MA-INF 4223- Lab Distributed Big Data Analytics

Smart Data Analytics

Spark Fundamentals I

02/05/2017

Dr. Hajira Jabeen, Gezim Sejdiu, Prof. Dr. Jens Lehmann

Lesson objectives

- After completing this lesson, you should be able to:
  - Justify the purpose of Spark
  - List and describe Spark Stack components
  - Understand the basics of Resilient Distributed Dataset (RDD) and Data Frames
  - Download and configure Spark- standalone
  - Scala overview
  - Launch and use Spark's Scala shell

Spark

- Zaharia, Matei, et al. "**Spark: Cluster Computing with Working Sets.**" HotCloud 10.10-10 (2010): 95.
- Zaharia, Matei, et al. "**Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing.**" Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012.
- Shi, Juwei, et al. "**Clash of the titans: Mapreduce vs. spark for large scale data analytics.**" Proceedings of the VLDB Endowment 8.13 (2015): 2110-2121.

Spark: Overview

- Focus : Applications that reuse a working set of data across multiple parallel operations
  - Iterative Machine learning
  - Interactive Data analysis
  - Systems like Pregel, did not provide ad hoc processing, or general reuse
  - General purpose programming interface
  - Resilient distributed datasets (RDDs)
    - Read only, persistent

RDDs

- Enable efficient data reuse in a broad range of applications
- Fault tolerant, parallel Data Structures
- Users can
  - Persist the RDDs in Memory
  - Control partitioning
  - Manipulate using rich set of operations
- coarse-grained transformations
  - Map, Filter, Join, applied to many data items concurrently
  - Keeps the lineage

RDDs

- RDD is represented by a Scala object
- Created in 4 ways
  - From a file in a shared file system
  - By "parallelizing" a Scala collection
  - Transforming an existing RDD
  - Changing the persistence of an existing RDD
    - Cache (hint, spill otherwise)
    - Save (to HDFS)

RDD / DSM

- RDDs can only be created through transformations
  - Immutable,
  - no need of check pointing
  - only lost partitions need to be recomputed
- Stragglers can be mitigated by running backup copies
- Tasks can be schedules based on data locality
- Degrade gracefully by spilling to disk
- Not suitable to asynchronous updates to shared state

Parallel Operations on RDDs

- Reduce
  - Combines dataset elements using an associative function to produce a result at the driver program.
- Collect
  - Sends all elements of the dataset to the driver program
- Foreach
  - Passes each element through a user provided function

Shared Variables

- Broadcast Variables
  - To Share a large read-only piece of data to be used in multiple parallel operations
- Accumulators:
  - These are variables that workers can only “add” to using an associative operation, and that only the driver can read.

MapReduce Vs Spark

- Shuffle: data is shuffled between computational stages, often involving sort, aggregation and combine operations.
- Execution model: parallelism among tasks, overlapping computation, data pipelining among stages
- Caching: reuse of intermediate data across stages at different levels

Problems

		Word Count	Sort	K-Means (LR)	Page-Rank
Type	One Pass	√	√		
	Iterative			√	√
Shuffle Sel.	High		√		
	Medium				√
	Low	√		√	
Job/Iter. Sel.	High		√		
	Medium				√
	Low	√		√	

Problems

		Word Count	Sort	K-Means (LR)	Page-Rank
Shuffle	Aggregation	√		√	√
	External sort		√		
	Data transfer		√		√
Execution	Task parallelism	√	√	√	√
	Stage overlap		√		
	Data pipelining				√
Caching	Input			√	√
	Intermediate data				√

WordCount

- Spark is 3x faster
- Fast initialization; hash-based combine better than sort-based combine
- For low shuffle selectivity workloads, hash-based aggregation in Spark is more efficient than sort-based aggregation in MapReduce due to the complexity differences in its in memory collection and combine components.

Sort

- In MapReduce, the reduce stage is faster than Spark because MapReduce can overlap the shuffle stage with the map stage, which effectively hides the network overhead.
- In Spark, the execution time of the map stage increases as the number of reduce tasks increase. This overhead is caused by and is proportional to the number of files opened simultaneously.

Iterative Algorithms

- CPU-bound iteration *caching the raw file* (to reduce disk I/O) may not help reduce the execution time since the disk I/O is hidden by the CPU overhead.
- In disk-bound, caching the raw file can significantly reduce the execution time.
- RDD caching can reduce not only disk I/O, but also the CPU overhead since it can cache any intermediate data during the analytic pipeline. For example, the main contribution of RDD caching for k-means is to cache the Point object to save the transformation cost from a text line, which is the

PageRank

- For graph analytic algorithms such as BFS and Community Detection that read the graph structure and iteratively exchange states through a shuffle, compared to MapReduce,
- Spark can avoid materializing graph data structures on HDFS across iterations, which reduces overheads for serialization/de-serialization, disk I/O, and network I/O.

Shortcomings of Mapreduce

- Force your pipeline into a Map and a Reduce steps
  - You might need to perform:
    - join
    - filter
    - map-reduce-map
- Read from disk for each Map/Reduce job
  - Expensive for some type of algorithm i.e:
    - Iterative algorithm : Machine learning

Shortcomings of Mapreduce

Solution?

- New framework: Support the same features of MapReduce and many more.

- Capable of reusing Hadoop ecosystem : e.g HDFS, YARN, etc.

  APACHE Spark™

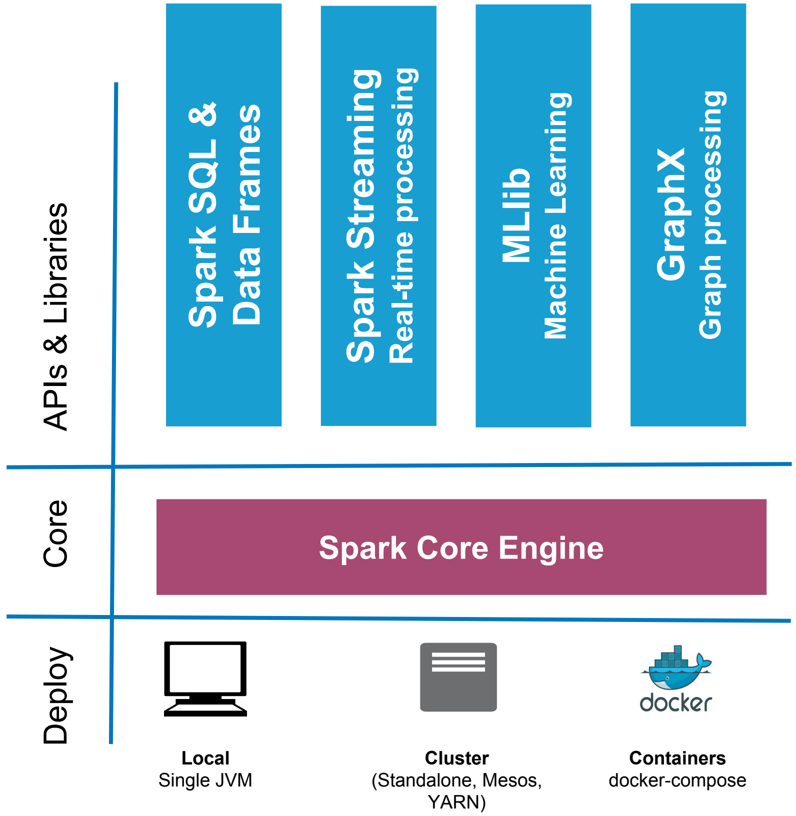
  - Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Introduction to Spark

- Apache Spark is an open-source **distributed** and highly scalable in-memory data processing and analytics system. It is a fast and general-purpose cluster computing system.
- It provides high-level APIs in Scala, Java, Python and R which and an optimized engine that supports general execution graphs.
- It also supports a rich set of higher-level tools including Spark SQL for SQL and structured data processing, MLlib for machine learning, GraphX for graph processing, and Spark Streaming.

Spark Stack

A unified analytics stack



Brief History of Spark

- Originally developed on UC Berkeley AMPLab in 2009.
- open-sourced in 2010.
- Spark paper came out.
- Spark Streaming was incorporated as a core component in 2011.
- In 2013, Spark was transferred to the Apache Software foundation.
- As of 2014, Spark is a top-level Apache project.
- July 26th, 2016 Spark 2.0 released with major improvements in every area over the 1.x branches

Who uses Spark and why?

Data Scientists:

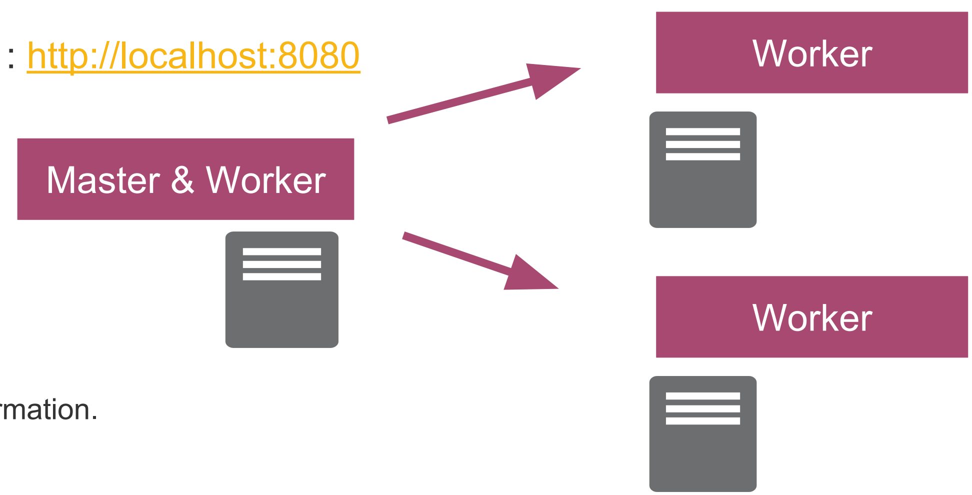
- Analyze and model data.
- Data transformations and prototyping.
- Statistics and Machine Learning.

Software Engineers:

- Data processing systems.
- API for distributed processing dataflow.
- Reliable, high performance and easy to monitor platform.

Installing Spark Standalone

- Spark runs on Windows and Linux-like operating systems.
- Download the Hadoop distribution you require under "Pre-build packages"
  - Place that compiled version on each node on the cluster
  - Run `./sbin/start-master.sh` to start a cluster
  - Once started, the master will show the spark://HOST:PORT url which you may need to connect workers to it.
    - Spark Master UI : http://localhost:8080



Check out Spark's website for more information.

Spark jobs and shell

- Spark jobs can be written on Scala, Java, Python or R.
- Spark native language is Scala, so it is natural to write Spark applications using Scala.
- The course will cover code examples written in Scala.

```
gezim@gezim-Latitude-E5550: /opt/spark-1.5.1/bin
gezim@gezim-Latitude-E5550:/opt/spark-1.5.1/bin$ ./spark-shell
17/03/20 14:17:30 WARN NativeCodeLoader: Unable to load native-hadoop library fo
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 1.5.1
      /_/

Using Scala version 2.10.4 (OpenJDK 64-Bit Server VM, Java 1.7.0_121)
Type in expressions to have them evaluated.
Type :help for more information.
17/03/20 14:17:39 WARN Utils: Your hostname, gezim-Latitude-E5550 resolves to a
17/03/20 14:17:39 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another
17/03/20 14:17:41 WARN MetricsSystem: Using default name DAGScheduler for source
Spark context available as sc.
17/03/20 14:17:45 WARN Connection: BoneCP specified but not present in CLASSPATH
17/03/20 14:17:45 WARN Connection: BoneCP specified but not present in CLASSPATH
17/03/20 14:17:56 WARN ObjectStore: Version information not found in metastore.
17/03/20 14:17:56 WARN ObjectStore: Failed to get database default, returning No
17/03/20 14:18:00 WARN NativeCodeLoader: Unable to load native-hadoop library fo
17/03/20 14:18:01 WARN Connection: BoneCP specified but not present in CLASSPATH
17/03/20 14:18:01 WARN Connection: BoneCP specified but not present in CLASSPATH
17/03/20 14:18:10 WARN ObjectStore: Version information not found in metastore.
17/03/20 14:18:11 WARN ObjectStore: Failed to get database default, returning No
SQL context available as sqlContext.

scala>
```

About Scala

- High-level language for the JVM
  - Object Oriented + Functional Programming
- Statically typed
  - Comparable in speed to Java
  - Type inference saves us from having to write explicit types most of the time
- Interoperates with Java
  - Can use any Java class (inherit from, etc.)
  - Can be called from Java code

Introduction to Scala

Scala syntax | Java equivalent

Declaring variables

Scala syntax:

```scala
var x: Int = 7
var x = 7 // type inferred
val y = "hi" // read-only
```

Java equivalent:

```java
int x = 7;
final String y = "hi";
```

Functions

Scala syntax:

```scala
def square(x: Int): Int = x*x
def square(x: Int): Int = {
    x*x
  }
def announce(text: String) = {
    println(text)
}
```

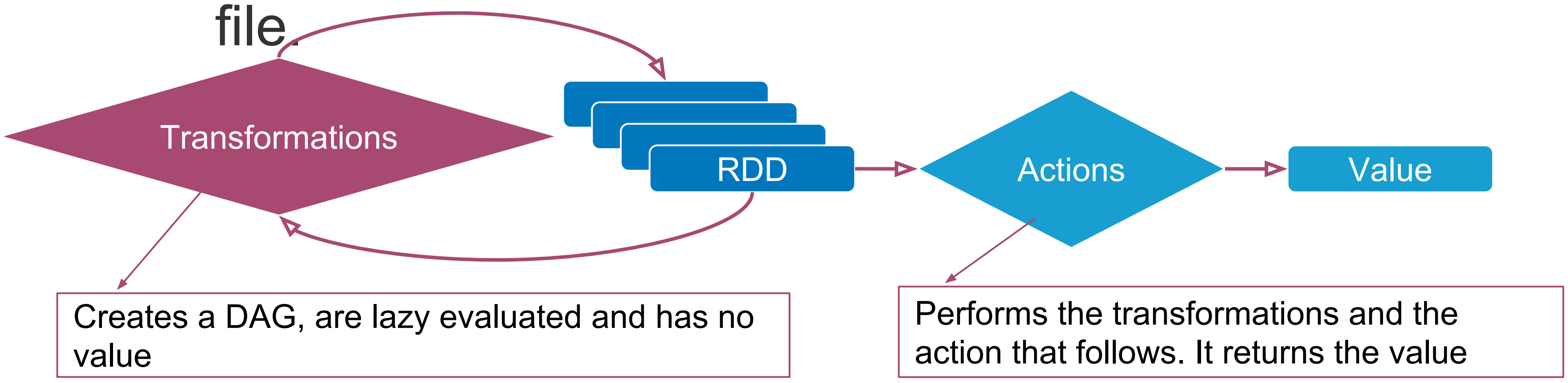
Java equivalent:

```java
int square(int x) {
    return x * x;
}

void announce(String text) {
    System.out.println(text);
}
```

Resilient Distributed Dataset (RDD)

- Spark's primarily abstraction.
- Distributed collection of elements, partitioned across the cluster.
  - **Resilient**: recreated, when data in-memory is lost.
  - **Distributed**: partitioned in-memory across the cluster
  - **Dataset**: list of collection or data that comes from file.

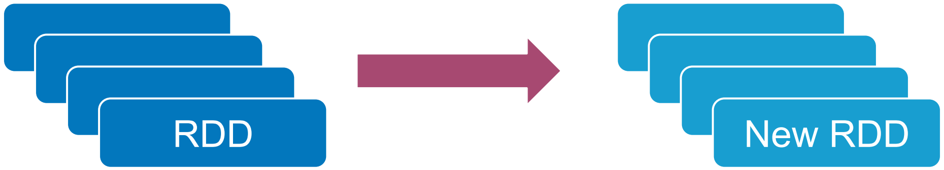


Creating RDDs

- Launch the Spark shell
  - `./bin/spark-shell` *//sc: SparkContext instance*
- Create some sample data and parallelize by creating and RDD
  - `val data = 1 to 1000`
  - `val distData =sc.parallelize(data)`
- Afterwards, you could perform any additional transformation or action on top of these RDDs:
  - `distData.map { x => ??? }`
  - `distData.filter { x => ??? }`
- An RDD can be created by external dataset as well:
  - `val readmeFile = sc.textFile("README.md")`

RDD Operations

- **Transformations**: define new RDDs based on existing one (f(RDD) => RDD), e.g filter, map, reduce, groupBy, etc.

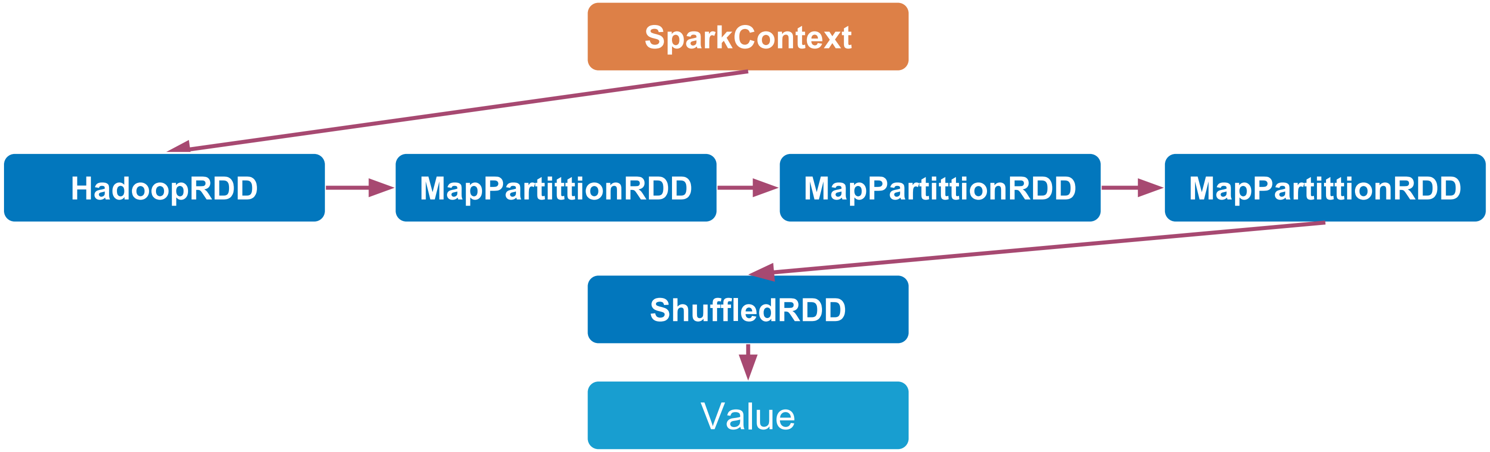


- **Actions**: returns values, e.g count, sum, collect, take, etc.

RDD Operations

Word Count example

```
val textFile = sparkSession.sparkContext.textFile("hdfs://...")
val wordCounts = textFile.flatMap(line => line.split(" "))
                         .filter(!_.isEmpty())
                         .map(word => (word, 1))
                         .reduceByKey(_ + _) //(a, b) => a + b
wordCounts.take(10)
```



Directed Acyclic Graph (DAG) for Word Count example

RDD Operations

Expressive and Rich API

map
filter
groupBy
sort
union
join
leftOuterJoin
rightOuterJoin

reduce
count
fold
reduceByKey
groupByKey
cogroup
cross
zip

sample
take
first
partitionBy
mapWith
pipe
save
...

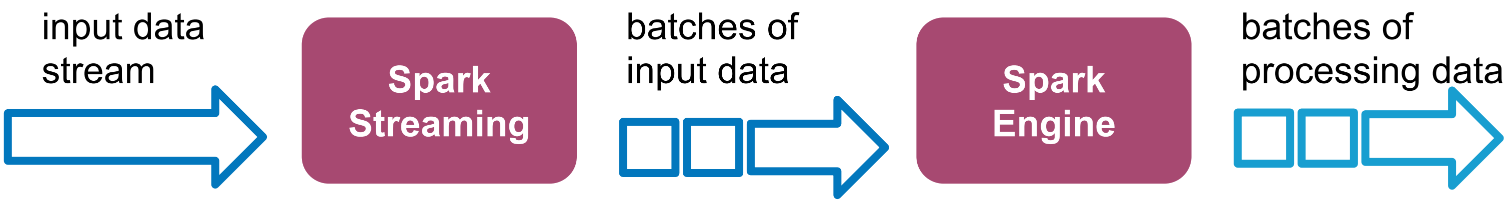
RDD Operations

Transformations and actions available on RDDs in Spark.

Transformations	$map(f : \mathrm{T} \Rightarrow \mathrm{U})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$filter(f : \mathrm{T} \Rightarrow \mathrm{Bool})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$
	$flatMap(f : \mathrm{T} \Rightarrow \mathrm{Seq[U]})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[U]}$
	$sample(fraction : \mathrm{Float})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{RDD[T]}$ (Deterministic sampling)
	$groupByKey()$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, Seq[V])]}$
	$reduceByKey(f : (\mathrm{V}, \mathrm{V}) \Rightarrow \mathrm{V})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$union()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[T]}) \Rightarrow \mathrm{RDD[T]}$
	$join()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (V, W))]}$
	$cogroup()$	:	$(\mathrm{RDD[(K, V)]}, \mathrm{RDD[(K, W)]}) \Rightarrow \mathrm{RDD[(K, (Seq[V], Seq[W]))]}$
	$crossProduct()$	:	$(\mathrm{RDD[T]}, \mathrm{RDD[U]}) \Rightarrow \mathrm{RDD[(T, U)]}$
	$mapValues(f : \mathrm{V} \Rightarrow \mathrm{W})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, W)]}$ (Preserves partitioning)
	$sort(c : \mathrm{Comparator[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
	$partitionBy(p : \mathrm{Partitioner[K]})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{RDD[(K, V)]}$
Actions	$count()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Long}$
	$collect()$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{Seq[T]}$
	$reduce(f : (\mathrm{T}, \mathrm{T}) \Rightarrow \mathrm{T})$	:	$\mathrm{RDD[T]} \Rightarrow \mathrm{T}$
	$lookup(k : \mathrm{K})$	:	$\mathrm{RDD[(K, V)]} \Rightarrow \mathrm{Seq[V]}$ (On hash/range partitioned RDDs)
	$save(path : \mathrm{String})$	:	Outputs RDD to a storage system, *e.g.*, HDFS

Spark Streaming

- Many important application must process large data streams at second-scale latencies.
- An extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.



- A high-level abstraction called discretized stream or DStream, a continuous stream of data. Internally, a DStream is represented as a sequence of RDDs.

Network Word Count example

```
// Create a DStream that will connect to hostname:port, like localhost:9999
val lines = ssc.socketTextStream("localhost", 9999)
val workCountss = lines.flatMap(_.split(" "))// Split each line into words
                              .map(x => (x, 1))
                              .reduceByKey(_ + _)// Count each word in each batch
workCountss.print()

ssc.start()// Start the computation
ssc.awaitTermination()// Wait for the computation to terminate
```

- After we submit our job we have to run the socket server for testing:
  - Login to your *shell* and run the following command to launch Netcat: `nc -l localhost 9999`
  - For example, if you type the following `hello hello world` text in the Netcat window, you should see the following output in the already running Spark Streaming job tab or window:

```
(hello, 2)
(world, 1)
```

DataFrames in Spark

- Distributed collection of data grouped into named columns (i.e. RDD with schema).

- DSL designed for common tasks
  - Metadata
  - Sampling
  - Project, filter, aggregation, join, …
  - UDFs

- Available in Scala, Java, Python, and R (via SparkR)

DataFrames in Spark

Word Count example

```
import sparkSession.sqlContext.implicits._ //necessary to convert the RDD to a DataFrame.

val textFile = sparkSession.sparkContext.textFile(input)

val wordCountDF = textFile.flatMap(line => line.split(" "))
                          .filter(!_.isEmpty())
                          .map(word => (word, 1))
                          .reduceByKey(_ + _)
                          .toDF("word","count")

wordCountDF.groupBy("word").count()
wordCountDF.show(100)
wordCountDF.select("word").show(5)
```

```
|                word|count|
+--------------------+-----+
|               Quail|    1|
|            "Created|    1|
|        "Lugares"@en|    1|
|               under|    1|
|           "Hunebeds|    1|
|           "File:Two|    1|
|               Diese|    1|
|                "Own|   19|
|"2004-07-22"^^<ht...|    5|
```

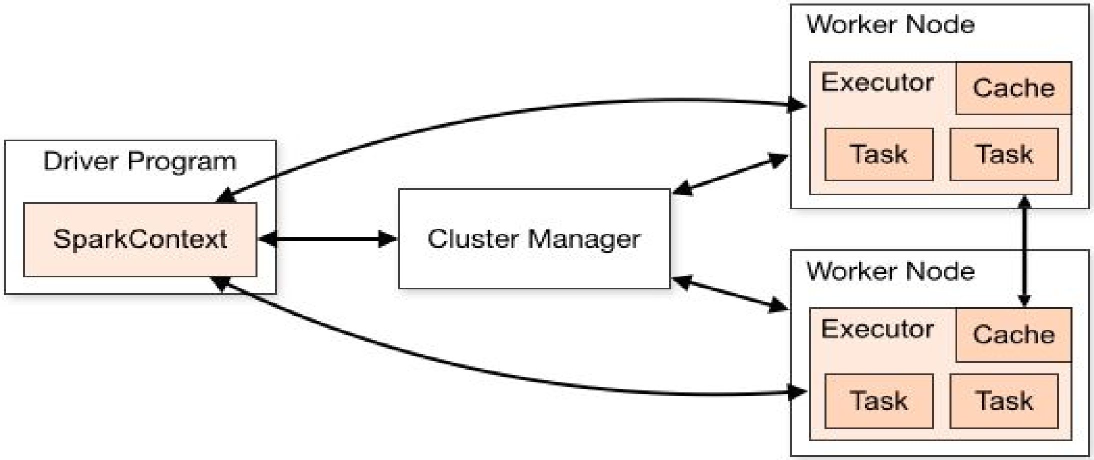
Spark application

- Spark Standalone Application on Scala
  - `git clone https://github.com/SANSA-Stack/SANSA-Template-Maven-Spark.git`
  - Import it on your IDE as Maven/General(sbt) project and create a new Scala class.

- Run a standalone application
  - Use any tools (sbt, maven) for defining dependencies
  - Generate a JAR package containing your application's code
    - Use sbt/mvn build package
  - Use `spark-submit` to run the program
    - ```
      ./bin/spark-submit   --class <main-class>   --master <master-url>
      <application-jar> \
      [application-arguments]
      ```
      ```

Spark configurations

- Spark Cluster Overview
  - **Components**
    - Driver aka `SparkContext`
    - Cluster Manager (Standalone, Apache Mesos, Hadoop YARN)
    - Executers



Spark monitoring

- Web Interfaces
  - **WebUI**
    - Every `SparkContext` launches a web UI, on port 4040, that displays useful information about the application.
  - **Metrices**
    - Spark has a configurable metrics system based on the Dropwizard Metrics Library. This allows users to report Spark metrics to a variety of sinks including HTTP, JMX, and CSV files.
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

Spark tuning

- Tuning Spark
  - **Data Serialization**
    - It plays an important role in the performance of any distributed application.
      - Java serialization
      - Kryo serialization
  - **Memory Tuning**
    - The amount of memory used by your objects (you may want your entire dataset to fit in memory), the cost of accessing those objects, and the overhead of garbage collection (if you have high turnover in terms of objects).
  - **Advanced Instrumentation**
    - Several external tools can be used to help profile the performance of Spark jobs.

References

[1]. “Spark Programming Guide” - http://spark.apache.org/docs/latest/programming-guide.html

[2]. “Spark Streaming Programming Guide”- http://spark.apache.org/docs/latest/streaming-programming-guide.html

[3]. “Spark Cluster Overview” - http://spark.apache.org/docs/latest/cluster-overview.html

[4]. “Spark Configuration” - http://spark.apache.org/docs/latest/configuration.html

[5]. “Spark Monitoring” - http://spark.apache.org/docs/latest/monitoring.html

[6]. “Spark tuning” - http://spark.apache.org/docs/latest/tuning.html