SPARK INTRODUCTION

Lecturer ***: Asso Prof. Dr. Thoai Nam***

Student : Le Nguyen Truong Giang – 157027

Pham Minh Khue – 1570213

Nguyen Xuan Trai – 1570751

Nguyen Hieu Thinh - 51203624

Index

- Spark Introduction
- Resilient Distributed Dataset - RDD
- Spark Application Programming
- Spark Libraries
- Spark Configuration

Spark Introduction

❑ Big Data and Spark

- ➢ Big Data: Volume, Velocity, Variety
  -> Need of faster results from analytics

- ➢ Apache Spark: a computing platform
  - ○ Speed
  - ○ Generality
  - ○ Ease of use

Spark Introduction

❑ Advance of Spark

- ➢ Parallel distributed processing, fault tolerance on commodity hardware, scalability, in-memory computing, high level APIs, …
- ➢ Save time and money
- ➢ Who?
  - ○ Data scientist
  - ○ Engineers
  - ○ Everyone

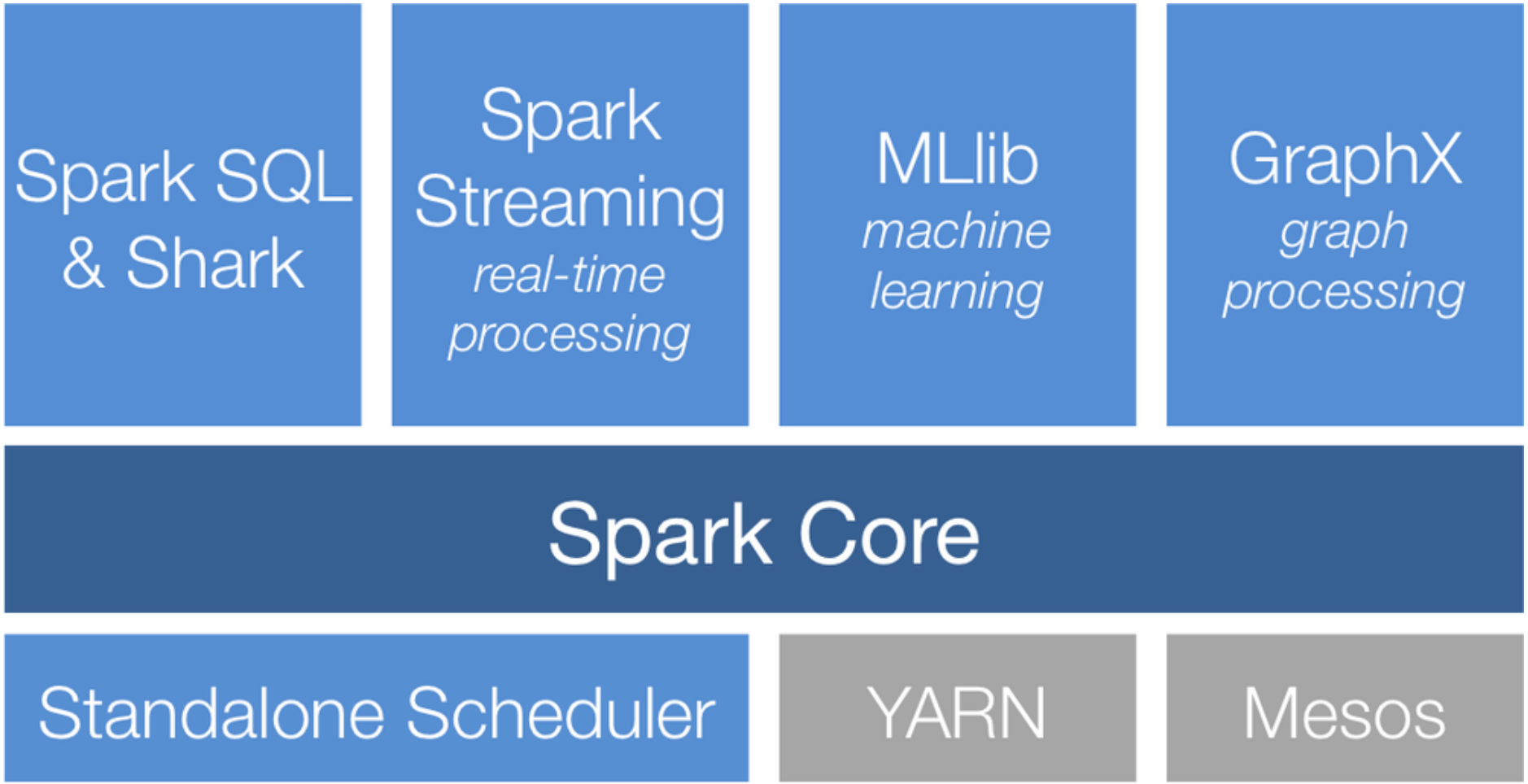
Spark Introduction

❑ Brief History of Spark

- ➢ 2002: MapReduce @ Google
- ➢ 2004: MapReduce paper
- ➢ 2006: Hadoop @ Yahoo
- ➢ 2008: Hadoop Summit
- ➢ 2010: Spark paper
- ➢ 2014: Apache Spark top-level

Spark Introduction

- Spark Unified Stack



Spark Introduction

- Downloading and install Spark standalone
  - Run on Windows, Linux, Mac OS
  - Pre-required: Java installed
  - Download: Hadoop distribution (Pre-built packages) in http://spark.apache.org
  - Start the cluster: /sbin/start-master.sh
  - Connect worker to it: http://localhost:8080
  - Scala shell: ./bin/spark-shell
  - Python shell: ./bin/pyspark

Resilient Distributed Dataset - RDD

❑ Resilient Distributed Datasets - RDD

- ➢ Spark’s primary abstraction
- ➢ Distributed collection of elements
- ➢ Parallelized across the cluster
- ➢ Three methods for creating RDD
  - ○ Parallelizing an existing collection
  - ○ Referencing a dataset
  - ○ Transformation from an existing RDD

Resilient Distributed Dataset - RDD

❑ Resilient Distributed Datasets - RDD

- ➢ Two types of RDD operations
  - ○ Transformations
    - • Create a DAG
    - • Lazy Evaluations
    - • No return value
  - ○Actions
    - • Performs the transformations and the action that follows
    - • Return a value
- ➢ Fault tolerance
- ➢ Caching

Resilient Distributed Dataset - RDD

❑ Resilient Distributed Datasets - RDD

- ➢ Dataset from any storage supported by Hadoop
  - ○ HDFS
  - ○ Cassandra
  - ○ HBase
  - ○ Amazon S3
- ➢ Type of files supported
  - ○ Text files
  - ○ SequenceFiles
  - ○ Hadoop InputFormat

Resilient Distributed Dataset - RDD

❑ Creating an RDD

- ➢ Launch the Spark shell

  ./bin/spark-shell

- ➢ Create some data

  val data = 1 to 10000

- ➢ Parallelize the data (creating the RDD)

  val distData = sc.parallelize(data)

- ➢ Or, create an RDD from an external dataset

  val readmeFile = sc.textFile(“Readme.md”)

Resilient Distributed Dataset - RDD

- RDD operations (basics)
  - Load a file

```
val lines = sc.TextFile("hdfs://data.txt")
```

  - Applying a transformation

```
val lineLengths = lines.map(s => s.length)
```

  - Invoking action

```
val totalLengths = lineLengths.reduce((a,b) =>
                                        a + b)
```

Resilient Distributed Dataset - RDD

- RDD operations (basics)
  - MapReduce example:

```
val wordCounts = textFile.flatMap(line =>.split(“ “)).
                          map(word => (word,1)).
                          reduceByKey((a,b) => a + b)

wordCounts.collect()
```

Spark Application Programming

Objectives:

- Understand the purpose and usage of the SparkContext
- Initialize Spark with the various programming languages (Scala, Java and Python)
- Describe and run some Spark examples
- Pass functions to Spark

SparkContext

- The main entry point for Spark functionally
- Represents the connection to a Spark cluster
- Create RDDs, accumulators, and broadcast variables on that cluster
- In the Spark shell, the SparkContext is automatically initialized.
- In Spark program, import some classes:

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

RDD

RDD Operations

```
16  object SparkWordCount extends WordCountBlueprint {
17
18    sc.textFile("./src/main/resources/data/words")
19      .flatMap(_.split("\\s+"))
20      .map(word => (clean(word), 1))
21      .reduceByKey(_ + _)
22      .collect foreach println
23  }
```

Transformation

Action

Accumulators

Accumulator Example

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

val sum = sc.accumulator(0)
val count = sc.accumulator(0)

input
  .filter(line => line.size > 0)
  .flatMap(line => line.split(" "))
  .map(word => word.size)
  .foreach{
      size =>
        sum += size // increment accumulator
        count += 1  // increment accumulator
  }

val average = sum.value.toDouble / count.value
```

initialize the accumulators

driver only

Broadcast variable

Example with broadcast variables

```
// RDD[(String, String)]
val names = … //load (URL, page name) tuples

// RDD[(String, Int)]
val visits = … //load (URL, visit counts) tuples

// Map[String, String]
val pageMap = names.collect.toMap

val bcMap = sc.broadcast(pageMap)

val joined = visits.map{
  case (url, counts) =>
    (url, (bcMap.value(url), counts))
}
```

Broadcast variable

pageMap is sent only to each node once

Linking Spark with Scala

Build a SparkConf object that contains information about the application

```
//Start the Spark context
val conf = new SparkConf().setAppName("WordCount").setMaster("local")
val sc = new SparkContext(conf)
```

The appName -> Name for the application to show on the cluster UI

The master parameter -> a Spark URL to run Spark in modes.

Passing functions to Spark

Spark's API relies on heavily passing functions in the driver program to run on the cluster

- Anonymous function syntax:

```
(x:Int) => x + 1
```

- Static methods in a global singleton object:

```
object MyFunctions {
  def func1(s: String): String = { ... }
}

myRdd.map(MyFunctions.func1)
```

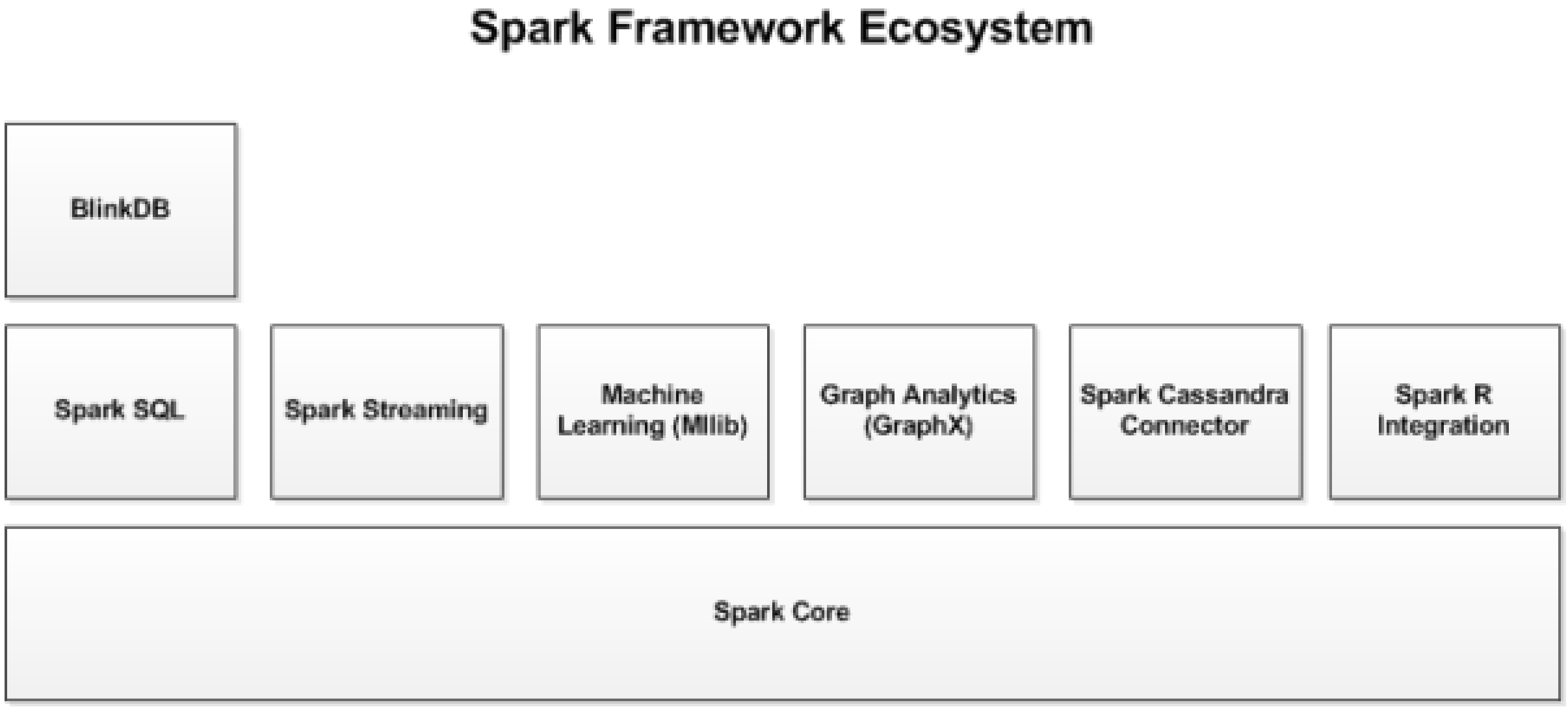
- Passing by reference:

```
class MyClass {
  def func1(s: String): String = { ... }
  def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }
}
```

Example

```scala
package test

import org.apache.spark.SparkConf
import org.apache.spark.SparkContext
import org.apache.spark.rdd.RDD.rddToPairRDDFunctions
object WordCount {
  def main(args: Array[String]) = {

    //Start the Spark context
    val conf = new SparkConf()
      .setAppName("WordCount")
      .setMaster("local")
    val sc = new SparkContext(conf)

    //Read some example file to a test RDD
    val test = sc.textFile("words.txt")

    test.flatMap { line => //for each line
      line.split(" ") //split the line in word by word.
    }
      .map { word => //for each word
        (word, 1) //Return a key/value tuple, with the word as key and 1 as value
      }
      .reduceByKey(_ + _) //Sum all of the value with same key
      .saveAsTextFile("output") //Save to a text file

    //Stop the Spark context
    sc.stop
  }
}
```

Spark Libraries



Spark Libraries - Spark SQL

- Spark SQL, part of Apache Spark big data framework, is used for structured data processing and allows running SQL like queries on Spark data
- Componence:
  - DataFrame
  - SQLContext

Spark Libraries - Spark SQL

- DataFrames

DataFrames can be created from different data sources such as:

- Existing RDDs
- Structured data files
- JSON datasets
- Hive tables
- External databases

Spark Libraries - Spark SQL

- **SQLContext**
  - Following code snippet shows how to create a SQLContext object.

```
val sqlContext = new
org.apache.spark.sql.SQLContext(sc)
```

  - There is also HiveContext which provides a superset of the functionality provided by SQLContext. It can be used to write queries using the HiveQL parser and read data from Hive tables.

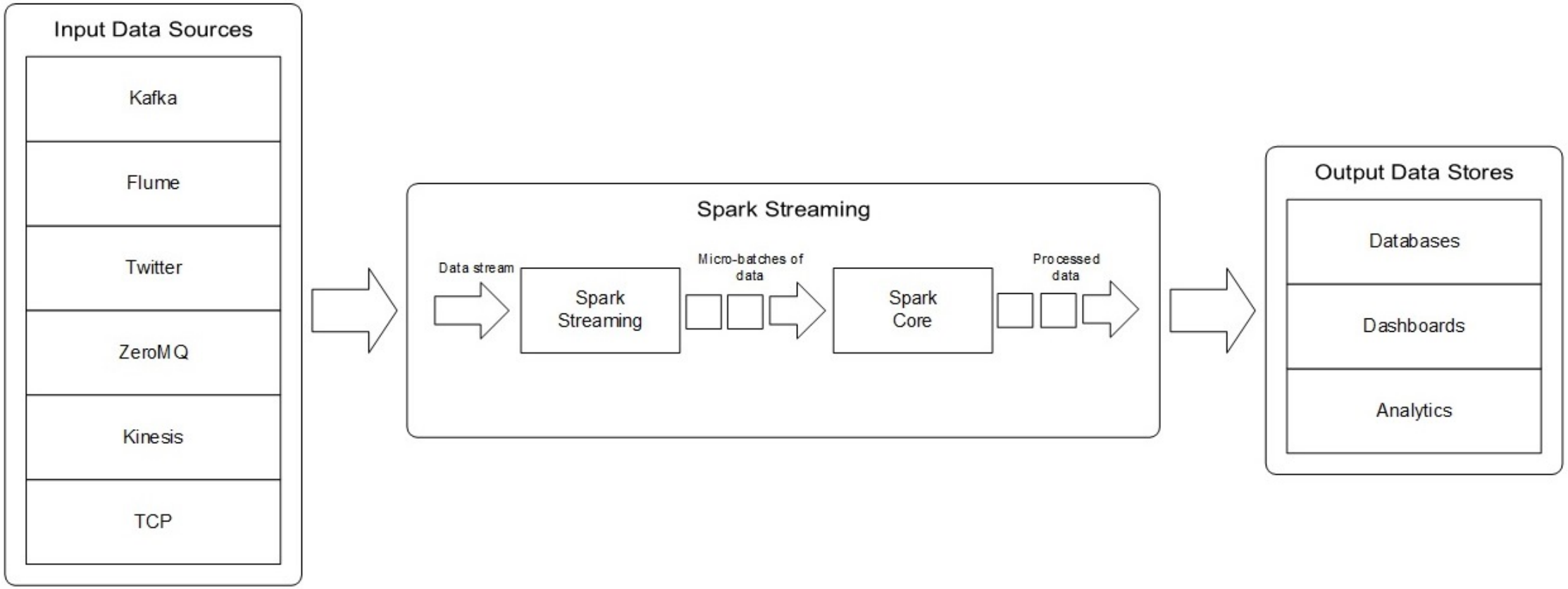
Spark Libraries - Spark Streaming

- Streaming data is basically a continuous group of data records generated from sources like sensors, server traffic and online searches.

- Streaming data processing applications help with live dashboards, real-time online recommendations, and instant fraud detection.

Spark Libraries - Spark Streaming

- Spark Streaming is an extension of core Spark API. Spark Streaming makes it easy to build fault-tolerant processing of real-time data streams



Spark Libraries - Spark MLlib

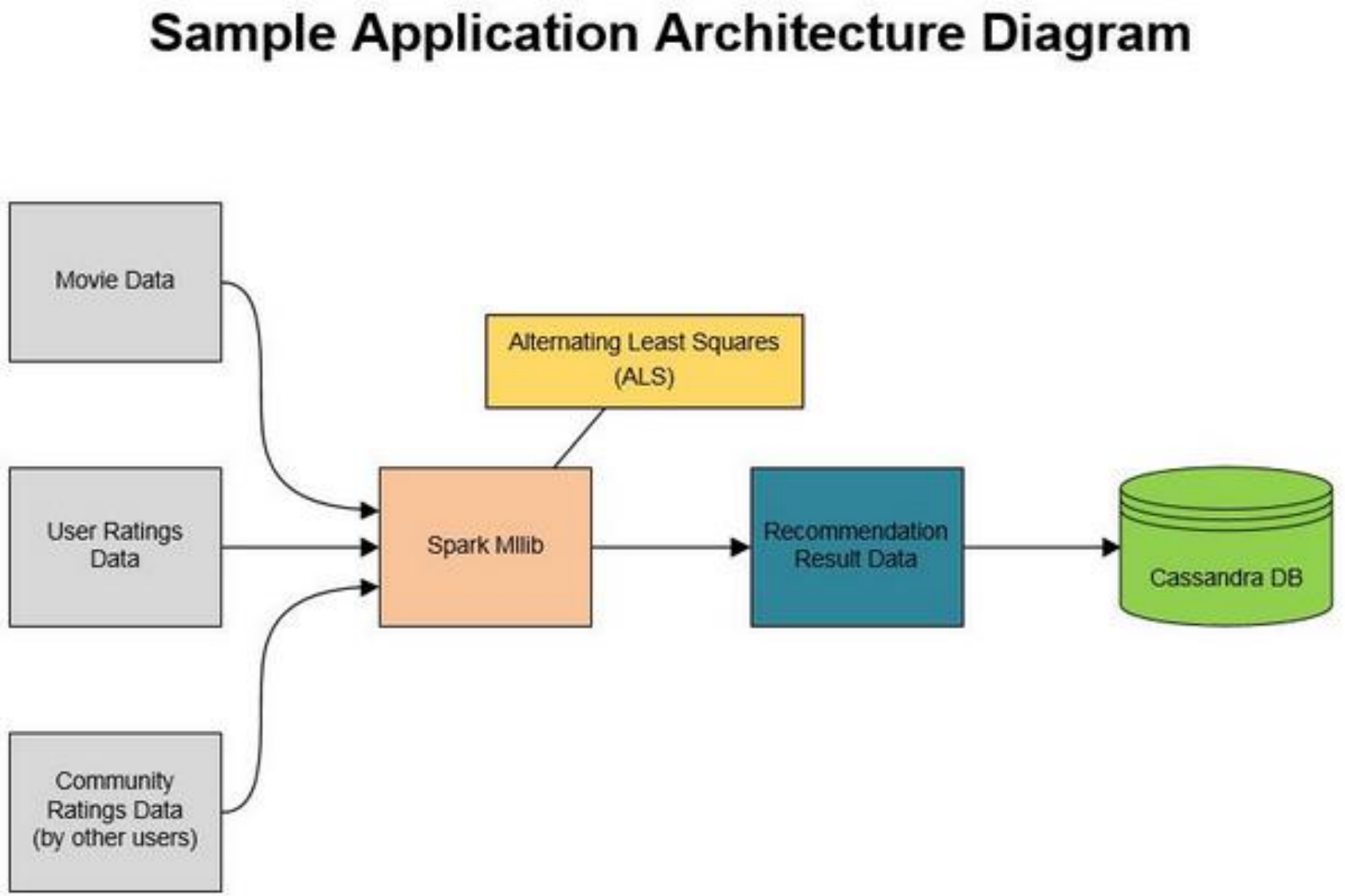
- MLlib is Spark's scalable machine learning library consisting of common learning algorithms and utilities, including classification, regression, clustering, collaborative filtering, dimensionality reduction, as well as underlying optimization primitives
- Spark Machine Learning API includes two packages called **spark.mllib** and **spark.ml**.

Spark Libraries - Spark MLlib

➢ Spark ML Data Pipelines

Step #	Name	Description
ML1	Data Ingestion	Loading the data from different data sources.
ML2	Data Cleaning	Data is pre-processed to get it ready for the machine learning data analysis.
ML3	Feature Extraction	Also known as Feature Engineering, this step is about extracting the features from the data sets.
ML4	Model Training	The machine learning model is trained in the next step using the training data sets.
ML5	Model Validation	Next, the machine learning model is evaluated based on different prediction parameters, for its effectiveness. We also tune the model during the validation step. This step is used to pick the best model.
ML6	Model Testing	The next step is to test the mode before it is deployed.
ML7	Model deployment	Final step is to deploy the selected model to execute in production environment.

Spark Libraries - Spark MLlib



Spark Libraries - Spark GraphX

- GraphX is a library for manipulating graphs (e.g., a social network's friend graph) and performing graph-parallel computations

- **Graph Data**
  - Graph Databases
  - Graph Data Analytics
  - Graph Data Visualization

Spark Libraries - Spark GraphX

➢ Graph Databases

- Unlike traditional data models, data entities as well as the relationships between those entities are the core elements in graph data models. When working on graph data, we are interested in the entities and the connections between the entities.
- The advantage of graph databases is to uncover patterns that are usually difficult to detect using traditional data models and analytics approaches.

Spark Libraries - Spark GraphX

- Graph Data Analytics

-Graph data modeling effort includes defining the nodes (also known as vertices), relationships (also known as edges), and labels to those nodes and relationships.

- Graph data processing mainly includes graph traversal to find specific nodes in the graph data set that match the specified patterns and then locate the associated nodes and relationships in the data so we can see the patterns of connections between different entities.

Spark Libraries - Spark GraphX

- Graph Data Visualization

Once we start storing connected data in a graph database and run analytics on the graph data, we need tools to visualize the patterns behind the relationships between the data entities.

Tools:

D3.js, Linkurious and GraphLab Canvas.

Similar Movie

- Map input rating to (userID, (movieID,rating))
- Find every movie pair rated by the same user
  - + This can be done with a “self-join” operation
  - + At this point we have (userID, ((movieID1, rating1), (movieID2, rating2)))
- Filter out duplicate pairs
- Make he movie pairs the key
  + map to ((movieID1, movieID2), (rating1,rating2))
- groupByKey() to get every rating pair found for each movie pair
- Compute similarity between ratings for each movie in the pair
- Sort, save and display results

Movie Recommendation

- MLlib build it all

```
data = sc.textFile('u.data')

ratings = data.map(lambda l: l.split()).map(lambda l: Rating(int(l[0]), int(l[1]), float(l[2]))).cache()


rank = 10

numIterations = 20

model = ALS.train(ratings, rank, numIterations)

recommendations = model.recommendProducts(userID,10)
```

THANK YOU