# Welcome to GKTCS Innovations Pvt Ltd

## IT Training & Consultancy

# Surendra Panpaliya

Director, GKTCS Innovations Pvt. Ltd, Pune.

## 18+ Years of Experience ( MCA, PGDCS, BSc. [Electronics] , CCNA)

- Founder,GKTCS Innovations Pvt. Ltd. Pune [ Nov 2009 – Till date ]
- 500 + Corporate Training for HP, IBM, Cisco,Wipro, Samsung etc.
- Skills
    - **Hadoop, Pig, Hive, Sqoop, Oozie, Spark, PySpark**
    - **Ruby, Rails,Cucumber, Calabash, Capybara, Rspec, Appium**
    - **Python, Django, Data Science, Machine Learning, Jython, Selenium**
    - **UNIX /Linux Shell Scripting, Perl, PHP, CakePHP, System Programming**
    - **CA Siteminder, Autosys, SSO, Service Desk, Service Delivery**
- Author of 4 Books
- National Paper Presentation Awards at BARC Mumbai

# Agenda

**Day 1**
**Module 1**
Introduction to Spark
What is Apache Spark?
Spark Jobs and APIs
Spark 2.0 architecture
Installation and Configuration

**Module 2**
Resilient Distributed Datasets
Internal workings of an RDD
Creating RDDs
Global versus local scope
Transformations
Actions
Hands on Session on RDD and Spark
Assignments 1
Best Practices 1

# Agenda

**Day 2**

**Module 3**

DataFrames
Python to RDD communications
Catalyst Optimiser refresh
Speeding up PySpark with DataFrames
Creating DataFrames
Simple DataFrame queries
Interoperating with RDDs
Querying with the DataFrame API
Hands On Session on Pandas DataFrame and PySpark
Assignments 2

**Module 4**

Prepare Data for Modelling
Checking for duplicates, missing observations, and outliers
Getting familiar with your data Visualisation
Hands on Session Data Modelling
Assignments 3

# Agenda

**Day 3**
**Module 5**
Introducing MLlib
Overview of the package
Loading and transforming the data
Getting to know your data
Creating the final dataset
Predicting infant survival
Hands on Session using PySpark MLib
Assignments 4

**Module 6**
Introducing the ML Package
Overview of the package
Predicting the chances of infant survival with ML
Parameter hyper-tuning
Other features of PySpark ML in action
Implementation of ML Algorithm
- Random Forest
- Regression
- K-means

Assignments 5

# Agenda

**Day 3**
**Module 7**
GraphFrames
Introducing GraphFrames
Installing GraphFrames
Preparing your flights dataset
Building the graph
Executing simple queries
Understanding vertex degrees
Determining the top transfer airports
Understanding motifs
Determining airport ranking using PageRank
Determining the most popular non-stop flights
Using Breadth-First Search
Visualizing flights using D3
Assignment 6
Conclusion and Summary

# SPARK Pyspark

Surendra R. Panpaliya
M:9975072320
surendrarp@gktcs.com
www.gktcs.com

# Agenda

**Day 1**
**Module 1**
Introduction to Spark
What is Apache Spark?
Spark Jobs and APIs
Spark 2.0 architecture
Installation and Configuration

**Module 2**
Resilient Distributed Datasets
Internal workings of an RDD
Creating RDDs
Global versus local scope
Transformations
Actions
Hands on Session on RDD and Spark
Assignments 1
Best Practices 1

# Agenda

**Day 1**
**Module 1**
Introduction to Spark
What is Apache Spark?
Spark Jobs and APIs
Spark 2.0 architecture
Installation and Configuration

# Module 1

## Introduction to Spark

# Introduction

Apache Spark is a powerful open source processing engine originally developed by Matei Zaharia as a part of his PhD thesis while at UC Berkeley.

The first version of Spark was released in 2012.

Since then, in 2013, Zaharia co-founded and has become the CTO at Databricks;

He also holds a professor position at Stanford, coming from MIT.

At the same time, the Spark codebase was donated to the Apache Software Foundation.

# Introduction

Apache Spark is fast, easy to use framework, that allows you to solve a wide variety of complex data problems whether semi-structured, structured, streaming, and/or machine learning / data sciences.

It also has become one of the largest open source communities in big data with more than 1,000 contributors from 250+ organizations and with 300,000+ Spark Meetup community members in more than 570+ locations worldwide.

# Module 1

## What is Apache Spark?

# What is Apache Spark?

- Apache Spark is an open-source powerful distributed querying and processing engine.
- It provides flexibility and extensibility of MapReduce but at significantly higher speeds: Up to 100 times faster than Apache Hadoop when data is stored in memory and up to 10 times when accessing disk.
- Apache Spark allows the user to read, transform, and aggregate data, as well as train and deploy sophisticated statistical models with ease.
- The Spark APIs are accessible in Java, Scala, Python, R and SQL.
- Apache Spark can be used to build applications or package them up as libraries to be deployed on a cluster or perform quick analytics interactively through notebooks (like, for instance, Jupyter, Spark-Notebook, Databricks notebooks, and Apache Zeppelin).

# What is Apache Spark?

| | |
|---|---|
| **Speed** | • Enables applications in Hadop clusters to run upto:<br>• 100x faster in memory<br>• 10X on disks |
| **Ease of Use** | • Allows quickly write applications in Java, SCALA or Python |
| **Sophisticated Analytics** | • Supports SQL queries, streaming data and complex analytics |

# What is Apache Spark?

- Apache Spark exposes a host of libraries familiar to data analysts, data scientists or researchers who have worked with Python's pandas or R's data.frames or data.tables.
- It is important to note that while Spark DataFrames will be familiar to pandas or data.frames / data.tables users, there are some differences so please temper your expectations.
- Users with more of a SQL background can use the language to shape their data as well.
- It also include several already implemented and tuned algorithms, statistical models, and frameworks: MLlib and ML for machine learning, GraphX and GraphFrames for graph processing, and Spark Streaming (DStreams and Structured).
- Spark allows the user to combine these libraries seamlessly in the same application.

# What does Apache Spark offer?

- Faster execution
- Spark encourages Hadoop application clusters to execute 100x faster in memory and 10x faster on disk.
- Owing to its advance DAG execution engine, it also possesses support for cyclic data flow and in-memory computing.

# What is Apache Spark?

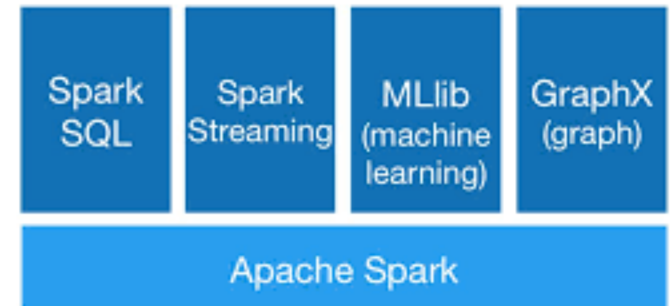
- Apache Spark is a lightning-fast cluster computing technology, designed for fast computation.
- It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing.
- The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

# What is Apache Spark?

- Alternative to MapReduce
- Compatibility with HDFS, HBASE and YARN
- Spark SQL component to access structured data
- Stable API
- Support for multiple languages
- Library support

# What is Apache Spark?

- Apache Spark has a well-designed and striking development API which lets the developers undergo data iteration with various data science methodologies which need quick in-memory processing.
- Also, with YARN, Spark can, in parallel be used for other data related workloads with all of them sharing the same data set Generality.



# Spark Architecture

| Spark SQL + DataFrames | Streaming | MLlib *Machine Learning* | GraphX *Graph Computation* |
|---|---|---|---|

Spark Core API

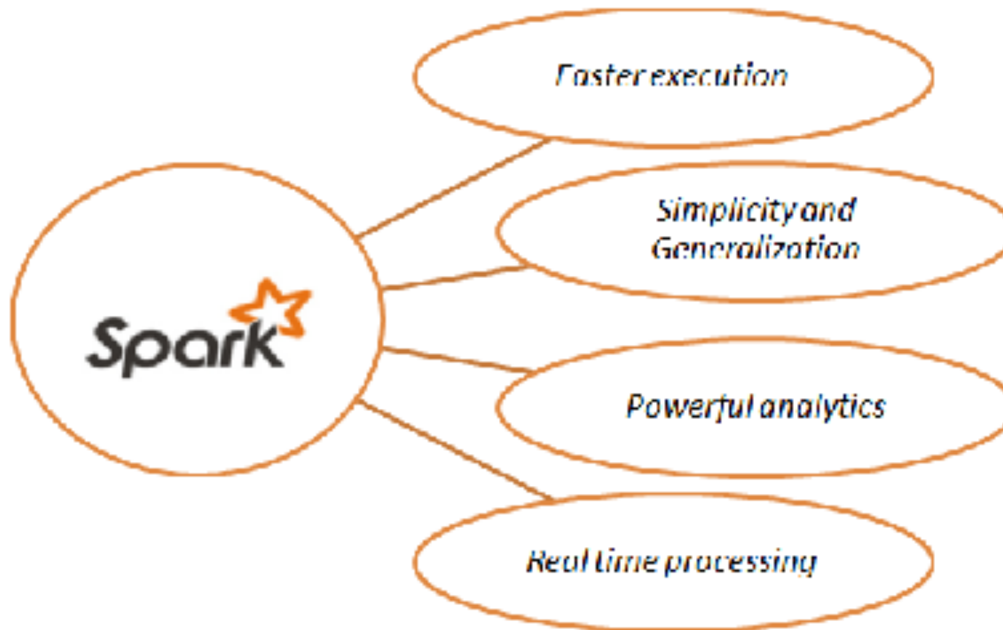
| R | SQL | Python | Scala | Java |
|---|---|---|---|---|

# What is Apache Spark?

- Spark is one of Hadoop's sub project developed in 2009 in UC Berkeley's AMPLab by Matei Zaharia.
- It was Open Sourced in 2010 under a BSD license.
- It was donated to Apache software foundation in 2013, and now Apache Spark has become a top level Apache project from Feb-2014.
- It has proven to be one of the largest communities contributing to Big Data.

# What is Apache Spark?

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming.



# What does Apache Spark offer?

- **Simplicity and Generalisation**
- Apache Spark allows writing applications with ease in Java, Scala or Python. There is availability of over 80 operators and it can used to query data within the shell. It has a perfect combination of SQL, streaming and complex analytics.
- There are high level tools like Spark SQL, MLlib (machine learning), GraphX and SparkStreaming. These libraries can be seamlessly combined in the same application.

# What does Apache Spark offer?

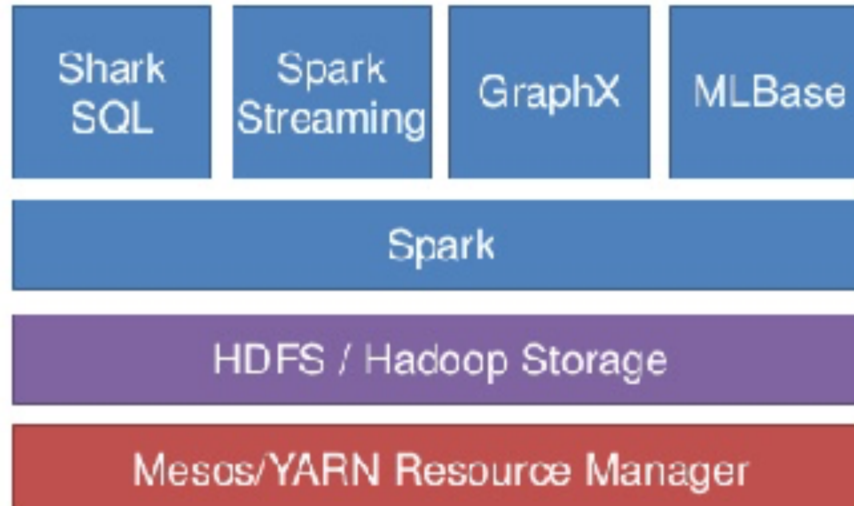
- **Powerful analytics**
- There is support for SQL queries, streaming and complex analytics. These combinations lead to a single workflow and give out sophisticated analytics.
- **Real time processing**
- It manages real time streaming and can manipulate real time data using Spark Streaming. Hence, streaming is also possible with Hadoop and other frameworks available.

# What does Apache Spark offer?

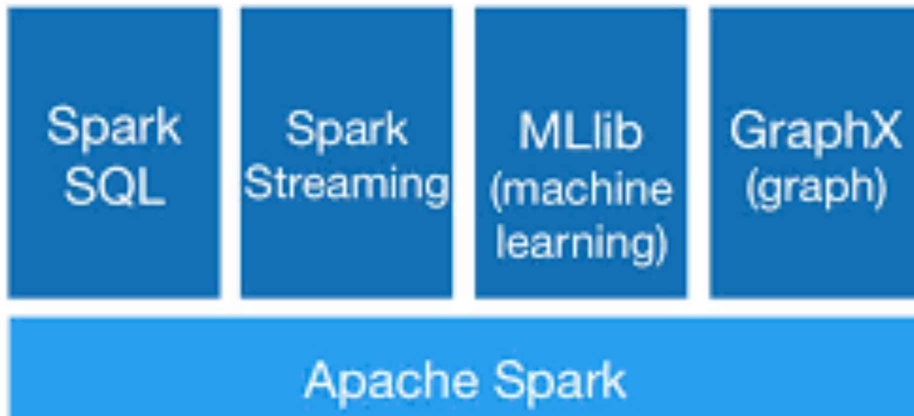
**Supports multiple languages**

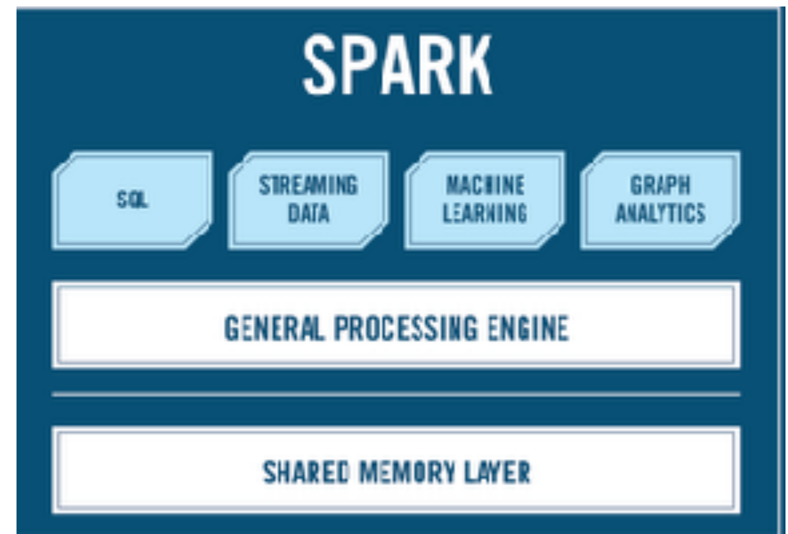
- Spark provides built-in APIs in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
- Advanced Analytics: Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

# Spark Ecosystem



# Spark Ecosystem



# Spark Ecosystem

**Apache Spark Core**

- Spark Core is the underlying general execution engine for 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 a component on top of Spark Core that introduces a new data abstraction called SchemaRDD, which provides support for structured and semi-structured data.

# Spark Ecosystem

**Spark Streaming**

Spark Streaming leverages Spark Core's fast scheduling capability to perform streaming analytics. It ingests data in mini-batches and performs RDD (Resilient Distributed Datasets) transformations on those mini-batches of data.

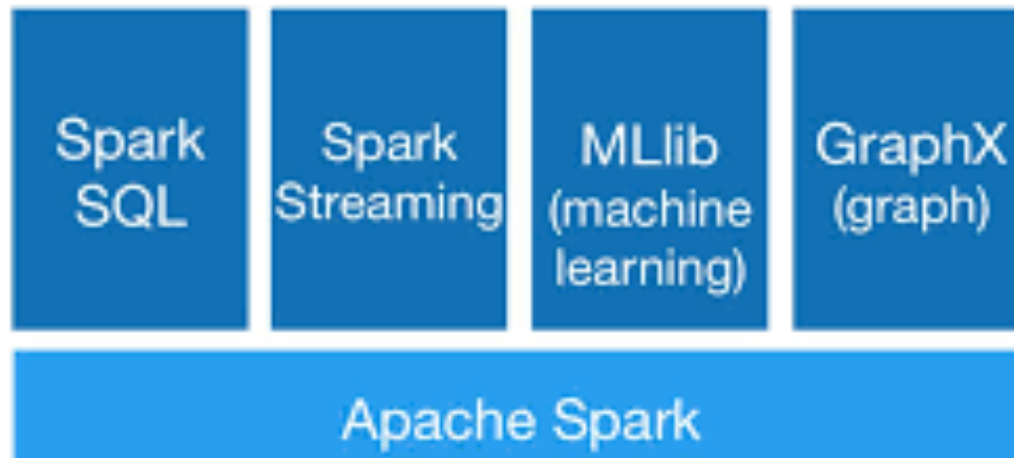
**MLlib (Machine Learning Library)**

MLlib is a distributed machine learning framework above Spark because of the distributed memory-based Spark architecture. It is, according to benchmarks, done by the MLlib developers against the Alternating Least Squares (ALS) implementations. Spark MLlib is nine times as fast as the Hadoop disk-based version ofApache Mahout (before Mahout gained a Spark interface).

# Spark Ecosystem

**GraphX**

GraphX is a distributed graph-processing framework on top of Spark. It provides an API for expressing graph computation that can model the user-defined graphs by using Pregel abstraction API. It also provides an optimised runtime for this abstraction.

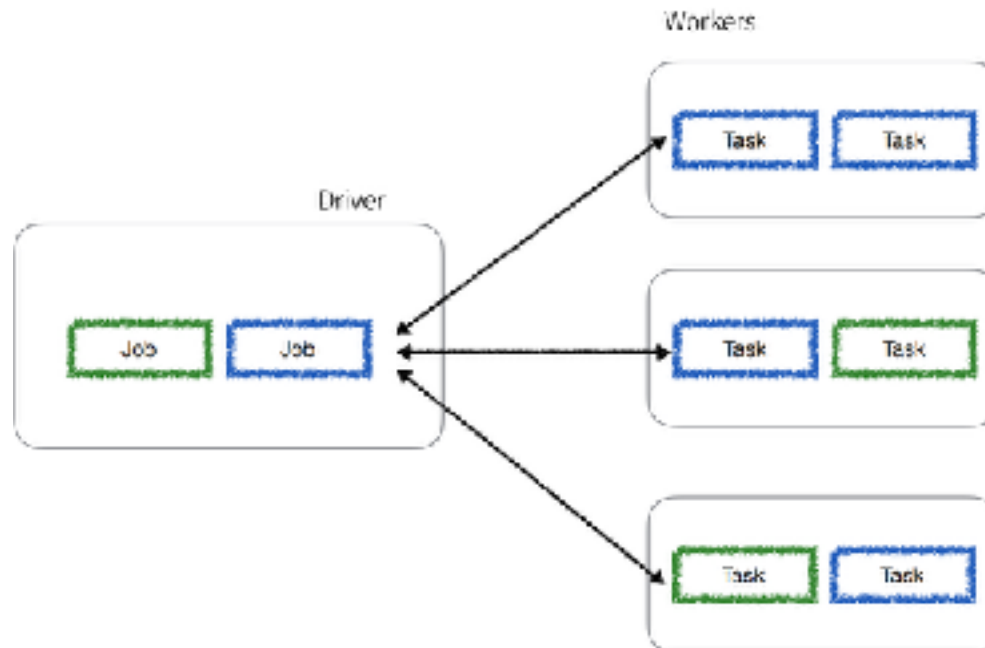


# Module 1

## Spark Jobs and APIs

# Execution process

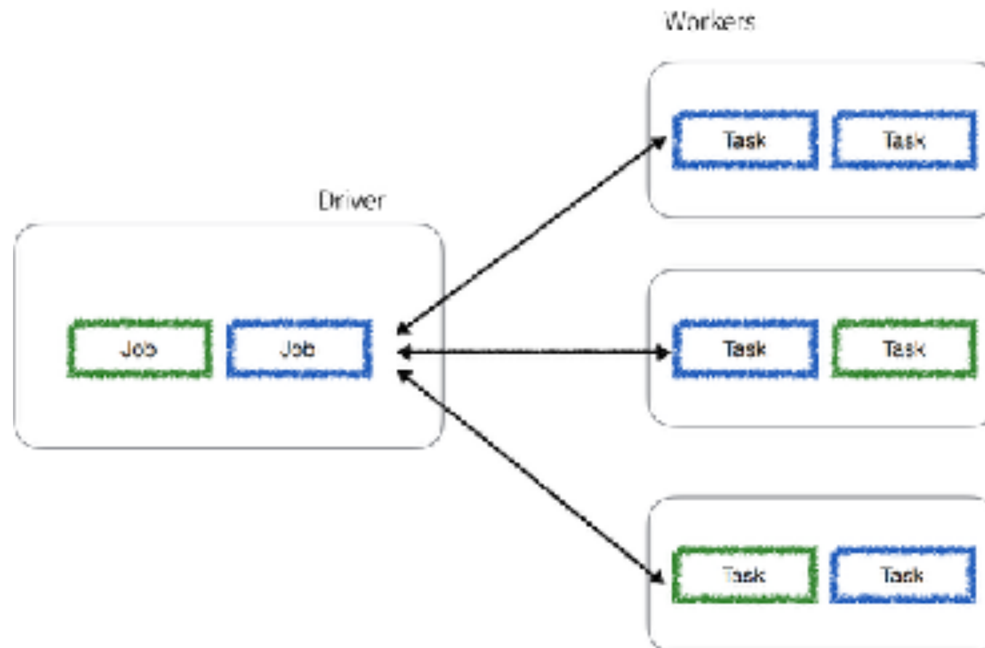
Spark application spins off a single **driver process** (that can contain **multiple jobs**) on the **master node** that then directs executor processes (that contain **multiple tasks**) distributed to a **number of worker nodes** as noted in the following diagram:



# Execution process

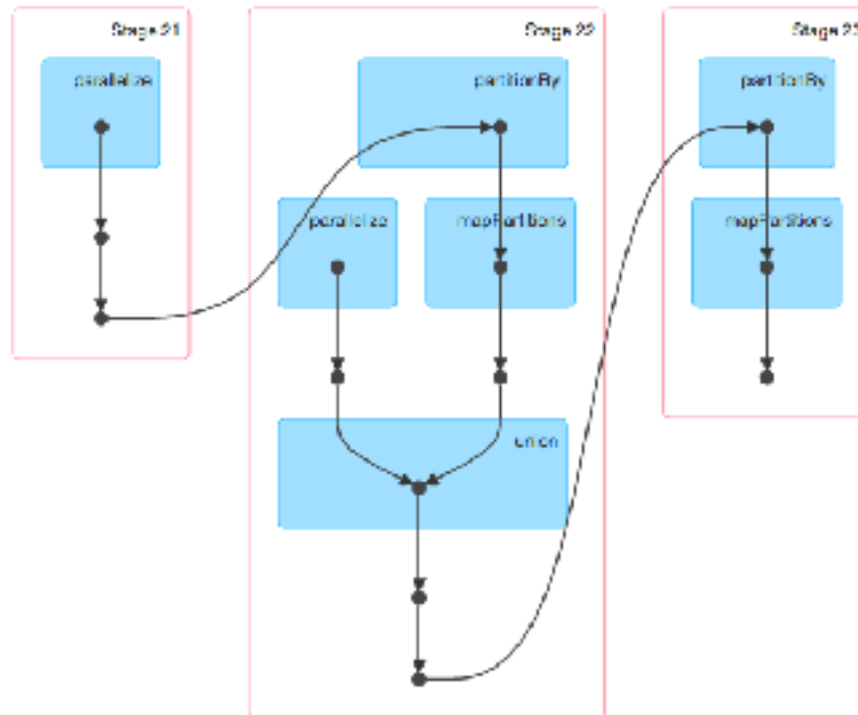
The driver process determines the number and the composition of the task processes directed to the executor nodes based on the graph generated for the given job.

Note, that any worker node can execute tasks from a number of different jobs.



# Spark Jobs

A Spark job is associated with a chain of object dependencies organized in a direct acyclic graph (DAG) such as the following example generated from the Spark UI. Given this, Spark can optimize the scheduling (for example, determine the number of tasks and workers required) and execution of these tasks:



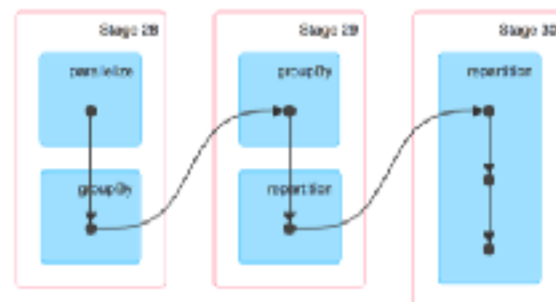
# DAG Scheduler

DAGScheduler is the scheduling layer of Apache Spark that implements stage-oriented scheduling. It transforms a logical execution plan (i.e. RDD lineage of dependencies built using RDD transformations) to a physical execution plan (using stages).

For details of DAG

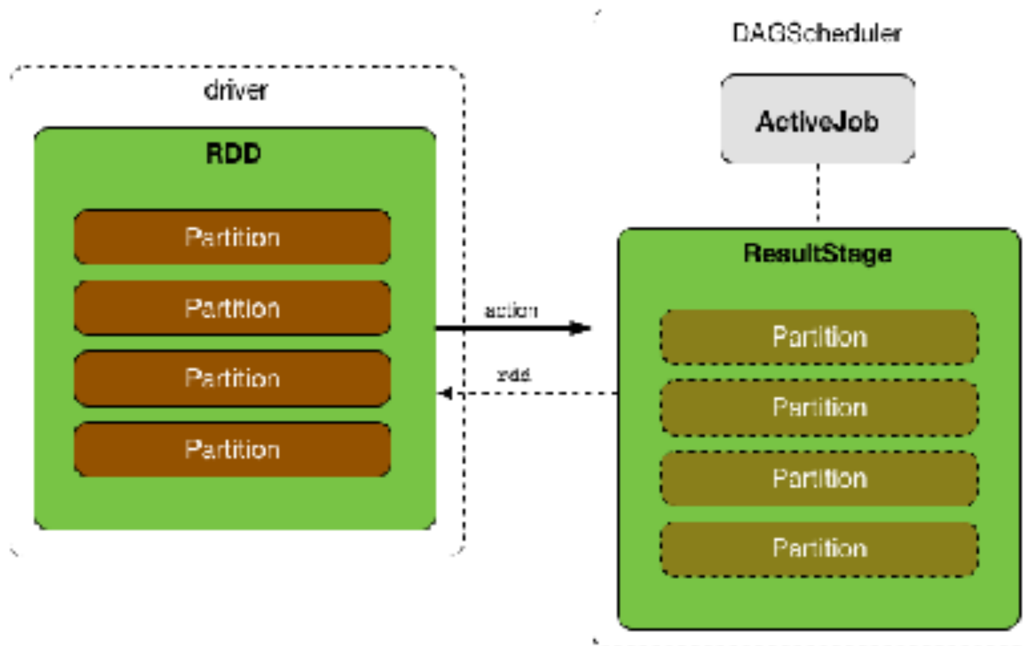
https://jaceklaskowski.gitbooks.io/mastering-apache-spark/content/spark-dagscheduler.html

DAGScheduler



# DAG Scheduler

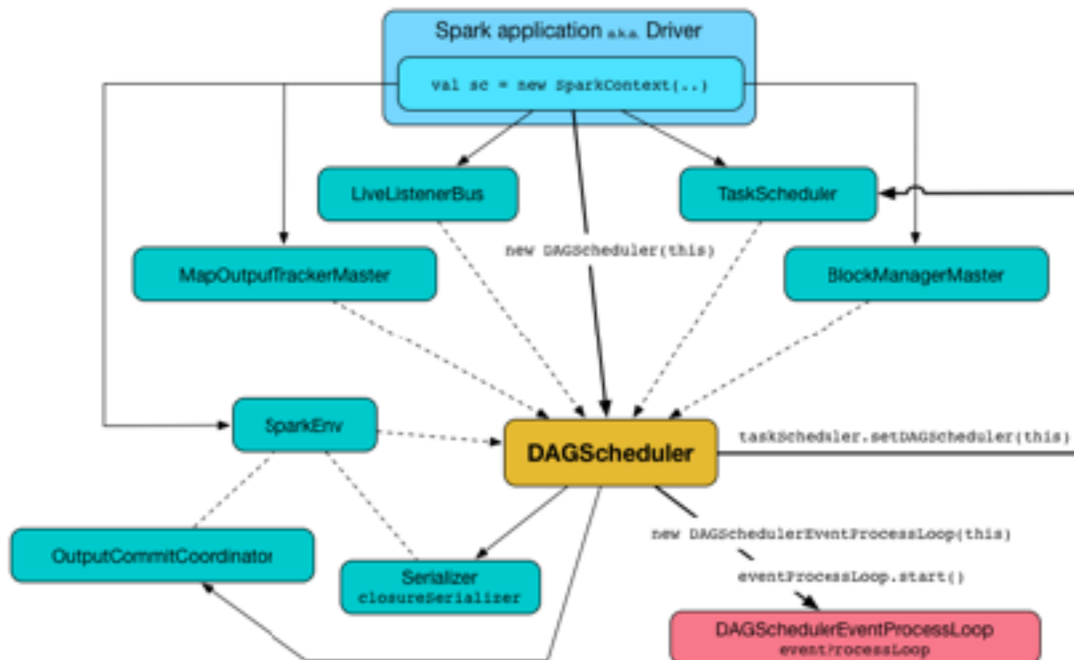
After an action has been called, SparkContext hands over a logical plan to DAGScheduler that it in turn translates to a set of stages that are submitted as TaskSets for execution.
The fundamental concepts of DAGScheduler are jobs and stages (refer to Jobs and Stages respectively) that it tracks through internal registries and counters.



# DAG Scheduler

DAGScheduler does three things in Spark as follows:

- Computes an execution DAG, i.e. DAG of stages, for a job.
- Determines the preferred locations to run each task on.
- Handles failures due to shuffle output files being lost.



# Spark Jobs and APIs

SQL

DataFrame

Logical Plan

Unresolved Logical Plan

Logical Plan

Optimized Logical Plan

Analysis

Logical Optimization

Physical Plan A

Physical Plan B

Physical Plan C

Physical Plan Z

Physical Planning

Cost Optimizer

Code Generation

Physical Pan A

RDD

# Module 1

## Spark 2.0 architecture

# Spark 2.0 Architecture

Apache Spark doesn't provide any storage (like HDFS) or any Resource Management capabilities. It is just a unified framework for processing large amount of data near to real time. In below figure, Apache Spark framework is organized in three major layers.

Resource Management
Standalone
YARN
Mesos
Spark Ecosystems
Spark SQL
Spark Streaming
BlinkDB
Spark Machine Learning
GraphX
Tachyon
Spark Core
BACK TO BAZICS
Spark DataFrame API
Java
Scala
Python
R
Spark Core

# Spark 2.0 Architecture

## Spark Core Layer:

As you can see Spark Core is the generalised layer of the framework. Spark core has the definition of all the basic functions. All other functionalities and extensions are built on top of Spark Core.

Other Language capabilities:

- Spark is totally written on Scala (a Functional as well as Object Oriented Programming Language) which runs on top of JVM
- Apart from Scala, Spark also supports languages like Java and Python
- Recently Spark has added the compatibility of statistical computing language R

# Spark 2.0 Architecture

**Spark DataFrame API:**

Spark also has real time query engine which is able to query data in a quite real time manner. To access that engine it has the DataFrame APIs in Scala, Java and Python language.

**Spark Ecosystems Layer:**

Spark Ecosystem Components are the additional libraries operating on top of Spark Core and DataFrames.

These components give the enrichment in the areas of **SQL capabilities, machine learning, real time big data computation** etc.

# Spark 2.0 Architecture

**Spark Ecosystems Layer:**

Following are the main components of Spark Ecosystem.

**Spark SQL:**

- Component on top of Spark Core with new RDD abstraction called SchemaRDD.
- Exposes Spark DataFrames through JDBC APIs and supports structured and semi-structured data
- Provides SQL like interface over DataFrames to query data in CSV, JSON, Sequence and Parquet file formats

# Spark 2.0 Architecture

**Spark Machine Learning (MLlib):**

- A common Machine Learning libraries for distributed, scalable and in memory computation
- Considerably faster than Apache Mahout in Hadoop MapReduce
- Supports common learning algorithms like dimension reduction, clustering, classification, regression, collaborative filtering etc..

**Spark Streaming:**

- Adds capability of processing streaming data near to real time
- Capable of ingesting data in micro batches (in the form of micro RDDs) and performs transformation on series of micro batches(RDDs)

# Spark 2.0 Architecture

**GraphX (Recently added):**

- Provides distributed graph processing APIs on top of Spark Core
- Allows user defined graph modeling with Pregel abstraction API

**BlinkDB (Recently added):**

- Approximate query engine over large volume data
- Allows to execute interactive SQL over the large volume data which returns approximate results
- Capable of executing queries faster with potential errors in aggregated values
- Useful in case of data insights where accuracy is not mandatory

# Spark 2.0 Architecture

**Tachyon (Recently added):**

- It is an in-memory distributed file system
- Enables faster file sharing across the cluster as there is no overhead of disk IO
- It caches frequently read file in memory so that scheduled job can read shared files directly from cache and can execute faster
- Can be used for in memory file sharing with MapReduce and Spark jobs

**Spark Resource Manager Layer:**

Apache Spark doesn't comes up with Resource Management module like YARN. It manage resource in standalone mode in single node cluster setup.

But for distributed cluster mode it can be integrated with resource management modules like YARN or Mesos.

# Spark 2.0 Architecture

# Spark 2.0 Architecture

## Spark Architecture

Spark Master

1. Request Resources

2. Allocate Resources

Spark Client

Driver

Spark Context

4. Perform Computation

Executor

Executor

Executor

3. Start Executors

Spark Worker

Spark Worker

Spark Worker

Credit: https://academy.datastax.com/courses/ds320-analytics-apache-spark/introduction-spark-architecture

# Spark 2.0 Architecture

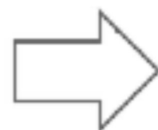
Tungsten Phase 2
speedups of 5-20x

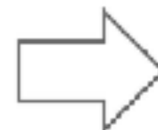
Structured Streaming

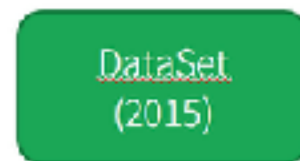
SQL 2003
& Unifying Datasets
and DataFrames

# Spark Jobs and APIs

## History of Spark APIs








Distribute collection of JVM objects

Functional Operators (map, filter, etc.)

Distribute collection of Row objects

Expression-based operations and UDFs

Logical plans and optimizer

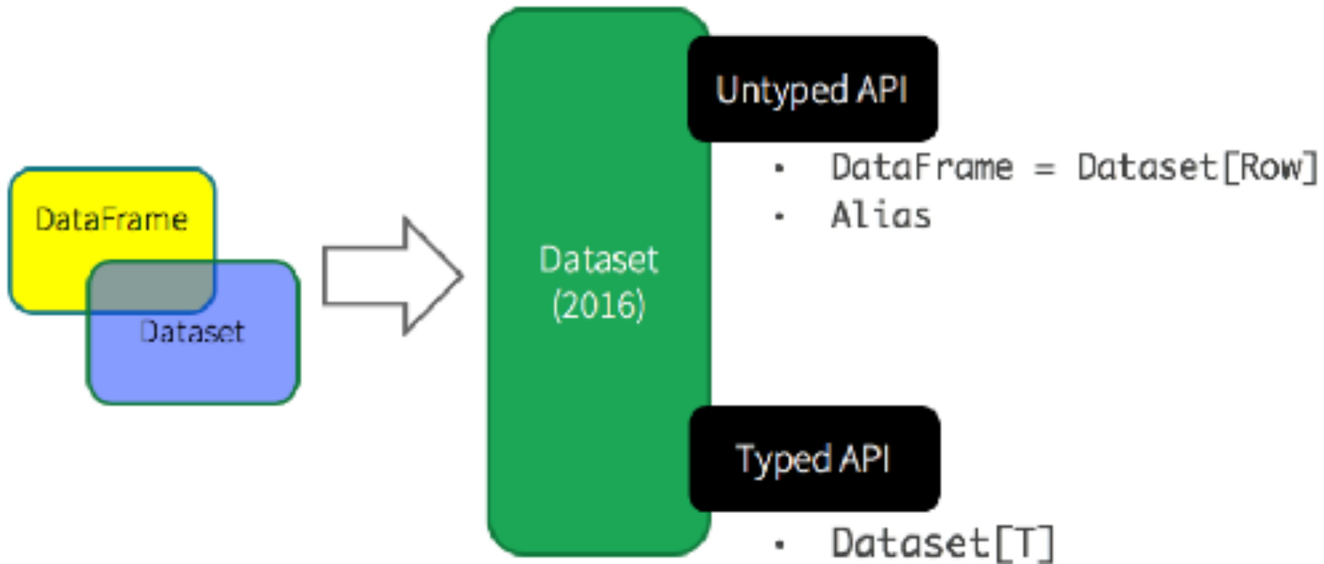
Fast/efficient internal representations

Internally rows, externally JVM objects

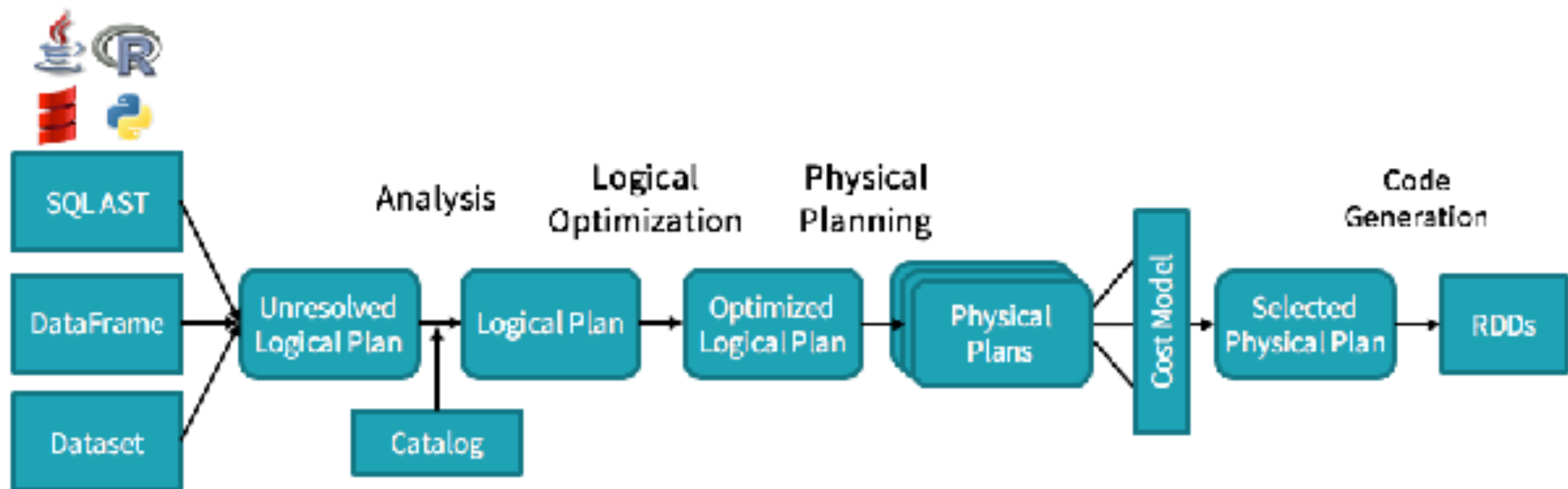
Almost the "Best of both worlds": type safe + fast

But slower than DF
Not as good for interactive analysis, especially Python

# Spark Jobs and APIs



# Spark Jobs and APIs




DataFrames, Datasets and SQL
share the same optimization/execution pipeline

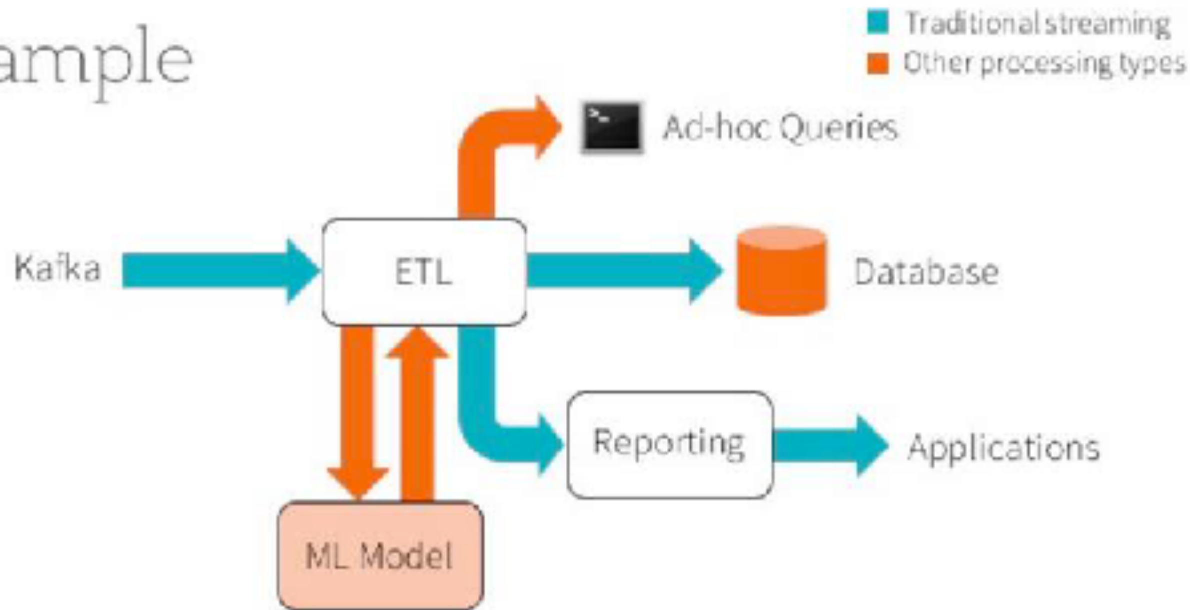
# Spark Jobs and APIs

Spark 1.3
Static DataFrames

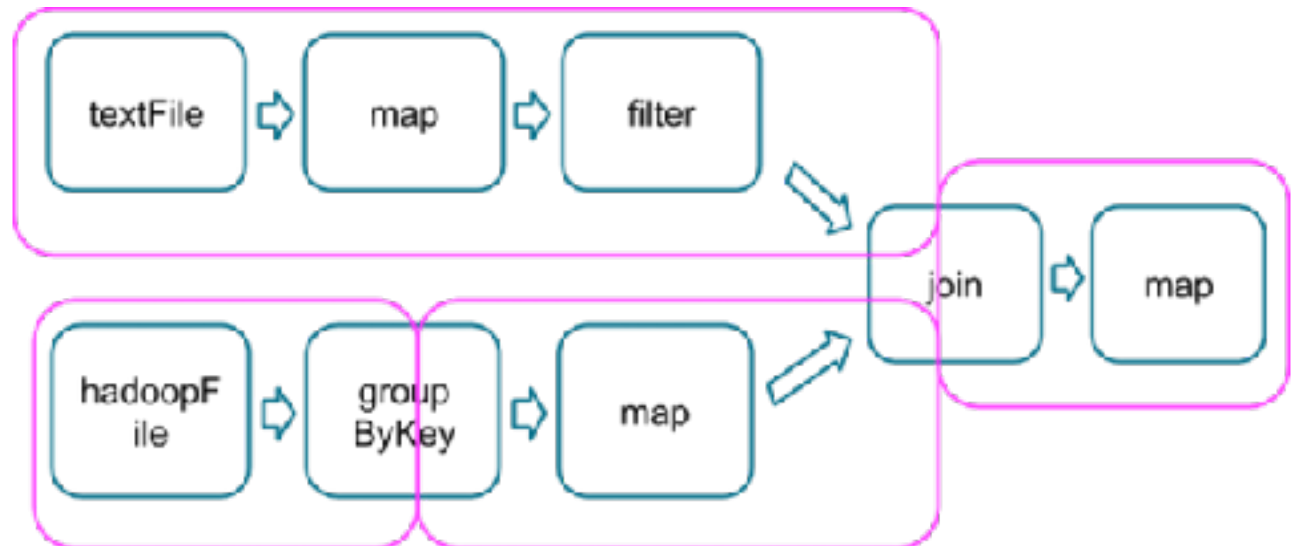
Spark 2.0
Infinite DataFrames

Single API !

# Spark Jobs and APIs



# Spark Jobs and APIs



# PySpark Installation on Windows

Prerequisites: Anaconda and GOW. If you already have anaconda and GOW installed, skip to step 5.

1. Download and install Gnu on windows (GOW) from the following link. Basically, GOW allows you to use linux commands on windows. In this install, we will need curl, gzip, tar which GOW provides.

```
C:\Users\mgalarny>gow --list
Available executables:

  awk, basename, bash, bc, bison, bunzip2, bzip2, bzip2recover, cat,
  chgrp, chmod, chown, chroot, cksum, clear, cp, csplit, curl, cut, dc,
  dd, df, diff, diff3, dirname, dos2unix, du, egrep, env, expand, expr,
  factor, fgrep, flex, fmt, fold, gawk, gfind, gow, grep, gsar, gsort,
  gzip, head, hostid, hostname, id, indent, install, join, jwhois, less,
  lesskey, ln, ls, m4, make, md5sum, mkdir, mkfifo, mknod, mv, nano,
  ncftp, nl, od, pageant, paste, patch, pathchk, plink, pr, printenv,
  printf, pscp, psftp, putty, puttygen, pwd, rm, rmdir, scp, sdiff, sed,
  seq, sftp, sha1sum, shar, sleep, split, ssh, su, sum, sync, tac, tail,
  tar, tee, test, touch, tr, uname, unexpand, uniq, unix2dos, unlink,
  unrar, unshar, uudecode, uuencode, vim, wc, wget, whereis, which,
  whoami, xargs, yes, zip
```

# PySpark Installation on Windows

2. Download and install Anaconda (windows version) from https://www.continuum.io/downloads

**Anaconda 4.4.0**

For Windows

Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

Changelog

1. Download the installer
2. Optional: Verify data integrity with MD5 or SHA-256  More info
3. Double-click the .exe file to install Anaconda and follow the instructions on the screen

Behind a firewall? Use these zipped Windows installers

# PySpark Installation on Windows

3.Select the default options when prompted during the installation of Anaconda.

4. Close and open a new command line (CMD).

5. Go to the Apache Spark website (link)

## Download Apache Spark™

1. Choose a Spark release: 2.1.0 (Dec 28 2016)
2. Choose a package type: Pre-built for Hadoop 2.7 and later
3. Choose a download type: Direct Download
4. Download Spark: spark-2.1.0-bin-hadoop2.7.tgz
5. Verify this release using the 2.1.0 signatures and checksums and project release KEYS.

Download Apache Spark
a) Choose a Spark release

b) Choose a package type

c) Choose a download type: (Direct Download)

d) Download Spark

# PySpark Installation on Windows

6. Move the file to where you want to unzip it.

mkdir C:\opt\spark

mv C:\Users\mgalarny\Downloads\spark-2.1.0-bin-hadoop2.7.tgz C:\opt\spark\spark-2.1.0-bin-hadoop2.7.tgz

7. Unzip the file. Use the bolded commands below

gzip -d spark-2.1.0-bin-hadoop2.7.tgz

tar xvf spark-2.1.0-bin-hadoop2.7.tar

8. Download winutils.exe into your spark-2.1.0-bin-hadoop2.7\bin

curl -k -L -o winutils.exe https://github.com/steveloughran/winutils/blob/master/hadoop-2.6.0/bin/winutils.exe?raw=true

# PySpark Installation on Windows

9. Make sure you have Java 7+ installed on your machine.

10. Next, we will edit our environmental variables so we can open a spark notebook in any directory.

setx SPARK_HOME C:\opt\spark\spark-2.1.0-bin-hadoop2.7

setx HADOOP_HOME C:\opt\spark\spark-2.1.0-bin-hadoop2.7

setx PYSPARK_DRIVER_PYTHON ipython

setx PYSPARK_DRIVER_PYTHON_OPTS notebook

Add ;C:\opt\spark\spark-2.1.0-bin-hadoop2.7\bin to your path.

Notes on the setx command: https://ss64.com/nt/set.html

# PySpark Installation on Windows

11. Close your terminal and open a new one. Type the command below.

pyspark local

```
C:\Users\mgalarny\Desktop>pyspark --master local[2]
```

Notes: The PYSPARK_DRIVER_PYTHON parameter and the PYSPARK_DRIVER_PYTHON_OPTS parameter are used to launch the PySpark shell in Jupyter Notebook.

The — master parameter is used for setting the master node address. Here we launch Spark locally on 2 cores for local testing.

# Module 2

## Resilient Distributed Datasets

# Resilient Distributed Datasets

RDD is a fundamental data structure of Spark. It is an immutable distributed collection of JVM objects.

Each dataset in RDD is divided into logical partitions, which may be computed on different nodes of the cluster.

RDDs can contain any type of Python, Java, or Scala objects, including user-defined classes.

Formally, an RDD is a read-only, partitioned collection of records.

RDDs can be created through deterministic operations on either data on stable storage or other RDDs.

RDD is a fault-tolerant collection of elements that can be operated on in parallel.

# Resilient Distributed Datasets (RDD)

RDD of Strings

Immutable **Collection** of Objects

**Partitioned** and **Distributed**

Stored in **Memory**

Partitions **Recomputed on Failure**

# Spark RDD

There are two ways to create RDDs: parallelizing an existing collection in your driver program, or referencing a dataset in an external storage system, such as a shared file system, HDFS, HBase, or any data source offering a Hadoop Input Format.

Spark makes use of the concept of RDD to achieve faster and efficient MapReduce operations. Let us first discuss how MapReduce operations take place and why they are not so efficient.

# Spark RDD

Data sharing is slow in MapReduce due to replication, serialisation, and disk IO. Most of the Hadoop applications, they spend more than 90% of the time doing HDFS read- write operations.

Recognising this problem, researchers developed a specialised framework called Apache Spark.

The key idea of spark is Resilient Distributed Datasets (RDD); it supports in- memory processing computation.

This means, it stores the state of memory as an object across the jobs and the object is sharable between those jobs. Data sharing in memory is 10 to 100 times faster than network and Disk.

Let us now try to find out how iterative and interactive operations take place in Spark RDD.

# Spark RDD

The transformations to the dataset are lazy.

This means that any transformation is only executed when an action on a dataset is called.

This helps Spark to optimize the execution.

For instance, consider the following very common steps that an analyst would normally do to get familiar with a dataset:

- Count the occurrence of distinct values in a certain column.
- Select those that start with an A.
- Print the results to the screen.

# Spark RDD

First, we order Spark to map the values of A using the .map(lambda v: (v, 1)) method, and then select those records that start with an 'A' (using the .filter(lambda val: val.startswith('A')) method).

If we call the .reduceByKey(operator.add) method it will reduce the dataset and add (in this example, count) the number of occurrences of each key.

All of these steps **transform the dataset.**

Second, we call the .collect() method to execute the steps.

This step is an **action on our dataset** - it finally counts the distinct elements of the dataset.
In effect, the action might reverse the order of transformations and filter the data first before mapping, resulting in a smaller dataset being passed to the reducer.

# Creating RDD

**There are two ways to create an RDD in PySpark:**

1. you can either .parallelize(...) a collection (list or an array of some elements):

data = sc.parallelize(

[('Amber', 22), ('Alfred', 23), ('Skye',4), ('Albert', 12),

('Amber', 9)])

**Or** you can reference a file (or files) located either locally or somewhere externally:

data_from_file = sc.\

textFile( '/Users/SurendraMac/Documents/PySpark_Data/VS14MORT.txt.gz', 4)

The last parameter in sc.textFile(..., n) specifies the **number of partitions** the dataset is divided into.

# Creating RDD

Note:

We downloaded the Mortality dataset VS14MORT.txt file from

ftp://ftp.cdc.gov/pub/Health_Statistics/NCHS/Datasets/DVS/mortality/mort2014us.zip;

the record schema is explained in this document

http://www.cdc.gov/nchs/ data/dvs/Record_Layout_2014.pdf.

We selected this dataset on purpose: For your convenience, we also host the file here: http://tomdrabas.com/data/VS14MORT. txt.gz

# Creating RDD

A rule of thumb would be to break your dataset into two-four partitions for each in your cluster.

Spark can read from a multitude of filesystems: Local ones such as NTFS, FAT, or Mac OS Extended (HFS+), or distributed filesystems such as HDFS, S3, Cassandra, among many others.

Be wary where your datasets are read from or saved to:

The path cannot contain special characters [].

Note, that this also applies to paths stored on Amazon S3 or Microsoft Azure Data Storage.

# Creating RDD

Multiple data formats are supported: Text, parquet, JSON, Hive tables, and data from relational databases can be read using a JDBC driver.

Note that Spark can automatically work with compressed datasets (like the Gzipped one in our preceding example).

Depending on how the data is read, the object holding it will be represented slightly differently.

The data read from a file is represented as MapPartitionsRDD instead of ParallelCollectionRDD when we .paralellize(...) a collection.

# Schema

RDDs are **schema-less** data structures (unlike DataFrames).

Thus, parallelizing a dataset, such as in the following code snippet, is perfectly fine with Spark when using RDDs:

```
data_heterogenous = sc.parallelize([
('Ferrari', 'fast'),
{'Porsche': 100000},
['Spain','visited', 4504]
]).collect()
```

So, we can mix almost anything: a tuple, a dict, or a list and Spark will not complain.

# Schema

Once you .collect() the dataset (that is, run an action to bring it back to the driver) you can access the data in the object as you would normally do in Python:

**data_heterogenous[1]['Porsche']**

It will produce the following:

100000

The **.collect()** method returns all the elements of the RDD to the driver where it is serialized as a list.

# Reading from files

When you read from a text file, each row from the file forms an element of an RDD.

The data_from_file.take(1) command will produce the following

(somewhat unreadable) output:

```
Out[7]: ['                    1
        2101  M1087 432311  4M4                2014U7CN
        I64 238 070   24 0111I64
        01 I64
        01  11                                 100 601']
```

To make it more readable, let's create a list of elements so each line is represented as a list of values.

# Reading from files

Lambda expressions

In this example, we will extract the useful information from the cryptic looking record of data_from_file.

```
data_from_file_conv = data_from_file.map(extractInformation)
data_from_file_conv.map(lambda row: row).take(1)
```

# Global versus local scope

Spark can be run in two modes:

Local and cluster. When you run Spark locally your code might not differ to what you are currently used to with running Python: Changes would most likely be more syntactic than anything else but with an added twist that data and code can be copied between separate worker processes.

In the cluster mode, when a job is submitted for execution, the job is sent to the driver (or a master) node.

The driver node creates a DAG for a job and decides which executor (or worker) nodes will run specific tasks.

The driver then instructs the workers to execute their tasks and return the results to the driver when done.

Before that happens, however, the driver prepares each task's closure: A set of variables and methods present on the driver for the worker to execute its task on the RDD.

# Transformations

## The .map(...) transformation

It can be argued that you will use the .map(...) transformation most often. The method is applied to each element of the RDD: In the case of the data_from_file_ conv dataset, you can think of this as a transformation of each row.

In this example, we will create a new dataset that will convert year of death into a numeric value:

```
data_2014 = data_from_file_conv.map(lambda row: int(row[16]))
```

Running data_2014.take(10) will yield the following result:

```
Out[11]: [2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, 2014, -99]
```

# Transformations

**The .map(...) transformation**

You can of course bring more columns over, but you would have to package them into a tuple, dict, or a list. Let's also include the 17th element of the row along so that we can confirm our .map(...) works as intended:

data_2014_2 = data_from_file_conv.map(

lambda row: (row[16], int(row[16]):)

data_2014_2.take(5)

The preceding code will produce the following result:

```
Out[12]: [('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('2014', 2014),
          ('-99', -99)]
```

# Transformations

**The .filter(...) transformation**

It allows you to select elements from your dataset that fit specified criteria.

As an example, from the data_from_file_conv dataset, let's count how many people died in an accident in 2014:

```
data_filtered = data_from_file_conv.filter( lambda row: row[16] == '2014' and row[21] == '0')
```

```
data_filtered.count()
```

# Transformations

**The .flatMap(...) transformation**

The .flatMap(...) method works similarly to .map(...), but it returns a flattened result instead of a list. If we execute the following code:

```
data_2014_flat = data_from_file_conv.flatMap(lambda row: (row[16], int(row[16]) + 1))
```

```
data_2014_flat.take(10)
```

It will yield the following output:

```
Out[14]: ['2014', 2015, '2014', 2015, '2014', 2015, '2014', 2015,
          '2014', 2015]
```

# Transformations

**The .distinct(...) transformation**

This method returns a list of distinct values in a specified column. It is extremely useful if you want to get to know your dataset or validate it. Let's check if the gender column contains only males and females; that would verify that we parsed the dataset properly. Let's run the following code:

```
distinct_gender = data_from_file_conv.map( lambda row: row[5]).distinct()

distinct_gender.collect()
```

This code will produce the following output:

```
Out[22]: ['-99', 'M', 'F']
```

First, we extract only the column that contains the gender.

Next, we use the .distinct() method to select only the distinct values in the list.

Lastly, we use the .collect() method to return the print of the values on the screen.

# Transformations

**The .sample(...) transformation**

The .sample(...) method returns a randomized sample from the dataset. The first parameter specifies whether the sampling should be with a replacement, the second parameter defines the fraction of the data to return, and the third is seed to the pseudo-random numbers generator:

```
fraction = 0.1

data_sample = data_from_file_conv.sample(False, fraction, 666)
```

In this example, we selected a randomized sample of 10% from the original dataset. To confirm this, let's print the sizes of the datasets:

```
print('Original dataset: {0}, sample: {1}'\

.format(data_from_file_conv.count(), data_sample.count()))
```

The preceding command produces the following output:

```
Original dataset: 2631171, sample: 263247
```

# Transformations

**The .leftOuterJoin(...) transformation**

.leftOuterJoin(...), just like in the SQL world, joins two RDDs based on the values found in both datasets, and returns records from the left RDD with records from the right one appended in places where the two RDDs match:

rdd1 = sc.parallelize([('a', 1), ('b', 4), ('c',10)])

rdd2 = sc.parallelize([('a', 4), ('a', 1), ('b', '6'), ('d', 15)])

rdd3 = rdd1.leftOuterJoin(rdd2)

Running .collect(...) on the rdd3 will produce the following:

```
Out[52]: [('c', (10, None)), ('b', (4, '6')), ('a', (1, 4)), ('a', (1, 1))]
```

# Transformations

## The .repartition(...) transformation

Repartitioning the dataset changes the number of partitions that the dataset is divided into. This functionality should be used sparingly and only when really necessary as it shuffles the data around, which in effect results in a significant hit in terms of performance:

```
rdd1 = rdd1.repartition(4)
len(rdd1.glom().collect())
```

The preceding code prints out 4 as the new number of partitions.

The .glom() method, in contrast to .collect(), produces a list where each element is another list of all elements of the dataset present in a specified partition; the main list returned has as many elements as the number of partitions.

# Actions

Actions, in contrast to transformations, execute the scheduled task on the dataset; once you have finished transforming your data you can execute your transformations.

**The .take(...) method**

This is most arguably the most useful (and used, such as the .map(...) method). The method is preferred to .collect(...) as it only returns the n top rows from a single data partition in contrast to .collect(...), which returns the whole RDD. This is especially important when you deal with large datasets:

data_first = data_from_file_conv.take(1)

If you want somewhat randomized records you can use .takeSample(...) instead, which takes three arguments: First whether the sampling should be with replacement, the second specifies the number of records to return, and the third is a seed to the pseudo-random numbers generator:

data_take_sampled = data_from_file_conv.takeSample(False, 1, 667)

# Actions

**The .collect(...) method**

This method returns all the elements of the RDD to the driver. As we have just provided a caution about it, we will not repeat ourselves here.

**The .reduce(...) method**

The .reduce(...) method reduces the elements of an RDD using a specified method.

You can use it to sum the elements of your RDD:

```
rdd1.map(lambda row: row[1]).reduce(lambda x, y: x + y)
```

This will produce the sum of 15.

# Actions

We first create a list of all the values of the rdd1 using the .map(...) transformation, and then use the .reduce(...) method to process the results. The reduce(...) method, on each partition, runs the summation method (here expressed as a lambda) and returns the sum to the driver node where the final aggregation takes place.

The .reduceByKey(...) method works in a similar way to the .reduce(...) method, but it performs a reduction on a key-by-key basis:

```
data_key = sc.parallelize(

[('a', 4),('b', 3),('c', 2),('a', 8),('d', 2),('b', 1),

('d', 3)],4)

data_key.reduceByKey(lambda x, y: x + y).collect()
```

The preceding code produces the following:

```
Out[122]: [('b', 4), ('c', 2), ('a', 12), ('d', 5)]
```

# Actions

**The .count(...) method**

The .count(...) method counts the number of elements in the RDD. Use the following code:

```
data_reduce.count()
```

This code will produce 6, the exact number of elements in the data_reduce RDD.

The .count(...) method produces the same result as the following method, but it does not require moving the whole dataset to the driver:

```
len(data_reduce.collect()) # WRONG -- DON'T DO THIS!
```

```
Out[132]: dict_items([('a', 2), ('b', 2), ('d', 2), ('c', 1)])
```

# Actions

**The .saveAsTextFile(...) method**

As the name suggests, the .saveAsTextFile(...) the RDD and saves it to text files: Each partition to a separate file:

```
data_key.saveAsTextFile( '/Users/drabast/Documents/PySpark_Data/data_key.txt')
```

To read it back, you need to parse it back as all the rows are treated as strings:

```
def parseInput(row):
    import re
    pattern = re.compile(r'\(\'([a-z])\', ([0-9])\)')
    row_split = pattern.split(row)
    return (row_split[1], int(row_split[2]))
data_key_reread = sc .textFile( '/Users/drabast/Documents/PySpark_Data/data_key.txt').map(parseInput)
data_key_reread.collect()
```

The list of keys read matches what we had initially:

```
Out[159]: [('a', 4), ('b', 3), ('c', 2), ('a', 8), ('d', 2), ('b', 1), ('d', 3)]
```

# Actions

**The .foreach(...) method**

This is a method that applies the same function to each element of the RDD in an iterative way; in contrast to .map(..), the .foreach(...) method applies a defined function to each record in a one-by-one fashion.

It is useful when you want to save the data to a database that is not natively supported by PySpark.

Here, we'll use it to print (to CLI - not the Jupyter Notebook) all the records that are stored in data_key RDD:

```
def f(x):

print(x)
```

# Iterative Operations on Spark RDD

The illustration given below shows the iterative operations on Spark RDD. It will store intermediate results in a distributed memory instead of Stable storage (Disk) and make the system faster.

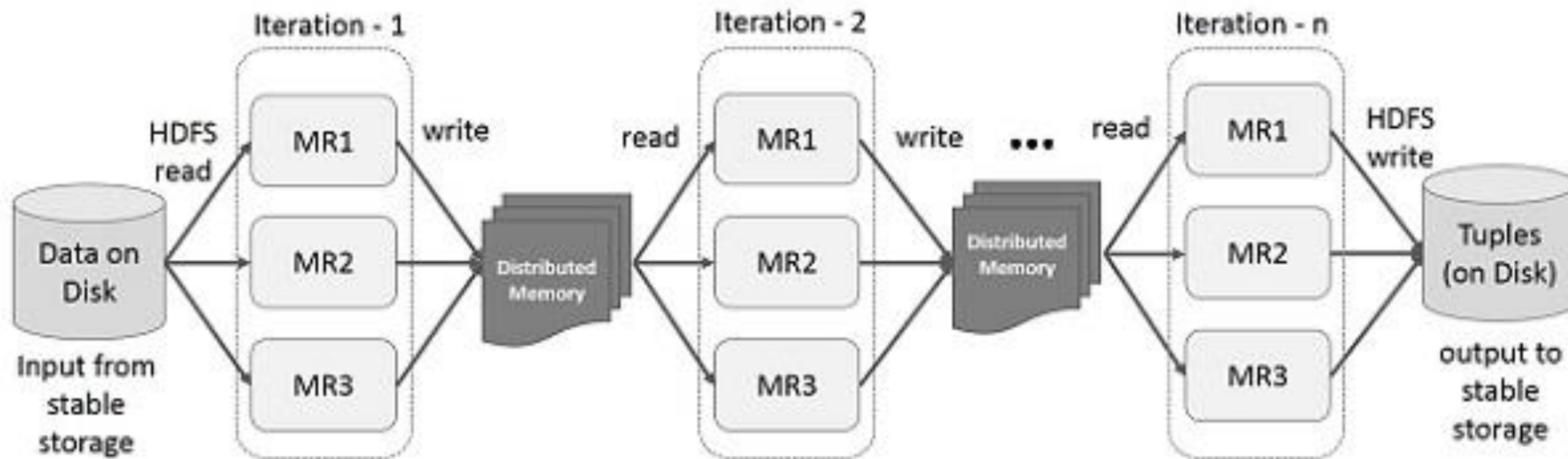
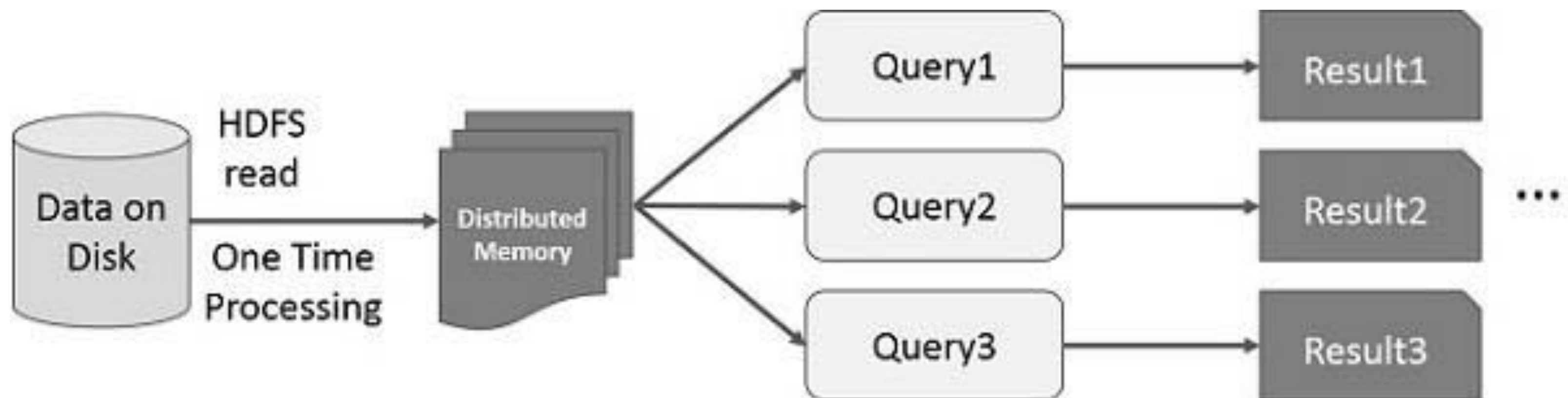



**Figure**: Iterative operations on Spark RDD

# Interactive Operations on Spark RDD

This illustration shows interactive operations on Spark RDD. If different queries are run on the same set of data repeatedly, this particular data can be kept in memory for better execution times.



# Spark Shell

Spark provides an interactive shell: a powerful tool to analyze data interactively.

It is available in either Scala or Python language.

Spark's primary abstraction is a distributed collection of items called a Resilient Distributed Dataset (RDD).

RDDs can be created from Hadoop Input Formats (such as HDFS files) or by transforming other RDDs.

## Open Spark Shell

The following command is used to open Spark shell.

**$ spark-shell**

```
scala> val inputfile = sc.textFile("input.txt")
```

# Module 2

## Resilient Distributed Datasets

# RDD Transformations

The Spark RDD API introduces few Transformations and few Actions to manipulate RDD.

## RDD Transformations

RDD transformations returns pointer to new RDD and allows you to create dependencies between RDDs.

Each RDD in dependency chain (String of Dependencies) has a function for calculating its data and has a pointer (dependency) to its parent RDD.

# RDD Transformations

Spark is lazy, so nothing will be executed unless you call some transformation or action that will trigger job creation and execution.

Therefore, RDD transformation is not a set of data but is a step in a program (might be the only step) telling Spark how to get data and what to do with it.

| Transformations | Actions |
| --- | --- |
| map(func) | take(N) |
| flatMap(func) | count() |
| filter(func) | collect() |
| groupByKey() | reduce(func) |
| reduceByKey(func) | takeOrdered(N) |
| mapValues(func) | top(N) |
| ... | ... |

# Iterative Operations on Spark RDD

Create RDD

Transformation

RDD

Lineage

Action

Result

# RDD Transformations

| Sr. | Transformation and Meaning |
|---|---|
| 1 | **map(func)**<br><br>Returns a new distributed dataset, formed by passing each element of the source through a function func. |
| 2 | **filter(func)**<br><br>Returns a new dataset formed by selecting those elements of the source on which func returns true. |
| 3 | **flatMap(func)**<br><br>Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item). |
| 4 | **mapPartitions(func)**<br><br>Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator<T> => Iterator<U> when running on an RDD of type T. |

# RDD Transformations

| Sr. | Transformation and Meaning |
| --- | --- |
| 5 | **mapPartitionsWithIndex(func)**<br><br>Similar to map Partitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator<T>) => Iterator<U> when running on an RDD of type T. |
| 6 | **sample(withReplacement, fraction, seed)**<br><br>Sample a fraction of the data, with or without replacement, using a given random number generator seed. |
| 7 | **union(otherDataset)**<br><br>Returns a new dataset that contains the union of the elements in the so dataset and the argument. |

# RDD Transformations

| Sr. | Transformation and Meaning |
|---|---|
| 8 | **intersection(otherDataset)**<br><br>Returns a new RDD that contains the intersection of elements in the source dataset and the argument. |
| 9 | **distinct([numTasks]))**<br><br>Returns a new dataset that contains the distinct elements of the source dataset. |
| 10 | **groupByKey([numTasks])**<br><br>When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable<V>) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. |

# RDD Transformations

| Sr. | Transformation and Meaning |
|---|---|
| 11 | **reduceByKey(func, [numTasks])**<br><br>When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V, V) => V. |
| 12 | **aggregateByKey(zeroValue)(seqOp, combOp, [numTasks])**<br><br>When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different from the input value type, while avoiding unnecessary allocations. |
| 13 | **sortByKey([ascending], [numTasks])**<br><br>When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the Boolean ascending argument. |

# RDD Transformations

| Sr. | Transformation and Meaning |
|---|---|
| 14 | **join(otherDataset, [numTasks])**<br><br>When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. |
| 15 | **cogroup(otherDataset, [numTasks])**<br><br>When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable<V>, Iterable<W>)) tuples. This operation is also called group With. |
| 16 | **cartesian(otherDataset)**<br><br>When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). |

# RDD Transformations

| Sr. | Transformation and Meaning |
|---|---|
| 17 | **pipe(command, [envVars])**<br><br>Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. |
| 18 | **coalesce(numPartitions)**<br><br>Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. |
| 19 | **repartition(numPartitions)**<br><br>Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. |
| 20 | **epartitionAndSortWithinPartitions(partitioner)**<br><br>Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. |

# Module 2

## RDD Actions

# RDD Actions

| Sr. | Actions and Meaning |
| --- | --- |
| 1 | reduce(func)<br><br>Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. |
| 2 | collect()<br><br>Returns 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. |
| 3 | count()<br>Returns the number of elements in the dataset. |
| 4 | first()<br>Returns the first element of the dataset (similar to take (1)). |

# RDD Actions

| Sr. | Actions and Meaning |
|---|---|
| 5 | take(n)<br>Returns an array with the first n elements of the dataset. |
| 6 | takeSample (withReplacement,num, [seed])<br><br>Returns an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. |
| 7 | takeOrdered(n, [ordering])<br><br>Returns the first n elements of the RDD using either their natural order or a custom comparator. |
| 8 | saveAsTextFile(path)<br><br>Writes the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark calls toString on each element to convert it to a line of text in the file. |

# RDD Actions

| Sr. | Actions and Meaning |
|---|---|
| 9 | saveAsSequenceFile(path) (Java and Scala)<br><br>Writes 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 available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). |
| 10 | saveAsObjectFile(path) (Java and Scala)<br><br>Writes the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile(). |
| 11 | countByKey()<br><br>Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. |
| 12 | foreach(func)<br><br>Runs a function func on each element of the dataset. This is usually, done for side effects such as updating an Accumulator or interacting with external storage systems. |

# Programming with RDD

Let us see the implementations of few RDD transformations and actions in RDD programming with the help of an example.

Example

Consider a word count example: It counts each word appearing in a document. Consider the

input.txt: input file.

people are not as beautiful as they look, as they walk or as they talk.they are only as beautiful as they love, as they care as they share.

# Programming with RDD

Follow the procedure given below to execute the given example.

**Open Spark-Shell**

The following command is used to open spark shell. Generally, spark is built using Scala. Therefore, a Spark program runs on Scala environment.

```
$ spark-shell
```

# Programming with RDD

Look at the last line of the output “Spark context available as sc” means the Spark container is automatically created spark context object with the name sc. Before starting the first step of a program, the SparkContext object should be created.

## Create an RDD

First, we have to read the input file using Spark-Scala API and create an RDD.

The following command is used for reading a file from given location. Here, new RDD is created with the name of inputfile. The String which is given as an argument in the textFile(“”) method is absolute path for the input file name. However, if only the file name is given, then it means that the input file is in the current location.

```
scala> val inputfile = sc.textFile("input.txt")
```

# Programming with RDD

## Execute Word count Transformation

- Our aim is to count the words in a file.
- Create a flat map for splitting each line into words (flatMap(line => line.split(“ ”)).
- Next, read each word as a key with a value ‘1’ (<key, value> = <word,1>)
- using map function (map(word => (word, 1)).

# Programming with RDD

Finally, reduce those keys by adding values of similar keys (reduceByKey(_+_)).

The following command is used for executing word count logic. After executing this, you will not find any output because this is not an action, this is a transformation; pointing a new RDD or tell spark to what to do with the given data)

```
scala> val counts = inputfile.flatMap(line => line.split(" ")).map(word => (word, 1)).reduceByKey(_+_);
```

# Programming with RDD

## Current RDD

While working with the RDD, if you want to know about current RDD, then use the following command. It will show you the description about current RDD and its dependencies for debugging.

```
scala> counts.toDebugString
```

## Caching the Transformations

You can mark an RDD to be persisted using the persist() or cache() methods on it. The first time it is computed in an action, it will be kept in memory on the nodes. Use the following command to store the intermediate transformations in memory.

```
scala> counts.cache()
```

# Programming with RDD

## Applying the Action

Applying an action, like store all the transformations, results into a text file. The String argument for **saveAsTextFile(" ")** method is the absolute path of output folder.

Try the following command to save the output in a text file.

In the following example, 'output' folder is in current location.

```
scala> counts.saveAsTextFile("output")
```

# Programming with RDD

**Checking the Output**

Open another terminal to go to home directory (where spark is executed in the other terminal). Use the following commands for checking output directory.

**[hadoop@localhost ~]$ cd output/**

**[hadoop@localhost output]$ ls -1**

part-00000
part-00001
_SUCCESS

The following command is used to see output from Part-00000 files.

**[hadoop@localhost output]$ cat part-00000**

# Programming with RDD

You will see the Output

(people,1)
(are,2)
(not,1)
(as,8)
(beautiful,2)
(they, 7)
(look,1)

# Programming with RDD

**UN Persist the Storage**

Before UN-persisting, if you want to see the storage space that is used for this application, then use the following URL in your browser.

**http://localhost:4040**

**Scala> counts.unpersist()**

verify after unpersist

**http://localhost:4040**

# Programming with RDD

**SparkWordCount.scala**

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

object SparkWordCount {
  def main(args: Array[String]) {
    val sc = new SparkContext( "local", "Word Count", "/usr/local/spark",
Nil,Map(), Map())
/* local = master URL; Word Count = application name; */
/* /usr/local/spark = Spark Home; Nil = jars; Map = environment */

 /* Map = variables to work nodes */

/*creating an inputRDD to read text file (in.txt) through Spark context*/

    val input = sc.textFile("in.txt")
```

# Programming with RDD

**SparkWordCount.scala**

```
val input = sc.textFile("in.txt")
/* Transform the inputRDD into countRDD */
val count=input.flatMap(line=>line.split(" "))
        .map(word=>(word, 1))
        .reduceByKey(_ + _)

/* saveAsTextFile method is an action that effects on the RDD */

count.saveAsTextFile("outfile")
System.out.println("OK");
} }
```

# Spark-Submit

$spark-submit [options] <app jar | python file> [app arguments]

Sr. Option Description

1 --master spark://host:port, mesos://host:port, yarn, or local.

2 --deploy-mode Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster") (Default: client).

3 --class Your application's main class (for Java / Scala apps).

4 --name A name of your application.

# Spark-Submit

$spark-submit [options] <app jar | python file> [app arguments]

Sr. Option Description

5 --jars Comma-separated list of local jars to include on the driver and executor classpaths.

6 --packages Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths.

7 --repositories Comma-separated list of additional remote repositories to search for the maven coordinates given with --packages.

8 --py-files Comma-separated list of .zip, .egg, or .py files to place on the PYTHON PATH for Python apps.

9 --files Comma-separated list of files to be placed in the working directory of each executor.

# Advance Spark Programming

Spark contains two different types of shared variables- one is broadcast variables and second is accumulators.

- Broadcast variables: used to efficiently, distribute large values.

- Accumulators: used to aggregate the information of particular collection.

. Broadcast variables are created from a variable v by calling SparkContext.broadcast(v). The broadcast variable is a wrapper around v, and its value can be accessed by calling the value method. The code given below shows this:

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

# Advance Spark Programming

- Accumulators
  An accumulator is created from an initial value v by calling SparkContext.accumulator(v). Tasks running on the cluster can then add to it using the add method or the += operator (in Scala and Python). However, they cannot read its value. Only the driver program can read the accumulator's value, using its value method. The code given below shows an accumulator being used to add up the elements of an array:

```
scala> val accum = sc.accumulator(0)
scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

If you want to see the output of above code then use the following command:

```
scala> accum.value
```

**output**

**res2: Int = 10**

# Numeric RDD Operations

Spark allows you to do different operations on numeric data, using one of the predefined API methods. Spark's numeric operations are implemented with a streaming algorithm that allows building the model, one element at a time.

These operations are computed and returned as a StatusCounter object by calling status() method.

The following is a list of numeric methods available in StatusCounter.

| S.No | Method & Meaning |
|---|---|
| 1 | count() Number of elements in the RDD. |
| 2 | Mean() Average of the elements in the RDD. |

# Numeric RDD Operations

**S.No Method & Meaning**

3 Sum() Total value of the elements in the RDD.

4 Max() Maximum value among all elements in the RDD.

5 Min() Minimum value among all elements in the RDD.

6 Variance() Variance of the elements.

7 Stdev() Standard deviation.

If you want to use only one of these methods, you can call the corresponding method directly on RDD.

# Summary

What did we learn today?

Recap and Memory Test.

Question and Answers

# Contact Us on:

**G K T C S Innovations Pvt. Ltd.**
**IT Training & Consultancy,**

**Mobile: +91- 9975072320, 8308761477**
**Email : surendra@gktcs.com**
**Web: www.gktcs.com**