark Shell using Scala and Python respectively.

Spark Web Console

When Spark is running in any mode, you can view the Spark job results and other statistics by accessing Spark Web Console via the following URL:

`http://localhost:4040`

Spark Console is shown in Figure 3 below with tabs for Stages, Storage, Environment, and Executors.

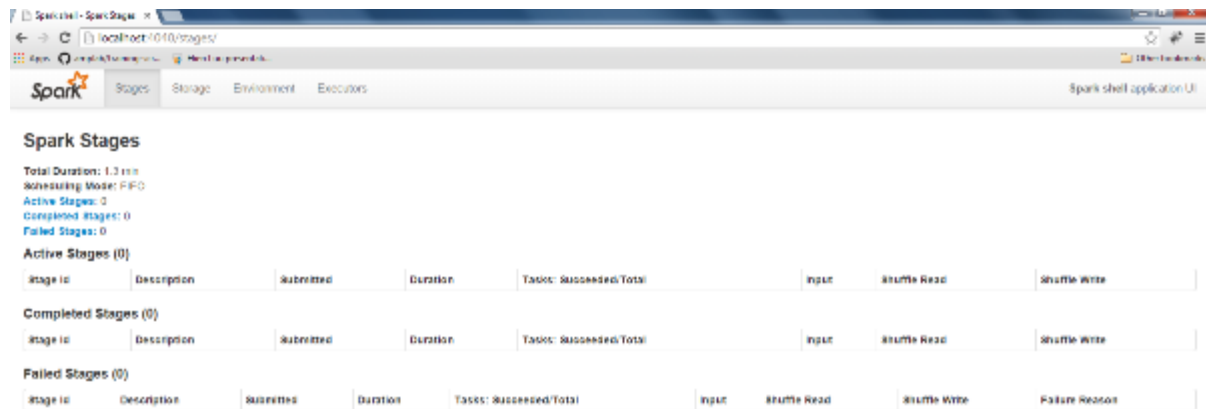


Figure 3. Spark Web Console

Shared Variables

Spark provides two types of shared variables to make it efficient to run the Spark programs in a cluster. These are Broadcast Variables and **Accumulators**.

Broadcast Variables: Broadcast variables allow to keep read-only variable cached on each machine

instead of sending a copy of it with tasks. They can be used to give the nodes in the cluster copies of large input datasets more efficiently.

Following code snippet shows how to use the broadcast variables.

```
//
// Broadcast Variables
//
val broadcastVar = sc.broadcast(Array(1, 2, 3))
broadcastVar.value
```

Accumulators: Accumulators are only added using an associative operation and can therefore be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Tasks running on the cluster can add to an accumulator variable using the add method. However, they cannot read its value. Only the driver program can read the accumulator's value.

The code snippet below shows how to use Accumulator shared variable:

```
//
// Accumulators
//

val accum = sc.accumulator(0, "My Accumulator")

sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

accum.value

Sample Spark Application

The sample application I cover in this module is a simple Word Count application. This is the same example one would cover when they are learning Big Data processing with Hadoop. We'll perform some data analytics queries on a text file. The text file and the data set in this example are small, but the same Spark queries can be used for large size data sets, without any modifications in the code.

To keep the discussion simple, we'll use the Spark Scala Shell.

First, let's look at how to install Spark on your local machine.

Pre-Requisites:

- You will need Java Development Kit (JDK) installed for Spark to work locally. This is covered in Step 1 below.
- You will also need to install Spark software on your laptop. The instructions on how to do this are covered in the Step 2 below.

Note: These instructions are for Windows environment. If you are using a different operating system environment, you'll need to modify the system variables and directory paths to match your environment.

I. INSTALL JDK:

1) Download JDK from Oracle website. JDK version 1.7 is recommended.

Install JDK in a directory name without spaces. For Windows users, install JDK in a folder like c:\dev, not in "c:\Program Files". "Program Files" directory has a space in the name and this causes problems when software is installed in this folder.

NOTE: DO NOT INSTALL JDK or Spark Software (described in Step 2) in "c:\Program Files" directory.

2) After installing JDK, verify it was installed correctly by navigating to "bin" folder under JDK 1.7 directory and typing the following command:

```
java -version
```

If JDK is installed correctly, the above command would display the Java version.

II. INSTALL SPARK SOFTWARE:

Download the latest Spark version from Spark website. Latest version at the time of publication of this article is Spark 1.2. You can choose a specific Spark installation depending on the Hadoop version. I downloaded Spark for Hadoop 2.4 or later, and the file name is spark-1.2.0-bin-hadoop2.4.tgz.

Unzip the installation file to a local directory (For example, c:\dev).

Word Count Application

Once you have Spark installed and have it up and running, you can run the **data analytics** queries using Spark API.

These are simple commands to read the data from a text file and process it. We'll look at advanced use cases of using Spark framework in the future articles in this series.

First, let's use Spark API to run the popular Word Count example. Open a new Spark Scala Shell if you don't already have it running. Here are the commands for this example.

```
import org.apache.spark.SparkContext
import org.apache.spark.SparkContext._
```

```
val txtFile = "README.md"
val txtData = sc.textFile(txtFile)
txtData.cache()
```

We call the cache function to store the RDD created in the above step in the cache, so Spark doesn't have to compute it every time we use it for further data queries. Note that cache() is a lazy operation. Spark doesn't immediately store the data in memory when we call cache. It actually takes place when an action is called on an RDD.

Now, we can call the count function to see how many lines are there in the text file.

```
txtData.count()
```

Now, we can run the following commands to perform the word count. The count shows up next to each word in the text file.

```
val wcData = txtData.flatMap(l => l.split(" ")).map(word => (word, 1)).reduceByKey(_ + _)
```

```
wcData.collect().foreach(println)
```

Lesson Proper for Week 4

What is Python?

Python: Dynamic programming language which supports several different programming paradigms:

Procedural programming

Object oriented programming

Functional programming

Standard: Python byte code is executed in the Python interpreter (similar to Java)
→ platform independent code

Why Python?

Extremely versatile language

Website development, data analysis, server maintenance, numerical analysis, ...

Syntax is clear, easy to read and learn (almost pseudo code)

Common language

Intuitive object oriented programming

Full modularity, hierarchical packages

Comprehensive standard library for many tasks

Big community

Simply extendable via C/C++, wrapping of C/C++ libraries

Focus: Programming speed

History

Start implementation in December 1989 by Guido van Rossum (CWI)

16.10.2000: Python 2.0

Unicode support

Garbage collector

Development process more community oriented

3.12.2008: Python 3.0

Not 100% backwards compatible

2007 & 2010 most popular programming language (TIOBE Index)

Recommendation for scientific programming (Nature News, NPG, 2015)

Current version: Python 3.9.2

Python2 is out of support!1

Zen of Python

20 software principles that influence the design of Python:

- 1 Beautiful is better than ugly.
- 2 Explicit is better than implicit.
- 3 Simple is better than complex.
- 4 Complex is better than complicated.
- 5 Flat is better than nested.
- 6 Sparse is better than dense.
- 7 Readability counts.
- 8 Special cases aren't special enough to break the rules.
- 9 Although practicality beats purity.
- 10 Errors should never pass silently.
- 11 Unless explicitly silenced.
- 12 ...

Is Python fast enough?

For user programs: Python is fast enough!

Most parts of Python are written in C

For compute intensive algorithms: Fortran, C, C++ might be better

Performance-critical parts can be re-implemented in C/C++ if necessary

First analyse, then optimise!

Jupyter Notebook

Jupyter is one of the powerful tools for development. However, it doesn't support Spark development implicitly. A lot of times Python developers are forced to use Scala for developing codes in Spark. This module aims to simplify that and enable the users to

use the Jupyter itself for developing Spark codes with the help of PySpark. Kindly follow the below steps to get this implemented and enjoy the power of Spark from the comfort of Jupyter. This exercise approximately takes 30 minutes.

Notebook document

Notebook documents (or “notebooks”, all lower case) are documents produced by the Jupyter Notebook App, which contain both computer code (e.g. python) and rich text elements (paragraph, equations, figures, links, etc...). Notebook documents are both human-readable documents containing the analysis description and the results (figures, tables, etc..) as well as executable documents which can be run to perform data analysis.

Jupyter Notebook App

The Jupyter Notebook App is a server-client application that allows editing and running notebook documents via a web browser. The Jupyter Notebook App can be executed on a local desktop requiring no internet access (as described in this document) or can be installed on a remote server and accessed through the internet.

In addition to displaying/editing/running notebook documents, the Jupyter Notebook App has a “Dashboard” (Notebook Dashboard), a “control panel” showing local files and allowing to open notebook documents or shutting down their kernels.

kernel

A notebook kernel is a “computational engine” that executes the code contained in a Notebook document. The ipython kernel, referenced in this guide, executes python code. Kernels for many other languages exist (official kernels).

When you open a Notebook document, the associated kernel is automatically launched. When the notebook is executed (either cell-by-cell or with menu Cell -> Run All), the kernel performs the computation and produces the results. Depending on the type of computations, the kernel may consume significant CPU and RAM. Note that the RAM is not released until the kernel is shut-down.

Notebook Dashboard

The Notebook Dashboard is the component which is shown first when you launch the Jupyter Notebook App. The Notebook Dashboard is mainly used to open notebook documents, and to manage the running kernels (visualize and shutdown).

The Notebook Dashboard has other features similar to a file manager, namely navigating folders and renaming/deleting files.

Installation of Pyspark (All operating systems)

This tutorial will demonstrate the installation of Pyspark and how to manage the environment variables in Windows, Linux, and Mac Operating System.

Pyspark = Python + Apache Spark

Apache Spark is a new and open-source framework used in the big data industry for real-time processing and batch processing. It supports different languages, like Python, Scala, Java, and R.

Apache Spark is initially written in a Java Virtual Machine(JVM) language called Scala, whereas Pyspark is like a Python API which contains a library called Py4J. This allows dynamic interaction with JVM objects.

Windows Installation

The installation which is going to be shown is for the Windows Operating System. It consists of the installation of Java with the environment variable and Apache Spark with the environment variable.

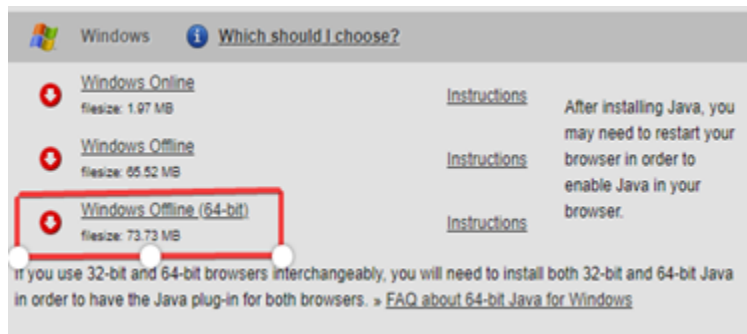
The recommended pre-requisite installation is Python, which is done from here.

Java installation

1. Go to Download Java JDK.

Visit Oracle's website for the download of the Java Development Kit(JDK).

2. Move to the download section consisting of the operating system Windows, and in my case, it's Windows Offline(64-bit). The installer file will be downloaded.

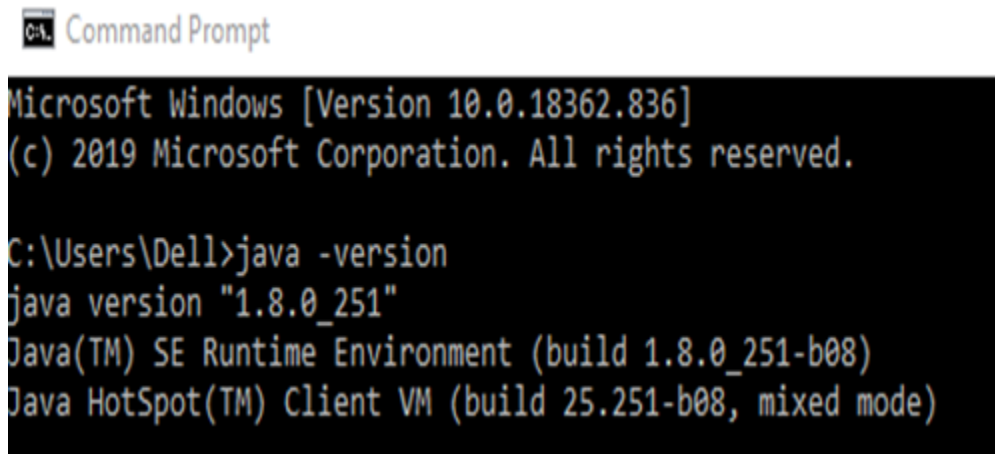


3. Open the installer file, and the download begins.

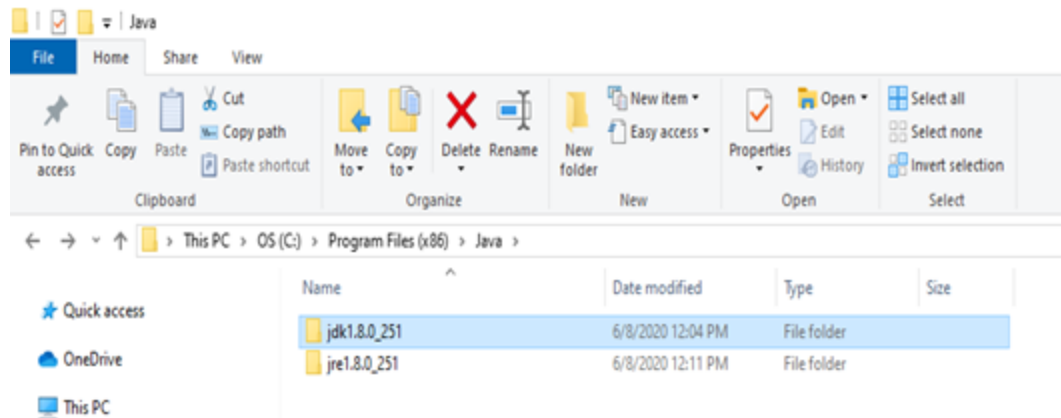


4.

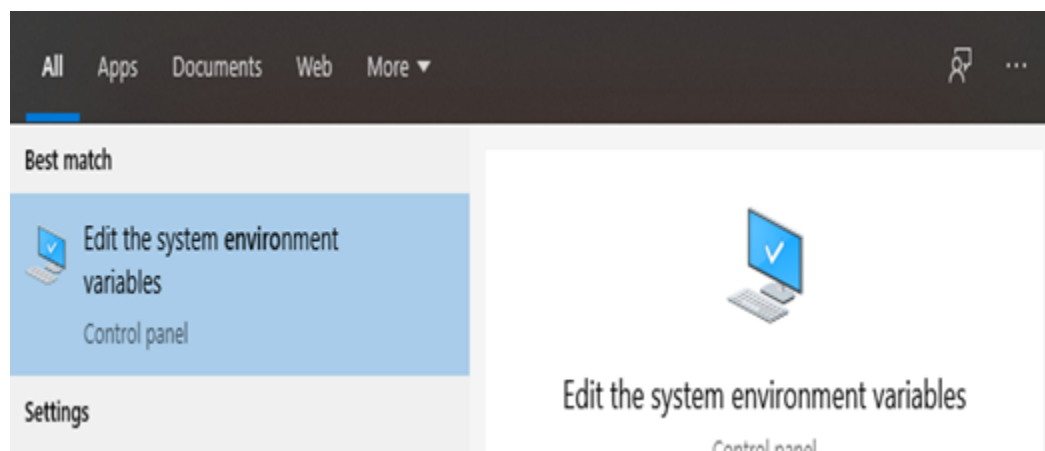
5. Go to "Command Prompt" and type "java -version" to know the version and know whether it is installed or not.



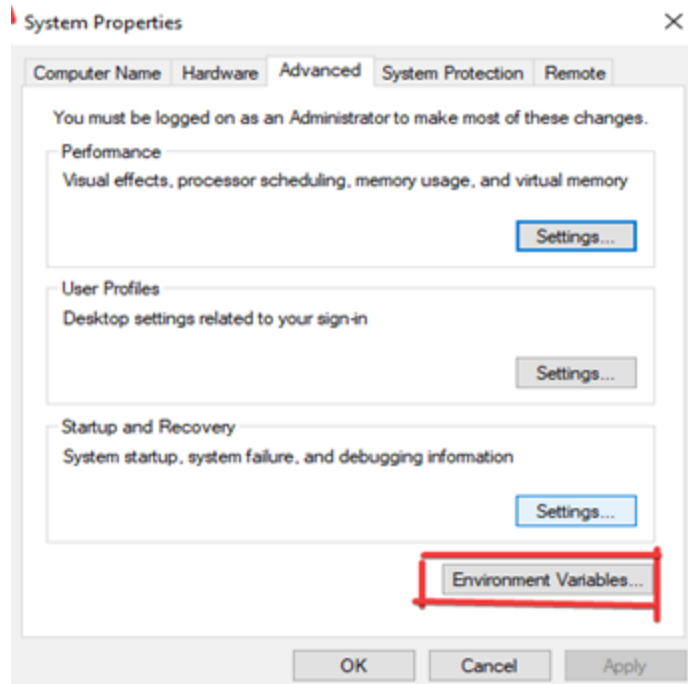
6. Add the Java path



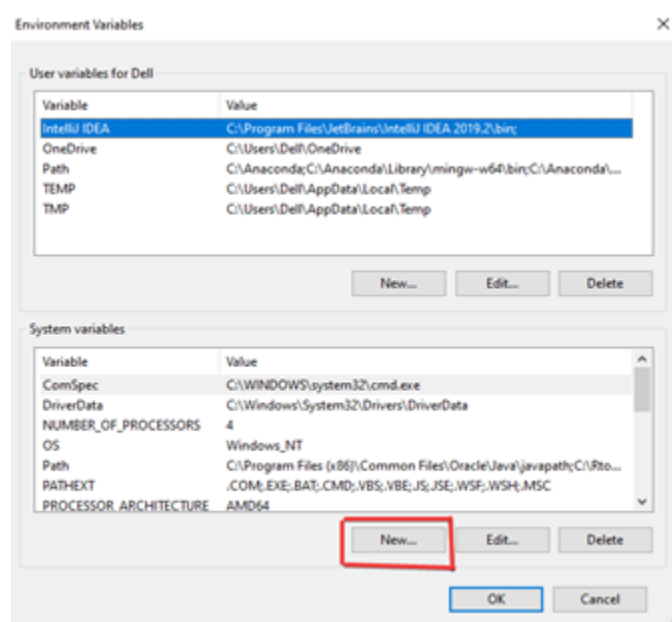
7. Go to the search bar and "EDIT THE ENVIRONMENT VARIABLES.



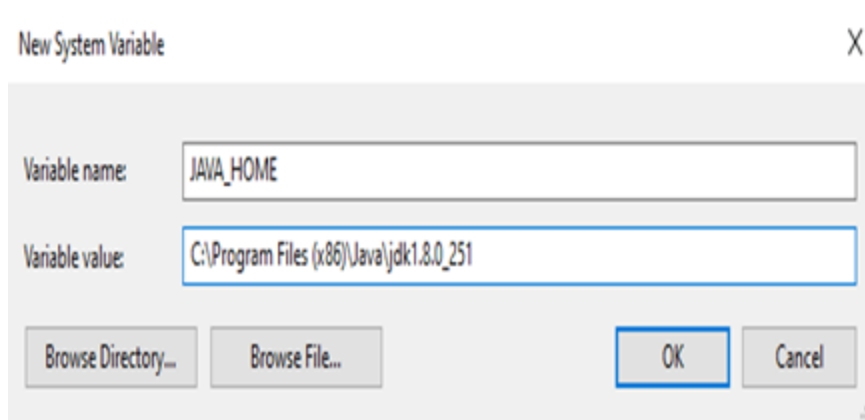
8. Click into the "Environment Variables"



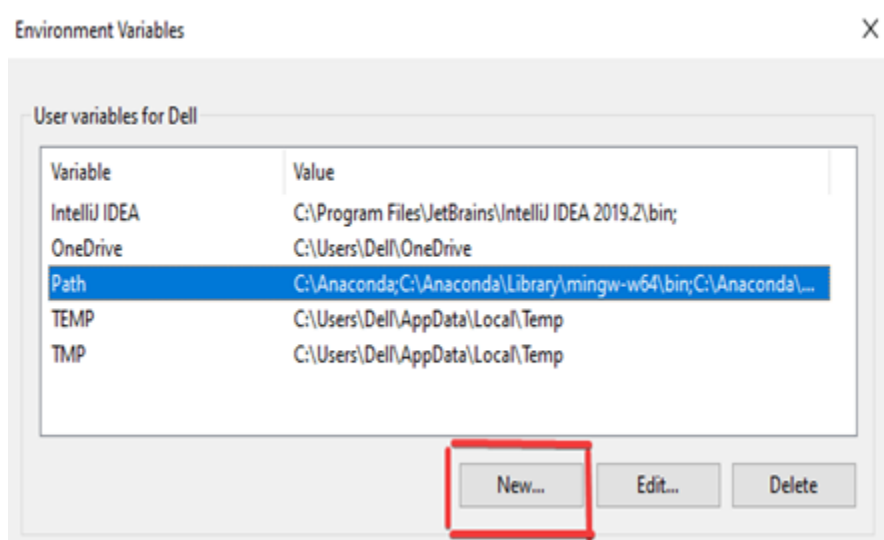
9. Click into "New" to create your new Environment variable



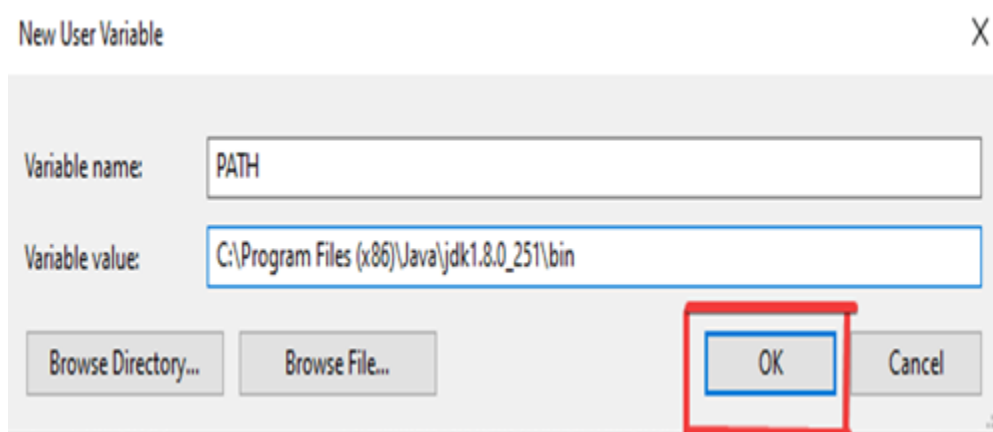
10. Use Variable Name as "JAVA_HOME" and your Variable Value as 'C:\Program Files (x86)\Java\jdk1.8.0_251'. This is your location of the Java file. Click 'OK' after you've finished the process.



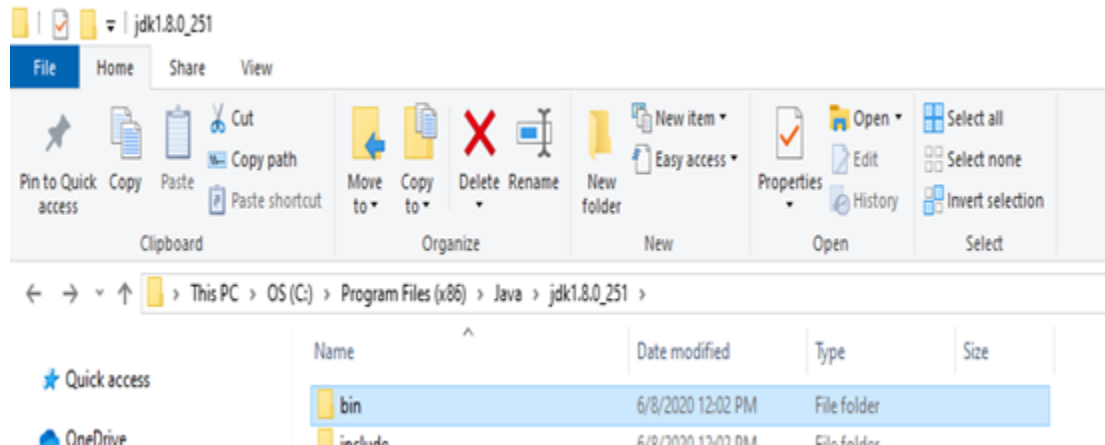
11. Let's add the User variable and select 'Path' and click 'New' to create it.



12. Add the Variable name as 'PATH' and path value as 'C:\Program Files (x86)\Java\jdk1.8.0_251\bin', which is your location of Java bin file. Click 'OK' after you've finished the process.



Note: You can locate your Java file by going to C drive, which is C:\Program Files (x86)\Java\jdk1.8.0_251' if you've not changed location during the download



Installing Pyspark

1. Head over to the Spark homepage.
2. Select the Spark release and package type as following and download the **.tgz file**.



You can make a new folder called 'spark' in the C directory and extract the given file by using 'Winrar', which will be helpful afterward.

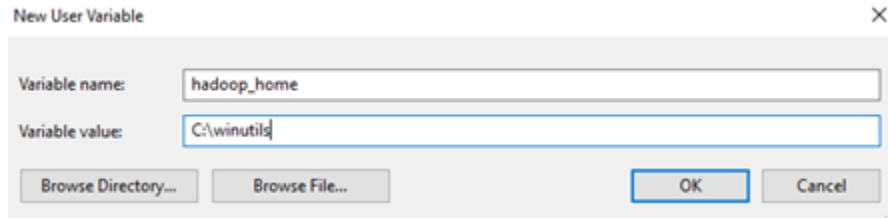
Download and setup winutils.exe

Go to Winutils choose your previously downloaded Hadoop version, then download the winutils.exe file by going inside 'bin'.

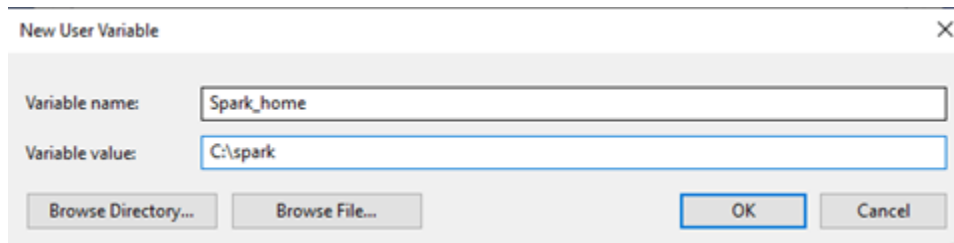
Make a new folder called 'winutils' and inside of it create again a new folder called 'bin'. Then put the file recently download 'winutils' inside it.

Environment variables

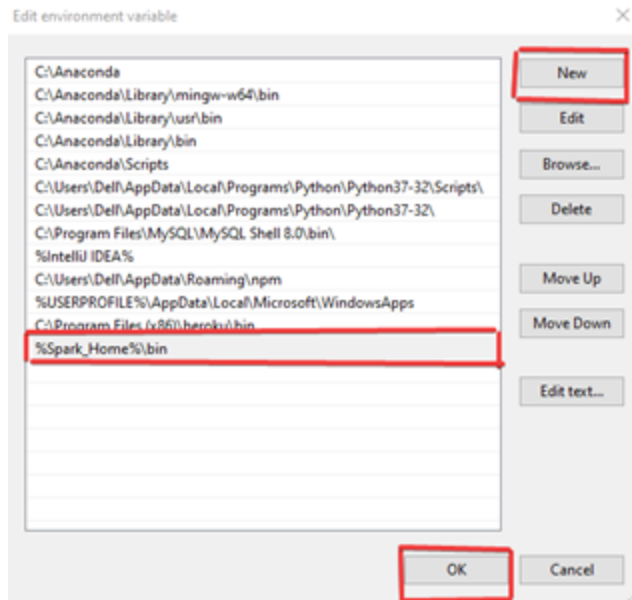
1. Let's create a new environment where variable name as "hadoop_home" and variable value to be the location of winutils, which is "C:\winutils" and click "OK".



2. For spark, also let's create a new environment where the variable name is "Spark_home" and the variable value to be the location of spark, which is "C:\spark" and click "OK".



3. Finally, double click the 'path' and change the following as done below where a new path is created "%Spark_Home%\bin" is added and click "OK".



Finalizing Pyspark Installation

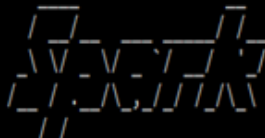
- 1. Open Command Prompt and type the following command.**

```
Microsoft Windows [Version 10.0.18362.900]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Dell>pyspark
```

- 2. Once everything is successfully done, the following message is obtained.**

```
Type "help", "copyright", "credits" or "license" for more information.  
20/06/17 17:36:48 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable  
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties  
Setting default log level to "WARN".  
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).  
Welcome to
```



```
 version 2.4.6  
  
Using Python version 3.7.4 (default, Aug 9 2019 18:34:13)  
SparkSession available as 'spark'.  
>>>
```

Lesson Proper for Week 5

Introduction to Spark DataFrame

A spark data frame can be said to be a distributed data collection that is organized into named columns and is also used to provide the operations such as filtering, computation of aggregations, grouping and also can be used with Spark SQL. **Data frames** can be created by making use of structured data files, along with existing RDDs, external databases, and Hive tables. It is basically termed and known as an abstraction layer which is built on top of RDD and is also followed by the dataset API, which was introduced in later versions of Spark (2.0 +). Moreover, the datasets were not introduced in Pyspark but only in Scala with Spark, but this was not the case in the case of Dataframes. Data frames, popularly known as DFs, are logical columnar formats that make working with RDDs easier and more convenient, also making use of the same functions as RDDs in the same way. If you talk more on the conceptual level, it is equivalent to the relational tables along with good optimization features and techniques.

How to Create a DataFrame?

A Data Frame is generally created by any one of the mentioned methods. It can be created by making use of Hive tables, external databases, Structured data files or even in the case of existing RDDs. These all ways can create these named columns known as Dataframes used for the processing in Apache Spark. By making use of SQLContext or SparkSession, applications can be used to create Dataframes.

Dataframe basics for PySpark

Spark has moved to a dataframe API since version 2.0. A dataframe in Spark is similar to a SQL table, an R dataframe, or a pandas dataframe. In Spark, dataframe is actually a wrapper around RDDs, the basic data structure in Spark. In my opinion, however, working with dataframes is easier than RDD most of the time.

There are a few ways to read data into Spark as a dataframe. In this post, I will load the first few rows of Titanic data on Kaggle into a pandas dataframe, then convert it into a Spark dataframe.

```
import findspark
```

```
findspark.init()
```

```
import pyspark # only run after findspark.init()
```

```
from pyspark.sql import SparkSession
```

```
spark = SparkSession.builder.getOrCreate()
```

```
import pandas as pd
```

```
sc = spark.sparkContext
```

Make a sample dataframe from Titanic data

```
data1 = {'PassengerId': {0: 1, 1: 2, 2: 3, 3: 4, 4: 5},
```

```
        'Name': {0: 'Owen', 1: 'Florence', 2: 'Laina', 3: 'Lily', 4: 'William'},
```

```
        'Sex': {0: 'male', 1: 'female', 2: 'female', 3: 'female', 4: 'male'},
```

```
        'Survived': {0: 0, 1: 1, 2: 1, 3: 1, 4: 0}}
```

```
data2 = {'PassengerId': {0: 1, 1: 2, 2: 3, 3: 4, 4: 5},
```

```
        'Age': {0: 22, 1: 38, 2: 26, 3: 35, 4: 35},
```

```
        'Fare': {0: 7.3, 1: 71.3, 2: 7.9, 3: 53.1, 4: 8.0},
```

```
        'Pclass': {0: 3, 1: 1, 2: 3, 3: 1, 4: 3}}
```

```
df1_pd = pd.DataFrame(data1, columns=data1.keys())
```

```
df2_pd = pd.DataFrame(data2, columns=data2.keys())
```

```
df1_pd
```

	Passeng erId	Name	Sex	Surviv ed
0	1	Owen	male	0
1	2	Floren ce	fem ale	1
2	3	Laina	fem ale	1
3	4	Lily	fem ale	1

```
4          Willia
5          m      male 0
```

df2_pd

	PassengerId	Age	Fare	Class
0	1	22	7.3	3
1	2	38	71.3	1
2	3	26	7.9	3
3	4	35	53.1	1
4	5	35	8.0	3

Convert pandas dataframe to Spark dataframe

```
df1 = spark.createDataFrame(df1_pd)
```

```
df2 = spark.createDataFrame(df2_pd)
```

```
df1.show()
```

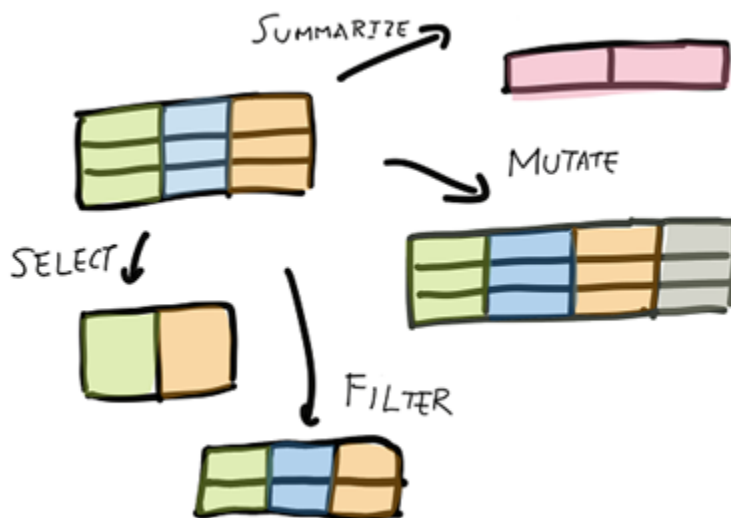
```
+-----+-----+-----+-----+
|PassengerId|      Name|  Sex|Survived|
+-----+-----+-----+-----+
|      1|    Owen| male|      0|
|      2|Florence|female|      1|
|      3|  Laina|female|      1|
|      4|    Lily|female|      1|
|      5|William| male|      0|
+-----+-----+-----+-----+
```

```
df1.printSchema()

root
 |-- PassengerId: long (nullable = true)
 |-- Name: string (nullable = true)
 |-- Sex: string (nullable = true)
 |-- Survived: long (nullable = true)
```

Basic dataframe verbs

In R's dplyr package, **Hadley Wickham** defined the 5 basic verbs — **select**, **filter**, **mutate**, **summarize**, and **arrange**. Here are the equivalents of the 5 basic verbs for Spark dataframes.



Select

I can select a subset of columns. The method `select()` takes either a list of column names or an unpacked list of names.

```
cols1 = ['PassengerId', 'Name']
```

```
df1.select(cols1).show()
```

```
+-----+-----+
```

```
| PassengerId | Name |
```

```
+-----+-----+
```

```
|  
1 | Owen |
```

```
|  
2 | Florence |
```

```
|  
3 | Laina |
```

```
|  
4 | Lily |
```

```
|  
5 | William |
```

```
+-----+-----+
```

Filter

I can filter a subset of rows. `The method filter()` takes **column expressions or SQL expressions**. Think of the WHERE clause in SQL queries.

- Filter with a column expression

```
df1.filter(df1.Sex == 'female').show()
```

```
+-----+-----+-----+-----+
```

```
| PassengerId | Name |  
Sex | Survived |
```

```
+-----+-----+-----+-----+
```

```
|  
2 | Florence | female | 1 |
```

```
|  
3 | Laina | female | 1 |
```

```
|  
4 | Lily | female | 1 |
```

```
+-----+-----+-----+-----+
```

- Filter with a SQL expression. Note the double and single quotes as I'm passing a SQL where clause into filter().

```
df1.filter("Sex='female']").show()
```

```
+-----+-----+-----+-----+
```

```
|PassengerId|      Name| Sex|Survived|
```

```
+-----+-----+-----+-----+
```

```
|      2|Florence|female|      1|
```

```
|      3|  Laina|female|      1|
```

```
|      4|    Lily|female|      1|
```

```
+-----+-----+-----+-----+
```

Mutate, or creating new columns

I can create new columns in Spark using `.withColumn()`. I have yet found a convenient way to create multiple columns at once without chaining multiple `.withColumn()` methods.

```
df2.withColumn('AgeTimesFare', df2.Age*df2.Fare).show()
```

```
+-----+---+----+-----+-----+
```

```
|PassengerId|Age|Fare|Pclass|AgeTimesFare|
```

```
+-----+---+----+-----+-----+
```

```
|      1| 22| 7.3|      3|      160.6|
```

```
|      2| 38|71.3|      1|     2709.4|
```

```
|      3| 26| 7.9|      3|      205.4|
```

```
|      4| 35|53.1|      1|     1858.5|
```

```
|      5| 35| 8.0|      3|      280.0|
```

```
+-----+---+----+-----+-----+
```

Summarize and group by

To summarize or aggregate a dataframe, first I need to convert the dataframe to a GroupedData object with `groupby()`, then call the aggregate functions.

```
gdf2 = df2.groupby('Pclass')
```

```
gdf2
```

```
<pyspark.sql.group.GroupedData at 0x9bc8f28>
```

I can take the average of columns by passing an *unpacked* list of column names.

```
avg_cols = ['Age', 'Fare']
```

```
gdf2.avg(*avg_cols).show()
```

```
+-----+-----+-----+
|Pclass|  avg(Age)|  avg(Fare)|
+-----+-----+-----+
|      1|      36.5|      62.2|
|      3|27.666666666666668|7.733333333333333|
+-----+-----+-----+
```

To call multiple aggregation functions at once, pass a dictionary.

```
gdf2.agg({'*': 'count', 'Age': 'avg', 'Fare': 'sum'}).show()
```

```
+-----+-----+-----+-----+
|Pclass|count(1)|  avg(Age)|sum(Fare)|
+-----+-----+-----+-----+
|      1|      2|      36.5|  124.4|
|      3|      3|27.666666666666668|  23.2|
+-----+-----+-----+-----+
```

To rename the columns `count(1)`, `avg(Age)` etc, use `toDF()`.

```
(
    gdf2
    .agg({'*': 'count', 'Age': 'avg', 'Fare': 'sum'})
    .toDF('Pclass', 'counts', 'average_age', 'total_fare')
```



```

        .show()
)
+-----+-----+-----+-----+
|Pclass|counts|    average_age|total_fare|
+-----+-----+-----+-----+
|      1|      2|           36.5| 124.4|
|      3|      3|27.666666666666668| 23.2|
+-----+-----+-----+-----+

```

Arrange (sort)

Use the `sort()` method to sort the dataframes. I haven't seen a good use case for this in Spark though.

```
df2.sort('Fare', ascending=False).show()
```

```

+-----+---+----+-----+
|PassengerId|Age|Fare|Pclass|
+-----+---+----+-----+
|      2| 38|71.3|    1|
|      4| 35|53.1|    1|
|      5| 35| 8.0|    3|
|      3| 26| 7.9|    3|
|      1| 22| 7.3|    3|
+-----+---+----+-----+

```

Joins and unions

There are two ways to combine data frames — joins and unions. The idea here is the same as joining and unioning tables in SQL.

Joins

For example, I can join the two titanic dataframes by the column PassengerId

```
df1.join(df2, ['PassengerId']).show()
```

```

+-----+-----+-----+-----+---+----+-----+

```

PassengerId	Name	Sex	Survived	Age	Fare	Pclass
-------------	------	-----	----------	-----	------	--------

5	William	male	0	35	8.0	3
1	Owen	male	0	22	7.3	3
3	Laina	female	1	26	7.9	3
2	Florence	female	1	38	71.3	1
4	Lily	female	1	35	53.1	1

I can also join by conditions, but it creates duplicate column names if the keys have the same name, which is frustrating. For now, the only way I know to avoid this is to pass a list of join keys as in the previous cell. If I want to make nonequi joins, then I need to rename the keys before I join.

Nonequi joins

Here is an example of nonequi join. They can be *very* slow due to skewed data, but this is one thing that Spark can do that Hive can not.

```
df1.join(df2, df1.PassengerId <= df2.PassengerId).show() # Note the duplicate col names
```

PassengerId	Name	Sex	Survived	PassengerId	Age	Fare	Pclass
1	Owen	male	0	1	22	7.3	3
1	Owen	male	0	2	38	71.3	1
1	Owen	male	0	3	26	7.9	3
1	Owen	male	0	4	35	53.1	1
1	Owen	male	0	5	35	8.0	3
2	Florence	female	1	2	38	71.3	1
2	Florence	female	1	3	26	7.9	3
2	Florence	female	1	4	35	53.1	1

	2	Florence female	1	5	35	8.0	3	
	3	Laina female	1	3	26	7.9	3	
	3	Laina female	1	4	35	53.1	1	
	3	Laina female	1	5	35	8.0	3	
	4	Lily female	1	4	35	53.1	1	
	4	Lily female	1	5	35	8.0	3	
	5	William	male	0	5	35	8.0	3

```

+-----+-----+-----+-----+-----+---+---+-----+

```

Unions

Union() returns a dataframe from the union of two dataframes

```
df1.union(df1).show()
```

```

+-----+-----+-----+-----+
|PassengerId|      Name|  Sex|Survived|
+-----+-----+-----+-----+
|      1|      Owen| male|      0|
|      2|Florence|female|      1|
|      3|  Laina|female|      1|
|      4|      Lily|female|      1|
|      5|William| male|      0|
|      1|      Owen| male|      0|
|      2|Florence|female|      1|
|      3|  Laina|female|      1|
|      4|      Lily|female|      1|
|      5|William| male|      0|
+-----+-----+-----+-----+

```

Some of my iterative algorithms create chained union() objects. There is a potential catch that the execution plan may grow too long, which cause performance problems or errors.

One common symptom of performance issues caused by chained unions in a for loop is it took longer and longer to iterate through the loop. In this case, `repartition()` and `checkpoint()` may help solving this problem.

Dataframe input and output (I/O)

There are two classes `pyspark.sql.DataFrameReader` and `pyspark.sql.DataFrameWriter` that handles dataframe I/O. Depending on the configuration, the files may be saved locally, through a Hive metastore, or to a Hadoop file system (HDFS).

Common methods on saving dataframes to files include `saveAsTable()` for Hive tables and `saveAsFile()` for local or Hadoop file system.

I will refer to the documentation for examples on how to read and write dataframes for different formats.

- [DataFrameReader documentation](#)
- [DataFrameWriter documentation](#)
- [Source code for reader and writer](#)

The spark.sql API

Many of the operations that I showed can be accessed by writing SQL (Hive) queries in `spark.sql()`. This is also a convenient way to read Hive tables into Spark dataframes. To make an existing Spark dataframe usable for `spark.sql()`, I need to register said dataframe as a temporary table.

Temp tables

As an example, I can register the two dataframes as temp tables then join them through `spark.sql()`.

```
df1.createOrReplaceTempView('df1_temp')
```

```
df2.createOrReplaceTempView('df2_temp')
```

```
query = ""
```

```
    select
```

```
    a.PassengerId,
```

```
    a.Name,
```

```
    a.Sex,
```

```
    a.Survived,
```

```

    b.Age,
    b.Fare,
    b.Pclass
from df1_temp a
join df2_temp b
    on a.PassengerId = b.PassengerId""
dfj = spark.sql(query)
dfj.show()
+-----+-----+-----+-----+---+---+-----+
|PassengerId|      Name|  Sex|Survived|Age|Fare|Pclass|
+-----+-----+-----+-----+---+---+-----+
|      5| William| male|    0| 35| 8.0|    3|
|      1|    Owen| male|    0| 22| 7.3|    3|
|      3|  Laina|female|    1| 26| 7.9|    3|
|      2|Florence|female|    1| 38|71.3|    1|
|      4|    Lily|female|    1| 35|53.1|    1|
+-----+-----+-----+-----+---+---+-----+

```

Spark DataFrames Operations

In Spark, **a data frame** is the **distribution and collection of an organized form of data** into named columns which is equivalent to a relational database or a schema or a data frame in a language such as **R or python** but along with a richer level of optimizations to be used. It is used to provide a specific domain kind of language that could be used for structured data manipulation.

The below mentioned are some basic Operations of Structured Data Processing by making use of Dataframes.

1. Reading a document which is of type: JSON: We would be making use of the command `sqlContext.read.json`.

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")
```

Output: In this case, the output will be that the field names will be automatically taken from the file student.json.

2. Showing of Data: In order to see the data in the Spark data frames, you will need to use the command:

```
dfs.show()
```

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")
```

```
dfs.show()
```

Output: The student data will be present to you in a tabular format.

3. Using printSchema method: If you are interested to see the structure, i.e. schema of the data frame, then make use of the following command: dfs.printSchema()

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")
```

```
dfs. printSchema ()
```

Output: The structure or the schema will be present to you

4. Use the select method: In order to use the select method, the following command will be used to fetch the names and columns from the list of data frames.

```
dfs.select("column-name").show()
```

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")
```

```
dfs.select("name").show()
```

Output: The values of the name column can be seen.

5. Using Age filter: The following command can be used to find the range of students whose age is more than 23 years.

```
dfs.filter(dfs("column-name") > value).show()
```

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")  
dfs.filter(dfs("age")>23).show()
```

Output: The filtered age for greater than 23 will appear in the results.

6. Using the groupBy method: The following method could be used to count the number of students who have the same age.

```
dfs.groupBy("column-name").count().show()
```

Example: Let us suppose our filename is student.json, then our piece of code will look like:

```
val dfs= sqlContext.read.json("student.json")  
dfs.groupBy("age").count().show()
```

7. Using SQL function upon a SparkSession: It enables the application to execute SQL type queries programmatically and hence returns the result in the form of a data frame.

```
spark.sql(query)
```

Example: Suppose we have to register the SQL data frame as a temp view then:

```
df.createOrReplaceTempView("student")  
sqlDF=spark.sql("select * from student")  
sqlDF.show()
```

Output: A temporary view will be created by the name of the student, and a spark.sql will be applied on top of it to convert it into a data frame.

8. Using SQL function upon a Spark Session for Global temporary view: This enables the application to execute SQL type queries programmatically and hence returns the result in the form of a data frame.

```
spark.sql(query)
```

Example: Suppose we have to register the SQL data frame as a temp view then:

```
df.createGlobalTempView("student")  
park.sql("select * from global_temp.student").show()
```

```
spark.newSession().sql("Select * from global_temp.student").show()
```

Output: A temporary view will be created by the name of the student, and a spark.sql will be applied on top of it to convert it into a data frame.

Advantages of Spark DataFrame

1. The data frame is the Data's distributed collection, and therefore the data is organized in named column fashion.
2. They are more or less similar to the table in the case of relational databases and have a rich set of optimization.
3. Dataframes are used to empower the queries written in SQL and also the data frame API
4. It can be used to process both structured as well as unstructured kinds of data.
5. The use of a catalyst optimizer makes optimization easy and effective.
6. The libraries are present in many languages such as Python, Scala, Java, and R.
7. This is used to provide strong compatibility with Hive and is used to run unmodified Hive queries on the already present hive warehouse.
8. It can scale very well, right from a few kbs on the personal system to many petabytes on the large clusters.
9. It is used to provide an easy level of integration with other big data technologies and frameworks.
10. The abstraction which they provide to RDDs is efficient and makes processing faster.

MIDTERM LESSON

Lesson Proper for Week 7

What is PySpark?

PySpark is the Python API written in python to support Apache Spark. **Apache Spark** is a **distributed framework that can handle Big Data analysis**. Apache Spark is written in Scala and can be integrated with Python, Scala, Java, R, SQL languages. Spark is basically a computational engine that works with huge sets of data by processing them in parallel and batch systems.

Who can learn PySpark?

Python is becoming a powerful language in the field of data science and machine learning. Through its library Py4j, one will be able to work with Spark using python. Python is a language that is widely used in machine learning and data science. Python supports parallel computing.

The prerequisites are

- Programming knowledge using python
- Big data knowledge and framework such as Spark

Spark Context

SparkContext is the **internal engine** that allows the connections with the clusters. If you want to run an operation, you need a SparkContext.

SparkContext is the entry point to any spark functionality. When we run any Spark application, a driver program starts, which has the main function and your SparkContext gets initiated here. The driver program then runs the operations inside the executors on worker nodes.

SparkContext uses Py4J to launch a JVM and creates a JavaSparkContext. By default, PySpark has SparkContext available as 'sc', so creating a new SparkContext won't work.

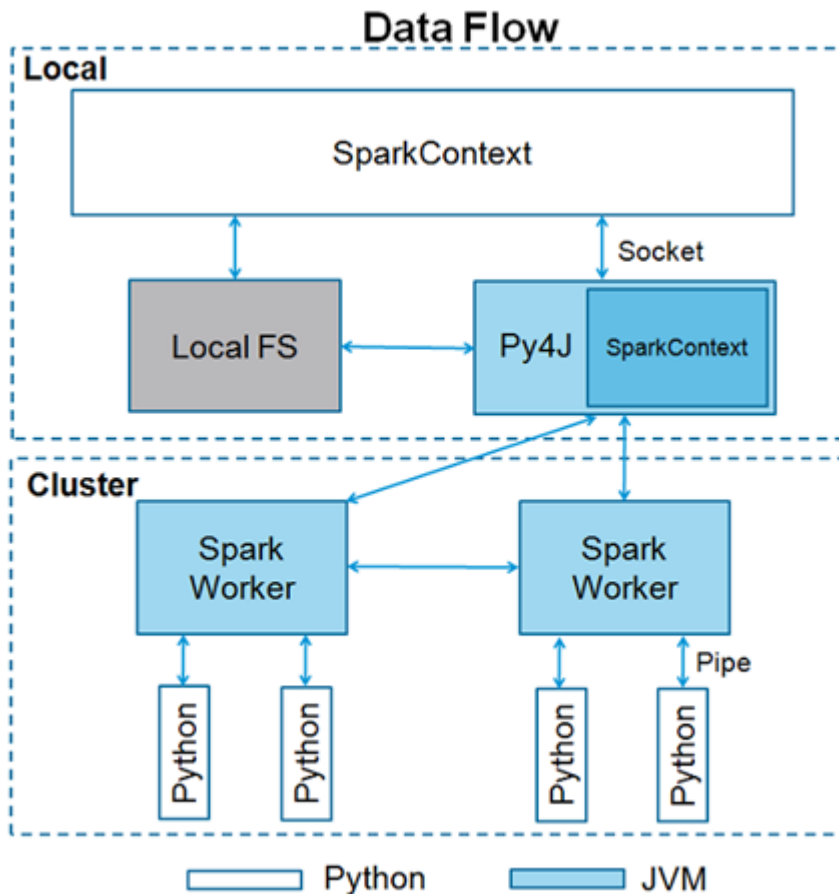
The following code block has the details of a PySpark class and the parameters, which a SparkContext can take.

```
class pyspark.SparkContext (  
    master = None,  
    appName = None,  
    sparkHome = None,  
    pyFiles = None,  
    environment = None,  
    batchSize = 0,
```

```

serializer = PickleSerializer(),
conf = None,
gateway = None,
jsc = None,
profiler_cls = <class 'pyspark.profiler.BasicProfiler'>
)

```



Parameters

Following are the parameters of a SparkContext.

- **Master** – It is the URL of the cluster it connects to.
- **appName** – Name of your job.
- **sparkHome** – Spark installation directory.
- **pyFiles** – The .zip or .py files to send to the cluster and add to the PYTHONPATH.
- **Environment** – Worker nodes environment variables.

- **batchSize** – The number of Python objects represented as a single Java object. Set 1 to disable batching, 0 to automatically choose the batch size based on object sizes, or -1 to use an unlimited batch size.
- **Serializer** – RDD serializer.
- **Conf** – An object of L{SparkConf} to set all the Spark properties.
- **Gateway** – Use an existing gateway and JVM, otherwise initializing a new JVM.
- **JSC** – The JavaSparkContext instance.
- **profiler_cls** – A class of custom Profiler used to do profiling (the default is pyspark.profiler.BasicProfiler).

SQLContext

A more convenient way is to use the DataFrame. SparkContext is already set, you can use it to create the DataFrame. You also need to declare the SQLContext

SQLContext allows connecting the engine with different data sources. It is used to initiate the functionalities of Spark SQL.

```
from pyspark.sql import Row
from pyspark.sql import SQLContext
```

```
sqlContext = SQLContext(sc)
```

Now in this Spark tutorial Python, let's create a list of tuple. Each tuple will contain the name of the people and their age. Four steps are required:

Step 1) Create the list of tuple with the information

```
[('John',19),('Smith',29),('Adam',35),('Henry',50)]
```

Step 2) Build a RDD

```
rdd = sc.parallelize(list_p)
```

Step 3) Convert the tuples

```
rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
```

Step 4) Create a DataFrame context

```
sqlContext.createDataFrame(ppl)
```

```
list_p = [('John',19),('Smith',29),('Adam',35),('Henry',50)]
```

```
rdd = sc.parallelize(list_p)
```

```
ppl = rdd.map(lambda x: Row(name=x[0], age=int(x[1])))
```

```
DF_ppl = sqlContext.createDataFrame(ppl)
```

If you want to access the type of each feature, you can use printSchema()

```
DF_ppl.printSchema()
```

```
root
```

```
-- age: long (nullable = true)
-- name: string (nullable = true)
```

Machine Learning Example with PySpark

Now that you have a brief idea of Spark and SQLContext, you are ready to build your first Machine learning program.

Following are the steps to build a Machine Learning program with PySpark:

- **Step 1)** Basic operation with PySpark
- **Step 2)** Data preprocessing
- **Step 3)** Build a data processing pipeline
- **Step 4)** Build the classifier: logistic
- **Step 5)** Train and evaluate the model
- **Step 6)** Tune the hyperparameter

In this PySpark Machine Learning tutorial, we will use the adult dataset. The purpose of this tutorial is to learn how to use Pyspark. For more information about the dataset, refer to this tutorial.

Note that, the dataset is not significant and you may think that the computation takes a long time. Spark is designed to process a considerable amount of data. Spark's performances increase relative to other machine learning libraries when the dataset processed grows larger.

Step 1) Basic operation with PySpark

First of all, you need to initialize the SQLContext is not already in initiated yet.

```
#from pyspark.sql import SQLContext
```

```
url =
```

```
"https://raw.githubusercontent.com/guru99-edu/R-Programming/master/adult_data.csv"
```

```
from pyspark import SparkFiles
```

```
sc.addFile(url)
```

```
sqlContext = SQLContext(sc)
```

then, you can read the cvs file with sqlContext.read.csv. You use inferSchema set to True to tell Spark to guess automatically the type of data. By default, it is turn to False.

```
df = sqlContext.read.csv(SparkFiles.get("adult_data.csv"), header=True,
inferSchema= True)
```

Let's have a look at the data type

```
df.printSchema()
```

```
root
```

```
|-- age: integer (nullable = true)
|-- workclass: string (nullable = true)
|-- fnlwgt: integer (nullable = true)
|-- education: string (nullable = true)
|-- education_num: integer (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true)
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain: integer (nullable = true)
|-- capital_loss: integer (nullable = true)
|-- hours_week: integer (nullable = true)
|-- native_country: string (nullable = true)
|-- label: string (nullable = true)
```

You can see the data with show.

```
df.show(5, truncate = False)
```

```
+---+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
|age|workclass  |fnlwgt|education|education_num|marital      |occupation
      |relationship |race |sex
|capital_gain|capital_loss|hours_week|native_country|label|
+---+-----+-----+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+-----+-----+
|39 |State-gov      |77516 |Bachelors|13          |Never-married
|Adm-clerical    |Not-in-family|White|Male |2174      |0   |40
|United-States |<=50K|
|50 |Self-emp-not-inc|83311 |Bachelors|13
|Married-civ-spouse|Exec-managerial |Husband |White|Male |0   |0
|13   |United-States |<=50K|
|38 |Private        |215646|HS-grad |9          |Divorced
|Handlers-cleaners|Not-in-family|White|Male |0   |0   |40
|United-States |<=50K|
```

```

|53 |Private      |234721|11th   |7
|Married-civ-spouse|Handlers-cleaners|Husband |Black|Male |0      |0
|40  |United-States |<=50K|
|28 |Private      |338409|Bachelors|13
|Married-civ-spouse|Prof-specialty  |Wife      |Black|Female|0      |0
|40  |Cuba         |<=50K|
+---+-----+-----+-----+-----+-----+-----+-----+
-+---+-----+-----+-----+-----+-----+-----+

```

only showing top 5 rows

If you didn't set `inferSchema` to `True`, here is what is happening to the type. There are all in string.

```

df_string = sqlContext.read.csv(SparkFiles.get("adult.csv"), header=True,
inferSchema= False)
df_string.printSchema()
root

```

```

|-- age: string (nullable = true)
|-- workclass: string (nullable = true)
|-- fnlwgt: string (nullable = true)
|-- education: string (nullable = true)
|-- education_num: string (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true)
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain: string (nullable = true)
|-- capital_loss: string (nullable = true)
|-- hours_week: string (nullable = true)
|-- native_country: string (nullable = true)
|-- label: string (nullable = true)

```

To convert the continuous variable in the right format, you can use `recast` the columns. You can use `withColumn` to tell Spark which column to operate the transformation.

```

# Import all from `sql.types`
from pyspark.sql.types import *

```

Write a custom function to convert the data type of DataFrame columns

```

def convertColumn(df, names, newType):
    for name in names:
        df = df.withColumn(name, df[name].cast(newType))
    return df
# List of continuous features
CONTI_FEATURES = ['age', 'fnlwgt', 'capital_gain', 'education_num',
'capital_loss', 'hours_week']
# Convert the type
df_string = convertColumn(df_string, CONTI_FEATURES, FloatType())
# Check the dataset
df_string.printSchema()
root
|-- age: float (nullable = true)
|-- workclass: string (nullable = true)
|-- fnlwgt: float (nullable = true)
|-- education: string (nullable = true)
|-- education_num: float (nullable = true)
|-- marital: string (nullable = true)
|-- occupation: string (nullable = true)
|-- relationship: string (nullable = true)
|-- race: string (nullable = true)
|-- sex: string (nullable = true)
|-- capital_gain: float (nullable = true)
|-- capital_loss: float (nullable = true)
|-- hours_week: float (nullable = true)
|-- native_country: string (nullable = true)
|-- label: string (nullable = true)

from pyspark.ml.feature import StringIndexer
#stringIndexer = StringIndexer(inputCol="label", outputCol="newlabel")
#model = stringIndexer.fit(df)
#df = model.transform(df)
df.printSchema()

```

Select columns

You can select and show the rows with select and the names of the features.
Below, age and fnlwgt are selected.

```
df.select('age','fnlwgt').show(5)
```

```
+---+-----+
|age|fnlwgt|
+---+-----+
| 39| 77516|
| 50| 83311|
| 38|215646|
| 53|234721|
| 28|338409|
+---+-----+
```

only showing top 5 rows

Count by group

If you want to count the number of occurrence by group, you can chain:

- groupBy()
- count()

together. In the PySpark example below, you count the number of rows by the education level.

```
df.groupBy("education").count().sort("count",ascending=True).show()
```

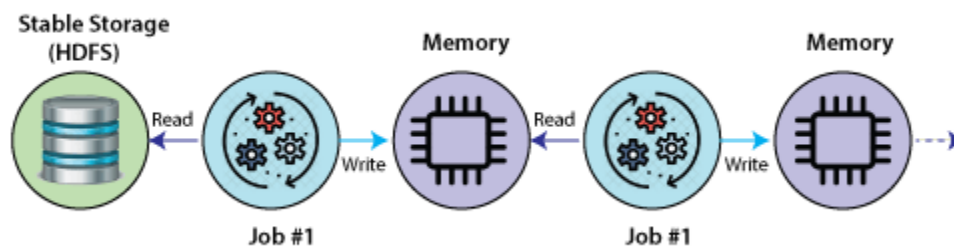
```
+-----+-----+
| education|count|
+-----+-----+
|  Preschool|  51|
|    1st-4th| 168|
|    5th-6th| 333|
|  Doctorate| 413|
|    12th| 433|
|    9th| 514|
| Prof-school| 576|
|    7th-8th| 646|
|    10th| 933|
| Assoc-acdm|1067|
|    11th|1175|
| Assoc-voc|1382|
|    Masters|1723|
```


Bachelors	5355
Some-college	7291
HS-grad	10501

Lesson Proper for Week 8

What are PySpark RDDs?

RDDs are most essential part of the PySpark or we can say backbone of PySpark. It is one of the fundamental schema-less data structures, that can handle both structured and unstructured data. It makes in-memory data sharing 10 - 100x faster in comparison of network and disk sharing.



"Resilient Distributed Datasets (RDD) is a distributed memory abstraction that helps a programmer to perform in-memory computations on large cluster." One of the important advantages of RDD is fault tolerance, it means if any failure occurs it recovers automatically.

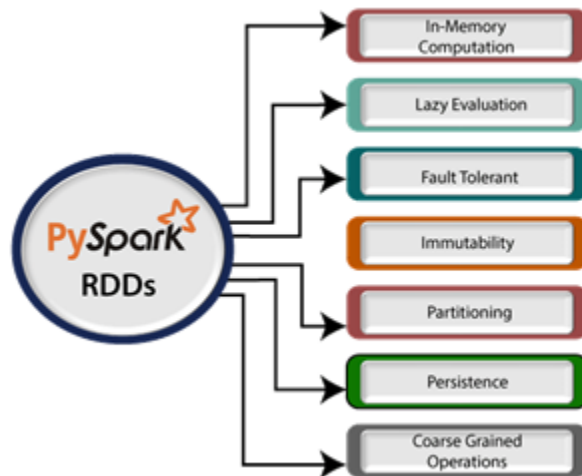
RDD divides data into smaller parts based on a key. The benefit of dividing data into smaller chunks is that if one executor node fails, another node will still process the data. These are able to recover quickly from any issues as the same data chunks are replicated across multiple executor nodes.

It provides the functionality to perform functional calculation against dataset very quickly by binding the multiple nodes.

RDD becomes immutable when it is created. Immutable mean, we cannot modify an object once it is created, but it can surely be transforme.

Features of RDD

Various features of PySpark RDDs are following:



In-memory Computation

PySpark provides provision of in-memory computation. Computed results are stored in distributed memory (RAM) instead of stable storage (disk). It provides very fast computation

Lazy Evolution

Transformation in PySpark RDDs is lazy. It doesn't compute the result immediately means that execution does not start until an action is triggered. When we call some operation in RDD for transformation, it does not execute immediately. **Lazy Evolution** plays an important role in saving calculation overhead. It provides the optimization by reducing the number of queries.

Fault Tolerant

RDDs track data lineage information to reconstruct lost data automatically. If failure occurs in any partition of RDDs, then that partition can be re-computed from the original fault tolerant input dataset to create it.

Immutability

The **created data can be retrieved anytime** but its value can't be changed. RDDs can only be created through deterministic operations.

Partitioning

RDDs are the collection of various data items that are so huge in size. Because of its size they cannot fit into a single node and must be partitioned across various nodes.

Persistence

It is an optimization technique where we can save the result of RDD evaluation. It stores the intermediate result so that we can use it further if required. It reduces the computation complexity.

Coarse-Grained Operation

The coarse grained operation means that we can transform the whole dataset but not individual element on the dataset. On the other hand, fine grained mean we can transform individual element on the dataset.

Create RDDs

PySpark provides two methods to create RDDs: loading an external dataset, or distributing a set of collection of objects. We can create RDDs using the `parallelize()` function which accepts an already existing collection in program and pass the same to the Spark Context. It is the simplest way to create RDDs. Consider the following code:

Output:

```
+---+---+---+---+
|col1|col2|col3|col4|
+---+---+---+---+
| 12| 20| 35|a b c|
| 41| 58| 64|d e f|
| 70| 85| 90|g h i|
+---+---+---+---+
```

-

Using `createDataFrame()` Function

1. `from pyspark.sql import SparkSession`
2. `spark = SparkSession \`
3. `.builder \`
4. `.appName("PySpark Create RDD example") \`
5. `.config("spark.some.config.option", "some-value") \`
6. `.getOrCreate()`
7. `Student = spark.createDataFrame([`
8. `('009001', 'Anuj', '70%', 'B.tech(cs)'),`

```

9. ('009002', 'Sachin', '80%', 'B.tech(cs)'),
10. ('008005', 'Yogesh', '94%', 'MCA'),
11. ('007014', 'Ananya', '98%', 'MCA')],
12. ['Roll_Num', 'Name', 'Percentage', 'Department']
13.)
14. Student.show()

```

The above code will give the following RDD data.

```

+-----+-----+-----+-----+
|Roll_Num|  Name|Percentage|Department|
+-----+-----+-----+-----+
|  009001|  Anuj|      70%|B.tech(cs)|
|  009002| Sachin|      80%|B.tech(cs)|
|  008005|Yogesh|      94%|      MCA|
|  007014|Ananya|      98%|      MCA|
+-----+-----+-----+-----+

```

- **Using read and load functions**

Here we read dataset from .csv file using the **read()** function.

```

1.
   ## set up SparkSession
2. from pyspark.sql import SparkSession
3. spark = SparkSession \
4. .builder \
5. .appName("PySpark create RDD example") \
6. .config("spark.some.config.option", "some-value") \
7. .getOrCreate()
8. df = spark.read.format('com.databricks.spark.csv').\
9. options(header='true', \
10. inferSchema='true').\
11. df = spark.read.format('com.databricks.spark.csv').\
12. options(header='true', \
13. inferSchema='true').\
14. load(r"C:\Users\DEVANSH SHARMA\top50.csv",
15. header=True)
16. df.show(5)
17. df.printSchema()

```

Output:

c0	Track.Name	Artist.Name
Genre Beats.Per.Minute Energy Danceability Loudness..dB.. Liveness Valence. Length. Acousticness.. Speechiness Popularity		
1 79 Seiorita Shawn Mendes canadian pop 117 55 76 -6 8 75 191 4		
2 92 China Anuel AA reggaeton flow 105 81 79 -4 8 61 302 8		
3 85 boyfriend (with S... Ariana Grande dance pop 190 80 40 -4 16 70 186 12		
4 86 Beautiful People ... El Sheeran pop 93 65 64 -8 8 55 198 12		
5 94 Goodbyes (Feat. Y... Post Malone dfw rap 150 65 58 -4 11 18 175 45		

```

only showing top 5 rows
root
|-- c0: integer (nullable = true)
|-- Track.Name: string (nullable = true)
|-- Artist.Name: string (nullable = true)
|-- Genre: string (nullable = true)
|-- Beats.Per.Minute: integer (nullable = true)
|-- Energy: integer (nullable = true)
|-- Danceability: integer (nullable = true)
|-- Loudness..dB..: integer (nullable = true)
|-- Liveness: integer (nullable = true)
|-- Valence.: integer (nullable = true)
|-- Length.: integer (nullable = true)
|-- Acousticness.: integer (nullable = true)
|-- Speechiness.: integer (nullable = true)
|-- Popularity: integer (nullable = true)

```

RDD Operations in PySpark

The RDD supports two types of operations:

1. Transformations

Transformations are the process which are used to create a new RDD. It follows the principle of Lazy Evaluations (the execution will not start until an action is triggered).

Few of transformations are given below:

- map
- flatMap
- filter
- distinct
- reduceByKey
- mapPartitions
- sortBy

2. Actions

Actions are the processes which are applied on an RDD to initiate Apache Spark to apply calculation and pass the result back to driver. Few actions are following:

- collect
- collectAsMap
- reduce
- countByKey/countByValue
- take
- first

Various Operations in RDDs

The operations applied on RDDs are following:

- **count()**

It returns the number of element available in RDD. Consider the following program.



1.

```
from pyspark import SparkContext
```
2.

```
words = sc.parallelize (
```
3.

```
["python",
```
4.

```
"java",
```
5.

```
"hadoop",
```
6.

```
"c",
```
7.

```
"C++",
```
8.

```
"spark vs hadoop",
```

```

9. "pyspark and spark"]
10.)
11. counts = words.count()
12. print("Number of elements present in RDD -> %i" % (counts))

```

Output:

Number of elements present in RDD : 7

- **collect()**

This function returns the entire elements in the RDD.

```

1.
   from pyspark import SparkContext
2. words = sc.parallelize (
3.   ["python",
4.    "java",
5.    "hadoop",
6.    "c",
7.    "C++",
8.    "spark vs hadoop",
9.    "pyspark and spark"]
10.)
11. counts = words.collect()
12. print(counts)

```

Output:

['python', 'java', 'hadoop', 'c', 'C++', 'spark vs hadoop', 'pyspark and spark']

- **foreach(f)**

The **foreach(f)** function returns only those elements which match the condition of the function inside foreach.

```

1.
   from pyspark import SparkContext

```

```

2. words = sc.parallelize (
3. ["python",
4.  "java",
5.  "hadoop",
6.  "C",
7.  "C++",
8.  "spark vs hadoop",
9.  "pyspark and spark"]
10.)
11. def f(x):
12.     print(x)
13.     fore = words.foreach(f)

```

Output:

```

python
java
hadoop
C
C++
spark vs hadoop
pyspark and spark

```

•
cc

The cc operation returns a new RDD which contains the elements; those satisfy the function inside the filter. In the following example, we filter out the strings containing "spark".

```

1.
   from pyspark import SparkContext
2. words = sc.parallelize (
3. ["scala",
4.  "java",
5.  "hadoop",
6.  "spark",

```



```

7.  "akka",
8.  "spark vs hadoop",
9.  "pyspark",
10. "pyspark and spark"]
11.)
12. words_filter = words.filter(lambda x: 'spark' in x)
13. filtered = words_filter.collect()
14. print("Filtered RDD : %s" % (filtered))

```

Output:

Filtered RDD : ['spark', 'spark vs hadoop', 'pyspark', 'pyspark and spark']



map(f, preservePartitioning = False)

It returns new RDD in a key-value pair and maps every string with a value of 1.
Consider the following example:

```

1.
   from pyspark import SparkContext
2. from pyspark import SparkContext
3. words = sc.parallelize (
4. ["python",
5.  "java",
6.  "hadoop",
7.  "C",
8.  "C++",
9.  "spark vs hadoop",
10. "pyspark and spark"]
11.)
12. words_map = words.map(lambda x: (x, 1))
13. mapping = words_map.collect()
14. print("Key value pair -> %s" % (mapping))

```

Output:

Key value pair -> [('python', 1), ('java', 1), ('hadoop', 1), ('c', 1), ('C++', 1), ('spark vs hadoop', 1), ('pyspark and spark', 1)]

-

reduce(f)

It performs the specified commutative and associative binary operation in RDD.
Consider the following example:

1.
from pyspark **import** SparkContext
2. from operator **import** add
3. sum = sc.parallelize([1, 2, 3, 4, 5])
4. adding = sum.reduce(add)
5. print("Adding all the elements in RDDs : %i" % (adding))

Output:

Adding all the elements : 15

-

cache()

We can check if the RDD is cached or not with **cache()** function.

1.
from pyspark **import** SparkContext
2. words = sc.parallelize (
3. ["python",
4. "java",
5. "hadoop",
6. "c",
7. "C++",
8. "spark vs hadoop",
9. "pyspark and spark"]
- 10.)
11. words.cache()
12. caching = words.persist().is_cached
13. print("Words got cached > %s" % (caching))

Output:

Words got cached > True

-

join(other, numPartition = None)

It returns RDD with the matching keys with their values in paired form. We will get two different RDDs for two pair of element. Consider the following code:

```
1.
    from pyspark import SparkContext
2. x = sc.parallelize([("pyspark", 1), ("hadoop", 3)])
3. y = sc.parallelize([("pyspark", 2), ("hadoop", 4)])
4. joined = x.join(y)
5. mapped = joined.collect()
6. print("Join RDD -> %s" % (mapped))
```

Output:

Join RDD -> [('hadoop', (3, 4)), ('pyspark', (1, 2))]

DataFrame from RDD

PySpark provides two methods to convert a RDD to DF. These methods are given following:

-

toDF()

When we create RDD by parallelize function, we should identify the same row element in DataFrame and wrap those element by the parentheses. The **row()** can accept the ****kwargs** argument.

```
1.
    from pyspark.sql.types import Row
2. from pyspark.sql import SparkSession
3. #here we are going to create a function
4. def f(x):
5.     d = {}
6.     for i in range(len(x)):
7.         d[str(i)] = x[i]
8.     return d
```

9. #Now populate that
 10. `df = rdd.map(lambda x: Row(**f(x))).toDF()`
- **`createDataFrame(rdd, schema)`**

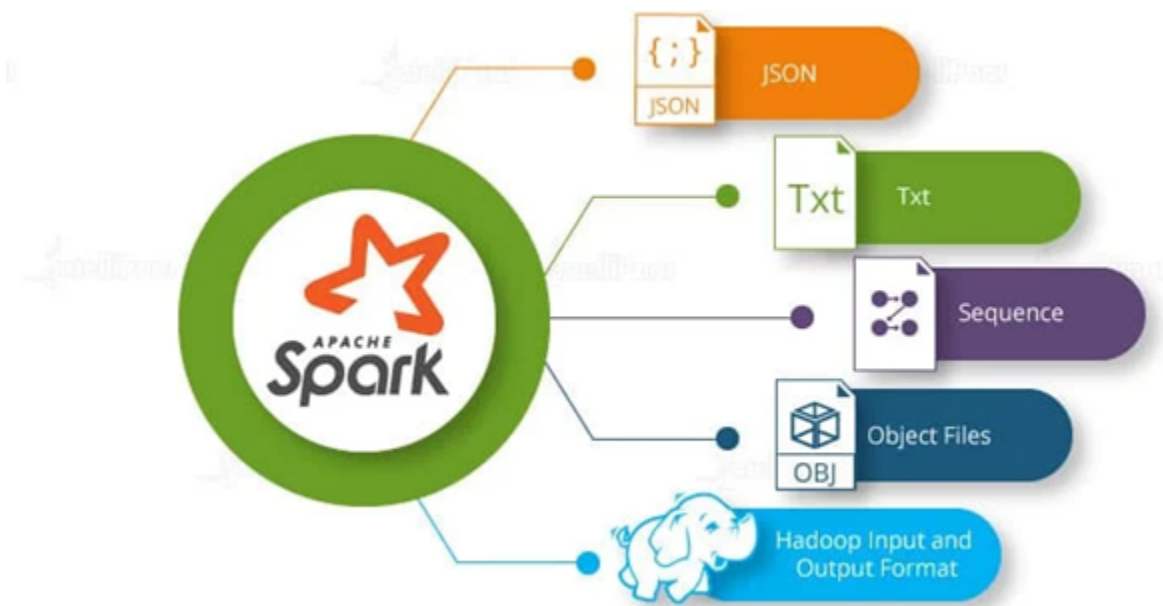
We can also convert the RDD to DataFrame by using `createDataFrame`. Consider the following example:

1.
`from pyspark.sql.types import StructType`
2. `from pyspark.sql.types import StructField`
3. `from pyspark.sql.types import StringType`
4. `schema = StructType([StructField(str(i), StringType(), True) for i in range(32)])`
5. `df = sqlContext.createDataFrame(rdd, schema)`

Lesson Proper for Week 9

File Formats

Spark provides a simple manner to load and save data files in a very large number of file formats. These formats may range from being unstructured, like text, to semi-structured, like JSON, to structured, like sequence files. The input file formats that Spark wraps are transparently handled in a compressed format based on the file extension specified.



Text Files

Text files are very simple and convenient to load from and save to Spark applications. When we load a single text file as an RDD, then each input line becomes an element in the RDD. It has the capacity to load multiple whole text files at the same time into a pair of RDD elements, with the key being the name given and the value the contents of each file format specified.

- **Loading the text files:** Loading a single text file is as simple as calling the `textFile()` function on our `SparkContext` with the pathname placed next to the file, as shown below:

```
input = sc.textFile("file:///home/holden/repos/spark/README.md")
```

- **Saving the text files:** Spark consists of a function called `saveAsTextFile()`, which saves the path of a file and writes the content of the RDD to that file. The path is considered as directory, and multiple outputs will be produced in that directory. This is how Spark becomes able to write output from multiple codes.

- **Example:**

```
result.saveAsTextFile(outputFile)
```

JSON Files

JSON stands for **JavaScript Object Notation, which is a light-weighted data interchange format.** It supports text only which can be easily sent to and received from a server.

- **Loading the JSON Files:** **For all supported languages, the approach of loading data in the text form and parsing the JSON data can be adopted.** Here, if the file contains multiple JSON records, the developer will have to download the entire file and parse each one by one.

- **Saving the JSON Files:** **In comparison to loading the JSON files, writing to it is much easier as, here, the developer does not have to worry about the wrong format of data values.** The same libraries can be used that were used to convert the RDDs into parsed JSON files; however, RDDs of the structured data will be taken and converted into RDDs of strings.

CSV and TSV Files

Comma-separated values (CSV) files are a very **common format used to store tables.** These files have a definite number of fields in each line the values of which are separated by a comma. Similarly, in **tab-separated values (TSV)** files, the field values are separated by tabs.

- **Loading the CSV Files:** The loading procedure of CSV and TSV files is quite similar to that of the JSON files. In order to load a CSV/TSV file, **its content in the text format is loaded at first and then it is processed.** Like the JSON files, CSV and TSV files also have different library files, but it is suggested to use only those corresponding to each language.

- **Saving the CSV Files:** Writing to CSV/TSV files is also quite easy. However, as the output cannot have the file name, mapping is required for better results. One easy way to perform this is to write a function that can convert the fields into positions in an array.

Sequence Files

A sequence file is a **flat file that consists of binary key/value pairs.** Sequence files are widely used in Hadoop. The sync markers in these files allow Spark to find a particular point in a file and re-synchronize it with record limits.

- **Loading the Sequence Files:** Spark comes with a specialized API that reads the sequence files. All we have to do is call a sequence file (`pat`, `keyClass`, `valueClass`, `minPartitions`), and access can be obtained from `SparkContext`.

- **Saving the Sequence Files:** In order to save the sequence files, a paired RDD, along with its types to write, is required. For several native types, implicit conversions between Scala and Hadoop Writables are possible. Hence, to

write a native type, we have to save the paired RDD by calling the `saveAsSequenceFile(path)` function. Then, we have to map over the data and convert it prior to saving if the conversion is not automatic.

Object Files

Object files are the packaging around sequence files that enables saving RDDs containing value records only. Saving an object file is quite simple as it just requires calling `saveAsObjectFile()` on an RDD.

Be familiar with these Top Spark Interview Questions and Answers and get a head start in your career!

RDD – The write operation in RDD is coarse grained

Hadoop Input and Output Formats

The input split is referred to as the data present in HDFS. Spark provides APIs to implement the InputFormat of Hadoop in Scala, Python, and Java. The old APIs were `hadoopRDD` and `hadoopFile`, but now the APIs have been improved and the new APIs are known as `newAPIHadoopRDD` and `newAPIHadoopFile`.

For `HadoopOutputFormat`, Hadoop takes `TextOutputFormat` in which the key and value pair is separated through comma and saved in part file. Spark has the APIs of Hadoop for both MapRed and MapReduce.

- **File Compression:** For most of the Hadoop outputs, a compression code can be specified which is easily accessible. It is used to compress the data.

File Systems

A wide array of file systems are supported by Apache Spark. Some of them are discussed below:

- **Local/Regular FS:** Spark is able to load files from the local file system, which requires files to remain on the same path on all nodes.
- **Amazon S3:** This file system is suitable for storing large amounts of files. It works faster when the computed nodes are inside Amazon EC2. However, at times, its performance goes down if we opt for the public network.
- **HDFS:** It is a distributed file system that works well on commodity hardware. It provides high throughput.

Structured Data with Spark SQL

It works effectively on semi-structured and structured data. Structured data can be defined as schemas, and it has a consistent set of fields.

Apache Hive

One of the common structured data sources on Hadoop is Apache Hive. Hive can store tables in a variety and different range of formats, from plain text to column-oriented formats, inside HDFS, and it also contains other storage systems. **Spark SQL** can load any amount of tables supported by Hive.

Databases

Spark supports a wide range of databases with the help of Hadoop Connectors or Custom Spark Connectors. Some of them are JDBC, Cassandra, HBase, and Elasticsearch.

Lesson Proper for Week 10

Introduction

This chapter introduces a variety of advanced Spark programming features that we didn't get to cover in the previous chapters. We introduce two types of shared variables: *accumulators* to aggregate information and *broadcast variables* to efficiently distribute large values. Building on our existing transformations on RDDs, we introduce batch operations for tasks with high setup costs, like querying a database. To expand the range of tools accessible to us, we cover Spark's methods for interacting with external programs, such as scripts written in R.

Throughout this chapter we build an example using ham radio operators' call logs as the input. These logs, at the minimum, include the call signs of the stations contacted. Call signs are assigned by country, and each country has its own range of call signs so we can look up the countries involved. Some call logs also include the physical location of the operators, which we can use to determine the distance involved. We include a sample log entry in Example 6-1. The book's sample repo includes a list of call signs to look up the call logs for and process the results.

Example 1. Sample call log entry in JSON, with some fields removed

```
{"address":"address here", "band":"40m", "callsign":"KK6JLK", "city":"SUNNYVALE",  
"contactlat":"37.384733", "contactlong":"-122.032164",  
"county":"Santa Clara", "dxcc":"291", "fullname":"MATTHEW McPherrin",  
"id":"57779", "mode":"FM", "mylat":"37.751952821", "mylong":"-122.4208688735", ...}
```

The first set of Spark features we'll look at are shared variables, which are a special type of variable you can use in Spark tasks. In our example we use Spark's shared variables to count nonfatal error conditions and distribute a large lookup table.

When our task involves a large setup time, such as creating a database connection or random-number generator, it is useful to share this setup work across multiple data items. Using a remote call sign lookup database, we examine how to reuse setup work by operating on a per-partition basis.

In addition to the languages directly supported by Spark, the system can call into programs written in other languages. This chapter introduces how to use Spark's language-agnostic pipe() method to interact with other programs through standard input and output. We will use the pipe() method to access an R library for computing the distance of a ham radio operator's contacts.

Finally, similar to its tools for working with key/value pairs, Spark has methods for working with numeric data. We demonstrate these methods by removing outliers from the distances computed with our ham radio call logs.

Accumulators

When we normally pass functions to Spark, such as a map() function or a condition for filter(), they can use variables defined outside them in the driver program, but each task running on the cluster gets a new copy of each variable, and updates from these copies are not propagated back to the driver. Spark's shared variables, *accumulators* and *broadcast variables*, relax this restriction for two common types of communication patterns: aggregation of results and broadcasts.

Our first type of shared variable, accumulators, provides a simple syntax for aggregating values from worker nodes back to the driver program. One of the most common uses of accumulators is to count events that occur during job execution for debugging purposes. For example, say that we are loading a list of all of the call signs for which we want to retrieve logs from a file, but we are also interested in how many lines of the input file were blank (perhaps we do not expect to see many such lines in valid input). Examples 6-2 through 6-4 demonstrate this scenario.

Example 2. Accumulator empty line count in Python

```
file = sc.textFile(inputFile)

# Create Accumulator[Int] initialized to 0

blankLines = sc.accumulator(0)

def extractCallSigns(line):

    global blankLines # Make the global variable accessible

    if (line == ""):

        blankLines += 1

    return line.split(" ")
```

```
callSigns = file.flatMap(extractCallSigns)

callSigns.saveAsTextFile(outputDir + "/callsigns")

print "Blank lines: %d" % blankLines.value
```

Example 3. Accumulator empty line count in Scala

```
val sc = new SparkContext(...)

val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0) // Create an Accumulator[Int] initialized to 0

val callSigns = file.flatMap(line => {

    if (line == "") {

        blankLines += 1 // Add to the accumulator

    }

    line.split(" ")

})

callSigns.saveAsTextFile("output.txt")

println("Blank lines: " + blankLines.value)
```


Example 4. Accumulator empty line count in Java

```
JavaRDD<String> rdd = sc.textFile(args[1]);

final Accumulator<Integer> blankLines = sc.accumulator(0);

JavaRDD<String> callSigns = rdd.flatMap(
    new FlatMapFunction<String, String>() { public Iterable<String> call(String line) {
        if (line.equals("")) {
            blankLines.add(1);
        }
        return Arrays.asList(line.split(" "));
    }
});

callSigns.saveAsTextFile("output.txt")

System.out.println("Blank lines: " + blankLines.value());
```

In these examples, we create an `Accumulator[Int]` called `blankLines`, and then add 1 to it whenever we see a blank line in the input. After evaluating the transformation, we print the value of the counter. Note that we will see the right count only *after* we run the `saveAsTextFile()` action, because the transformation above it, `map()`, is lazy, so the side-effect incrementing of the accumulator will happen only when the lazy `map()` transformation is forced to occur by the `saveAsTextFile()` action.

Of course, it is possible to aggregate values from an entire RDD back to the driver program using actions like `reduce()`, but sometimes we need a simple way to aggregate values that, in the process of transforming an RDD, are generated at different scale or granularity than that of the RDD itself. In the previous example, accumulators let us count errors as we load the data, without doing a separate `filter()` or `reduce()`.

To summarize, accumulators work as follows:

- We create them in the driver by calling the `SparkContext.accumulator(initialValue)` method, which produces an accumulator holding an initial value. The return type is an `org.apache.spark.Accumulator[T]` object, where `T` is the type of `initialValue`.
- Worker code in Spark closures can add to the accumulator with its `+=` method (or `add` in Java).
- The driver program can call the `value` property on the accumulator to access its value (or call `value()` and `setValue()` in Java).

Note that tasks on worker nodes cannot access the accumulator's `value()`—from the point of view of these tasks, accumulators are *write-only* variables. This allows accumulators to be implemented efficiently, without having to communicate every update.

The type of counting shown here becomes especially handy when there are multiple values to keep track of, or when the same value needs to increase at multiple places in the parallel program (for example, you might be counting calls to a JSON parsing library throughout your program). For instance, often we expect some percentage of our data to be corrupted, or allow for the backend to fail some number of times. To prevent producing garbage output when there

are too many errors, we can use a counter for valid records and a counter for invalid records. The value of our accumulators is available only in the driver program, so that is where we place our checks.

Continuing from our last example, we can now validate the call signs and write the output only if most of the input is valid. The ham radio call sign format is specified in Article 19 by the International Telecommunication Union, from which we construct a regular expression to verify conformance, shown in Example 6-5.

Example 5. Accumulator error count in Python

```
# Create Accumulators for validating call signs
```

```
validSignCount = sc.accumulator(0)
```

```
invalidSignCount = sc.accumulator(0)
```

```
def validateSign(sign):
```

```
    global validSignCount, invalidSignCount
```

```
    if re.match(r"Ad?[a-zA-Z]{1,2}d{1,4}[a-zA-Z]{1,3}", sign):
```

```
        validSignCount += 1
```

```
        return True
```

```
    else:
```

```
        invalidSignCount += 1
```

```
        return False
```

```
# Count the number of times we contacted each call sign
```

```
validSigns = callSigns.filter(validateSign)
```

```
contactCount = validSigns.map(lambda sign: (sign, 1)).reduceByKey(lambda (x, y): x + y)
```

```
# Force evaluation so the counters are populated
```

```
contactCount.count()
```

```
if invalidSignCount.value < 0.1 * validSignCount.value:
```

```
    contactCount.saveAsTextFile(outputDir + "/contactCount")
```

```
else:
```

```
    print "Too many errors: %d in %d" % (invalidSignCount.value, validSignCount.value)
```

Accumulators and Fault Tolerance

Spark automatically deals with failed or slow machines by re-executing failed or slow tasks. For example, if the node running a partition of a `map()` operation crashes, Spark will rerun it on another node; and even if the node does not crash but is simply much slower than other nodes, Spark can preemptively launch a “speculative” copy of the task on another node, and take its result if that finishes. Even if no nodes fail, Spark may have to rerun a task to rebuild a cached value that falls out of memory. The net result is therefore that the same function may run multiple times on the same data depending on what happens on the cluster.

How does this interact with accumulators? The end result is that *for accumulators used in actions, Spark applies each task's update to each accumulator only once*. Thus, if we want a reliable absolute value counter, regardless of failures or multiple evaluations, we must put it inside an action like `foreach()`.

For accumulators used in RDD transformations instead of actions, this guarantee does not exist. An accumulator update within a transformation can occur more than once. One such case of a probably unintended multiple update occurs when a cached but infrequently used RDD is first evicted from the LRU cache and is then subsequently needed. This forces the RDD to be recalculated from its lineage, with the unintended side effect that calls to update an accumulator within the transformations in that lineage are sent again to the driver. Within transformations, accumulators should, consequently, be used only for debugging purposes.

While future versions of Spark may change this behavior to count the update only once, the current version (1.2.0) does have the multiple update behavior, so accumulators in transformations are recommended only for debugging purposes.

Custom Accumulators

So far we've seen how to use one of Spark's built-in accumulator types: integers (`Accumulator[Int]`) with addition. Out of the box, Spark supports accumulators of type `Double`, `Long`, and `Float`. In addition to these, Spark also includes an API to define custom accumulator types and custom aggregation operations (e.g., finding the maximum of the accumulated values instead of adding them). Custom accumulators need to extend `AccumulatorParam`, which is covered in the Spark API documentation. Beyond adding to a numeric value, we can use any operation for add, provided that operation is commutative and associative. For example, instead of adding to track the total we could keep track of the maximum value seen so far.

Tip

An operation *op* is commutative if $a \text{ op } b = b \text{ op } a$ for all values *a*, *b*.

An operation *op* is associative if $(a \text{ op } b) \text{ op } c = a \text{ op } (b \text{ op } c)$ for all values *a*, *b*, and *c*.

For example, sum and max are commutative and associative operations that are commonly used in Spark accumulators.

Broadcast Variables

Spark's second type of shared variable, *broadcast variables*, allows the program to efficiently send a large, read-only value to all the worker nodes for use in one or more Spark operations. They come in handy, for example, if your application needs to send a large, read-only lookup table to all the nodes, or even a large feature vector in a machine learning algorithm.

Recall that Spark automatically sends all variables referenced in your closures to the worker nodes. While this is convenient, it can also be inefficient because (1) the default task launching mechanism is optimized for small task sizes, and (2) you might, in fact, use the same variable in *multiple* parallel operations, but Spark will send it separately for each operation. As an example, say that we wanted to write a Spark program that looks up countries by their call signs by prefix matching in an array. This is useful for ham radio call signs since each country gets its own prefix, although the prefixes are not uniform in length. If we wrote this naively in Spark, the code might look like Example 6-6.

Example 6. Country lookup in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = loadCallSignTable()
```

```
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes)
    count = sign_count[1]
    return (country, count)
```

```
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+ y)))
```

This program would run, but if we had a larger table (say, with IP addresses instead of call signs), the signPrefixes could easily be several megabytes in size, making it expensive to send that Array from the master alongside each task. In addition, if we used the same signPrefixes object later (maybe we next ran the same code on *file2.txt*), it would be sent *again* to each node.

We can fix this by making signPrefixes a broadcast variable. A broadcast variable is simply an object of type `spark.broadcast.Broadcast[T]`, which wraps a value of type `T`. We can access this value by calling `value` on the Broadcast object in our tasks. The value is sent to each node only once, using an efficient, BitTorrent-like communication mechanism.

Using broadcast variables, our previous example looks like Examples 6-7 through 6-9.

Example 7. Country lookup with Broadcast values in Python

```
# Look up the locations of the call signs on the
# RDD contactCounts. We load a list of call sign
# prefixes to country code to support this lookup.
signPrefixes = sc.broadcast(loadCallSignTable())
```

```
def processSignCount(sign_count, signPrefixes):
    country = lookupCountry(sign_count[0], signPrefixes.value)
    count = sign_count[1]
    return (country, count)
```

```
countryContactCounts = (contactCounts
    .map(processSignCount)
    .reduceByKey((lambda x, y: x+ y)))
```

```
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 8. Country lookup with Broadcast values in Scala

```
// Look up the countries for each call sign for the
```

```
// contactCounts RDD. We load an array of call sign
// prefixes to country code to support this lookup.
val signPrefixes = sc.broadcast(loadCallSignTable())
val countryContactCounts = contactCounts.map{case (sign, count) =>
    val country = lookupInArray(sign, signPrefixes.value)
    (country, count)
}.reduceByKey((x, y) => x + y)
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt")
```

Example 9. Country lookup with Broadcast values in Java

```
// Read in the call sign table
// Look up the countries for each call sign in the
// contactCounts RDD
final Broadcast<String[]> signPrefixes = sc.broadcast(loadCallSignTable());
JavaPairRDD<String, Integer> countryContactCounts = contactCounts.mapToPair(
    new PairFunction<Tuple2<String, Integer>, String, Integer> (){
        public Tuple2<String, Integer> call(Tuple2<String, Integer> callSignCount) {
            String sign = callSignCount._1();
            String country = lookupCountry(sign, callSignInfo.value());
            return new Tuple2(country, callSignCount._2());
        }
    }).reduceByKey(new SumInts());
countryContactCounts.saveAsTextFile(outputDir + "/countries.txt");
```

As shown in these examples, the process of using broadcast variables is simple:

1. Create a `Broadcast[T]` by calling `SparkContext.broadcast` on an object of type `T`. Any type works as long as it is also `Serializable`.
2. Access its value with the `value` property (or `value()` method in Java).
3. The variable will be sent to each node only once, and should be treated as read-only (updates will *not* be propagated to other nodes).

The easiest way to satisfy the *read-only* requirement is to broadcast a primitive value or a reference to an immutable object. In such cases, you won't be able to change the value of the broadcast variable except within the driver code. However, sometimes it can be more convenient or more efficient to broadcast a mutable object. If you do that, it is up to you to maintain the read-only condition. As we did with our call sign prefix table of `Array[String]`, we must make sure that the code we run on our worker nodes does not try to do something like `val theArray = broadcastArray.value; theArray(0) = newValue`. When run in a worker node, that line will assign `newValue` to the first array element only in the copy of the array local to the worker node running the code; it will not change the contents of `broadcastArray.value` on any of the other worker nodes.

Spark SQL provides a domain-specific language to manipulate _____ in Scala, Java, or Python.

- a) Spark Streaming
- b) Spark SQL
- c) RDDs
- d) All of the mentioned

Spark written in SCALA

Spark comes packaged with higher-level libraries, including support for SQL queries, streaming data, machine learning and graph processing.

RDD – The read operation in RDD is either coarse grained or fine grained.

Df.show - triggers evaluation

RDD- is a immutable collection of objects

printSchema()- is not an action

foreach()- is an action

Df.repartition- wide transform commands

Lesson Proper for Week 11

Optimizing Broadcasts

When we are broadcasting large values, it is important to choose a data serialization format that is both fast and compact, because the time to send the value over the network can quickly become a bottleneck if it takes a long time to either serialize a value or to send the serialized value over the network. In particular, Java Serialization, the default serialization library used in Spark's Scala and Java APIs, can be very inefficient out of the box for anything except arrays of primitive types. You can optimize serialization by selecting a different serialization library using the spark.serializer property or by implementing your own serialization routines for your data types (e.g., using the java.io.Externalizable interface for Java Serialization, or using the reduce() method to define custom serialization for Python's pickle library).

Working on a Per-Partition Basis

Working with data on a per-partition basis allows us to avoid redoing setup work for each data item. Operations like opening a database connection or creating a random-number generator are examples of setup steps that we wish to avoid doing for each element. Spark has *per-partition* versions of map and foreach to help reduce the cost of these operations by letting you run code only once for each partition of an RDD.

Going back to our example with call signs, there is an online database of ham radio call signs we can query for a public list of their logged contacts. By using partition-based operations, we can share a connection pool to this database to avoid setting up many connections, and reuse our JSON parser. As Examples 6-10 through 6-12 show, we use the mapPartitions() function, which gives us an iterator of the elements in each partition of the input RDD and expects us to return an iterator of our results.

Example 10. Shared connection pool in Python

```
def processCallSigns(signs):
```

```
    """Lookup call signs using a connection pool"""
    # Create a connection pool
    http = urllib3.PoolManager()
    # the URL associated with each call sign record
    urls = map(lambda x: "http://73s.com/qsos/%s.json" % x, signs)
    # create the requests (non-blocking)
    requests = map(lambda x: (x, http.request('GET', x)), urls)
    # fetch the results
    result = map(lambda x: (x[0], json.loads(x[1].data)), requests)
    # remove any empty results and return
    return filter(lambda x: x[1] is not None, result)
```

```
def fetchCallSigns(input):
```

```
""Fetch call signs""
```

```
return input.mapPartitions(lambda callSigns : processCallSigns(callSigns))
```

```
contactsContactList = fetchCallSigns(validSigns)
```

Example 6-11. Shared connection pool and JSON parser in Scala

```
val contactsContactLists = validSigns.distinct().mapPartitions{
  signs =>
    val mapper = createMapper()
    val client = new HttpClient()
    client.start()
    // create http request
    signs.map {sign =>
      createExchangeForSign(sign)
    }
    // fetch responses
    }.map{ case (sign, exchange) =>
      (sign, readExchangeCallLog(mapper, exchange))
    }.filter(x => x._2 != null) // Remove empty CallLogs
}
```

Example 12. Shared connection pool and JSON parser in Java

```
// Use mapPartitions to reuse setup work.
```

```
JavaPairRDD<String, CallLog[]> contactsContactLists =
  validCallSigns.mapPartitionsToPair(
    new PairFlatMapFunction<Iterator<String>, String, CallLog[]>() {
      public Iterable<Tuple2<String, CallLog[]>> call(Iterator<String> input) {
```



```
// List for our results.
ArrayList<Tuple2<String, CallLog[]>> callsignLogs = new ArrayList<>();
ArrayList<Tuple2<String, ContentExchange>> requests = new ArrayList<>();
ObjectMapper mapper = createMapper();
HttpClient client = new HttpClient();
try {
    client.start();
    while (input.hasNext()) {
        requests.add(createRequestForSign(input.next(), client));
    }
    for (Tuple2<String, ContentExchange> signExchange : requests) {
        callsignLogs.add(fetchResultFromRequest(mapper, signExchange));
    }
} catch (Exception e) {
}
return callsignLogs;
}});

System.out.println(StringUtils.join(contactsContactLists.collect(), ","));
```

When operating on a per-partition basis, Spark gives our function an Iterator of the elements in that partition. To return values, we return an Iterable. In addition to mapPartitions(), Spark has a number of other per-partition operators, listed in Table 6-1.

Table 1. Per-partition operators

Function name	We are called with	We return	Function signature on RDD[T]
---------------	--------------------	-----------	---------------------------------

<code>mapPartitions()</code>	Iterator of the elements in that partition	Iterator of our return elements	<code>f: (Iterator[T]) → Iterator[U]</code>
<code>mapPartitionsWithIndex()</code>	Integer of partition number, and Iterator of the elements in that partition	Iterator of our return elements	<code>f: (Int, Iterator[T]) → Iterator[U]</code>
<code>foreachPartition()</code>	Iterator of the elements	Nothing	<code>f: (Iterator[T]) → Unit</code>

In addition to avoiding setup work, we can sometimes use `mapPartitions()` to avoid object creation overhead. Sometimes we need to make an object for aggregating the result that is of a different type. Thinking back to Chapter 3, where we computed the average, one of the ways we did this was by converting our RDD of numbers to an RDD of tuples so we could track the number of elements processed in our reduce step. Instead of doing this for each element, we can instead create the tuple once per partition, as shown in Examples 6-13 and 6-14.

Example 13. Average without `mapPartitions()` in Python

```
def combineCtrs(c1, c2):
```

```
    return (c1[0] + c2[0], c1[1] + c2[1])
```

```
def basicAvg(nums):
```

```
    """Compute the average"""
```

```
    nums.map(lambda num: (num, 1)).reduce(combineCtrs)
```

Example 14. Average with `mapPartitions()` in Python

```
def partitionCtr(nums):
```

```
    """Compute sumCounter for partition"""
```

```
    sumCount = [0, 0]
```

```
for num in nums:
    sumCount[0] += num
    sumCount[1] += 1
return [sumCount]
```

```
def fastAvg(nums):
    """Compute the avg"""
    sumCount = nums.mapPartitions(partitionCtr).reduce(combineCtrs)
    return sumCount[0] / float(sumCount[1])
```

Piping to External Programs

With three language bindings to choose from out of the box, you may have all the options you need for writing Spark applications. However, if none of Scala, Java, or Python does what you need, then Spark provides a general mechanism to pipe data to programs in other languages, like R scripts.

Spark provides a `pipe()` method on RDDs. Spark's `pipe()` lets us write parts of jobs using any language we want as long as it can read and write to Unix standard streams. With `pipe()`, you can write a transformation of an RDD that reads each RDD element from standard input as a String, manipulates that String however you like, and then writes the result(s) as Strings to standard output. The interface and programming model is restrictive and limited, but sometimes it's just what you need to do something like make use of a native code function within a map or filter operation.

Most likely, you'd want to pipe an RDD's content through some external program or script because you've already got complicated software built and tested that you'd like to reuse with Spark. A lot of data scientists have code in R,¹² and we can interact with R programs using `pipe()`.

In Example 15 we use an R library to compute the distance for all of the contacts. Each element in our RDD is written out by our program with newlines as separators, and every line that the program outputs is a string element in the resulting RDD. To make it easy for our R program to parse the input we will

reformat our data to be mylat, mylon, theirlat, theirlon. Here we have a comma as the separator.

Example 15. R distance program

```
#!/usr/bin/env Rscript
library("lmap")
f <- file("stdin")
open(f)
while(length(line <- readLines(f,n=1)) > 0) {
  # process line
  contents <- Map(as.numeric, strsplit(line, ","))
  mydist <- gdist(contents[[1]][1], contents[[1]][2],
                  contents[[1]][3], contents[[1]][4],
                  units="m", a=6378137.0, b=6356752.3142, verbose = FALSE)
  write(mydist, stdout())
}
```

If that is written to an executable file named `./src/R/finddistance.R`, then it looks like this in use:

```
$ ./src/R/finddistance.R
```

```
37.75889318222431,-122.42683635321838,37.7614213,-122.4240097
```

```
349.2602
```

```
coffee
```

```
NA
```

```
ctrl-d
```

So far, so good—we've now got a way to transform every line from stdin into output on stdout. Now we need to make `finddistance.R` available to each of our

worker nodes and to actually transform our RDD with our shell script. Both tasks are easy to accomplish in Spark, as you can see in Examples 6-16 through 6-18.

Example 16. Driver program using pipe() to call finddistance.R in Python

```
# Compute the distance of each call using an external R program
distScript = "./src/R/finddistance.R"
distScriptName = "finddistance.R"
sc.addFile(distScript)
def hasDistInfo(call):
    """Verify that a call has the fields required to compute the distance"""
    requiredFields = ["mylat", "mylong", "contactlat", "contactlong"]
    return all(map(lambda f: call[f], requiredFields))
def formatCall(call):
    """Format a call so that it can be parsed by our R program"""
    return "{0},{1},{2},{3}".format(
        call["mylat"], call["mylong"],
        call["contactlat"], call["contactlong"])

pipeInputs = contactsContactList.values().flatMap(
    lambda calls: map(formatCall, filter(hasDistInfo, calls)))
distances = pipeInputs.pipe(SparkFiles.get(distScriptName))
print distances.collect()
```

Example 17. Driver program using pipe() to call finddistance.R in Scala

```
// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
```

```

val distScript = "./src/R/finddistance.R"
val distScriptName = "finddistance.R"
sc.addFile(distScript)
val distances = contactsContactLists.values.flatMap(x => x.map(y =>
    s"$y.contactlay,$y.contactlong,$y.mylat,$y.mylong")).pipe(Seq(
    SparkFiles.get(distScriptName)))
println(distances.collect().toList)

```

Example 18. Driver program using pipe() to call finddistance.R in Java

```

// Compute the distance of each call using an external R program
// adds our script to a list of files for each node to download with this job
String distScript = "./src/R/finddistance.R";
String distScriptName = "finddistance.R";
sc.addFile(distScript);
JavaRDD<String> pipeInputs = contactsContactLists.values()
    .map(new VerifyCallLogs()).flatMap(
    new FlatMapFunction<CallLog[], String>() {
        public Iterable<String> call(CallLog[] calls) {
            ArrayList<String> latLons = new ArrayList<String>();
            for (CallLog call: calls) {
                latLons.add(call.mylat + "," + call.mylong +
                    "," + call.contactlat + "," + call.contactlong);
            }
            return latLons;
        }
    });

```

```
JavaRDD<String> distances = pipeInputs.pipe(SparkFiles.get(distScriptName));  
System.out.println(StringUtils.join(distances.collect(), ","));
```

With `SparkContext.addFile(path)`, we can build up a list of files for each of the worker nodes to download with a Spark job. These files can come from the driver's local filesystem (as we did in these examples), from HDFS or other Hadoop-supported filesystems, or from an HTTP, HTTPS, or FTP URI. When an action is run in the job, the files will be downloaded by each of the nodes. The files can then be found on the worker nodes in `SparkFiles.getRootDirectory`, or located with `SparkFiles.get(filename)`. Of course, this is only one way to make sure that `pipe()` can find a script on each worker node. You could use another remote copying tool to place the script file in a knowable location on each node.

Tip

All the files added with `SparkContext.addFile(path)` are stored in the same directory, so it's important to use unique names.

Once the script is available, the `pipe()` method on RDDs makes it easy to pipe the elements of an RDD through the script. Perhaps a smarter version of `findDistance` would accept `SEPARATOR` as a command-line argument. In that case, either of these would do the job, although the first is preferred:

- `rdd.pipe(Seq(SparkFiles.get("finddistance.R"), ","))`
- `rdd.pipe(SparkFiles.get("finddistance.R") + " ,")`

In the first option, we are passing the command invocation as a sequence of positional arguments (with the command itself at the zero-offset position); in the second, we're passing it as a single command string that Spark will then break down into positional arguments.

We can also specify shell environment variables with `pipe()` if we desire. Simply pass in a map of environment variables to values as the second parameter to `pipe()`, and Spark will set those values.

You should now at least have an understanding of how to use `pipe()` to process the elements of an RDD through an external command, and of how to distribute such command scripts to the cluster in a way that the worker nodes can find them.

Numeric RDD Operations

Spark provides several descriptive statistics operations on RDDs containing numeric data. These are in addition to the more complex statistical and machine learning methods we will describe later in Chapter 11.

Spark's numeric operations are implemented with a streaming algorithm that allows for building up our model one element at a time. The descriptive statistics are all computed in a single pass over the data and returned as a `StatsCounter` object by calling `stats()`. Table 6-2 lists the methods available on the `StatsCounter` object.

Table 2. Summary statistics available from `StatsCounter`

Method	Meaning
<code>count()</code>	Number of elements in the RDD
<code>mean()</code>	Average of the elements
<code>sum()</code>	Total
<code>max()</code>	Maximum value
<code>min()</code>	Minimum value
<code>variance()</code>	Variance of the elements
<code>sampleVariance()</code>	Variance of the elements, computed for a sample
<code>stdev()</code>	Standard deviation
<code>sampleStdev()</code>	Sample standard deviation

If you want to compute only one of these statistics, you can also call the corresponding method directly on an RDD—for example, `rdd.mean()` or `rdd.sum()`.

In Examples 6-19 through 6-21, we will use summary statistics to remove some outliers from our data. Since we will be going over the same RDD twice (once to compute the summary statistics and once to remove the outliers), we may wish to cache the RDD. Going back to our call log example, we can remove the contact points from our call log that are too far away.

Example 19. Removing outliers in Python

```
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.

distanceNumerics = distances.map(lambda string: float(string))

stats = distanceNumerics.stats()

stddev = std.stdev()

mean = stats.mean()

reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stddev)

print reasonableDistances.collect()
```

Example 20. Removing outliers in Scala

```
// Now we can go ahead and remove outliers since those may have misreported
locations

// first we need to take our RDD of strings and turn it into doubles.

val distanceDouble = distance.map(string => string.toDouble)

val stats = distanceDoubles.stats()

val stddev = stats.stdev

val mean = stats.mean

val reasonableDistances = distanceDoubles.filter(x => math.abs(x-mean) < 3 *
stddev)

println(reasonableDistance.collect().toList)
```

Example 21. Removing outliers in Java

```
// First we need to convert our RDD of String to a DoubleRDD so we can
// access the stats function
```

```

JavaDoubleRDD distanceDoubles = distances.mapToDouble(new
DoubleFunction<String>() {
    public double call(String value) {
        return Double.parseDouble(value);
    }
});

final StatCounter stats = distanceDoubles.stats();

final Double stddev = stats.stdev();

final Double mean = stats.mean();

JavaDoubleRDD reasonableDistances =
    distanceDoubles.filter(new Function<Double, Boolean>() {
        public Boolean call(Double x) {
            return (Math.abs(x-mean) < 3 * stddev);
        }
    });

System.out.println(StringUtils.join(reasonableDistance.collect(), ","));

```

With that final piece we have completed our sample application, which uses accumulators and broadcast variables, per-partition processing, interfaces with external programs, and summary statistics. The entire source code is available in `src/python/ChapterSixExample.py`, `src/main/scala/com/oreilly/learningsparkexamples/scala/ChapterSixExample.scala`, and `src/main/java/com/oreilly/learningsparkexamples/java/ChapterSixExample.java`, respectively.

Actions

Action	Meaning
reduce (<i>func</i>)	Aggregate the elements of the dataset using a function <i>func</i> (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel.

collect()	Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data.
count()	Return the number of elements in the dataset.
first()	Return the first element of the dataset (similar to <code>take(1)</code>).
take(<i>n</i>)	Return an array with the first <i>n</i> elements of the dataset. Note that this is currently not executed in parallel. Instead, the driver program computes all the elements.
takeSample (<i>withReplacement</i> , <i>num</i> , <i>seed</i>)	Return an array with a random sample of <i>num</i> elements of the dataset, with or without replacement, using the given random number generator seed.
saveAsTextFile (<i>path</i>)	Write 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 <code>toString</code> on each element to convert it to a line of text in the file.

saveAsSequenceFile(path)	Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is only available on RDDs of key-value pairs that either implement Hadoop's Writable interface or are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc).
countByKey()	Only available on RDDs of type (K, V). Returns a `Map` of (K, Int) pairs with the count of each key.
foreach(func)	Run a function <i>func</i> on each element of the dataset. This is usually done for side effects such as updating an accumulator variable (see below) or interacting with external storage systems.

2.3 Create DataFrame with schema

If you want to specify the column names along with their data types, you should create the StructType schema first and then assign this while creating a DataFrame.

1.2 Using Spark createDataFrame() from SparkSession

Using `createDataFrame()` from SparkSession is another way to create and it takes rdd object as an argument. and chain with toDF() to specify names to the columns.