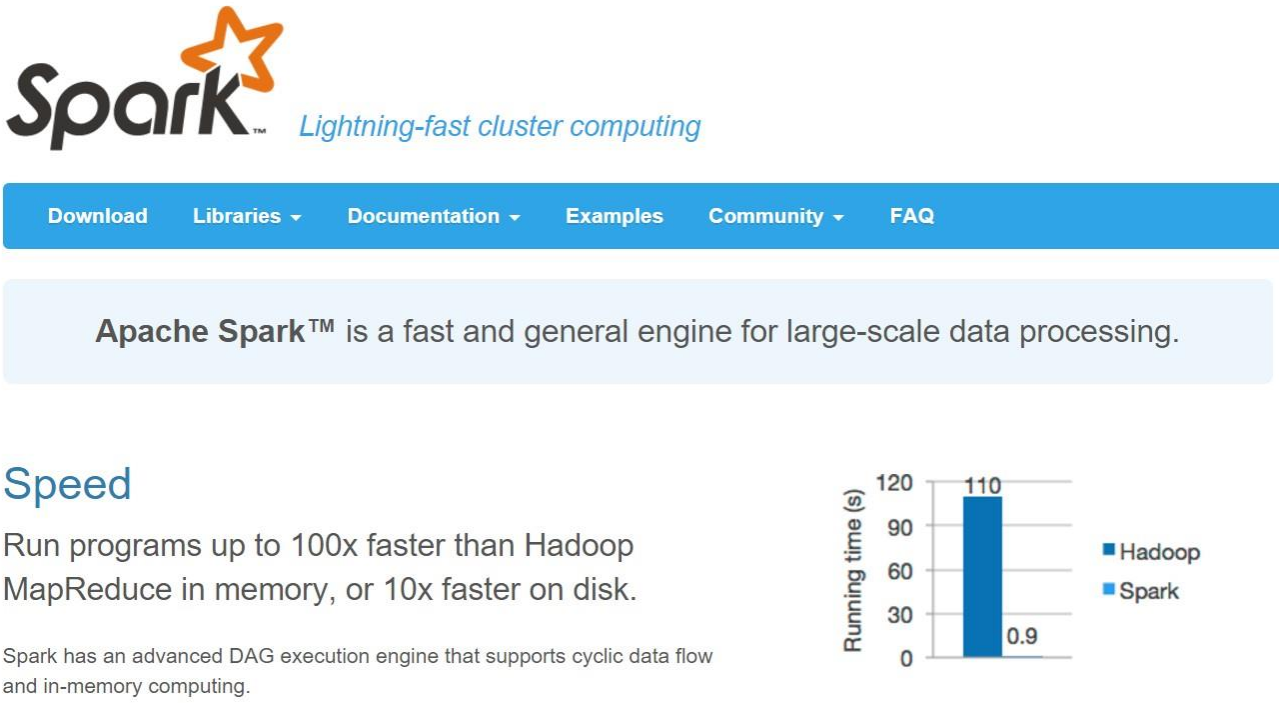
What is Spark?

- Apache Spark is a fast and general engine for large-scale data processing
- Written in Scala – Functional programming language that runs in a JVM
- Spark shell – Interactive—for learning, data exploration, or ad hoc analytics – Python or Scala
- Spark applications – For large scale data processing – Python, Scala, or Java



Reference: http://spark.apache.org

MapReduce VS Spark

How does MapReduce work and what is its constraint?

- In MapReduce, all the data is written back to the physical storage medium after each operation. This means that it reads data from the disk and once this is completed, it writes the data back to the disk.
- This process becomes inefficient when we need low latency. Since it has to read all the data from disk at the beginning of each stage of the process, it is very time consuming. Data Sharing is Slow in MapReduce
- Unfortunately, in most current frameworks, the only way to reuse data between computations (Ex – between two MapReduce jobs) is to write it to an external table storage system (Ex – HDFS). Although this framework provides numerous abstractions for accessing a cluster's computational resources, users still want more.
- Both Iterative and Interactive applications require faster data sharing across parallel jobs. Data sharing is slow in MapReduce due to replication, serialization, and disk IO. Regarding storage system, most of the Hadoop applications, they spend more than 90% of the time doing HDFS read-write operations

Apache Spark Stack

Spark provides a stack of libraries built on core Spark

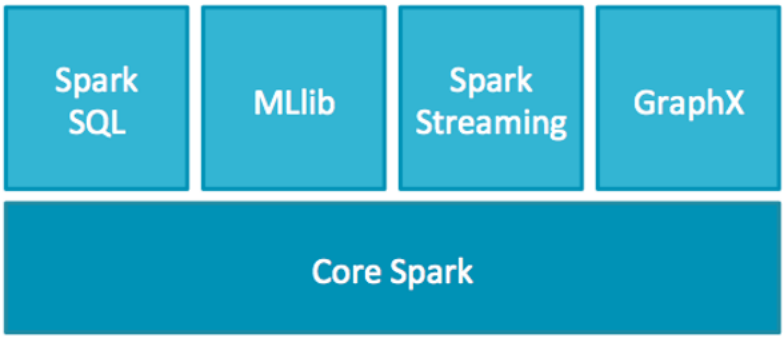
— Core Spark provides the fundamental Spark abstraction: Resilient Distributed Datasets (RDDs)

— Spark SQL works with structured data

— MLlib supports scalable machine learning

— Spark Streaming applications process data in real time

— GraphX works with graphs and graph-parallel computation



Spark's Language APIs

Spark's language APIs allow you to run Spark code from other languages

Scala

Spark is primarily written in Scala, making it Spark's "default" language.

Java

Even though Spark is written in Scala, Spark's authors have been careful to ensure that you can write Spark code in Java.

Python

Python supports nearly all constructs that Scala supports.

SQL

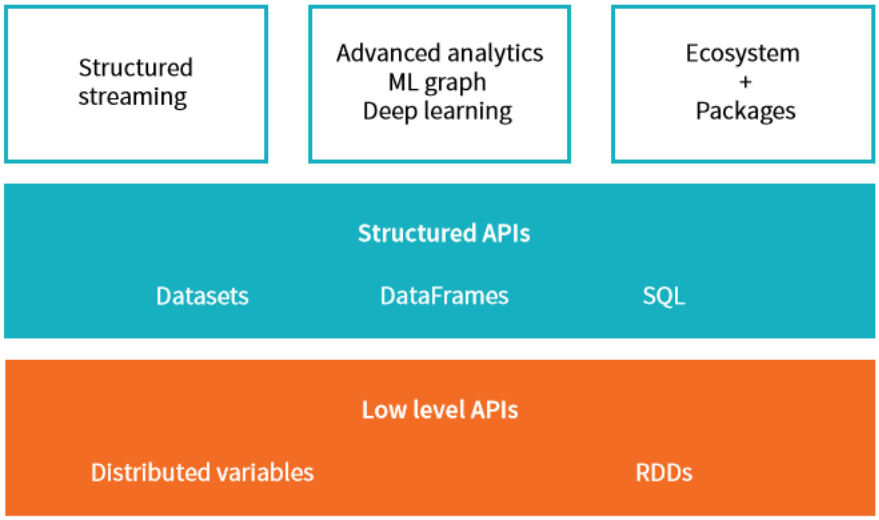
Spark supports ANSI SQL 2003 standard. This makes it easy for analysts and non-programmers to leverage the big data powers of Spark.

R

Spark has two commonly used R libraries, one as a part of Spark core (SparkR) and another as a R community driven package (sparklyr).

A Tour of Spark's Tool sets and Spark's APIs:

Spark is composed of the simple primitives, the lower level APIs and the Structured APIs, then a series of "standard libraries" included in Spark.



While Spark is available from a variety of languages, what Spark makes available in those languages is worth mentioning.

- RDD Programming Guide: overview of Spark basics - RDDs (core but old API), accumulators, and broadcast variables
- Spark SQL, Datasets, and DataFrames: processing structured data with relational queries (newer API than RDDs)
- Structured Streaming: processing structured data streams with relation queries (using Datasets and DataFrames, newer API than DStreams)
- Spark Streaming: processing data streams using DStreams (old API)
- MLlib: applying machine learning algorithms
- GraphX: processing graphs

Spark has two fundamental sets of APIs: the low level "Unstructured" APIs and the higher level Structured APIs.

Unstructured API: RDD was the primary user-facing API in Spark since its inception. At the core, an RDD is an immutable distributed collection of elements of your data, partitioned across nodes in your cluster that can be operated in parallel with a low-level API that offers transformations and actions.

Structured API: DataFrames, Datasets & SparkSql

Like an RDD, a **DataFrame** is an immutable distributed collection of data. Unlike an RDD, data is organized into named columns, like a table in a relational database. Designed to make large data sets processing even easier, DataFrame allows developers to impose a structure onto a distributed collection of data, allowing higher-level abstraction; it provides a domain specific language API to manipulate your distributed data; and makes Spark accessible to a wider audience, beyond specialized data engineers.

Datasets:

Dataset takes on two distinct APIs characteristics: a strongly-typed API and an untyped API. Conceptually, consider DataFrame as an alias for a collection of generic objects Dataset[Row], where a Row is a generic untyped JVM object.

Dataset, by contrast, is a collection **of strongly-typed JVM objects**, dictated by a **case class** you define in Scala or a class in Java.

Spark Shell and Spark Applications

The Spark Shell

- **The Spark shell provides an interactive Spark environment**
  - Often called a *REPL*, or Read/Evaluate/Print Loop
  - For learning, testing, data exploration, or ad hoc analytics
  - You can run the Spark shell using either Python or Scala
- **You typically run the Spark shell on a gateway node**

Starting the Spark Shell

- **On a Cloudera cluster, the command to start the Spark 2 shell is**
  - `pyspark2` for Python
  - `spark2-shell` for Scala
- **The Spark shell has a number of different start-up options, including**
  - `master`: specify the cluster to connect to
  - `jars`: Additional JAR files (Scala only)
  - `py-files`: Additional Python files (Python only)
  - `name`: the name the Spark Application UI uses for this application
    - Defaults to `PySparkShell` (Python) or `Spark shell` (Scala)
  - `help`: Show all the available shell options

```
$ pyspark2 --name "My Application"
```

```
C:\spark\bin>spark-shell2.cmd
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).
18/04/07 20:52:38 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfor
m... using builtin-java classes where applicable
Spark context Web UI available at http://192.168.56.1:4040
Spark context available as 'sc' (master = local[*], app id = local-1523114559504).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Scala version 2.11.8 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_161)
Type in expressions to have them evaluated.
Type :help for more information.
```

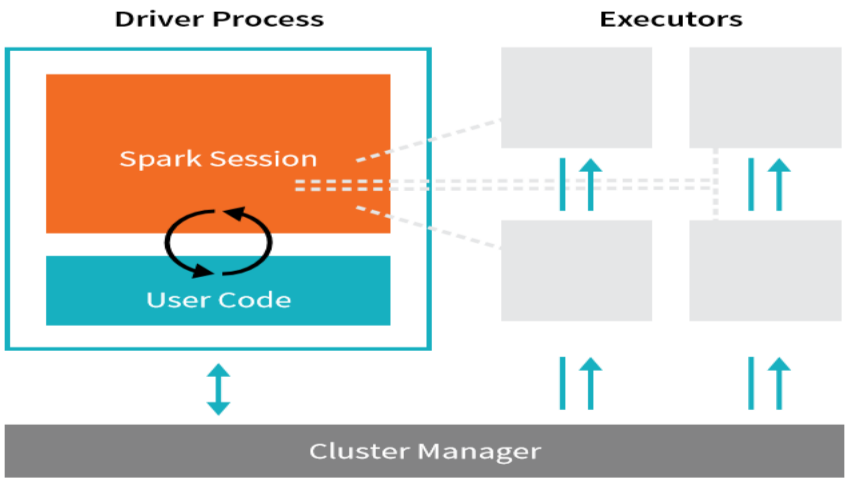
```
C:\spark\bin>pyspark2.cmd
Python 3.6.1 (v3.6.1:69c0db5, Mar 21 2017, 17:54:52) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/04/07 22:05:16 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfor
m... using builtin-java classes where applicable
18/04/07 22:05:26 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObject
Exception
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 2.2.1
      /_/

Using Python version 3.6.1 (v3.6.1:69c0db5, Mar 21 2017 17:54:52)
SparkSession available as 'spark'.
>>>
```

```
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
18/04/07 22:09:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platfo
rm... using builtin-java classes where applicable
18/04/07 22:09:34 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjec
tException

 Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version  2.2.1
      /_/

 SparkSession available as 'spark'.
>
```

Spark Applications

Spark Applications consist of a driver process and a set of executor processes. The driver process runs your main() function, sits on a node in the cluster, and is responsible for three things: maintaining information about the Spark Application; responding to a user's program or input; and analyzing, distributing, and scheduling work across the executors (defined momentarily). The driver process is absolutely essential - it's the heart of a Spark Application and maintains all relevant information during the lifetime of the application.

The executors are responsible for actually executing the work that the driver assigns them. This means, each executor is responsible for only two things: executing code assigned to it by the driver and reporting the state of the computation, on that executor, back to the driver node.



The cluster manager controls physical machines and allocates resources to Spark Applications. This can be one of several core cluster managers: Spark's standalone cluster manager, YARN, or Mesos.

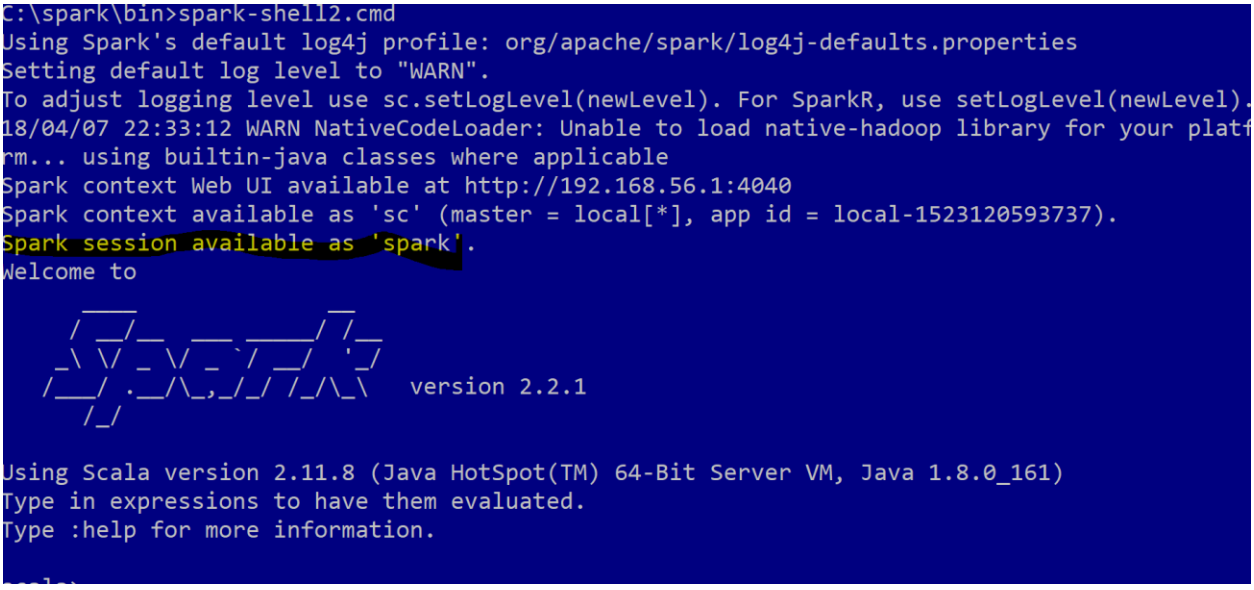
As a short review of Spark Applications, the key points to understand at this point are that:

• Spark has some cluster manager that maintains an understanding of the resources available.

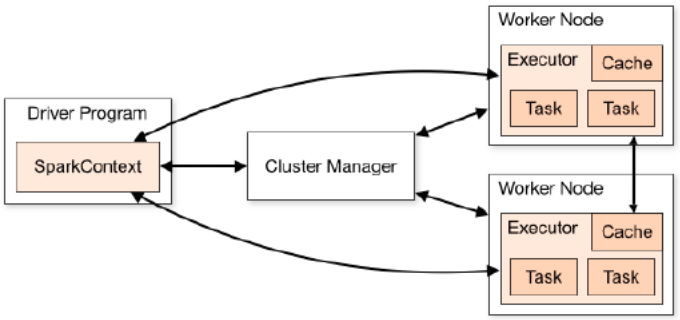
• The driver process is responsible for executing our driver program's commands across the executors in order to complete our task.

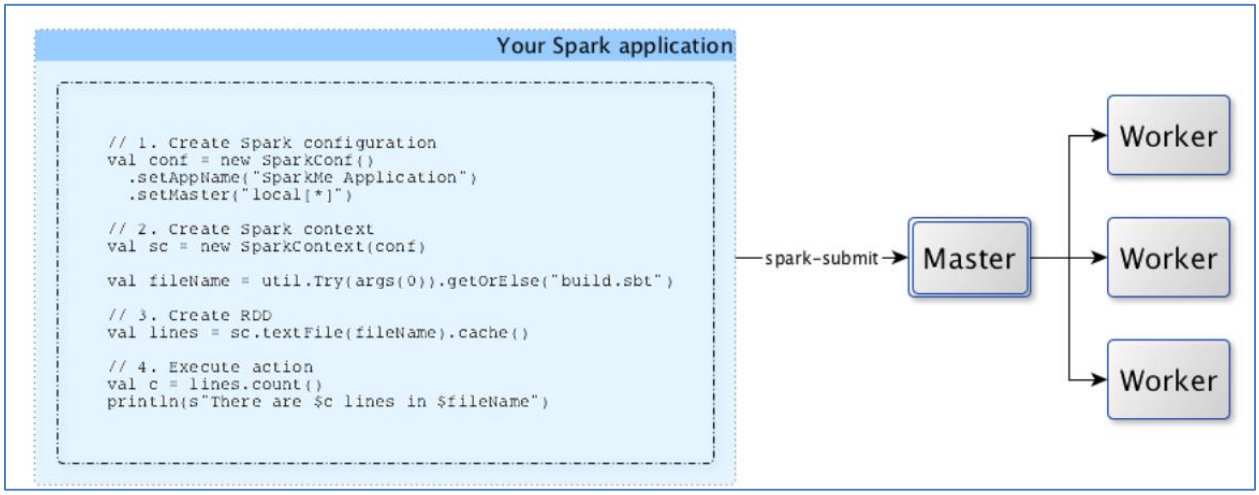
Working with Spark Session

▪ The main entry point for the Spark API is a Spark session
▪ The Spark shell provides a preconfigured SparkSession object called spark
▪ The SparkSession class provides functions and attributes to access all of Spark functionality
▪ Examples include
— sql: execute a Spark SQL query
— catalog: entry point for the Catalog API for managing tables
— read: function to read data from a file or other data source
— conf: object to manage Spark configuration settings
— SparkContext : entry point for core Spark API



SparkContext & SparkSession





As shown in the diagram, a SparkContext is a conduit to access all Spark functionality; only a single SparkContext exists per JVM. The Spark driver program uses it to connect to the cluster manager to communicate, submit Spark jobs and knows what resource manager (YARN, Mesos or Standalone) to communicate to. It allows you to configure Spark configuration parameters. And through SparkContext, the driver can access other contexts such as SQLContext, HiveContext, and StreamingContext to program Spark.

Example: Refer SparkSession.rtf

SparkSession Encapsulates SparkContext

Each language API will maintain the same core concepts that we described above. There is a **SparkSession** available to the user, the SparkSession will be the entrance point to running Spark code. When using Spark from a Python or R, the user never writes explicit JVM instructions, but instead writes Python and R code that Spark will translate into code that Spark can then run on the executor JVMs

```
import org.apache.spark.sql.SparkSession
val sparkSession = SparkSession.builder
  .master("local")
  .appName("my-spark-app")
  .config("spark.some.config.option", "config-value")
  .getOrCreate()
```

```
import org.apache.spark.sql.SparkSession sparkSession: org.apache.spark.sql.S
parkSession = org.apache.spark.sql.SparkSession@46d6b87c
```

Anatomy of Spark Job

Job Submission

Spark job is submitted automatically as soon as action is performed on RDD. Internally it call runJob() on SparkContext which will pass the call to scheduler. Scheduler is made of two parts:

DAG scheduler: that breaks down the job into DAG of stages and

Task scheduler: that is responsible for submitting the task from each stage to the cluster

DAG scheduler construct:

Shuffle map tasks:

Each shuffle map task run on one RDD partition. Based on the partition function it writes output to new set of partitions

Result task:

Each results task runs a computation on its RDD partition then send results back to the driver, and the driver assembles the results from each partition into a final results

DAG scheduler is responsible for splitting stages into tasks for submission to the task scheduler

Task Scheduling

When the task scheduler is sent a set of tasks, it uses its list of executors that are running for the application and construct mapping of task to executor

Executor first assign task to local process then node-level then rack level and finally cluster level

Task Execution:

Executor first makes sure that jar and file dependencies are copied to local

Executor and Cluster Manager

Local

In the local mode there is single executor running in the same JVM as driver. The Master url for local mode is local [n]

Standalone

Simple distributed implementation that runs a single Spark Master and one or more workers. When a Spark Application starts, the master will ask workers to spawn executor process on behalf of the application

Master url is spark://localhost:7077

Spark on YARN

Running Spark on YARN provides the tightest integration with other Hadoop components. YARN has resource manager used in HADOOP

Each running Spark application corresponds to an instance of a YARN application and each executor runs in its own YARN container. The Master URL is yarn-client or yarn-cluster

Yarn client mode is required for programs that have any interactive component, such as spark-shell or pyspark

Yarn cluster mode is for batch mode and useful for production deployment

Syntax for submitting of Spark Job in batch mode

```
./bin/spark-submit \
  --class <main-class>
  --master <master-url> \
  --deploy-mode <deploy-mode> \
  --conf <key>=<value> \
  ... # other options
```

```
  <application-jar> \
  [application-arguments]
```

Some of the commonly used options are:

--class: The entry point for your application (e.g. org.apache.spark.examples.SparkPi)

--master: The master URL for the cluster (e.g. spark://23.195.26.187:7077)

--deploy-mode: Whether to deploy your driver on the worker nodes (cluster) or locally as an external client (client) (default: client) †

--conf: Arbitrary Spark configuration property in key=value format. For values that contain spaces wrap “key=value” in quotes (as shown).

application-jar: Path to a bundled jar including your application and all dependencies. The URL must be globally visible inside of your cluster, for instance, an hdfs:// path or a file:// path that is present on all nodes.

application-arguments: Arguments passed to the main method of your main class, if any

Examples:

 # Run application locally on 8 cores

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master local[8] \
  /path/to/examples.jar \
  100
```

 # Run on a Spark standalone cluster in client deploy mode

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

 # Run on a Spark standalone cluster in cluster deploy mode with supervise

```
./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master spark://207.184.161.138:7077 \
  --deploy-mode cluster
  --supervise
  --executor-memory 20G \
  --total-executor-cores 100 \
  /path/to/examples.jar \
  1000
```

Note: Spark standalone cluster with cluster deploy mode, you can also specify --supervise to make sure that the driver is automatically restarted if it fails with non-zero exit code

 # Run on a YARN cluster

```
export HADOOP_CONF_DIR=XXX

./bin/spark-submit \
  --class org.apache.spark.examples.SparkPi \
  --master yarn \
  --deploy-mode cluster \  # can be client for client mode
  --executor-memory 20G \
  --num-executors 50 \
  /path/to/examples.jar \
  1000
```

 # Run a Python application on a Spark standalone cluster

```
./bin/spark-submit \
  --master spark://207.184.161.138:7077 \
  examples/src/main/python/pi.py \
  1000
```

The master URL passed to Spark can be in one of the following formats:

Master URL: Meaning

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.

Yarn: Connect to a YARN cluster in client or cluster mode depending on the value of --deploy-mode. The cluster location will be found based on the HADOOP_CONF_DIR or YARN_CONF_DIR variable.

yarn-client: Equivalent to yarn with --deploy-mode client, which is preferred to `yarn-client`

yarn-cluster: Equivalent to yarn with --deploy-mode cluster, which is preferred to `yarn-cluster`

Yarn-client

In YARN client mode, the interaction with YARN starts when a new SparkContext instance is constructed by the driver programs.

- The context submits a YARN application to the YARN resource manager

- YARN RM start a YARN container on a Node Manager in the cluster and runs SparkExecutorLauncher application Master in it.

- The job of ExecutorLauncher is to start executors in YARN container, which it does by requesting resources from the RM

Then launching ExecutorBackend process as the container are allocated to it

Yarn-cluster

In YARN cluster mode, the users' driver program runs in a YARN application master process.

- The spark-submit client will launch the YARN application but it does not run any user code.

- Application Master starts the driver program before allocating resources for the executor

Spark Cluster Options (1)

- **Spark applications can run on these types of clusters**
  - Apache Hadoop YARN
  - Apache Mesos
  - Spark Standalone
- **They can also run locally instead of on a cluster**
- **The default cluster type for the Spark 2 Cloudera add-on service is YARN**
- **Specify the type or URL of the cluster using the `master` option**

Spark Cluster Options (2)

- **The possible values for the `master` option include**
  - `yarn`
  - `spark://masternode:port` (Spark Standalone)
  - `mesos://masternode:port` (Mesos)
  - `local[*]` runs locally with as many threads as cores (default)
  - `local[n]` runs locally with *n* threads
  - `local` runs locally with a single thread

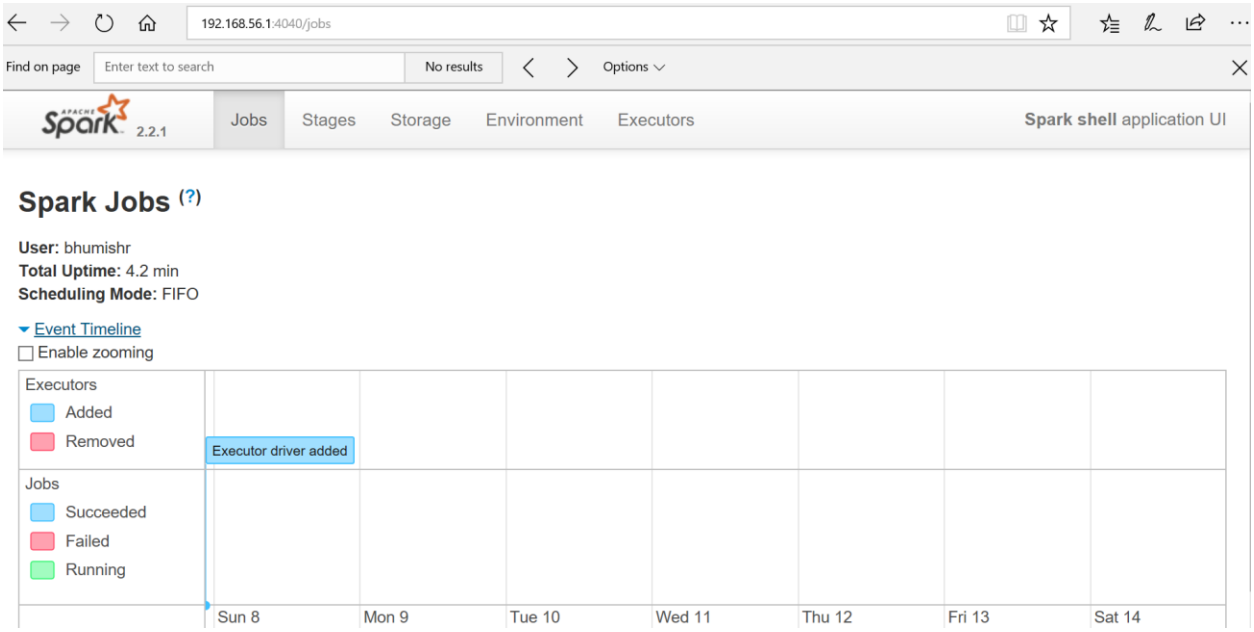
```
$ pyspark2 --master yarn
```

Language: *Python*

```
$ spark2-shell --master yarn
```

Language: *Scala*

Debugging and Monitoring Spark Application:



Scalable language

Scala is a **modern multi-paradigm*** programming language designed to express **common** programming **patterns** in a **concise**, **elegant**, and **type-safe** way.

**In the context of programming, "paradigm" often refers to object-oriented, procedural, functional, logical, etc. So "multi paradigm" would be referring to something as relating to features of several paradigms*

Introduction to Scala

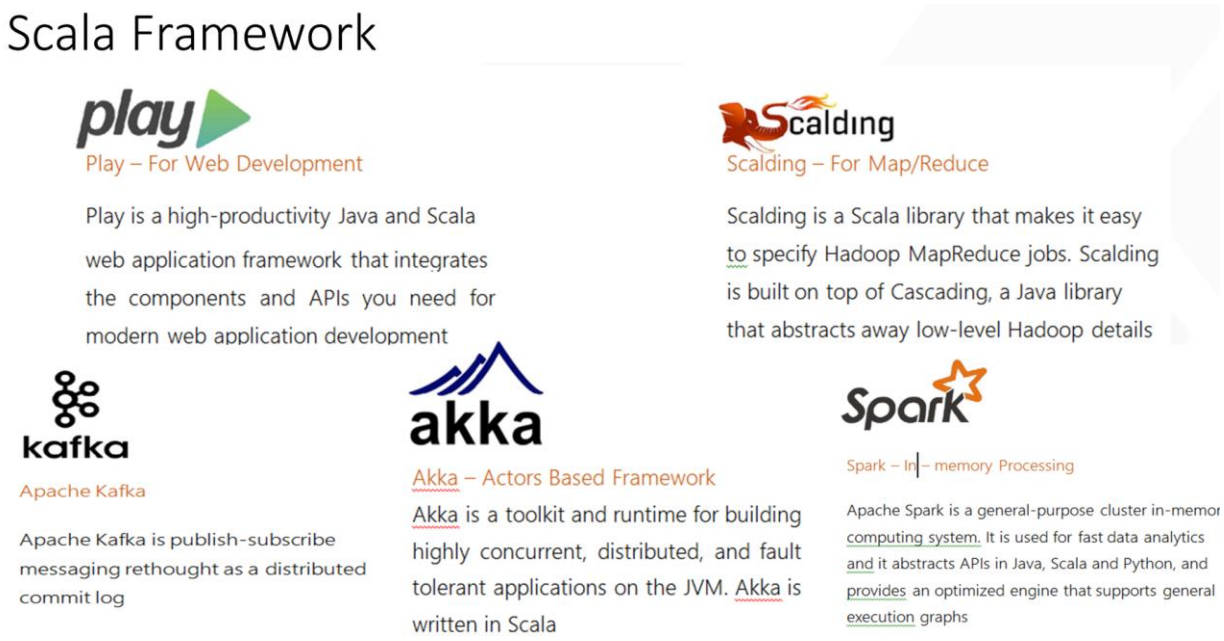
- ✓ Martin Odersky and his team started developing Scala in 2001
- ✓ Scala is a general purpose programming language, multi paradigm object oriented, functional, scalable
- ✓ Aimed to implement common programming patterns in a concise, elegant, and type-safe way
- ✓ Supports both object-oriented and functional programming styles, thus helping programmers to be more productive
- ✓ Publicly released in January 2004 on the JVM platform and a few months later on the .NET platform
- ✓ Scala is Statically Typed (Statically typed language binds the type to a variable for its entire scope)
- ✓ Dynamically typed languages bind the type to the actual value referenced by a variable .Example : python
- ✓ Fully supports Object Oriented Programming i.e. everything is an object in Scala
- ✓ Unlike Java, Scala does not have primitives
- ✓ Supports “static” class members through Singleton Object Concept
- ✓ Improved support for OOP through Traits

Why Scala?

- ✓ Scala is pure object-oriented language. Conceptually, every value is an object and every operation is a method-call
- ✓ Scala is also a functional language and supports immutable data structures
- ✓ Many big data technologies use Scala like Spark, Kafka, Storm, Akka, Scalding and web frameworks like Play

Java Code	Scala Code

`List<String> list = new ArrayList<String>();`
`list.add("1");`
`list.add("2");`
`list.add("3");`
	`val list = List("1", "2", "3")`



Download and Install Scala

Latest version can be downloaded from: http://www.scala-lang.org/download/

Install the Scala and Set the Scala Path in Machine

For Scala Crash Course refer following materials:

LearningScala1.sc
LearningScala2.sc
LearningScala3.sc
LearningScala4.sc

Spark RDD

Resilient Distributed Datasets (RDDs)

- **RDDs are part of core Spark**
- ***Resilient Distributed Dataset* (RDD)**
  - *Resilient*: If data in memory is lost, it can be recreated
  - *Distributed*: Processed across the cluster
  - *Dataset*: Initial data can come from a source such as a file, or it can be created programmatically
- **Despite the name, RDDs are *not* Spark SQL `Dataset` objects**
  - RDDs predate Spark SQL and the DataFrame/Dataset API

RDD Data Types

- **RDDs can hold any serializable type of element**
  - Primitive types such as integers, characters, and booleans
  - Collections such as strings, lists, arrays, tuples, and dictionaries (including nested collection types)
  - Scala/Java Objects (if serializable)
  - Mixed types
- **Some RDDs are specialized and have additional functionality**
  - Pair RDDs
    - RDDs consisting of key-value pairs
  - Double RDDs
    - RDDs consisting of numeric data

Creating RDDs from Files

- **Use `SparkContext` object, not `SparkSession`**
  - `SparkContext` is part of the core Spark library
  - `SparkSession` is part of the Spark SQL library
  - One Spark context per application
  - Use `SparkSession.sparkContext` to access the Spark context
    - Called `sc` in the Spark shell
- **Create file-based RDDs using the Spark context**
  - Use `textFile` or `wholeTextFiles` to read text files
  - Use `hadoopFile` or `newAPIHadoopFile` to read other formats
    - The Hadoop "new API" was introduced in Hadoop .20
    - Spark supports both for backward compatibility

Creating RDDs from Text Files (1)

- **`SparkContext.textFile` reads newline-terminated text files**
  - Accepts a single file, a directory of files, a wildcard list of files, or a comma-separated list of files
  - Examples
    - `textFile("myfile.txt")`
    - `textFile("mydata/")`
    - `textFile("mydata/*.log")`
    - `textFile("myfile1.txt,myfile2.txt")`

```
myRDD = spark.sparkContext.textFile("mydata/")
```

Language: *Python*

Creating RDDs from Text Files (2)

- `textFile` maps each line in a file to a separate RDD element
  - Only supports newline-terminated text

```
myRDD = spark. \
  sparkContext. \
  textFile("purplecow.txt")
```

Language: *Python*

```
I've never seen a purple cow.\n
I never hope to see one;\n
But I can tell you, anyhow,\n
I'd rather see than be one.\n
```

myRDD

I've never seen a purple cow.
I never hope to see one;
But I can tell you, anyhow,
I'd rather see than be one.

Multi-line Text Elements

- `textFile` maps each line in a file to a separate RDD element
  - What about files with a multi-line input format, such as XML or JSON?
- Use `wholeTextFiles`
  - Maps entire contents of each file in a directory to a single RDD element
  - Works only for small files (element must fit in memory)

user1.json

```
{
 "firstName":"Fred",
 "lastName":"Flintstone",
 "userid":"123"
}
```

user2.json

```
{
 "firstName":"Barney",
 "lastName":"Rubble",
 "userid":"234"
}
```

Example: Using wholeTextFiles

```
userRDD = spark.sparkContext. \
  wholeTextFiles("userFiles")
```

Language: *Python*

userRDD

("user1.json",{"firstName":"Fred", "lastName":"Flintstone","userid":"123"})
("user2.json",{"firstName":"Barney", "lastName":"Rubble","userid":"234"})
("user3.json",…)
("user4.json",…)

Creating RDDs from Collections

- **You can create RDDs from collections instead of files**
  - `SparkContext.parallelize(`*collection*`)`
- **Useful when**
  - Testing
  - Generating data programmatically
  - Integrating with other systems or libraries
  - Learning

```
myData = ["Alice","Carlos","Frank","Barbara"]
myRDD = sc.parallelize(myData)
```

Language: *Python*

Saving RDDs

- **You can save RDDs to the same data source types supported for reading RDDs**
  - Use `RDD.saveAsTextFile` to save as plain text files in the specified directory
  - Use `RDD.saveAsHadoopFile` or `saveAsNewAPIHadoopFile` with a specified Hadoop `OutputFormat` to save using other formats
- **The specified save directory cannot already exist**

```
myRDD.saveAsTextFile("mydata/")
```

RDD Operations

- **Two general types of RDD operations**
  - *Actions* return a value to the Spark driver or save data to a data source
  - *Transformations* define a new RDD based on the current one(s)
- **RDDs operations are performed lazily**
  - Actions trigger execution of the base RDD transformations

RDD Action Operations

- **Some common actions**
  - `count` returns the number of elements
  - `first` returns the first element
  - `take(n)` returns an array (Scala) or list (Python) of the first *n* elements
  - `collect` returns an array (Scala) or list (Python) of all elements
  - `saveAsTextFile(dir)` saves to text files

```
myRDD = sc. \
  textFile("purplecow.txt")

for line in myRDD.take(2):
  print line
I've never seen a purple cow.
I never hope to see one;
```

Language: *Python*

```
val myRDD = sc.
  textFile("purplecow.txt")

for (line <- myRDD.take(2))
  println(line)
I've never seen a purple cow.
I never hope to see one;
```

Language: *Scala*

RDD Transformation Operations (1)

- **Transformations create a new RDD from an existing one**
- **RDDs are immutable**
  - Data in an RDD is never changed
  - Transform data to create a new RDD
- **A transformation operation executes a *transformation function***
  - The function transforms elements of an RDD into new elements
  - Some transformations implement their own transformation logic
  - For many, you must provide the function to perform the transformation

RDD Transformation Operations (2)

- **Transformation operations include**
  - `distinct` creates a new RDD with duplicate elements in the base RDD removed
  - `union(`*`rdd`*`)` creates a new RDD by appending the data in one RDD to another
  - `map(`*`function`*`)` creates a new RDD by performing a function on each record in the base RDD
  - `filter(`*`function`*`)` creates a new RDD by including or excluding each record in the base RDD according to a Boolean function

Example: `distinct` and `union` Transformations

cities1.csv

```
Boston,MA
Palo Alto,CA
Santa Fe,NM
Palo Alto,CA
```

cities2.csv

```
Calgary,AB
Chicago,IL
Palo Alto,CA
```

```
distinctRDD = sc.\
  textFile("cities1.csv").distinct()
for city in distinctRDD.collect(): \
  print city
Boston,MA
Palo Alto,CA
Santa Fe,NM

unionRDD = sc.textFile("cities2.csv"). \
  union(distinctRDD)
for city in unionRDD.collect():
  print city
Calgary,AB
Chicago,IL
Palo Alto,CA
Boston,MA
Palo Alto,CA
Santa Fe,NM
```

Language: *Python*