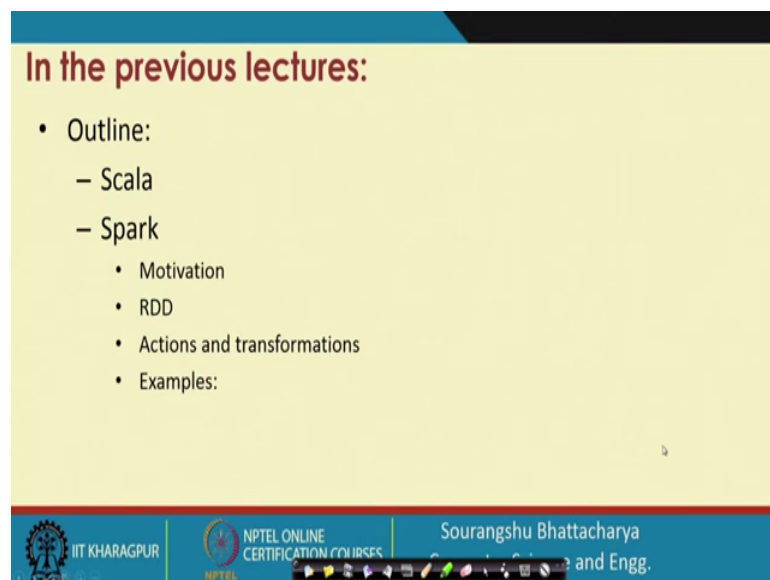


Scalable Data Science
Prof. Sourangshu Bhattacharya
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 19c
Spark (Contd.)

Hello everyone, welcome to the 19th lecture on NPTEL course on scalable data science. I am Prof. Sourangshu Bhattacharya from Computer Science and Engineering in IIT Kharagpur. Today, we are going to see the 3rd lecture on spark.

(Refer Slide Time: 00:34)



In the previous lectures:

- Outline:
 - Scala
 - Spark
 - Motivation
 - RDD
 - Actions and transformations
 - Examples:

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Sourangshu Bhattacharya
Department of Computer Science and Engineering

So, we have already seen we have got an introduction on Scala, and how concepts of scala can be used in spark. And we have also seen the motivation and RDD, and we have seen the concept of actions and transformations, and we have also seen some programming examples in spark.

(Refer Slide Time: 01:00)

In this Lecture:

- Outline:
 - Spark: Programming examples:
 - Alternating least squares
 - User log mining
 - Partitioning
 - Accumulators
 - Scheduling of tasks

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | Sourangshu Bhattacharya
and Engg.

So, we continue with our programming examples. So, we give two more programming examples in spark. And then we go into the concepts of partitioning, and accumulators in spark. And then we go into details of implementation of spark jobs, and how actually jobs are run in spark.

(Refer Slide Time: 01:32)

Collaborative filtering

Predict movie ratings for a set of users based on their past ratings of other movies

$$R = \begin{pmatrix} 1 & ? & ? & 4 & 5 & ? & 3 \\ ? & ? & 3 & 5 & ? & ? & 3 \\ 5 & ? & 5 & ? & ? & ? & 1 \\ 4 & ? & ? & ? & ? & 2 & ? \end{pmatrix}$$

Users (vertical axis)
Movies (horizontal axis)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, the first example we are going to see today is that of collaborative filtering. So, the problem is the following that you are given a matrix of user's cross movies. So, you have a site where users are watching movies. And you, certain users have rated certain movies.

And you want to predict the ratings of other users on these movies. So, basically in the matrix there are certain entries, which are filled and certain entries are question mark. And the question is can you fill in the question mark entries.

(Refer Slide Time: 02:21)

Matrix Factorization

Model R as product of user and movie matrices
A and B of dimensions $U \times K$ and $M \times K$

$R = AB^T$

Problem: given subset of R, optimize A and B

The slide features a diagram where a purple square labeled 'R' is circled in red. To its right is a red vertical rectangle labeled 'A' with handwritten dimensions 'U x K' above it. Further right is a blue horizontal rectangle labeled 'B^T' with handwritten dimensions 'K x M' below it. A red arrow points from the top of 'A' to the top of 'B^T'. To the right of the diagram is a handwritten red 'U' with a dot above it. The slide footer includes the IIT Kharagpur logo and 'NPTEL ONLINE CERTIFICATION COURSES'.

So, a popular approach to solving these collaborative filtering problem is using, what is called a matrix factorization model, which is to say that the you factorize the ratings matrix R. Here as a product of two matrices. So, if so this is a user cross movie matrix, so this is user cross movie matrix, so the first matrix is a user cross K matrix, where K is the latent dimension. And the second matrix B is or B transpose is a K cross movie matrix. In other words, you are you have to find out some latent features for users.

(Refer Slide Time: 03:27)

Matrix Factorization

Model R as product of user and movie matrices
A and B of dimensions $U \times K$ and $M \times K$

$$R = A B^T$$

Problem: given subset of R, optimize A and B

Handwritten notes: $R_{um} = A_u^T B_m$

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

And some latent features for movies such that if you multiply this, so if you multiply for user U the latent feature yeah, so if you take the user U features, and take dot product with the features of the matrix B mth column of this you get the rating, which is $u^T B_m$. So, so this is the this is the setting of the collaborative filtering.

(Refer Slide Time: 04:00)

Alternating Least Squares

Start with random A and B

Repeat:

1. Fixing B, optimize A to minimize error on scores in R
2. Fixing A, optimize B to minimize error on scores in R

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, the algorithm for solving this problem is very simple. It is called the alternating least squares algorithm. So, we start with random matrices of A and B which is the user matrix and the movie matrix. And then first we fix the matrix B. And then find A which

optimizes the error on scores in R. And the second step is you fix A to the A that you have obtained in the previous step. And then you optimize for B such that it minimizes the error on scores of R again. So, you do this repeatedly, first fixing B and then next fixing A. And every time you update the B matrix and the A matrix or the movie matrix and the user matrix.

(Refer Slide Time: 05:10)

```
val R = readRatingsMatrix(...)

var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
    .map(i => updateUser(i, B, R))
    .toArray()
  B = spark.parallelize(0 until M, numSlices)
    .map(i => updateMovie(i, A, R))
    .toArray()
}
```

The slide includes a video player interface at the bottom with logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES.

So, how you will implement this in spark? So first, so the first simple implementation here I have provided is that you have the ratings matrix, which is a matrix in a local variable. And then you create this you create this A matrix which is matrix of random vectors, and also the B matrix, which is a matrix of random vector. So, initialize the A and the B matrix.

Now, for parallel updating what you do is you first parallelize the set of users. So, you want to update the vectors for each user. So, you pass the current B and the current R matrix. And then in the *i*th machine, so this operation in inside the map is happening on the *i*th machines. So, in the *i*th machine you update for the user *i*. And then again you bring it back to the local machine.

And similarly you do this for each and every movie. So, you again the number of machines that you have you parallelize the movies in M movies that you have in to each of these machines. So, numSlices here is saying how many machines you have. So, if you have 10 machines, you split the group of users in to 10 machines. And then again for

each user in that machine in the i th machine you update the movie vector using the optimization.

So, each both these update user and update movie functions are local functions, which are running on each of these machines. And these are optimizing the vectors for this particular user or movie. Now, the problem with this implementation is that you are doing a lot of communication. So, every time you are sending the appropriate values for this ratings matrix which is not changing.

So, as you can see the A matrix, and the B matrix are changing, but the ratings matrix remains fixed throughout this computation. So, how can we take advantage of this situation and not send these A and B matrix as a closure. So because as we have discussed already whenever you try to run a mapper function, which is a closure. The whole variables all the variables in this closure have to be passed to the corresponding machine, where this map function is running. So, these are local variables on the controlling machine, which have to be passed to the machine in which this map function is actually getting executed. So, we want to avoid that.

(Refer Slide Time: 09:08)

Efficient Spark ALS

```
val R = spark.broadcast(readRatingsMatrix(...))
var A = (0 until U).map(i => Vector.random(K))
var B = (0 until M).map(i => Vector.random(K))

for (i <- 1 to ITERATIONS) {
  A = spark.parallelize(0 until U, numSlices)
    .map(i => updateUser(i, B, R.value))
    .toArray()
  B = spark.parallelize(0 until M, numSlices)
    .map(i => updateMovie(i, A, R.value))
    .toArray()
}
```

Solution:
mark R as "broadcast variable"

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, we implement this using what is called a broadcast variable. So, what is a broadcast variable? Broadcast variable is a variable which you can access just like a local variable, but these are actually pre distributed on all the nodes of the machine. So, whenever you say that this you declare this rating matrix, which you have read using let us say read

rating matrix function. As a spark dot broadcast variable, spark as and when required will send this the values of this matrix R on or rather it will send the whole matrix to all the servers on which the jobs are running.

And instead of R now, we just have to use R dot value, because dot value actually gives you the value of this broadcast variable. So, R now is actually the broadcast variable, whose value field actually contains the matrix, and it is stored in each and every server. And rest of the code remains as it is. So, then you can see that this, the ratings matrix does not have to be transmitted to each and every server all the time.

(Refer Slide Time: 10:47)

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

Result: full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)

The diagram illustrates the Spark architecture. A central 'Driver' node is connected to three worker nodes labeled 'Cache 1', 'Cache 2', and 'Cache 3'. Each worker node contains a 'Wor' (Worker) component and a 'Block' component (Block 1, Block 2, Block 3). Arrows labeled 'tasks' point from the Driver to the Worker components. Arrows labeled 'results' point from the Worker components back to the Driver.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, we take the another example, which is the example of log mining. So, here the idea is that you have various users who are, so you want to search interactively for various patterns in an error log. So, the first task is basically that you read the lines of the error log file using the text file command as we have seen already.

The next is that you read the; you read or you filter out the lines, which do not have error or rather you fill filter the lines which have error. And then you store this errors RDD after getting the messages into this cached messages. So, we have seen this part of the coat. Now, suppose you want to get a error message, which has let us say the term foo. So, you want to get the error message, which has the term foo. So, you call on these cached messages, the filter command.

And then you call the count command, so because you call the count command the action happens. So, then this RDD is materialized, and you want to count basically the number of error messages which contains foo. So, what will happen is that all these blocks will be created from the cached RDD. So, this all these so this entire code will now be executed, up till now up till this no code has been executed. So, up till this line, no code has been executed. Now, it will or rather code has been executed that no RDD has been materialized. So, no materialization has been done. Now, one RDD will be materialized. And hence, this RDD blocks will be created on this worker node. So, this is block 1, block 2, block 3. So, all this RDD blocks will be created.

Now, next time and then the results will be brought back to the driver, which is the total count in this case. Now, because we had called this cache function here so, it will not only create this blocks, but it will also create this cache for this cached messages. So, for this cached messages RDD, it will create this cached on each of these worker machines.

Now, suppose you want to execute this same command, but this time you want to search for the string bar instead of the string foo. So, earlier for the same set of you know filtered messages, you were searching for foo, now you are searching for bar. Now, what will happen is again the tasks will go to different servers, but this time instead of hitting the blocks or the actual lines here, it will actually hit the cached.

So, all this portion of code will now not be executed. Instead, it will directly hit the cached messages RDD. And it will filter those with bar, and then it will compute the count and come back with the result. So, this is the advantage of the caching mechanisms. So, it says for all these read operations and write operation. So, in case of interactive or iterative computation, this cache command helps reduce the cost of execution or time of execution by this mechanism. So, for example, a full-text search on Wikipedia will take less than 1 second if you do cache and if you do it without caching it will take 20 seconds.

(Refer Slide Time: 15:53)

Example: Log Mining

Load error messages from a log into memory, then interactively search for various patterns

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split('\t')(2))
cachedMsgs = messages.cache()

cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

Result: scaled to 1 TB data in 5-7 sec
(vs 170 sec for on-disk data)

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

And similarly for other things.

(Refer Slide Time: 15:57)

Spark Scheduler

Dryad-like DAGs
Pipelines functions within a stage
Cache-aware work reuse & locality
Partitioning-aware to avoid shuffles

Stage 1: A, B (groupBy)
Stage 2: C, D, E, F (map, union)
Stage 3: G (join)

Legend: = cached data partition

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, with this we stop our programming examples. So, we have given many programming examples and scenarios, where you can write machine learning programs using spark, very complex machine learning programs. And you can also feed them up somewhat. Now, we actually go into how this programs are executed. So, we recall that, we had said that whenever a spark a program is written and executed, until an action is called the only the lineage graph is created.

So, this is an example of a lineage graph, but in addition so A, B, C, D, E, F are the RDDs in this lineage graph. So, in the lineage graph all the nodes are RDDs, and the arrows are showing the dependencies. But, we have now gone deep into how the dependencies look like in terms of partitions. So, not only do the RDDs so each of the RDDs will have a certain number of partitions. So, each RDD is actually present in multiple machines. So, let us say this RDD A is present in three machines and so on and so forth.

So, now these partitions can either so for example, the RDD B which is created by a transformation from RDD A, can either have this kind of all to all dependency that is all partitions of B depend on all partitions of A or it can have this kind of a one to one dependency. So, there are basically two types of transformation. So, edges on in this graph are of course marked by transformation as you can see. So, one set of transformations like map, filter, etcetera union, etcetera. These side type of transformations produce this one to one dependency or also called lean dependency.



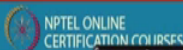
And other transformations like groupBy, groupby key, reduced by key, join etcetera. They produce many to one transformations or many to one dependencies, depending on again what happens. So, now the way the spark scheduler works is that it pipelines the one to one dependencies. So, all the one to one dependencies are executed together in one machine.

So, even though there are two transformations here, this whole thing is compressed into one stage. Whereas, the many to one dependencies or the fag dependencies cause create the stage boundaries. So, basically this whole graph will be executed in three stages. Stage 1, where this RDD will be computed. Stage 2, where this RDD will be computed. And then stage 3, where this RDD will be computed, where the input is basically the RDD from the stage 1 and the RDD from the stage 2. So, this is how the spark scheduler, schedules the jobs or schedule the task. So, each computation here will be done using one task.

(Refer Slide Time: 19:58)

User Log Mining



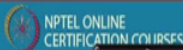
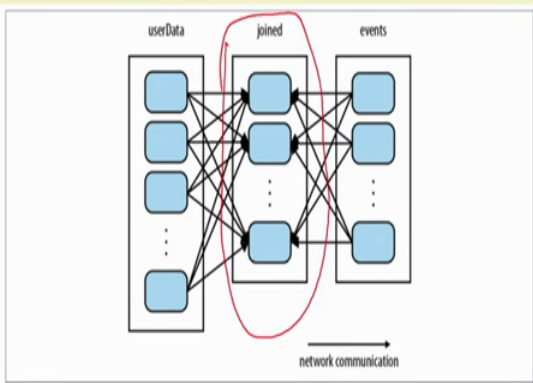
```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...").persist()
def processNewLogs(logFileName: String) {
  val events = sc.sequenceFile[UserID, LinkInfo](logFileName)
  val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo))
  pairs
  val offTopicVisits = joined.filter {
    case (userId, (userInfo, linkInfo)) => // Expand the tuple into its
    components
    userInfo.topics.contains(linkInfo.topic)
  }.count()
  println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```



So, we will skip this user log mining.

(Refer Slide Time: 20:10)

User Log Mining



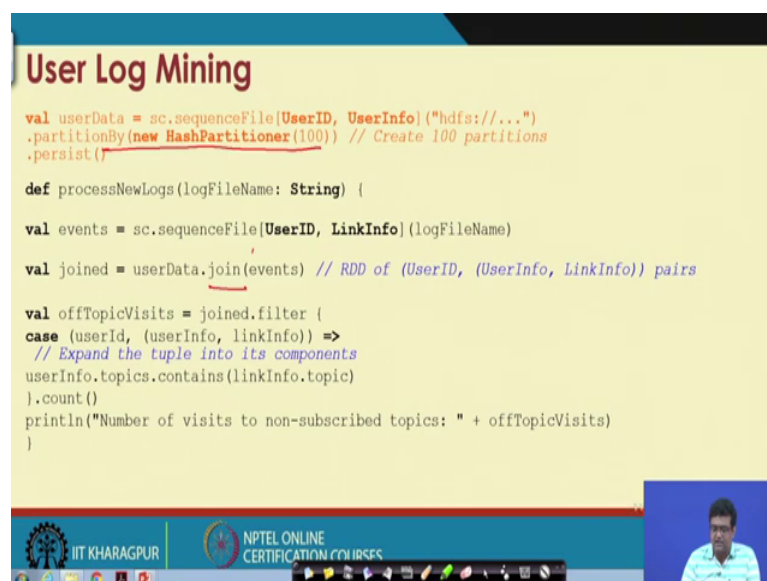
So, now we will look at the task of another task of user log mining and we will see an important concept called the partition in concept. So, the task here is that you are given two types of data. One is the UserData, which stores the user ID and the UserInfo. So, think about it that users are visiting links in a particular on the internet and you are collecting that data.

So, now what you want to report is how which users are visiting off topic links. So, for each user, you have certain number of topics in which the user is interested. And similarly for each link you have a certain number of topics, which this link pertains to. So, you want to find out, which user is visiting and off topic or the number of user who are visiting an off topic link. So, what you do is you get the events RDD, where the fields are UserID and the LinkInfo.

And then you have the joint RDD, which basically joins the on the user ID. So, you know which user has visited which link in one table. And now what you want is you want to filter userInfo dot topics dot contain. So, that topics in UserInfo is giving the topics at the user is interested in. And similarly the topic in link in phase giving the topics the link pertains to. And if this is true, then you just want to filter out those topics or as a you do not want to use those topics.

So, basically you just want to filter out the cases, where actually this should be not contains. So, you want to see where user topics are not contained in linked topics. So, and then you want to call count to see off topic visits, see the number of off topic visits for each user. Now, if you see your RDD will look something like this, so here on one hand you have the userData, on the other side you have events data. And you are having this communications of for creating this joined RDD which is there. Now, you see that this is unfortunately a lot of communication.

(Refer Slide Time: 23:37)



The slide displays the following Scala code:

```
val userData = sc.sequenceFile[UserID, UserInfo]("hdfs://...")
                .partitionBy(new HashPartitioner(100)) // Create 100 partitions
                .persist()

def processNewLogs(logFileName: String) {

    val events = sc.sequenceFile[UserID, LinkInfo](logFileName)

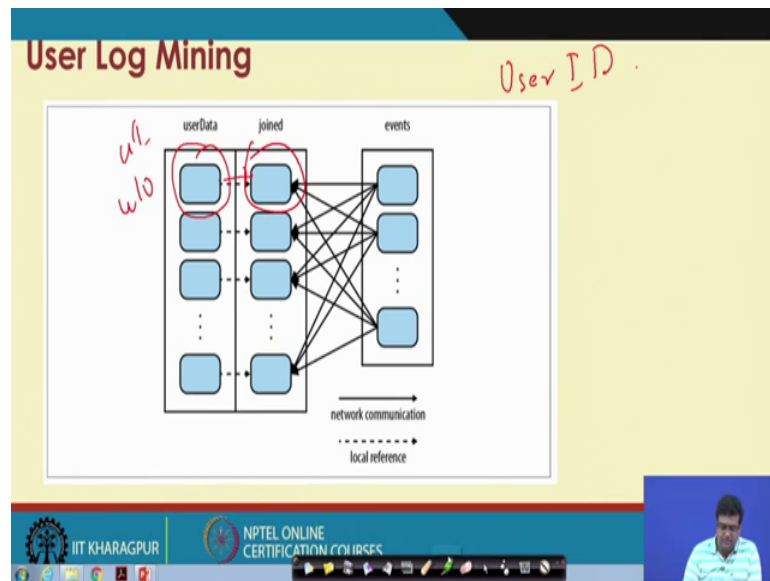
    val joined = userData.join(events) // RDD of (UserID, (UserInfo, LinkInfo)) pairs

    val offTopicVisits = joined.filter {
        case (userId, (userInfo, linkInfo)) =>
            // Expand the tuple into its components
            userInfo.topics.contains(linkInfo.topic)
    }.count()
    println("Number of visits to non-subscribed topics: " + offTopicVisits)
}
```

The slide also features the IIT Kharagpur logo and NPTEL Online Certification Courses branding at the bottom. A small video inset in the bottom right corner shows a presenter.

Instead, what you can do is you can use what is called a partitioner.

(Refer Slide Time: 23:42)



Basically, what you need is this kind of a graph. So, you know that your key for the joined RDD. So, for all these three RDD your key is User ID. So, what you can do is you can have the joined RDDs such that the keys of this partitions are collocated with the keys of this partition. So, this will have exactly the number of partition as the user data. So, the data for the same user will be stored in the same machines as the userData RDD was stored. So, then only the events data for this particular user, for let us say you are storing from user 1 to user 10 in this particular.

So, then for all these event partitions only user 1 to user 10 data needs to be transferred to this. So, this shapes a lot of bandwidth. So, this is done using the construct called partitioner. So, what you can do is while reading the userData, you can provide this partition by because, and then you can persist it. So, when you do this partition by thing it basically creates a partitioned RDD. So, when you join a partitioned RDD with another RDD, then the data flow happens from the events RDD to the partitioned RDD. So, this produces a more efficient communication.

(Refer Slide Time: 25:42)

Partitioning

- Operations **benefiting** from partitioning:
cogroup(), groupWith(), join(), leftOuterJoin(), rightOuterJoin(), groupByKey(), reduceByKey(), combineByKey(), and lookup().
- Operations **affecting** partitioning:
cogroup(), groupWith(), **join()**, leftOuterJoin(), rightOuterJoin(), groupByKey(), reduceByKey(), combineByKey(), partitionBy(), sort()

mapValues() (if the parent RDD has a partitioner),
flatMapValues() (if parent has a partitioner)
filter() (if parent has a partitioner).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, there are many operations or rather transformations, which benefit from partitioning such a join, cogroup etcetera. And many of this operations effect partitioning. So, basically these operations create partitioned RDDs, if the input RDD is already partitioned.

(Refer Slide Time: 26:09)

Page Rank (Revisited)

```
val links = sc.objectFile[(String, Seq[String])]("links") .  
partitionBy(new HashPartitioner(100)).persist()  
  
var ranks = links.mapValues(v => 1.0)  
  
for(i<-0 until 10) {  
  val contributions = links.join(ranks).flatMap {  
    case (pageId, (nbh, rank)) => nbh.map(dest => (dest, rank / nbh.size))  
  }  
  ranks = contributions.reduceByKey((x, y) => x + y).  
  mapValues(v => 0.15 + 0.85*v)  
}  
ranks.saveAsTextFile("ranks")
```

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

And then you can see that your page rank algorithm can also be speeded up using this partitioning, because what you can do is you can create the links RDD as a partitioned RDD. And in that case every time you join the links RDDs with the ranks RDD, only the

ranks get copied to the links instead of every time moving both links and ranks to a third machine.

(Refer Slide Time: 26:55)

Accumulators

```
val sc = new SparkContext(...) val file = sc.textFile("file.txt")

val blankLines = sc.accumulator(0)
// Create an Accumulator[Int] initialized to 0

val callSigns = file.flatMap(
  line => { if (line == "")
    blankLines += 1 // Add to the accumulator
  }
  line.split(" ") })

callSigns.saveAsTextFile("output.txt")

println("Blank lines: " + blankLines.value)
```

The diagram illustrates the Spark architecture. A central 'Master' node is connected to several 'Task' nodes. A 'Closure' is shown as a box containing a task, with an arrow pointing from the Master to the Task. The code on the slide shows how a closure is used to pass the accumulator variable to the task.

So, another important topic and is what is called accumulators. So, up till now what we have seen that the machines can be thought of as the master machine. So, the machines that are used in spark can be thought of as the master machine, and the task or client machines. So, this is where all the tasks are running. Now, we know that so all the communication happen. So, all the communications for spark happen between the task machines.

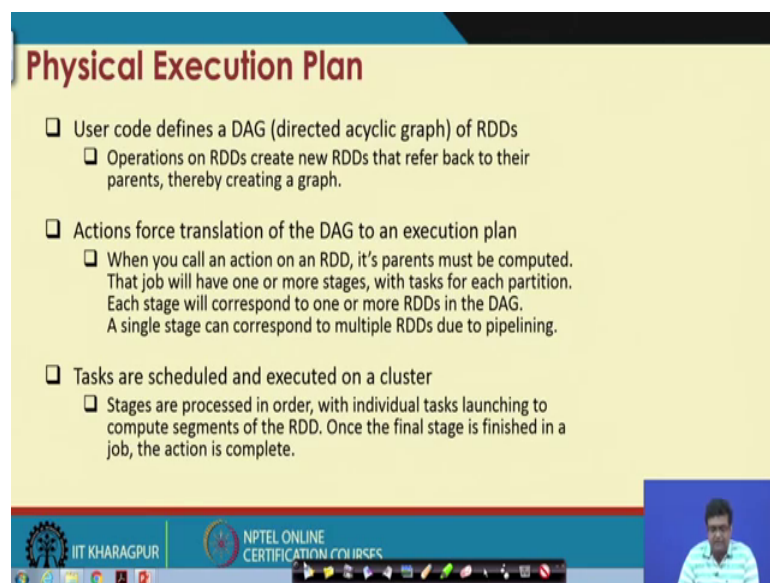
So, for communicating a master variable to the task machines, we use what are called closures. So, if you define a local variable on the master and use it in a function in the mapper or the task, then this variable can be accessed in the task. But, how can you use some variable which is defined in the task machine on to the function machine. So, this cannot always be done, but it can be done in certain restricted ways.

So, in order to do this you have to use the accumulator variable. So, what is the accumulated variable? So for example, in this case you can define for a given spark context and accumulator variable. Now, say you are, you have this RDD which takes this file RDD and calls a flat map. And basically, it counts the number of blank lines. And then you can say so if there is a blank, so it is so you have this file for which you are

processing the lines. And if you have line which is blank, you just increment this blank line.

So, note that this line RDD is distributed on to the task machines. But, when the increment happens, this blank line is actually a accumulated variable, which is defined on the master machines. So, the blank lines gets incremented. So, at the end of it, you can use the blank lines dot value to get the total number of blank lines from all these machines.

(Refer Slide Time: 29:58)



Physical Execution Plan

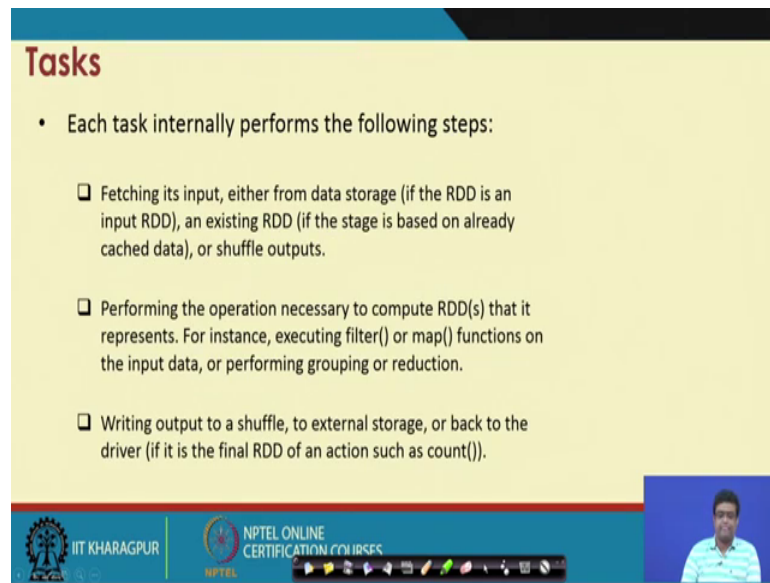
- ❑ User code defines a DAG (directed acyclic graph) of RDDs
 - ❑ Operations on RDDs create new RDDs that refer back to their parents, thereby creating a graph.
- ❑ Actions force translation of the DAG to an execution plan
 - ❑ When you call an action on an RDD, it's parents must be computed. That job will have one or more stages, with tasks for each partition. Each stage will correspond to one or more RDDs in the DAG. A single stage can correspond to multiple RDDs due to pipelining.
- ❑ Tasks are scheduled and executed on a cluster
 - ❑ Stages are processed in order, with individual tasks launching to compute segments of the RDD. Once the final stage is finished in a job, the action is complete.

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, finally we get into how the execution plan is made. So, user code defines so we have already seen that user code defines a DAG for the RDDs and so operations on the RDDs create new RDDs that refer back to their parents, thereby creating a graph. So, actions force translation of the DAG in to an execution plan. So, basically whenever you call an action on an RDD, it space so the RDD itself and its parents must be computed.

The job has to be divided into stages, as we have already explained. And each stage has to be computed and the basically the RDD can have one or more the computation of RDD may involve more one or more than one stages. And tasks are basically scheduled and executed on the cluster. So, stages are processed in a certain order, and individual tasks for computing this stages are computed, are launched on each of the machines until the whole job or the action is complete.

(Refer Slide Time: 31:35)



Tasks

- Each task internally performs the following steps:
 - ❑ Fetching its input, either from data storage (if the RDD is an input RDD), an existing RDD (if the stage is based on already cached data), or shuffle outputs.
 - ❑ Performing the operation necessary to compute RDD(s) that it represents. For instance, executing filter() or map() functions on the input data, or performing grouping or reduction.
 - ❑ Writing output to a shuffle, to external storage, or back to the driver (if it is the final RDD of an action such as count()).

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES

So, each task internally performs the following steps. So, it fetches the input from either the data storage. If the RDD is an input RDD, like using either a text file or parallelized command or an existing RDD. And then it performs the operations necessary to compute the a particular partition of that RDD. Typically tasks are used for computing partitions of an RDD. So, for instance executing a filter or map operations on the input data or performing a group operation using a shuffle if necessary.

And then write the output of a shuffle, to an external storage. So, external storage can either be memory or a local file system as we have seen in case of map reduce. And then this RDD is materialized. And we go forward from there to computing other RDDs as defined in the execution plan, which has already been designed. So, this is how the spark internals work.

(Refer Slide Time: 33:09)

Conclusion:

- We have seen:
 - Spark: Programming examples:
 - Alternating least squares
 - User log mining
 - Partitioning
 - Accumulators
 - Scheduling of tasks

The slide footer includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, the name Sourangshu Bhattacharya, and the text 'and Engg'. A small video inset of the speaker is visible in the bottom right corner.

So, in conclusion we have seen in this lecture, which has three parts; the spark the scala programming language, the spark programming platform. And we have seen how to implement some machine learning algorithm using spark. And then we have seen also seen, how the spark scheduler internally operates. And how we can and how basically we can speed up the various spark program using constructs such as portioning and also constructs such as cache and also how we can use accumulators and closures.

(Refer Slide Time: 34:07)

References:

- Learning Spark: Lightning-Fast Big Data Analysis. Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. O Reilly Press 2015.
- Any book on scala and spark.

The slide footer includes the IIT Kharagpur logo, NPTEL Online Certification Courses logo, the name Sourangshu Bhattacharya, and the text 'and Engg'. A small video inset of the speaker is visible in the bottom right corner.

So, the reference is for this are learning spark, which is the book by Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia. So, Matei Zaharia is also the inventor of spark or any other book on spark, there are many books on spark available and also on scala.

Thank you.