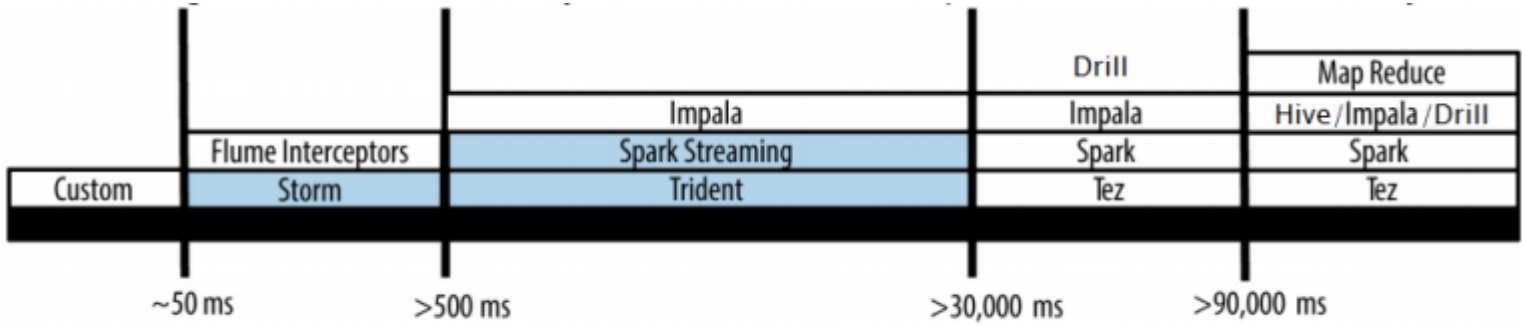
Spark & SparkStreaming

ixAT Solutions – ixatsolutions@gmail.com

Introduction

- Apache Spark is an open source big data processing framework built around speed, ease of use, and sophisticated analytics. It was originally developed in 2009 in UC Berkeley's AMPLab, and open sourced in 2010 as an Apache project.

Realtime, NRTime and Batch Processing

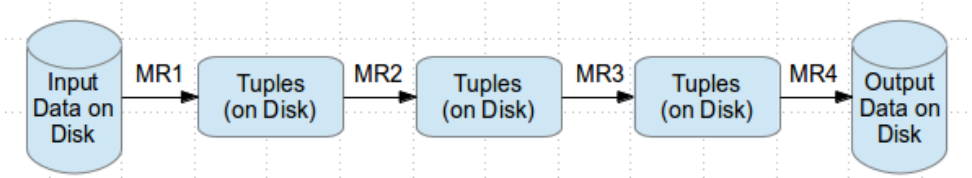


Hadoop and Spark

- MapReduce is a great solution for one-pass computations, but not very efficient for use cases that require multi-pass computations and algorithms. Each step in the data processing workflow has one Map phase and one Reduce phase and you'll need to convert any use case into MapReduce pattern to leverage this solution.

- The Job output data between each step has to be stored in the distributed file system before the next step can begin. Hence, this approach tends to be slow due to replication & disk storage.

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



Hadoop and Spark

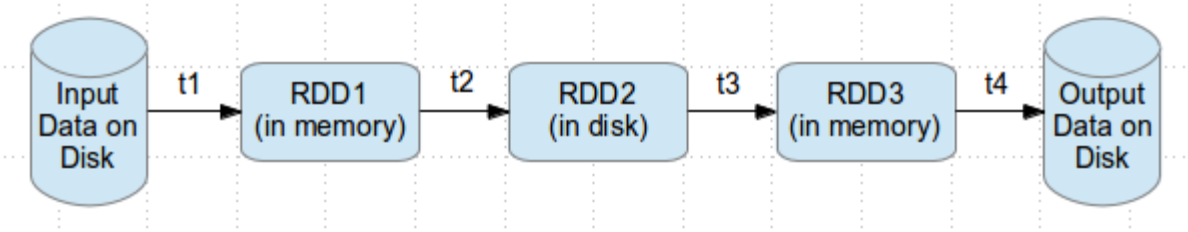
- Spark allows programmers to develop complex, multi-step data pipelines using directed acyclic graph (DAG) pattern. It also supports in-memory data sharing across DAGs, so that different jobs can work with the same data.

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

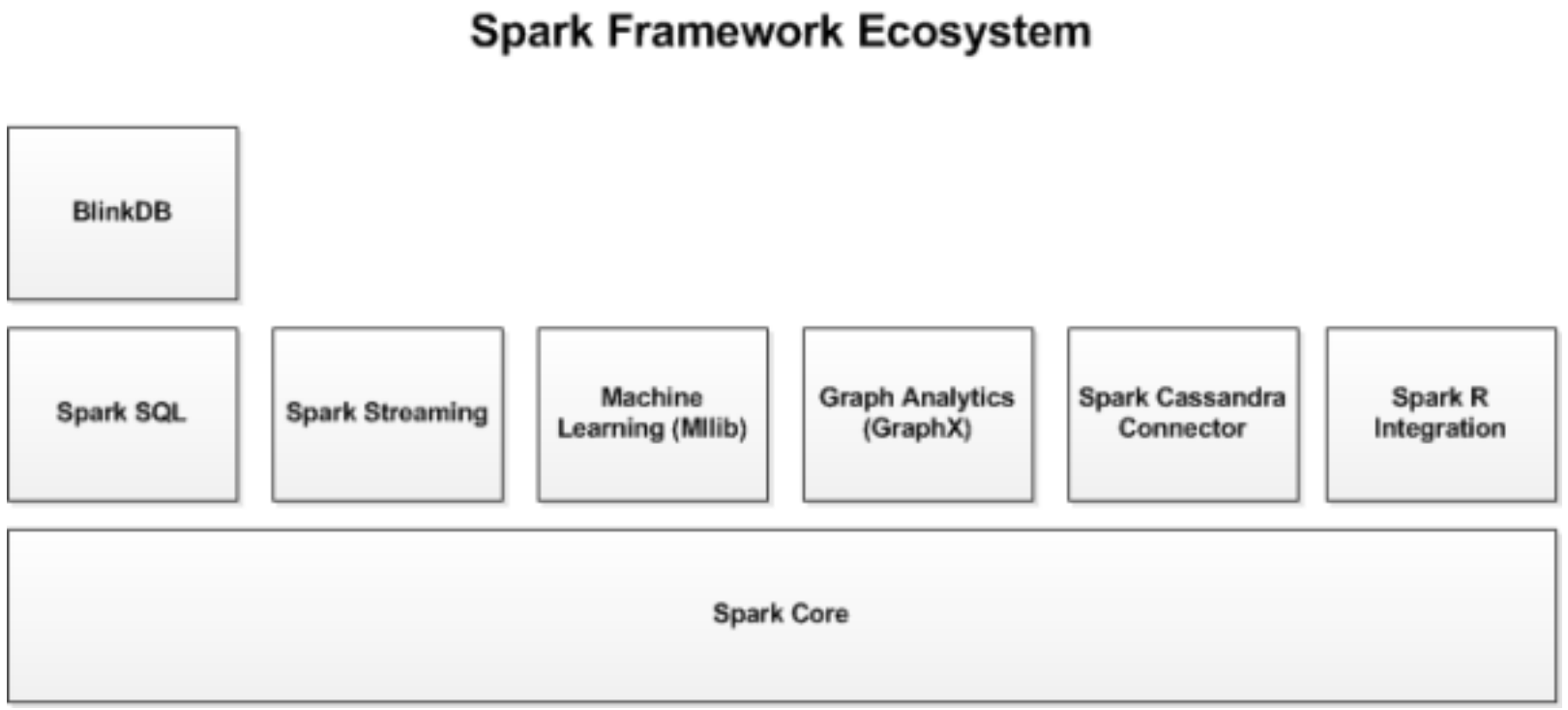
- We should look at Spark as an alternative to Hadoop MapReduce rather than a replacement to Hadoop. It's not intended to replace Hadoop but to provide a comprehensive and unified solution to manage different big data use cases and requirements.

Spark features

- Spark is written in Scala Programming Language and runs on Java Virtual Machine (JVM) environment
- Supports more than just Map and Reduce functions.
- Optimizes arbitrary operator graphs.
- Lazy evaluation of big data queries which helps with the optimization of the overall data processing workflow.
- Provides concise and consistent APIs in Scala, Java and Python.
- Offers interactive shell for Scala and Python. This is not available in Java yet.



Spark Ecosystem



Spark Architecture

- **Data Storage** - Spark uses HDFS file system for data storage purposes. It works with any Hadoop compatible data source including HDFS, HBase, Cassandra, etc. Other Dist File systems such as S3, Gluster etc... are also supported. One could also use Memory centric Distributed Filesystems such as Alluxio in Spark.

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

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

Spark Installation

- Download spark from http://d3kbcqa49mib13.cloudfront.net/spark-1.6.0-bin-hadoop2.6.tgz
- Untar and set path etc… (to bin and sbin folders of unarchived tar)
- Create $SPARK_HOME/conf/spark-env.sh file with following content
  - SPARK_JAVA_OPTS=-Dspark.driver.port=53411
  - HADOOP_CONF_DIR=$HADOOP_HOME/etc/hadoop
  - SPARK_MASTER_IP=localhost
- Create $SPARK_HOME/conf/spark-defaults.conf with the following content
  - spark.master spark://localhost:7077
  - spark.serializer org.apache.spark.serializer.KryoSerializer
- Also create $SPARK_HOME/conf/slaves file with all slave node IP's/Names (for now, its localhost)
- Start DFS, YARN
- Start Spark via $SPARK_HOME/sbin/start-all.sh

Run a test - wordcount

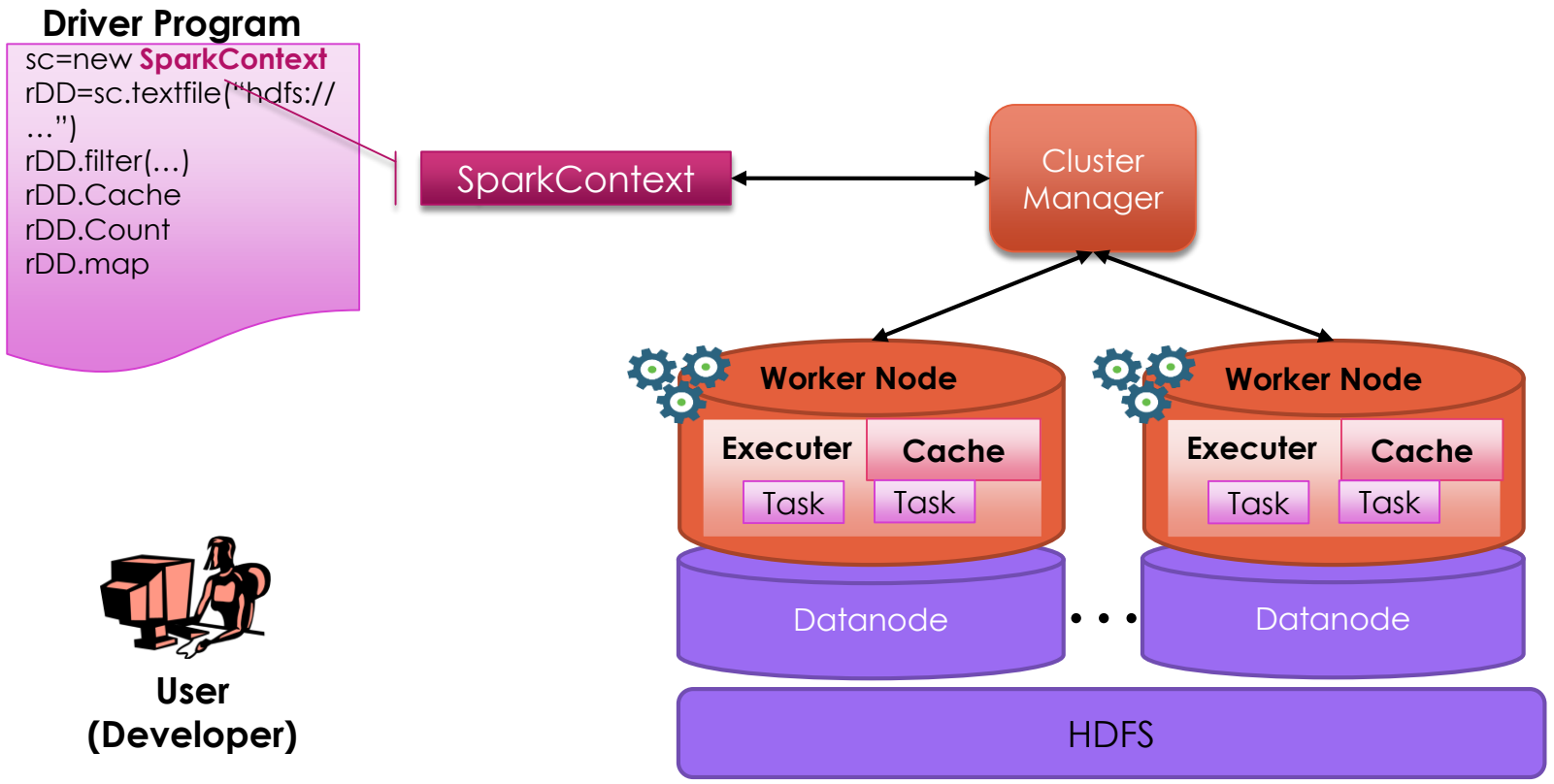
- Create a text file
- Upload the file to HDFS, say /test.txt
- Open spark-shell
- Run the following in spark-shell

```
sc.textFile("/test.txt").flatMap( line => line.split(" ")).map(word =>
(word,1)).reduceByKey(_+_).sortBy(_._2, false).collect().foreach(println)
```

- The above scala code does an MR for wordCount including Sort By Values in descending order, you could store the results back to HDFS using saveAsTextFile API

```
sc.textFile("/test.txt").flatMap( line => line.split(" ")).map(word =>
(word,1)).reduceByKey(_+_).sortBy(_._2, false).saveAsTextFile("/output")
```

Architecture



RRD – Resilient Distributed Dataset

- Resilient Distributed Dataset (RDD) is a basic Abstraction in Spark
- Immutable, Partitioned collection of elements that can be operated in parallel
- The **distributed memory abstractions** that lets programmer perform **in-memory parallel computations** on **large clusters**. And that too in a **highly fault tolerant** manner.

RRD Operations

- **Programming Interface:** Programmer can perform 3 types of operations:

Transformations

- Create a new dataset from and existing one.
- Lazy in nature. They are executed only when some action is performed.
- Example :
  - Map(func)
  - Filter(func)
  - Distinct()

Actions

- Returns to the driver program a value or exports data to a storage system after performing a computation.
- Example:
  - Count()
  - Reduce(funct)
  - Collect
  - Take()

Persistence

- For caching datasets in-memory for future operations.
- Option to store on disk or RAM or mixed (Storage Level).
- Example:
  - Persist()
  - Cache()

The RRD's in WordCount

```
val tf = sc.textFile("/test.txt") → RRD0

tf.count() →An action on RRD0

val fm = tf.flatMap( line => line.split(" ")) → RRD1

fm.collect()  → An action

val themap = fm.map(word => (word,1)) → RRD2

themap.collect → An Action

val reducedMap = themap.reduceByKey(_+_) → RRD3
reducedMap.collect → An Action

val sortedreducedMap = reducedMap.sortBy(_._2, false) → RRD4
sortedreducedMap.collect → An Action
sortedreducedMap.saveAsTextFile("/output") → Persistance
```

Spark In Java

```
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {

  public static void main(String[] args) {

    String logFile = "/test.txt";
    SparkConf conf = new
SparkConf().setAppName("SparkWC").setMaster("local");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> lines = sc.textFile("hdfs://test.txt");

// Map each line to multiple words
JavaRDD<String> words = lines.flatMap(
  new FlatMapFunction<String, String>() {
    public Iterable<String> call(String line) {
      return Arrays.asList(line.split(" "));
    }
});
```

```
// Turn the words into (word, 1) pairs
JavaPairRDD<String, Integer> ones = words.mapToPair(
  new PairFunction<String, String, Integer>() {
    public Tuple2<String, Integer> call(String w) {
      return new Tuple2<String, Integer>(w, 1);
    }
});

// Group up and add the pairs by key to produce counts
JavaPairRDD<String, Integer> counts = ones.reduceByKey(
  new Function2<Integer, Integer, Integer>() {
    public Integer call(Integer i1, Integer i2) {
      return i1 + i2;
    }
});

counts.saveAsTextFile("hdfs://output/counts.txt");

    System.out.println("Done….");
  }
}
```

Spark In Java8

```
/* SimpleApp.java */
import org.apache.spark.api.java.*;
import org.apache.spark.SparkConf;
import org.apache.spark.api.java.function.Function;

public class SimpleApp {

  public static void main(String[] args) {

    String logFile = "/test.txt";
    SparkConf conf = new SparkConf().setAppName("SparkWC").setMaster("local");
    JavaSparkContext sc = new JavaSparkContext(conf);
    JavaRDD<String> lines = sc.textFile("hdfs://test.txt");

     JavaRDD<String> words =  lines.flatMap(line -> Arrays.asList(line.split(" ")));
     JavaPairRDD<String, Integer> counts =   words.mapToPair(w -> new Tuple2<String, Integer>(w, 1)).reduceByKey((x, y) -> x +
     y);
     counts.saveAsTextFile("hdfs://output/counts.txt");
     System.out.println("Done….");
       }
     }
```

Stocks processing

- sc.textFile("/stocks.csv").map(_.split(",")).map(rec=>(rec(0),rec(6).toLong)).reduceByKey(_+_).collect.foreach(println)

- Spark has a great set of expressive API's which are a fun to work with in scala

- map
- filter
- groupBy
- sort
- union
- join
- leftOuterJoin
- rightOuterJoin
- reduce
- count
- fold
- reduceByKey
- groupByKey
- cogroup
- cross
- zip

sample
take
first
partitionBy
mapWith
pipe
save
...

Spark Streaming

- Spark Streaming is an extension of the core Spark API that enables scalable, high-throughput, fault-tolerant stream processing of live data streams.
- Data can be ingested from many sources like Kafka, Flume, Twitter, ZeroMQ, Kinesis, or TCP sockets, and can be processed using complex algorithms expressed with high-level functions like map, reduce, join and window.
- Finally, processed data can be pushed out to filesystems, databases, and live dashboards
- Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.

Window Streaming on Stocks

- Create a file stocks.scala with below content

```
import org.apache.spark.streaming.{Seconds, StreamingContext}
import org.apache.log4j.{Level, Logger}

val rootLogger = Logger.getRootLogger()
rootLogger.setLevel(Level.ERROR)

val ssc = new StreamingContext(sc, Seconds(5))

ssc.textFileStream("/stocks").map(_.split(",")).map(rec=>(rec(0),rec(6).toLong)).reduceByK
ey(_+_).print()
ssc.start()
ssc.awaitTermination()
```

- Run stocks.scala using the below command

```
spark-shell -i stocks.scala
```

- In another window, create some stocks file and push that to /stocks directory in HDFS, and observe the streaming application