iTechArt

Software
Engineering
Services


# Introduction

# GOALS



- Get familiarized with reasons of Spark's popularity in Big Data.
- Learn how to solve simple tasks with Apache Spark.
- Get familiarized with basic concepts and basic components of Apache Spark.

# APACHE SPARK IN BIG DATA WORLD

# WHAT IS APACHE SPARK?



Open source general-purpose fast compute engine for distributed data processing at scale

- Generalizes MapReduce model
- Uses memory both to compute and store objects

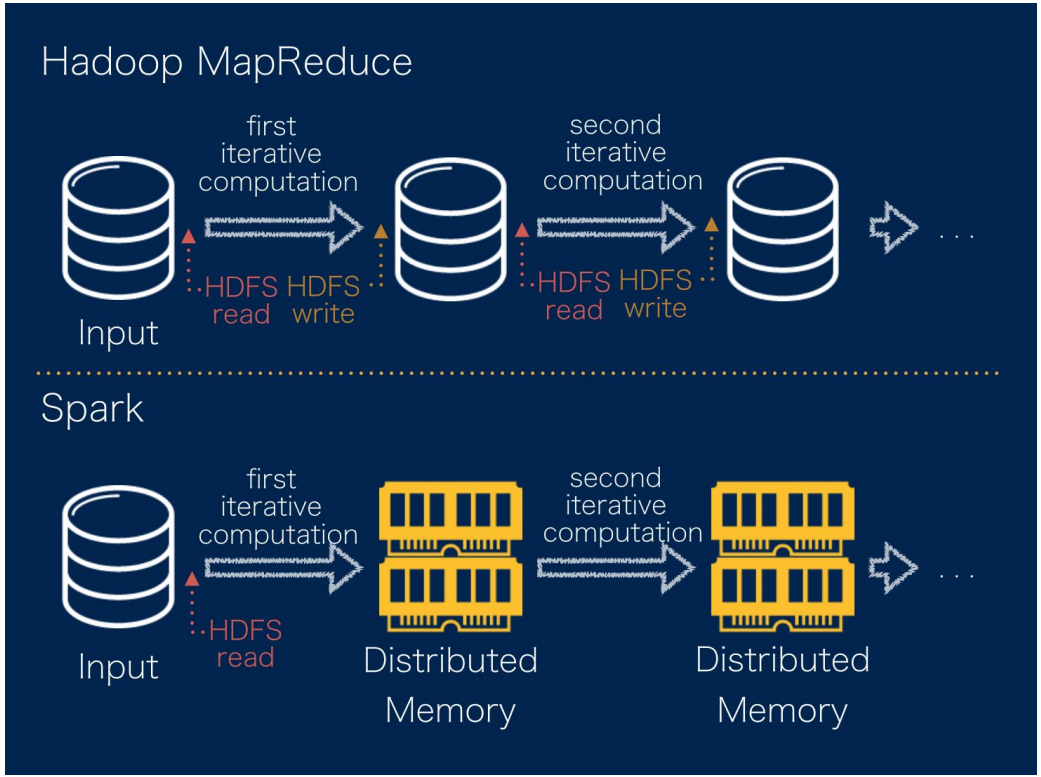
Rich set of APIs and libraries

- In Scala, Java, Python, R

Large community of 1000+ contributors from 250+ organizations

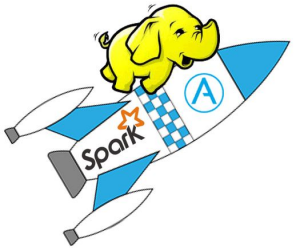
# SPARK VS HADOOP MAPREDUCE





# RELATIONSHIP BETWEEN SPARK AND HADOOP






- Widely used outside of Hadoop environments.
- Can read from any input source that MapReduce supports, ingest data directly from Apache Hive warehouses, and runs on top of the Apache Hadoop YARN resource manager.
- Represents a modern alternative to MapReduce, based on a more performance oriented and feature rich design.

# USE CASES



- Data integration and ETL
  - Cleansing and combining data from diverse sources for visualization or processing or analyzing in the future.
  - E.g. Netflix's productionizing ETL at petascale.


- Machine learning and advanced analytics
  - Sophisticated algorithms to predict outcomes or make decisions based on input data.
  - E.g. Alibaba's analysis of its marketplace in the retail sector;
  - E.g. Spotify's music recommendation engine in the media sector.

# USE CASES



- Interactive analytics or business intelligence
  - Gaining insight from massive data.
  - E.g. Huawei's query platform in the telecom sector.


- Real-time data processing
  - Capturing and processing data continuously with low latency and high reliability.
  - E.g. Netflix's streaming recommendation engine in the media sector.

NETFLIX

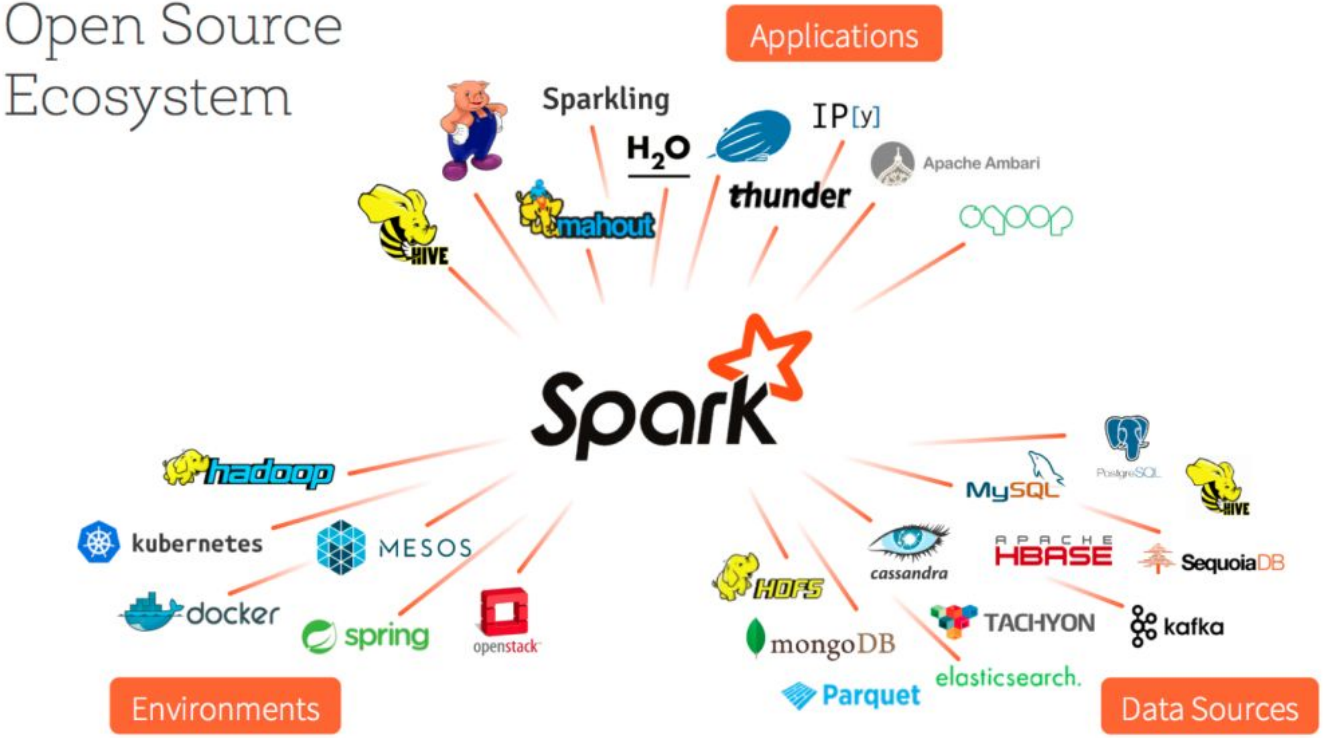
# USE CASES



- Simple
  - Easy-to-use, high-level declarative and unified APIs
- Simplified and Unified Engine
  - Supports for SQL queries, structured streaming, machine learning and graph processing
- Speed
  - Running 100x faster than Apache® Hadoop™ by exploiting in memory computing and Tungsten's and Catalyst's code optimizations
- Integrate Broadly
  - Built-in support for many data sources and data formats.
- Spark itself is written in Scala and runs on the Java Virtual Machine (JVM).

# SPARK ECOSYSTEM





# BRIEF HISTORY



- Spark was initially started by Matei Zaharia at UC Berkeley AMPLab in 2009, and open sourced in 2010 under a BSD license.
- After being released, Spark grew into a broad developer community, and moved to the Apache Software Foundation in 2013.
- In February 2014, Spark became a Top-Level Apache Project and one of the most active open source big data projects.
- Today, the project is developed collaboratively by a community of hundreds of developers from hundreds of organizations.

# BRIEF HISTORY



The main versions:

| Version | Original release date | Latest version | Release date |
|---|---|---|---|
| 0.5 | 2012-06-12 | 0.5.1 | 2012-10-07 |
| 1.0 | 2014-05-26 | 1.0.2 | 2014-08-05 |
| 1.6 | 2016-01-04 | 1.6.3 | 2016-11-07 |
| 2.0 | 2016-07-26 | 2.0.2 | 2016-11-14 |
| **2.4** | 2018-11-02 | 2.4.2 | 2019-04-23 |
| **3.0** | 2020-06-16 | 3.0.3 | 2021-06-23 |
| **3.1** | 2021-03-02 | 3.1.2 | 2021-05-27 |

# WHO IS DATABRICKS?



Databricks is the largest contributor to the open source Apache Spark project providing 10x more code than any other company.

The company has also trained over 40,000 users on Apache Spark, and has the largest number of customers deploying Spark to date.

Databricks provides a virtual analytics data platform, to simplify data integration, real-time experimentation, and robust deployment of production applications.

# SIMPLE TASKS WITH APACHE SPARK

# SPARK SHELL



**Spark shell** is an interactive environment in Python or Scala that provides a simple way to learn the API, as well as a powerful tool to analyze data interactively.

It is an extension of Python/Scala REPL with automatic instantiation of SparkContext as `sc`.

**SparkContext** is an entry point to Spark's functionality.

Spark shell offers an environment with *auto-completion* (using `TAB` key) where you can run ad-hoc queries and get familiar with the features of Spark.

Spark shell is particularly helpful for fast interactive prototyping.

# USING SPARK SHELL



- Spark's primary abstraction is a *Resilient Distributed Dataset* (RDD).
- Formally, an **RDD** is a read-only, partitioned collection of records.
- One could compare RDDs to collections in Scala, i.e. a RDD is computed on many JVMs while a Scala collection lives on a single JVM.

Let's make a new RDD from the text file and call some actions.

# USING SPARK SHELL



CODE SNIPPET

```
./bin/spark-shell
...
scala> val textFile = sc.textFile("README.md")
textFile: org.apache.spark.rdd.RDD[String] = README.md
MapPartitionsRDD[1] at textFile at <console>:24

scala> textFile.count() // Number of items in this RDD
res0: Long = 105

scala> textFile.first() // First item in this RDD
res1: String = # Apache Spark
```

# USING SPARK SHELL



CODE SNIPPET

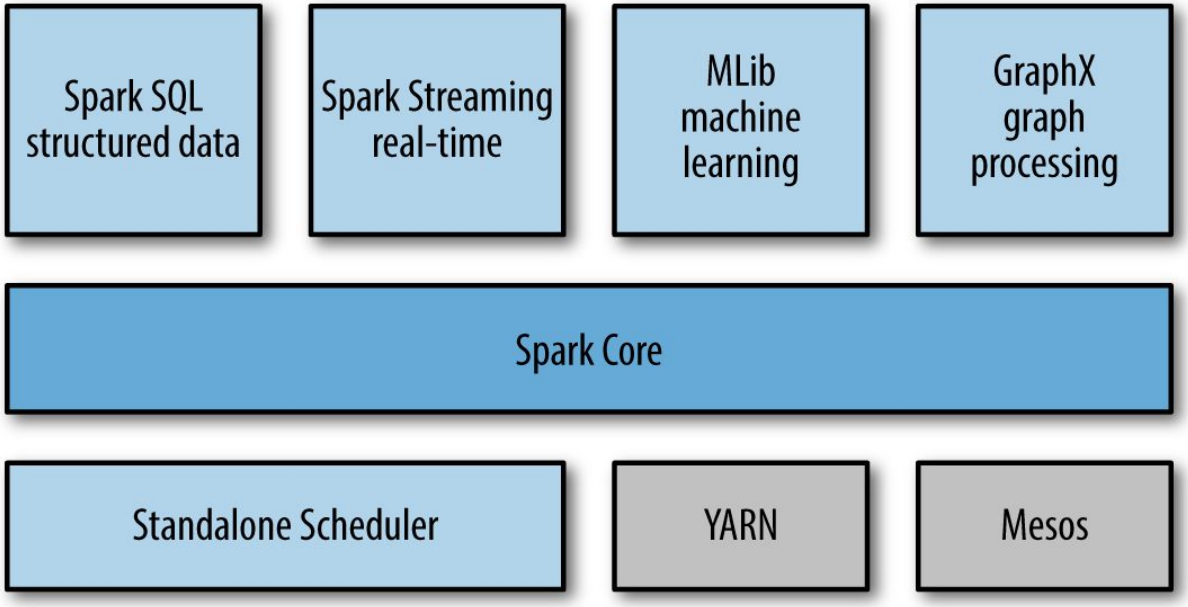
```
./bin/pyspark
...
>>> sc.textFile("README.md")
README.md MapPartitionsRDD[11] at textFile at
NativeMethodAccessorImpl.java:0

>>> textFile.count() // Number of items in this RDD
105

>>> textFile.first() // First item in this RDD
Row(value=u'# Apache Spark')
```

# BASIC CONCEPTS OF APACHE SPARK

# SPARK COMPONENTS



Spark unifies batch processing, advanced analytics, interactive exploration, and real-time stream processing into a single data processing framework.



# SPARK CORE



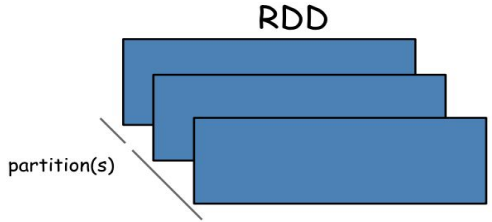
Spark Core is the general execution engine for the Spark platform that other functionality is built atop. It provides:

- Memory management and fault recovery
- Scheduling, distributing and monitoring jobs on a cluster
- Interacting with storage systems
- Basic I/O functionalities, exposed through an application programming interface centered on the resilient distributed dataset (**RDD**) abstraction.

# KEY CHARACTERISTICS OF RDDS



- Represents partitioned collection of elements that can be operated on in parallel.
- RDDs automatically recover from node failures.
- RDDs are immutable and their operations are lazy.
- RDDs can contain any type of Python, Java, or Scala objects.
- RDD class contains the basic operations available on all RDDs, such as `*map*` and `*filter*`.
- Users may also ask Spark to *persist* an RDD in memory, allowing it to be reused efficiently across parallel operations.



# SPARK SQL



Spark SQL is a Spark *module* for structured data processing.

After Spark 2.0, *RDDs* are replaced by *Dataset*, but RDDs are still supported.

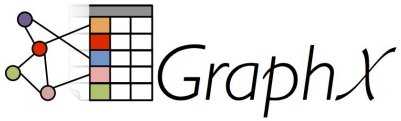
Spark SQL provide Spark with more information about the *structure* of both the data and the computation being performed.

Internally, Spark SQL uses this extra information to perform extra optimizations.

# OTHER COMPONENTS



- Spark Streaming
  - An extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Spark MLlib
  - Machine learning (ML) library that makes practical machine learning scalable and easy.
- GraphX
  - A new component in Spark for graphs and graph-parallel computation.



# CLUSTER MANAGERS



Spark's runtime runs on top of a variety of cluster managers:

- YARN
- Mesos
- Spark's own cluster manager called Standalone mode
- Kubernetes

# APACHE SPARK EXAMPLES

# MORE EXAMPLES USING SPARK-SHELL



CODE SNIPPET

```
scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))
linesWithSpark: org.apache.spark.rdd.RDD[String] = MapPartitionsRDD[2]
// How many lines contain "Spark"?
scala> linesWithSpark.count()

res3: Long = 20
```

CODE SNIPPET

```
>>> linesWithSpark = textFile.filter(textFile.value.contains ("Spark"))

# How many lines contain "Spark"?
>>> linesWithSpark.count()
20
```

# MORE EXAMPLES USING SPARK-SHELL



**Caching** - Spark also supports pulling data sets into a cluster-wide in-memory cache.

This is very useful when data is accessed repeatedly.

# MORE EXAMPLES USING SPARK-SHELL



CODE SNIPPET

```
scala> linesWithSpark.cache()
res7: linesWithSpark.type =
MapPartitionsRDD[2]

scala> linesWithSpark.count()
res8: Long = 15

scala> linesWithSpark.count()
res9: Long = 15
```

CODE SNIPPET

```
>>> linesWithSpark.cache()

>>> linesWithSpark.count()
15

>>> linesWithSpark.count()
15
```

# MORE EXAMPLES USING SPARK-SHELL



CODE SNIPPET

```
scala> import java.lang.Math
import java.lang.Math

scala> textFile.map(line => line.split(" ").size).reduce((a, b) =>
Math.max(a, b))
res5: Int = 22
```

CODE SNIPPET

```
>>> from pyspark.sql.functions import *

>>> textFile.select(size(split(textFile .value,
"\s+")).name("numWords")).agg(max(col("numWords"))).collect()
[Row(max(numWords)=22)]
```

# APACHE SPARK APPLICATION

# SELF-CONTAINED APPLICATION



Suppose we wish to write a self-contained application using the Spark API.

# SELF-CONTAINED APPLICATION



CODE SNIPPET

```
/* SimpleApp.scala */
import org.apache.spark.{SparkConf, SparkContext}

object SimpleApp {
 def main(args: Array[String]) {
   val logFile = "YOUR_SPARK_HOME/README.md"  // Should be some file on
your system
   val spark = new SparkContext(new SparkConf().setAppName("Simple
Application"))
   val logData = spark.textFile(logFile).cache()
   val numAs = logData.filter(line => line.contains("a")).count()
   val numBs = logData.filter(line => line.contains("b")).count()
   println(s"Lines with a: $numAs, Lines with b: $numBs")
   spark.stop()
 }
}
```

# SELF-CONTAINED APPLICATION



Our application depends on the Spark API, so we'll also include an sbt configuration file, `build.sbt`, which explains that Spark is a dependency.

This file also adds a repository that Spark depends on.

# SELF-CONTAINED APPLICATION



CODE SNIPPET

```
name := "Simple Project"

version := "1.0"

scalaVersion := "2.13.6"

libraryDependencies += "org.apache.spark" %% "spark-core" %
"3.2.0"
```

# SELF-CONTAINED APPLICATION

Once layout `SimpleApp.scala` and `build.sbt` according to the typical directory structure, we can create a JAR package with the application's code.

# SELF-CONTAINED APPLICATION



CODE SNIPPET

```
# Your directory layout should look like this
$ find .
.
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/SimpleApp.scala

# Package a jar containing your application
$ sbt package
...
[info] Packaging
{..}/{..}/target/scala-2.13/simple-project_2.13-1.0.jar
```

# SELF-CONTAINED APPLICATION



Then we can use the `spark-submit` script to run our program.

# SELF-CONTAINED APPLICATION



CODE SNIPPET

```
# Use spark-submit to run your application
$ YOUR_SPARK_HOME/bin/spark-submit \
  --class "SimpleApp" \
  --master local[4] \
  target/scala-2.13/simple-project_2.13-1.0.jar

...
Lines with a: 62, Lines with b: 31
```

# Various code examples



Sources

# USEFUL LINKS



1. https://databricks.com/glossary/hadoop-ecosystem
2. https://spark.apache.org/history.html
3. https://spark.apache.org/docs/latest/quick-start.html
4. Books:
    1. Apache-Spark-2.x-Cookbook.pdf
    2. Learning Spark
    3. Spark Definitive guide
5. Matei Zaharia abou Spark 2.0 - youtube
6. https://github.com/awesomedata/awesome-public-datasets
7. Databricks Scala Style Guide