

## **Lecture - 20**

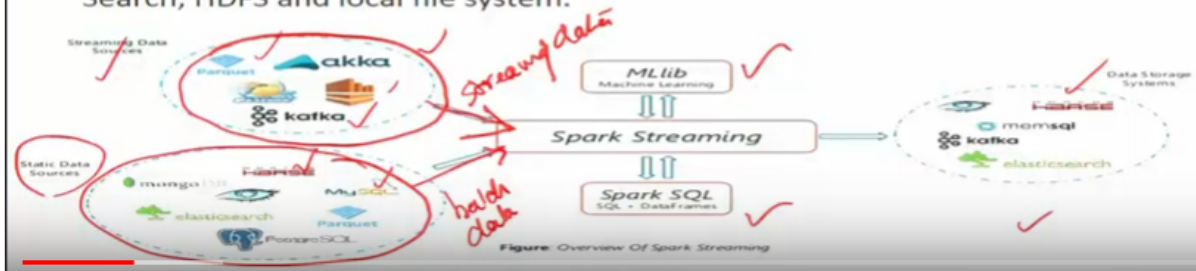
### **Spark Streaming and Sliding Window Analytics (Part-II)**

Spark Streaming Workflow.

Refer slide time :( 0:15)

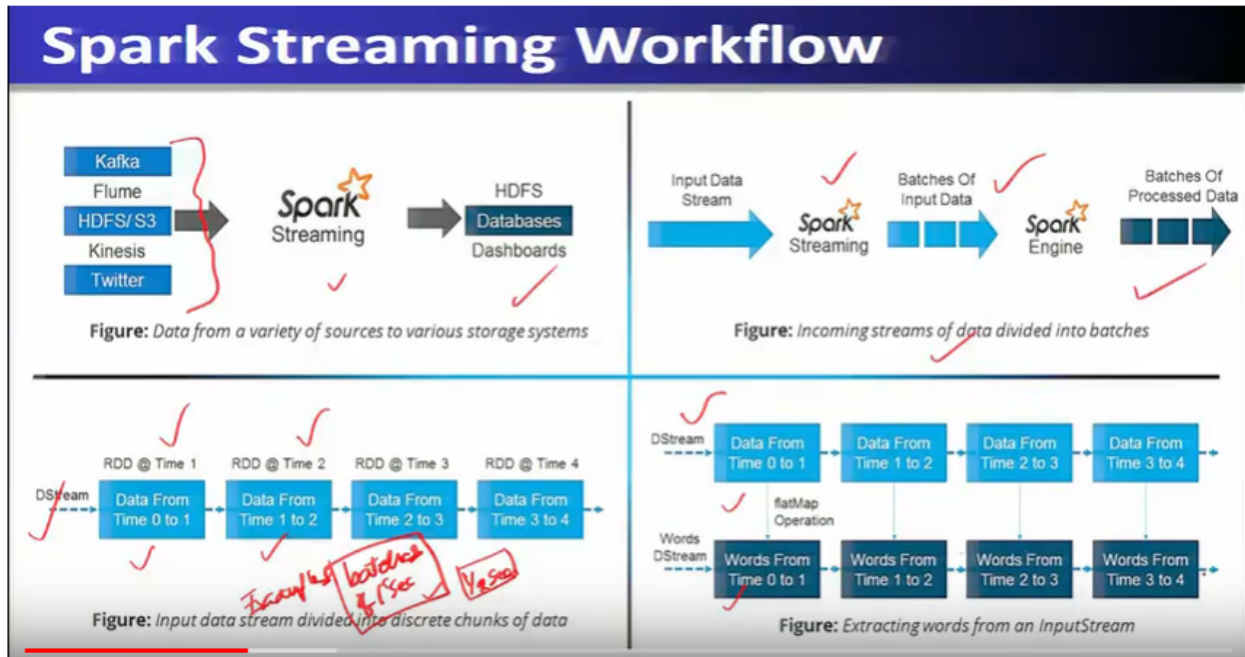
# Spark Streaming Workflow

- Spark Streaming workflow has four high-level stages. The first is to stream data from various sources. These sources can be streaming data sources like Akka, Kafka, Flume, AWS or Parquet for real-time streaming. The second type of sources includes HBase, MySQL, PostgreSQL, Elastic Search, Mongo DB and Cassandra for static/batch streaming.
- Once this happens, Spark can be used to perform Machine Learning on the data through its MLlib API. Further, Spark SQL is used to perform further operations on this data. Finally, the streaming output can be stored into various data storage systems like HBase, Cassandra, MemSQL, Kafka, Elastic Search, HDFS and local file system.



So, it is partly streaming but, flow has four high-level stages and let us see all these stages in, in a brief manner. So, the first is to stream, the data from various sources and these sources, are such as akka system, Kafka system, flume AWS or Park West for the real-time streaming, data input. So, this particular streaming data, is to be input using these different sources, akka Kafka flume and Park West, they will give the live feed, of streaming data, to the to the system. Now second type of sources, of the data include, they are called the static data sources. So, they include, HBase that we have discussed, my sequel post grass SQL, then Mongo DB Cassandra and they are for the static are the batch streaming data, feed into the SPAR streaming system. Once these once these data, is streamed into the spark system, the spark and we used to perform on, on it the machine learning using, the machine learning, API that is ma lib, further the spark SQL, can also be used to perform further, operations on this particular data, finally the streaming output can be stored, into the into the various data storage system, like HBase and Cassandra, SQL Kafka elastic search HDFS local system and so, on.

Refer slide time :( 02:40)

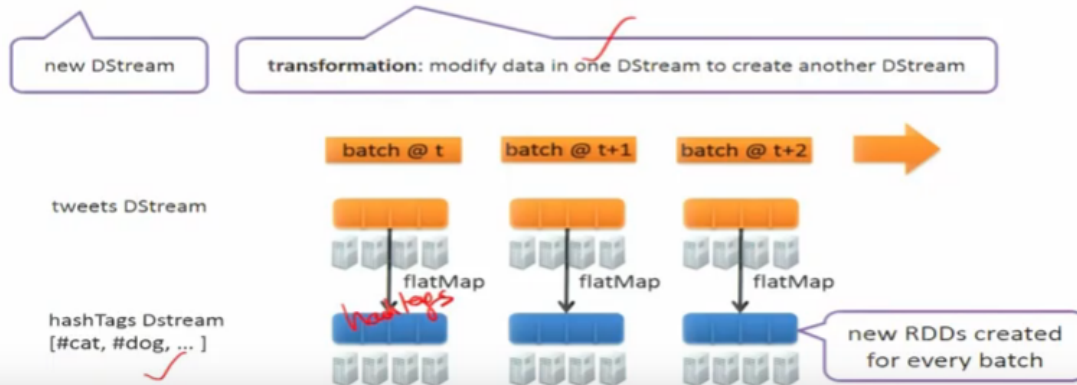


So, therefore the, the spark is streaming workflow, looks like this that input, of the streaming data, comes from Kafka flume HDFS Genesis Twitter. And finally after processing they will be stored, on to the either HDFS databases dashboard and so on. So, incoming these, incoming data, streams are now divided into the, into the batches of input data, which is given back to the spark engine, for computation and they will process, it and gives back output this particular data. Now this batch the input data streams, is now divided into the, discrete chunks of data, for example the streams, which is handled by the spark streaming is called the, 'Discretized Stream', or add stream. So, d stream is, is the batches, of data of X seconds. So, here the batch, is of let us say one second, so from zero to one second, batch is called as, 'RDD', at the rate type one. So, the batch of timing from time one, to time two that is that next one second is called the, 'RDD,' at the rate time 2 and so on. So, the batches, are do I into the into the discrete chunks, in this example this is the batches, of one second it is divided into the batches of one second in this example, it can be minimum half a second batches, for better latency from n to n. So, once this particular, data stream or d stream is decided or is broken by, the Spark streaming system, then various transformation can be applied, which is also a part of the Sparky streaming system, for example a flat map operation, can be applied on every d stream. So, for example when a flat map operation is applied, it will give different words, which are input or which are divided, in the form of d stream of one second duration, similarly this flat map is when applied on all the case then it will extract the words from the input stream.

Refer slide time :( 05:19)

## Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
```

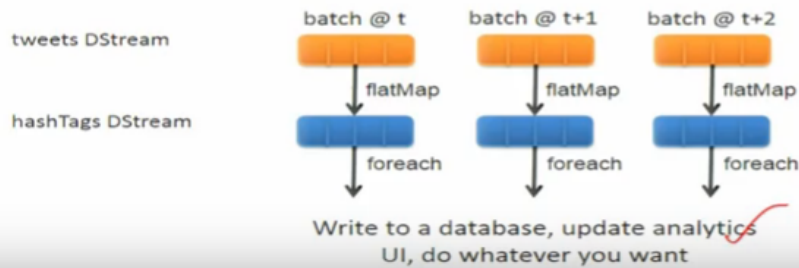


And this can be shown, here in this example, of getting the hashtags, from on the Twitter. So, Twitter stream is input into the, into the system into the spark streaming system. So, this is called. So, Twitter stream is given into the system and then after dividing, into the different D streams, it will perform the transformation, on top of it which is called the, 'Flat Map'. And the flat map will be defined on, the tweets, which is given into the system and the flat map, will perform the get, tag the hashtag it will get, as the status and this hashtag will be, extracted as per the transformation. So, the transformation will modify, one D stream to create, another d stream in this particular manner and finally it will, from this tweet d stream it will extract the hashtags, in this particular example. So, this is shown over here that, this tweet D streams when a flat map is applied, it will give the hashtag D streams which is hashtag in, in the terms of for example number of hash cat hash dog and so on different topics, it will extract and it will generate the new RDD

Refer slide time :( 07:03)

## Example 1 – Get hashtags from Twitter

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.foreach(hashTagRDD => { ... })
```



out of every batch, therefore after that these hashtags, will be saved, into the Hadoop as a file and this will be the output, which is to be. So, output operation is to push this particular transform data, in the form of a hashtag to the external storage. And here that is shown over here, but not every time we are going to store, we can perform various analytics on this transform hashtag. And maybe that sometimes this transformed hashtag, analytics will require to update the website or perform various other applications, depending upon whatever we want to do on this output so, for each. So, therefore various,

Refer slide time :( 08:01)

## Java Example

### ~~Scala~~

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

### Java

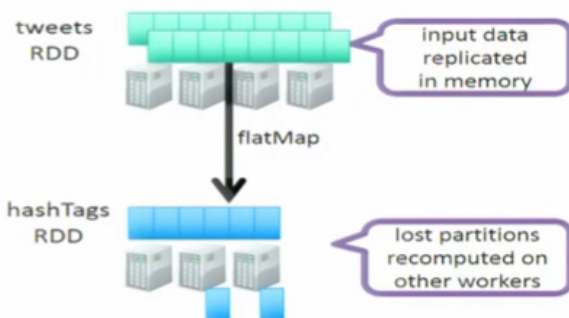
```
JavaDStream<Status> tweets = ssc.twitterStream()
JavaDStream<String> hashTags = tweets.flatMap(new Function<...> { })
hashTags.saveAsHadoopFiles("hdfs://...")
```

different programming languages are supported. So, the example which we have seen was written in scalar, the same application it can be written using the Java. So, Java API also is available, with the Sparky streaming system.

Refer slide time :( 08:21)

## Fault-tolerance

- RDDs remember the sequence of operations that created it from the original fault-tolerant input data
- Batches of input data are replicated in memory of multiple worker nodes, therefore fault-tolerant
- Data lost due to worker failure, can be recomputed from input data



And now let us see about the fault tolerance. So, RDDs remember the sequence of operation and that created it from the original fault tolerant data. So, therefore RDDs are knowing, using lineage about the sequence of operation, how they are created, from the original. So, this we know from the spark, fault tolerance system and when the batches of input data are replicated in the memory of multiple worker, nodes and therefore we are trying to achieve the fault tolerance in this case. So, whenever the data is lost to the worker failure it can be recomputed, using lineage from the input why because these RDDs remember all that things. So, so therefore this fault tolerance can also be ensured here, in the part of this spark system, in a spark streaming.

Refer slide time :( 09:17)

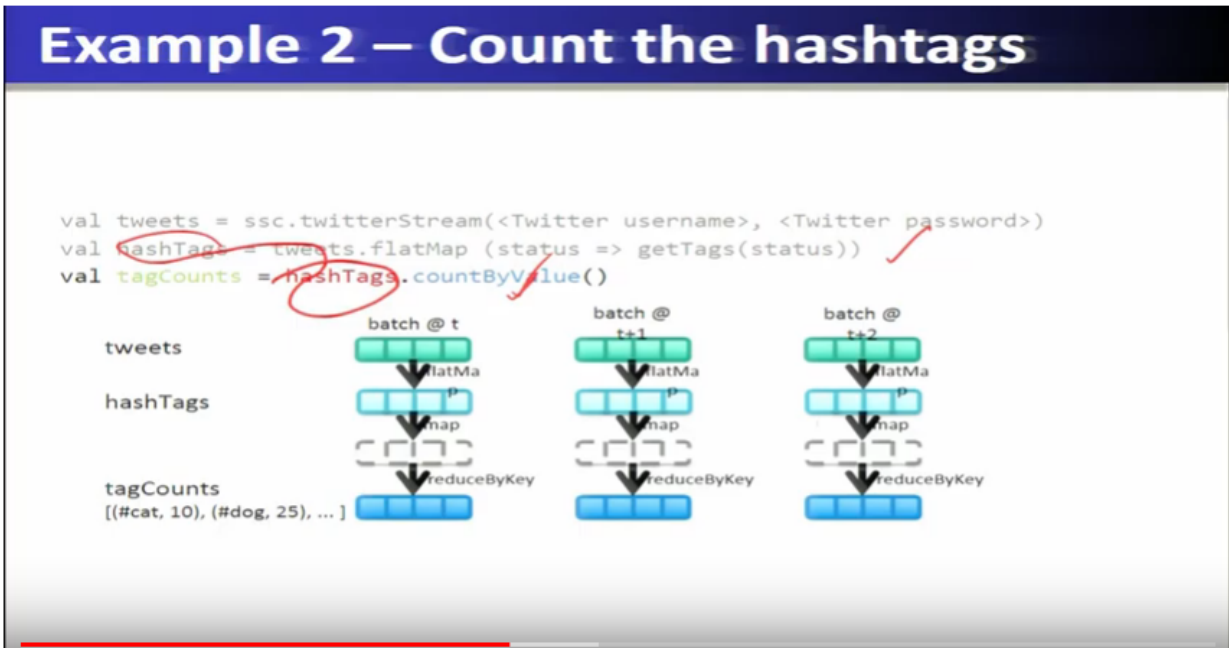
## Key concepts

- **DStream** – sequence of RDDs representing a stream of data
  - Twitter, HDFS, Kafka, Flume, ZeroMQ, Akka Actor, TCP sockets
- **Transformations** – modify data from on DStream to another
  - Standard RDD operations – map, countByValue, reduce, join, ... ✓
  - Stateful operations – window, countByValueAndWindow, ... ✓
- **Output Operations** – send data to external entity
  - saveAsHadoopFiles – saves to HDFS ✓
  - foreach – do anything with each batch of results ✓

So, let us see the key concepts. So, key concepts so, far we have seen about the D stream which is a sequence of RDDs representing the stream of data and these D streams are created out of the stream of data, from Twitter HDFS Kafka flume and so on. And there are various transformations, can be applied on D stream, which can modify, one from, from, from the given D stream to another form of these team. So, the standard RDD transformations, which are operations which are available for the transformations, are the map, count by value reduce, join and so, on similarly there are other stateful operations, which are available in the form of transformations such as window operations. And count by value and window and so, on we will see all these stateful operations. Now besides the transformation, there are actions or the output operations, also to be performed on the D streams, which are available as part of the sparkie streaming system. Now these output operation, will send the data to the external entities, will save as Hadoop files will say to HDFS, for each do anything with each batch of results. So, we will see that whenever an action or an output operation, which we are going to perform either, it will save or to the HDFS file save as, a file or It will further actions, using for each command.



Refer slide time :( 10:53)



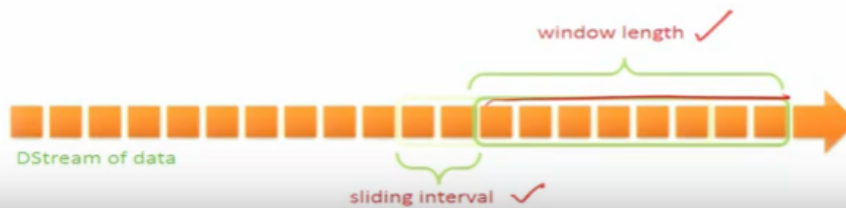
So, again we will now count the hashtags, in this particular example. So, that means once we get, the hashtags that we are going to count these hashtags. So, the se hashtags which is now available, using this spark streaming system. Now we are going to perform an action or the operation as an output. So, the output here is to be the count by value. So, it will count how many hashtags about these hashtags, are there into the stream processing into the data. So, here we can see, that we perform this count, by value.

Refer slide time :( 11:42)



### Example 3 – Count the hashtags over last 10 mins

```
val tweets = ssc.twitterStream()
val hashTags = tweets.flatMap(status => getTags(status))
val tagCounts = hashTags.window(Minutes(1), Seconds(5)).countByValue()
```

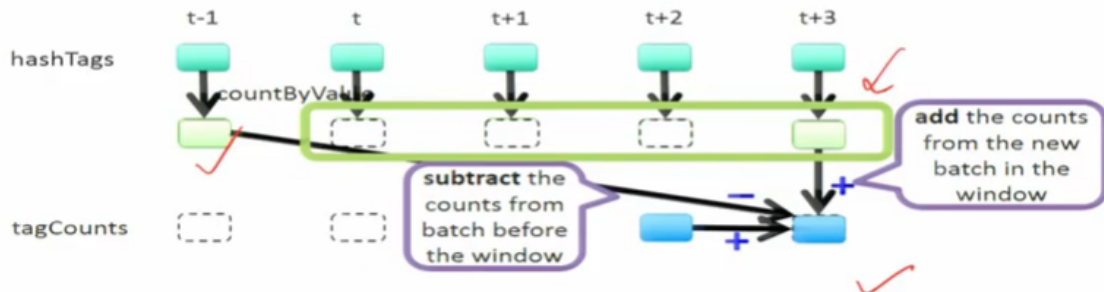


Now we will also see some more functionality, in the terms of this example, which will count the hashtag over the last 10 minutes. So, is so, basically for that we have to use, the been doing operation. So, we have to use the window of one minute and within the one minute after every five seconds, we are now monitoring, this hashtag and then performing the count by value. So, this particular command this operation, will have the windows window has two different arguments, one is the length of the window, window size and window length. And so, here it is in the window of one minute duration for every five second, we are going to count, this by value. So, just window is can be seen here, the window has sliding window operation, has the window length and the window interval, sliding interval. So, we can see, using this particular diagram, that this is the window length, this is of one minute and sliding interval, is of that another duration. So, sliding interval, will give the data for processing so, whenever we say count by value. So, this particular window length and sliding interval, together will give the data for this computation.

Refer slide time :( 13:35)

# Smart window-based countByValue

```
val tagCounts = hashtags.countByValueAndWindow(Minutes(10), Seconds(1))
```



So, here, we can see that in this particular example that that we are going to count, all the data in that particular window. Now one important thing is that, when the window, slides can see the sliding interval, there are two things one is called, 'Window Length'. And the sliding interval. So, when the window slides, then the, this particular old data will be out of that window and a new data will enter into the system. So, the values which we are now counting every time is going to be changed in this way. So, what will be the new count, word count, count by value that is required to subtract, this is the previous value and the new value is to be added this concept, requires, the window based different algorithm, to do the analysis.

Refer slide time :( 14:48)

## Smart window-based *reduce*

- Technique to incrementally compute count generalizes to many reduce operations
  - Need a function to “inverse reduce” (“subtract” for counting)
- Could have implemented counting as:  
`hashTags.reduceByKeyAndWindow(_ + _, _ - _, Minutes(1), ...)`

That we will see in the further slides. So, smart window based, reduction we will see different commands are there so, techniques to incrementally compute count, generally generalizes to the many reduce operations, that needs a function to inverse the reduce that is subtract for counting. I could have implemented counting as, the hashtag reduced by key and window. So, within that particular time that will, do this operation that is after sliding the new value will come and the old value will be subtracted. And this is performed in the reduced by key and window operation.

Refer slide time :( 15:31)

# Arbitrary Stateful Computations

Specify function to generate new state based on previous state and new data

- Example: Maintain per-user mood as state, and update it with their tweets

```
def updateMood(newTweets, lastMood) => newMood  
moods = tweetsByUser.updateStateByKey(updateMood _)
```

Arbitrary stateful computation, this is also very important computation, in the spark streaming system, why because this allows you to, maintain a count of particular words or event which is occurring, into the stream of data. So, this is to maintain the stateful computation. So, especially by function to generate the new state, based on the previous state and a new data. So, for example maintain per user mood as the state and update it when, when it sees that tweet. So, update mode definition, of a function can be defined as, the new tweets and the last mood, which it has seen and then it will update to the new mood and so, this is an update function so, whenever. So, this particular function, will do this update mood, using the parameters which is given in the in the tweet so, so whenever a new tweet it comes, it will perform this update move function. And this will be the update state by the key and so, whenever the Twitter tweets by the user is given. So, here the mood will be extracted and, and the state variable, is maintained to measure the to understand the mood. So, here that is that is why it is called a, 'Stateful Computation'. So, state is man is maintained all the time so and this state will be updated whenever not did the stream of data is, coming and now there is a change of mood. So, it will be extracted out of the stream and updated as the state.

Refer slide time :( 17:34)

## Arbitrary Combinations of Batch and Streaming Computations

### Inter-mix RDD and DStream operations!

- Example: Join incoming tweets with a spam HDFS file to filter out bad tweets

```
tweets.transform(tweetsRDD => {  
    tweetsRDD.join(spamHDFSFile).filter(...)  
})
```

So, arbitrary combination of batch and stream, computation example we are going to see now. So, there will be an intermix RDDs and the d stream computation, operation. So, for example join incoming stream with the with a spam, HDFS file to filter out the bad tweets. So, using this particular way of joining, we can see that so, so the tweet RDDs is now, joined with the HDFS file system and we will perform various, filter on top of it and this will transform, the tweets and the transform to X will be given as the output. So, all these functions are written in the scalar, to understand these particular operations or the commands or the programs. So, I advise you to refer the scalar programming language, to have a better understanding of the streaming.

Refer slide time :( 18:36)

## Spark Streaming-Dstreams, Batches and RDDs



- These steps repeat for each batch.. Continuously
- Because we are dealing with Streaming data. Spark Streaming has the ability to “remember” the previous RDDs...to some extent.

So, spark streaming these 3dstreams batches and RDDs. So, again let us summarize all this is that whenever the data, input data stream is coming, input streaming data is coming. So, in the spark streaming system, it will divide into the batches, of one second duration in this example, and these batches are now performed various transformations. And an action and given back to the spark engine for giving the output. So, these steps are repeated for each batch continuously, because we are dealing with the streaming data. So, the data is continuously coming, in the form of streams. So, is partly streaming has the ability to remember the previous RDDs and to some extent.

Refer slide time :( 19:26)

## DStreams + RDDs = Power

- Online machine learning
  - Continuously learn and update data models (*updateStateByKey* and *transform*)
- Combine live data streams with historical data
  - Generate historical data models with Spark, etc.
  - Use data models to process live data stream (*transform*)
- CEP-style processing
  - window-based operations (*reduceByWindow*, etc.)

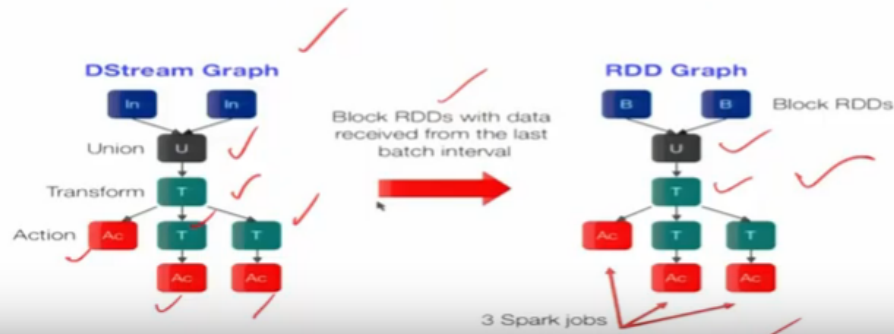
Therefore this d stream and RDDs together that is d stream, is basically the, the streaming data, and RDDs are the batch data, together if we add it will become the more power for example we can apply or we can introduce the online machine learning. So, that means we can apply the, the RDDs, when it becomes an RDD we can apply the machine learning that the library is on top of it hence, it becomes a online machine learning, technique that means machine learning applied, on the d streaming using this model. So, continuously learn and update the data model this can be performed, using update state by the key and transformations. So, also there is a in this manner, we can combine the live stream data, with the historical data, we generally we can generate, the historical data model with the spark etcetera. Now we can use the data model to process, live data streams using transformations, we can also do the CP style processing, such as window based operations reduced by window etc.

Refer slide time :( 20:26)



# From DStreams to Spark Jobs

- Every interval, an RDD graph is computed from the DStream graph
- For each output operation, a Spark action is created
- For each action, a Spark job is created to compute it



So, from D stream to the spark jobs, we can see that every interval, an RDD graph is computed from D stream graph. And for each output operation spark action, is created for each action a spark job is computed. So, this is shown here in this particular example, graph that it is a D stream graph that input streams are coming and all these input different streams, we can perform a union on it. And we can perform different transformations and then perform various actions. So, these block of RDDs are then with the data received from the last batch interval is given back to the spark system, in the form of RDD graph. And RDD again will perform these RDDs and perform the, union and again apply the transformation and give the output. So, this particular from a spark is streaming, weather and when the when the jobs are given for this part then again another level of transformations, can be applied before it can output.

Refer slide time :( 21:34)

## Input Sources

- Out of the box, we provide
  - Kafka, HDFS, Flume, Akka Actors, Raw TCP sockets, etc.
- Very easy to write a receiver for your own data source
- Also, generate your own RDDs from Spark, etc. and push them in as a “stream”

So, there are input streams which we have already seen let us summarize that that out of the box ,we have provided for the input sources, Kafka HDFS flume ARCA actors raw TCP sockets and it is very easy to write a receiver, for your own data source. So, these are different receivers, which are inbuilt and you can also write down, your own receiver for your data source also generate your own RDDs from the spark and push them as the stream.

Refer slide time :( 22:09)

## Current Spark Streaming I/O

- Input Sources
  - Kafka, Flume, Twitter, ZeroMQ, MQTT, TCP sockets
  - Basic sources: sockets, files, Akka actors
  - Other sources require receiver threads
- Output operations
  - Print(), saveAsTextFiles(), saveAsObjectFiles(), saveAsHadoopFiles(), foreachRDD()
  - foreachRDD can be used for message queues, DB operations and more



So, current as park streaming input output, we can say now summarizes as cough-cough loon Twitter Onq and so on, the basic sources are sockets file a character and so on. So, the output operations are print save as a file, save as object file save as Hadoop files, for each RDDs for each RDD, can be used as a message queue and DB operations and many more things.

Refer slide time :( 22:38)

## Dstream Classes

- Different classes for different languages (Scala, Java)
- Dstream has 36 value members
- Multiple types of Dstreams
- Separate Python API

org.apache.spark.input	hide focus
└─ PortableDataStream	
org.apache.spark.serializer	hide focus
└─ DeserializationStream	
org.apache.spark.streaming.api.java	hide focus
├─ JavaDStream	
├─ JavaDStreamLike	
├─ JavaInputDStream	
├─ JavaPairDStream	
├─ JavaPairInputDStream	
├─ JavaPairReceiverInputDStream	
├─ JavaReceiverInputDStream	
org.apache.spark.streaming.dstream	hide focus
├─ ConstantInputDStream	
├─ DStream	
├─ InputDStream	
├─ PairDStreamFunctions	
├─ ReceiverInputDStream	

So, d stream classes different classes for different languages are ported Scala and Java these three miles36 different values, value members and multiple type of rest reams and separate Python API will be provided.

Refer slide time :( 22:54)

# Spark Streaming Operations

- All the Spark **RDD** operations
  - Some available through the transform() operation

map/flatMap	filter	repartition	union
count	reduce	countByValue	reduceByKey
join	cogroup	transform	updateStateByKey

- Spark Streaming **window** operations

window	countByWindow	reduceByWindow
reduceByKeyAndWindow	countByValueAndWindow	

- Spark Streaming **output** operations

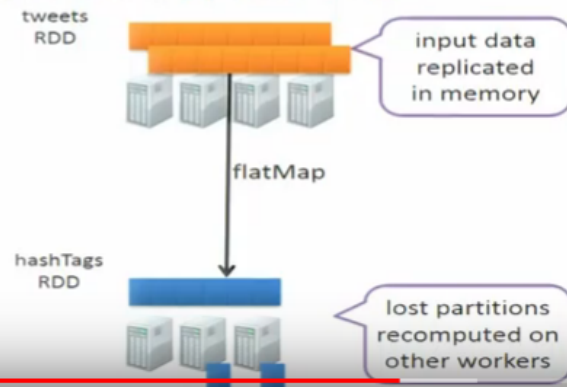
print	saveAsTextFiles	saveAsObjectFiles
saveAsHadoopFiles	foreachRDD	

Now Spark streaming operations are summarized here, as the RDD operations and some of the transformations, such as map and flat map that we have seen filter also we have seen repartition, Union, count reduce, count by value reduced by key join, code group, transform and update state by key are different transformations, available in the spark RDDs. Now it's park swimming window operations are available, such as window count by, window count really reduced by window, reduced by key and window count by value and window, similarly for output operations in spark streaming, various commands such as print save as a text files save as object files, save as Hadoop files for each RDD and so on.

Refer slide time :( 23:46)

# Fault-tolerance

- Batches of input data are replicated in memory for fault-tolerance
- Data lost due to worker failure, can be recomputed from replicated input data



- All transformations are fault-tolerant, and *exactly-once* transformations

So, therefore the batches of input data are replicated, in the memory for fault tolerance data lost, due to the worker can be recomputed, from the replicated input data. And all transformed transformation or fault tolerant and follow the exactly ones transformations.

Refer slide time :( 24:02)

# Fault-tolerance

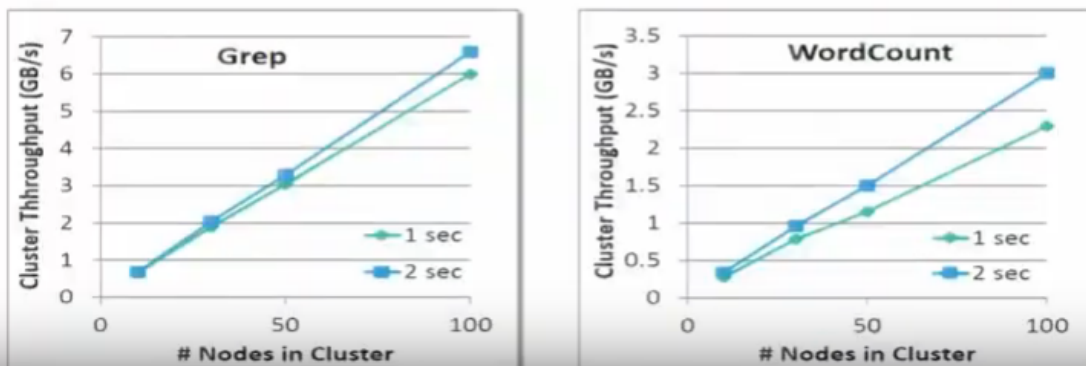
- Received data is **replicated** among multiple Spark executors
  - Default factor: 2
- **Checkpointing**
  - Saves state on regular basis, typically every 5-10 batches of data
  - A failure would have to replay the 5-10 previous batches to recreate the appropriate RDDs
  - Checkpoint done to HDFS or equivalent
- Must protect the **driver program**
  - If the driver node running the Spark Streaming application fails
  - Driver must be restarted on another node.
  - Requires a checkpoint directory in the StreamingContext
- **Streaming Backpressure**
  - `spark.streaming.backpressure.enabled`
  - `spark.streaming.receiver.maxRate`

So, fault tolerance, is that is the receive data, is replicated among, multiple spark executors the default is to and this must protect the driver program there is only one driver running. So, if the driver node is running, on the spark streaming and application fails driver, must be restarted on another node. And this can be handled using the zookeeper, our yarn, this was engine requires a check point directly here, in the streaming context which. So, check pointing saves the state on the regular intervals, typically every five to ten batches of data; the check point's being made. So, a failure would have to replay, the five to ten previous batches, to recreate the appropriate RDDs. So, checkpoint done to SDF s or equivalent so, streaming back pressure exclaiming black pressure will be enabled and all these will achieve the fault tolerance. So, in nutshell we can say that a check pointing and replication together, will ensure the, the, the recovery of the failures.

Refer slide time :( 25:19)

## Performance

Can process **60M records/sec (6 GB/sec)** on  
**100 nodes** at **sub-second** latency



So, performance if we see that even the grape and the Whatcom performance. So, on sub second latency, it is four it is achieved in one second and two second a very good performance that we are seeing here.

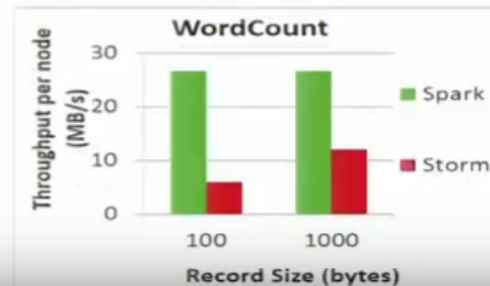
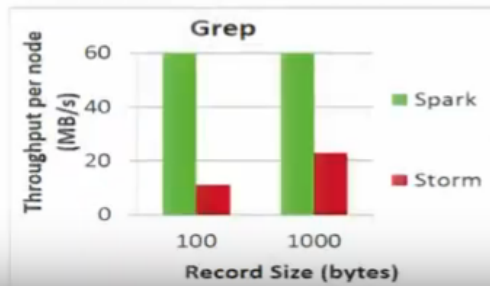
Refer slide time :( 25:39)



## Comparison with other systems

### Higher throughput than Storm

- Spark Streaming: **670k** records/sec/node
- Storm: **115k** records/sec/node
- Commercial systems: **100-500k** records/sec/node

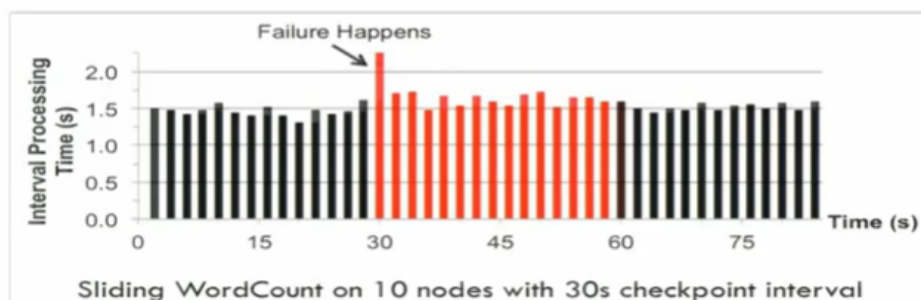


Compared to the other systems, this is also showing, the better performance that is a spark system, is having good performance and storm is basically achieving and the but the higher throughput, a low throughput and a spark is having achieved the better throughput.

Refer slide time :( 26:02)

## Fast Fault Recovery

Recovers from faults/stragglers within 1 sec





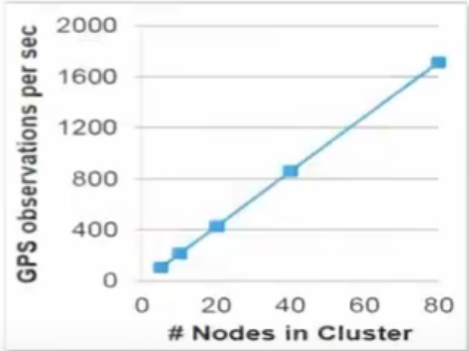
So, fault tolerance recovery systems are there so, it recovers from false or stragglers within one second.

Refer slide time :( 26:10)

## Real time application: Mobile Millennium Project

Traffic transit time estimation using online machine learning on GPS observations

- Markov-chain Monte Carlo simulations on GPS observations
- Very CPU intensive, requires dozens of machines for useful computation
- Scales linearly with cluster size



# Nodes in Cluster	GPS observations per sec
10	100
20	400
40	800
80	1700

So, this also is reported and,

Refer slide time :( 26:18)

## Spark program vs Spark Streaming program

**Spark Streaming program on Twitter stream**

```
val tweets = ssc.twitterStream(<Twitter username>, <Twitter password>)
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFiles("hdfs://...")
```

**Spark program on Twitter log file**

```
val tweets = sc.hadoopFile("hdfs://...")
val hashTags = tweets.flatMap(status => getTags(status))
hashTags.saveAsHadoopFile("hdfs://...")
```

Now let us see the spark program, versus spark streaming program and here again those issues we have seen that. So, whether it is coming from, the spark streaming or coming from the Hadoop file, these particular things are going to be handled in the sparks training system,

Refer slide time :( 26:50)

## Advantage of an unified stack

- Explore data interactively to identify problems
- Use same code in Spark for processing large logs
- Use similar code in Spark Streaming for realtime processing

```
$ ./spark-shell
scala> val file = sc.hadoopFile("smallLogs")
...
scala> val filtered = file.filter(_.contains("ERROR"))
...
scala> val mapped = filtered.map(...)
...
object ProcessProductionData {
  def main(args: Array[String]) {
    val sc = new SparkContext(...)
    val file = sc.hadoopFile("productionLogs")
    val filtered = file.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
object ProcessLiveStream {
  def main(args: Array[String]) {
    val sc = new StreamingContext(...)
    val stream = sc.kafkaStream(...)
    val filtered = stream.filter(_.contains("ERROR"))
    val mapped = filtered.map(...)
    ...
  }
}
```

both as a batch, as well as streaming data. So, all these things we have already, discussed there, that is for having the unified stack and explore, the data interactively to identify the problems and use the same code in the spark for processing, large logs and use similar code in the Spark streaming for the real time. And in the code we can see that, we can apply the, the filter and then we can also, invoke this particular, fault tolerant aspect and then invoking the spark context and once the spark context is invoked then, the driver and the executors are allocated

Refer slide time :( 27:42)

# Roadmap

- Spark 0.8.1
  - Marked alpha, but has been quite stable
  - Master fault tolerance – manual recovery
    - Restart computation from a checkpoint file saved to HDFS
- Spark 0.9 in Jan 2014 – out of alpha!
  - Automated master fault recovery
  - Performance optimizations
  - Web UI, and better monitoring capabilities
- Spark v2.4.0 released in November 2, 2018

and they are basically able to get ready for the computation.