Programming for Big Data

Lecture 2

Data Processing with Spark

Dr. Bojan Božić
Dublin institute of Technology

Agenda

Preliminaries
1. Spark Architecture
2. Programming RDDs
   - Core Concepts
   - RDD Operations
   - Persistence
3. Spark Streaming
   - Micro-Batching
   - Example
   - Challenges
5. Summary

PRELIMINARIES

My Details

Name:	Dr. Bojan Božić
Email:	bojan.bozic@dit.ie

Details on SPARK labs:

https://ceadar.dit.ie/bojan.bozic/SPARK/

Final assignment will be published on webcourses.

Any Questions, Please email me!

Books and Resources

Mastering Spark

www.packtpub.com/big-data-and-business-intelligence/mastering-apache-spark

Apache Spark From Inception to Production

http://info.mapr.com/rs/mapr/images/Getting_Started_With_Apache_Spark.pdf

Spark Programming Guide

http://spark.apache.org/docs/latest/programming-guide.html

Learning Spark - Lightning-Fast Big Data Analysis

http://shop.oreilly.com/product/0636920028512.do

Course Details

Classes: Wednesday 18:30 - 21:30

- 4 labs with 3 lab assignments
- Final assignment starts in last (4th) lab

Grading: Only final assignment is graded (100%)

Software

http://spark.apache.org/

SPARK ARCHITECTURE

What is Spark?

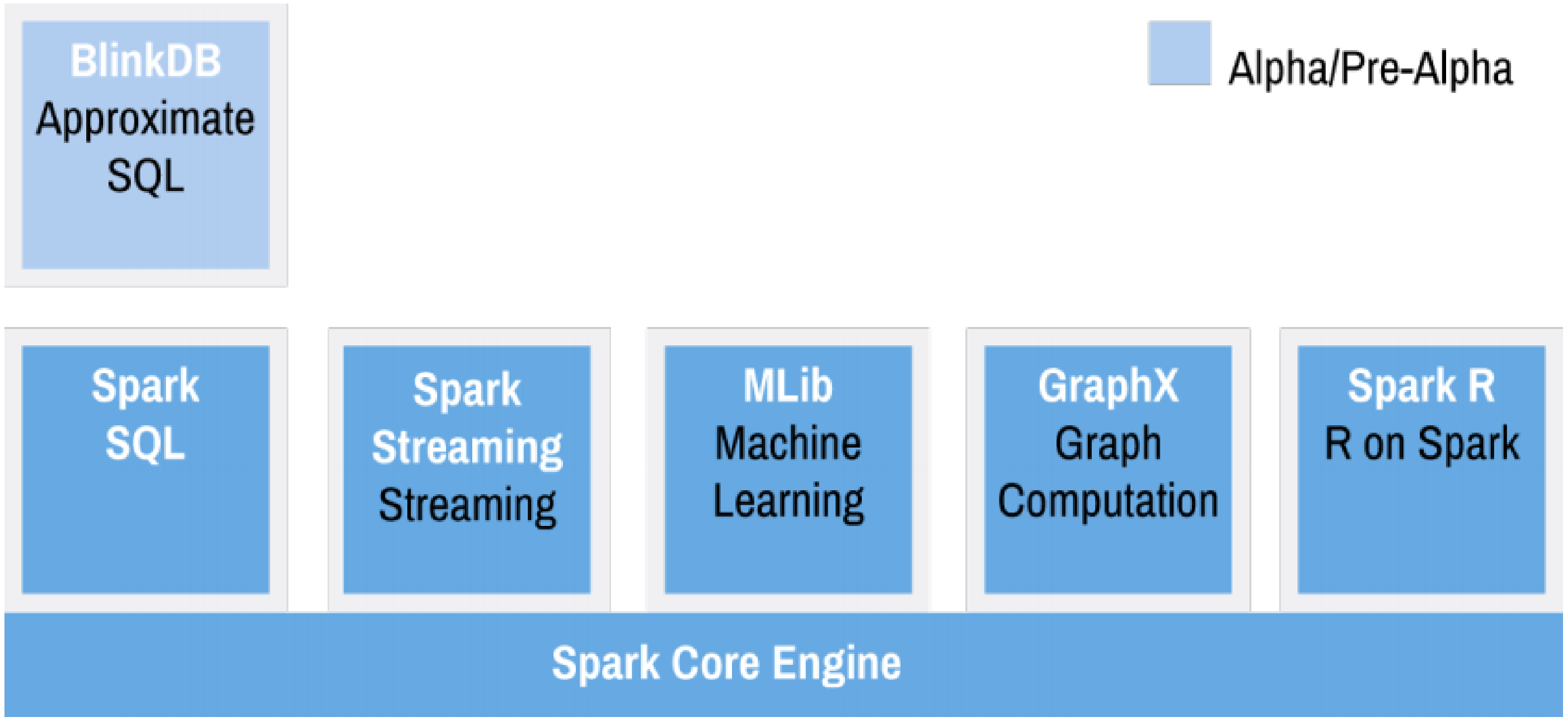
Spark, an incubated project of the Apache Software Foundation, is a general-purpose data processing engine, suitable for use in a wide range of circumstances including, **interactive queries across large data sets**, **processing of streaming data, large scale machine learning** and **large-scale ETL**.

What is Spark?

Spark has several core benefits:

1) **Simplicity** - its a single environment that can address a lot of tasks
2) **Speed** - Spark is an **in-memory Big Data solution** and is thus blistering fast compare to standard on-disk Hadoop
3) **Support** - Spark has APIs for Java, Python, R, Scala and is currently supported by a range of different commercial applications

Spark Architecture



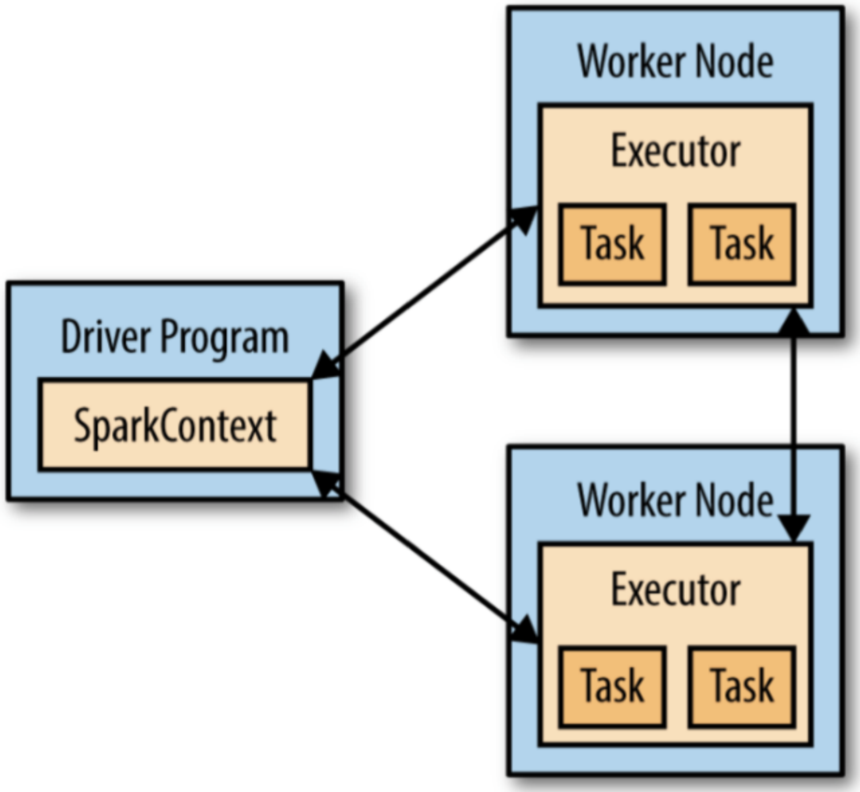
PROGRAMMING RDDS

Programmings RDDs

Core concepts:

1) Every Spark application consists of a ***driver*** *program* that launches various parallel operations on a cluster.
2) Driver programs access Spark through a SparkContext object, which represents a connection to a computing cluster.
3) The driver program manages a number of nodes, called executers, which perform operations on RDDs.

Programmings RDDs



Spark context with two worker nodes - executers

Programmings RDDs

Creating an RDD:

1) An RDD in Spark is simply an **immutable** distributed collection of objects.
2) Each RDD is split into multiple *partitions*, which may be computed on different nodes of the cluster.
3) An RDD is created by **loading a dataset** or **distributing a collection of objects** (list or set) in the driver program.

Programmings RDDs

1) **Load a Dataset:**

```
val lines = sc.textFile("/path/to/README.md")
```

2) **Distribute a collection:**

```
val lines = sc.parallelize(List("pandas", "i like pandas"))
```

Programmings RDDs

RDD operations are divided between:

1) **Transformations** result in a new RDD
2) **Actions** return a result (computation) to the driver

Programmings RDDs

Transformations (Arrays):

Filter(func) (return array of elements matching filter):

```
val inputRDD = sc.textFile("log.txt")
val errorsRDD = inputRDD.filter(line => line.contains("error"))
val warningsRDD = inputRDD.filter(line => line.contains("warning"))
```

Map(func) (Pass each element through func):

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
```

Flatmap (return an array):

```
val lines = sc.parallelize(List("hello world", "hi"))
val words = lines.flatMap(line => line.split(" "))
```

Sample (take a sample):

```
val common = lines1.sample(false, 0.5)
```

Programmings RDDs

Union(RDD) (join two RDDs):

__val__ badLinesRDD = errorsRDD.union(warningsRDD)

Intersection(RDD) (return the common values of two RDDs):

__val__ lines1 = sc.parallelize(__List__("hello", "world", "tennis", "Japan"))
__val__ lines2 = sc.parallelize(__List__("hello", "Japan"))
__val__ common = lines1.intersection(lines2)
common.collect().foreach(println)

Distinct (return distinct values from and RDD):

__val__ lines = sc.parallelize(__List__("hello", "hello","tennis", "Japan"))
lines.collect().foreach(println)

Subtract (take one RDD from another and return):

__val__ common = lines1.subtract(lines2)

Programmings RDDs

Transformations (Key, Value Pairs):

ReduceByKey (combines values with the same key):

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
val result = sc.parallelize(data).reduceByKey((x, y) => x + y))
result.collect().foreach(println)
```

GroupByKey (returns a dataset of (K, Iterable<V>) pairs):

```
result.groupByKey.collect.foreach(println))
```

MapValues (apply to each value):

```
val result = sc.parallelize(Seq(("a", 3), ("b", 4), ("a", 1)))
result.mapValues(x => x+1).foreach(println)
```

keys (returns the keys from a key.value pair):

```
val result = sc.parallelize(Seq(("a", 3), ("b", 4), ("a", 1)))
result.keys.foreach(println)
```

Programmings RDDs

Transformations contd... (Key, Value Pairs):

Values (return an RDD of just the values):

```
val data = Seq(("a", 3), ("b", 4), ("a", 1))
data.values.collect().foreach(println)
```

SortByKey (returns a sorted by key RDD):

```
data.sortByKey().collect().foreach(println)
```

flatMapValues (returns an RDD of just the values):

```
data.sortByKey().collect().foreach(println)
```

subtractByKey (return an RDD of just the values):

```
data.sortByKey().collect().foreach(println)
```

Programmings RDDs

Actions:

Reduce (reduces two elements to one):

```
val input = sc.parallelize(List(1, 2, 3, 4))
val result = input.map(x => x * x)
val sumInput = input.reduce((x,y) => x + y)
val sumResult = result.reduce((x,y) => x + y)
```

Collect (returns all the elements of a dataset as an array):

```
val lines = sc.parallelize(List(“hello”, “hello”,“tennis”, “Japan”))
lines.collect().foreach(println)
```

Count (take the first element):

```
val common = lines.first
```

Take (return an array of the first n elements):

```
val common = lines.take(10)
```

Programmings RDDs

foreach (apply a function to every element in an array):

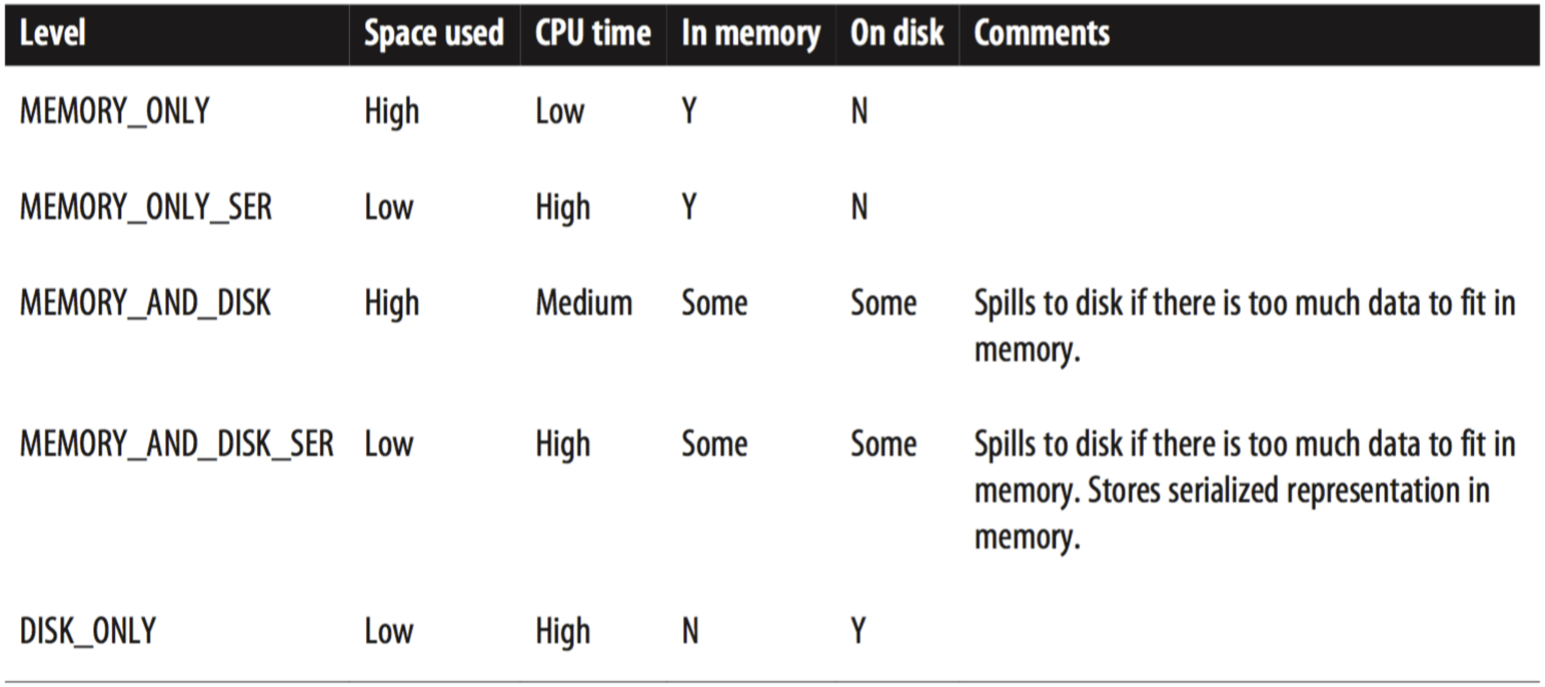
```
val input = sc.parallelize(List(1, 2, 3, 4))
input.collect().foreach(println)
```

takeOrdered (takes an ordered set of elements from an array):

```
val input = sc.parallelize(List(1, 2, 3, 4))
input.collect().foreach(println)
```

Programmings RDDs

Because of lazy evaluation, persistence is very important in Spark otherwise every single operation will be computed each time a calculation is required.



Level	Space used	CPU time	In memory	On disk	Comments
MEMORY_ONLY	High	Low	Y	N	
MEMORY_ONLY_SER	Low	High	Y	N	
MEMORY_AND_DISK	High	Medium	Some	Some	Spills to disk if there is too much data to fit in memory.
MEMORY_AND_DISK_SER	Low	High	Some	Some	Spills to disk if there is too much data to fit in memory. Stores serialized representation in memory.
DISK_ONLY	Low	High	N	Y	

Programmings RDDs

Persistence:

Persist (saves the output to memory):

```
import org.apache.spark.storage.StorageLevel
val lines = sc.parallelize(List(1,2,3,4))
lines.persist(StorageLevel.MEMORY_ONLY)
lines.collect().foreach(println)
```

Cache (persist method with "MEMORY_ONLY" storage level):

```
val lines = sc.parallelize(List("hello", "hello","tennis", "Japan"))
lines.cache
```

Unpersist (removes persistence):

```
lines.unpersist
```

Programmings RDDs

Shuffling:

1) On some operations, Spark needs to reorganise data across the cluster - for joins for example
2) Shuffling data is very expensive as data must be moved around the cluster - lots of serialisation, network latency and IOPS.
3) Data might not now reside completely in memory - resulting in a degradation of performance.

Loading and Saving Data

As with most modern open source programming languages or frameworks, Spark can work with a multitude of data formats from TSV, CSV, JSON, SQL, Sequence Files (parallelised Hadoop files for fast ingestion), Object Files (Java Serialised Objects) from local file, from HDFS or S3.

Read in File:

```
val rdd = sc.textFile("file:///home/holden/repos/spark/README.md") )
```

Write out File:

```
rdd.saveAsTextFile("foo")
```

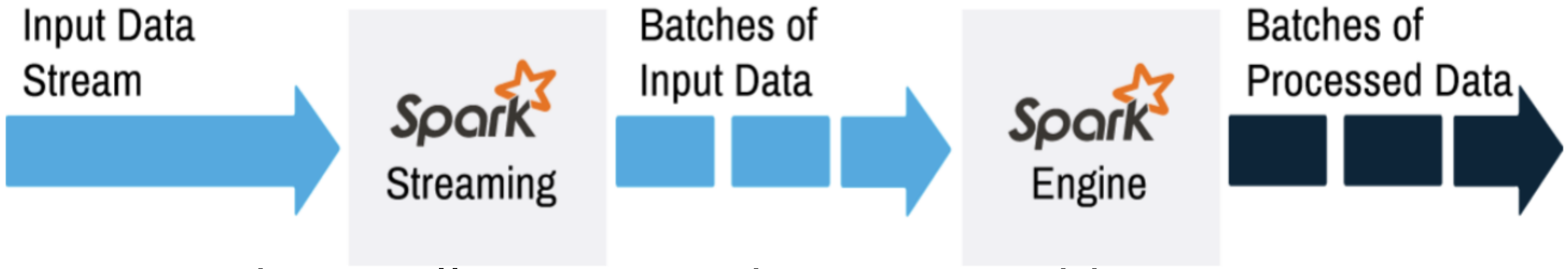
SPARK STREAMING

Spark Streaming

1) Can ingest data from a wide range of data sources, including Apache Kafka, Apache Flume, Amazon Kinesis, Twitter, or sensors and other devices connected via TCP sockets
2) Data is processed in streams using a range of algorithms and high-level data processing functions like *map*, *reduce*, *join* and *window*

Spark Streaming

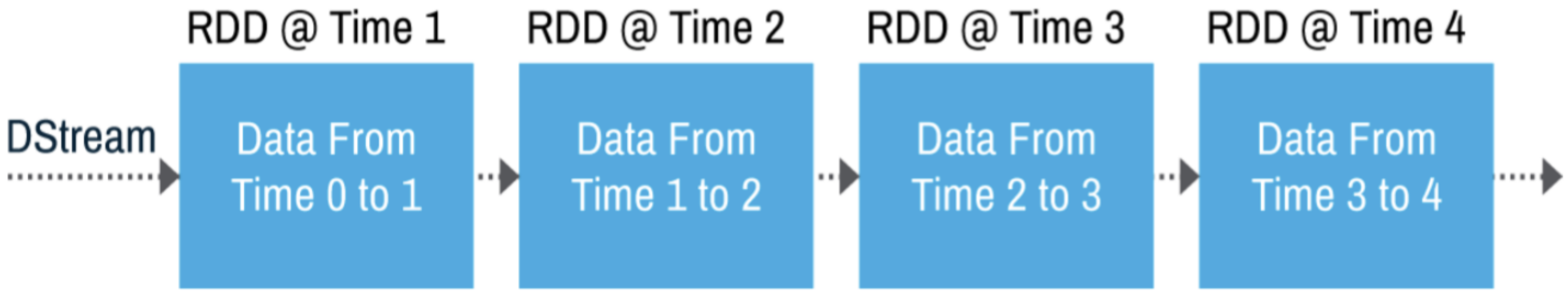
1. Logically, Spark Streaming represents a continuous stream of data as a discretised stream or DStream.



3. Spark actually stores and processes this DStream as a sequence of RDDs.
4. Each RDD is actually a snapshot of data ingested during a time period allowing Spark to apply existing batch processes.

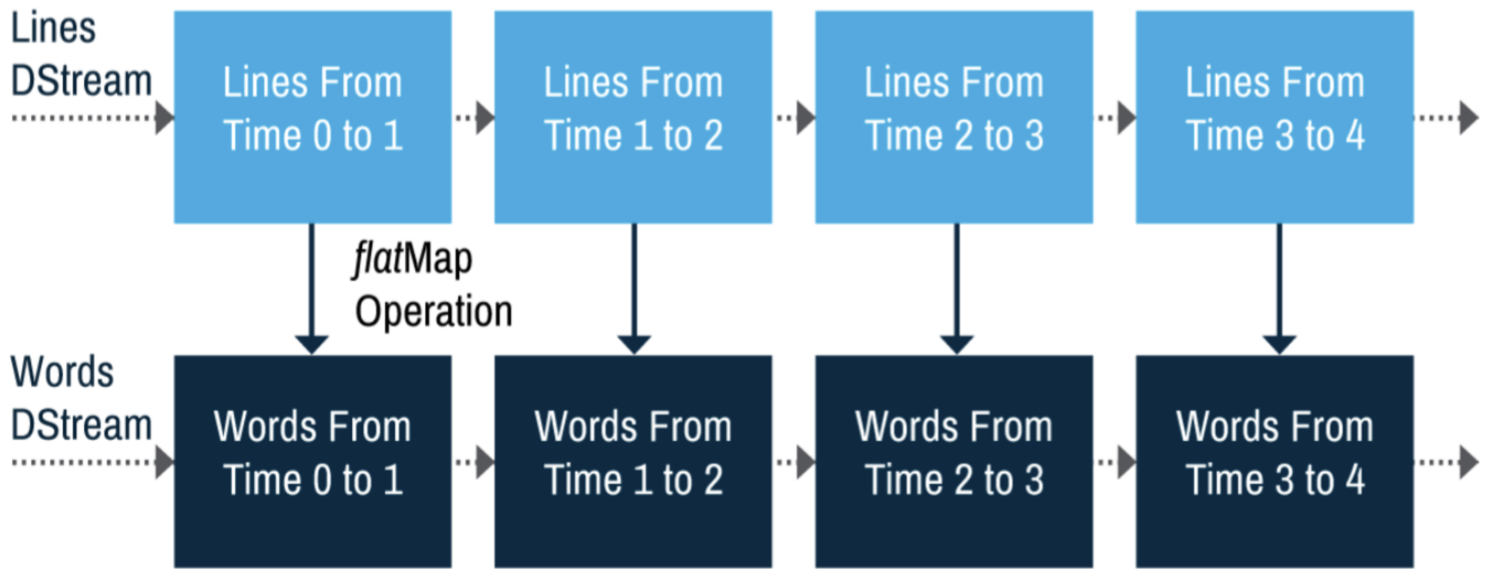
Spark Streaming

Any Data Processing code - such as algorithms or pipelines are applied in exactly the same manner as if applied on large batch data.



Spark Streaming

For example, a flat map can be used to extract individual words from lines of text in an input source. This same function behaves the same on RDDs in Stream and in Batch.



Spark Streaming

1) Spark streaming performs functions on micro-batches therefore Spark streaming is not “a real” streaming framework.
2) Setting the batch interval sets the interval time between processing 500ms - 5000ms.
3) The closer to real-time the more overhead endured by the system.
4) Other platforms such as Apache Storm and FLink work on **real-time streams**

Spark Streaming

Challenges:

1) Back-pressure - when the volume of events coming across a stream is more than the stream processing engine can handle - is a challenge for all streaming platforms including Spark
2) Out-of-Order data - can be difficult to ascertain as the process works with micro-batches.

SUMMARY

Summary

Preliminaries
1. Spark Architecture
2. Programming RDDs
 - Core Concepts
 - RDD Operations
 - Persistence
3. Spark Streaming
   - Micro-Batching
   - Example
   - Challenges
5. Summary

Questions

?