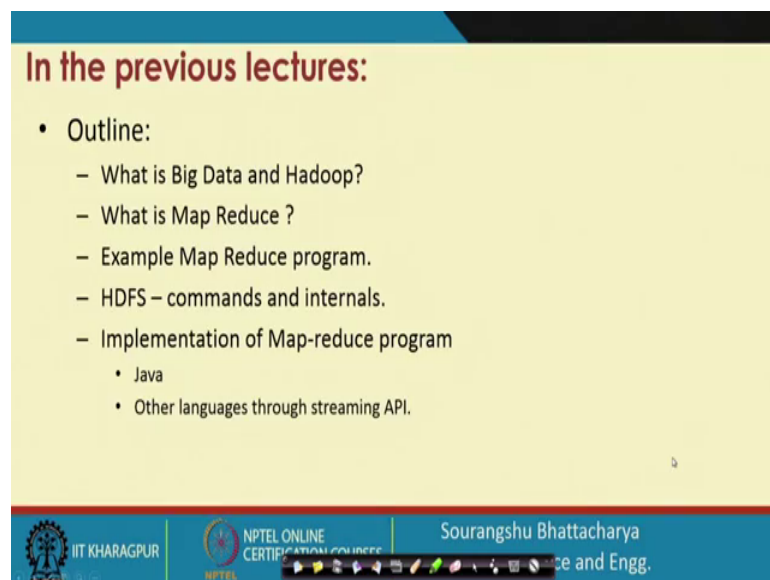


**Scalable Data Science**  
**Prof. Sourangshu Bhattacharya**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture – 29**  
**Hadoop System ( Contd. )**

Hello everyone. Welcome to the 3rd portion of 18th lecture on Scalable Data Science. I am Professor Sourangshu Bhattacharya from Computer Science and Engineering Department at IIT, Kharagpur and the topic of today's discussion is Hadoop Systems.

(Refer Slide Time: 00:39)



**In the previous lectures:**

- Outline:
  - What is Big Data and Hadoop?
  - What is Map Reduce ?
  - Example Map Reduce program.
  - HDFS – commands and internals.
  - Implementation of Map-reduce program
    - Java
    - Other languages through streaming API.

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

So, in that previous lecture,s we have discussed what is big data and hadoop, what is map reduce. We have seen some examples of map reduce programs, we have seen HDFS commands and internals and we have seen some implementation of map reduce programs in java and other languages through streaming API.

(Refer Slide Time: 01:01)

**In this Lecture:**

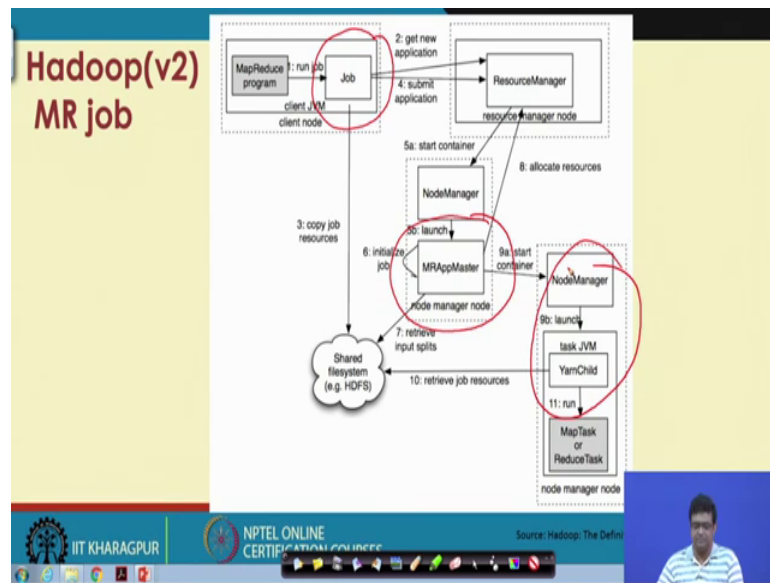
- Outline:
  - Map-reduce implementation details
    - Sort and shuffle
    - Coordination
    - Fault-tolerance
    - Pipelining
    - Refinements

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE | Sourangshu Bhattacharya  
Computer Science and Engg.

So, in this lecture, we will go into the implementation details of the map reduce framework and specifically we will look at the implementation of sort and shuffle. As we discussed this is an important component and this is the main component which is important for scalability of any map reduce program. We will see how the master or the app master coordinates with the mapper and the reducer tasks, then we will see how fault tolerance is implemented which is that if a particular mapper or a reducer tasks fails to respond, how does the app master recover from the situation.

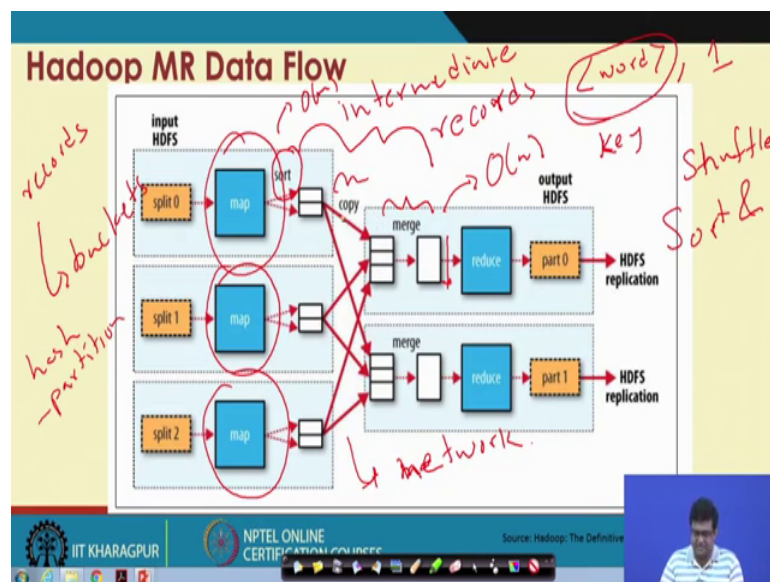
Then, we will see some we will see the notion of pipelining of map and reduced tasks and finally, we will see some refinements which help speed up a map reduce program.

(Refer Slide Time: 02:13)



So, we have seen this overall picture where basically a map reduce program consists of multiple jobs and each job has an app master which co-ordinates between various map or reduced tasks which are needed to be executed in order to complete this job.

(Refer Slide Time: 02:45)



Now, as you have seen; one of the, so as you have seen the the parallel picture for for a map reduce tasks is that several mapper tasks get executed in parallel possibly in different machines. Now, these mapper tasks produce some intermediate results, intermediate records let us say this mapper tasks produce some intermediate record. So,

for example, in case of your in case of your word count program, this intermediate records where of this form and then, 1 and there where as many records of this form as there are words in the whole file and now we are having a big data. So, possibly the number of intermediate records is going to be very high.

So, now what do we do with this? What does the map reduce system have to do with this intermediate records? So, they have to ensure that the intermediate records with the same key in this case the same word. So, word is the key here. So, the intermediate records with the same key reach the same reducer and this operation is called the shuffle or sometimes sort and shuffle operation, ok. So, the question is how can this operation be executed in a scalable manner and the key to this lies in distributing the tasks in different portion.

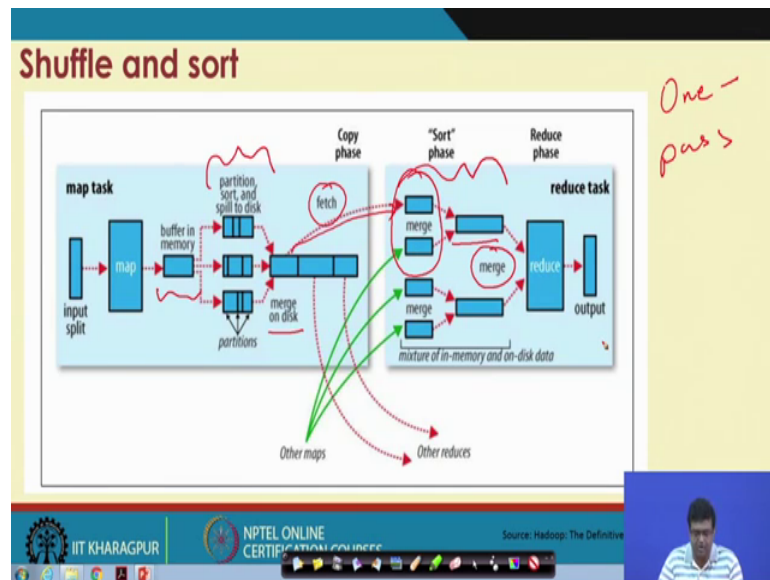
So, for example, if the, so the first thing that t is, so that is done on the mapper side is what is called the sort operation. So, sort operation sorts each record into different buckets as you can see the buckets here. So, in some sense it creates the outputs for each of the reducers, ok. So, how does we will come to how the sort operation computes which records will be put in the same bucket. So, in order to map records to buckets, the mapper function may use something like a hash function or some other partition function, ok.

So, this basically partitions the output mapper output records into different buckets based on the mapper output key. So, this is the first step. The second step is that it should know where the output of each bucket must go. So, both the mapper and the reducer must know. So, the reducer should know. So, for example the first bucket should go to the reducer 1 and the second bucket should go to the reducer 2 and this should happen for each mapper.

Now, both the mapper and the reducer should use a consistent hash function, the same hash function in order for this to happen because the reducer knows the records or the keys that will come to it and hence, it can provide the hash to the mapper. Now, this phase is called the copy phase. So, fast inside the mapper, there is the sort phase, then there is a copy phase where the reducer asks for records in a particular bucket from each of the mappers. So, this phase is called the copy phase.

Then, finally in the reducer there is a phase called the merge phase. So, the merge phase basically merges the records which have come to a particular reducer into one single sorted list. Now, here note that since individual lists retrieved from all the 3 mappers are sorted, this merge function can be easily executed. So, this is an order  $n$  function. Similarly the sort function is an order  $n$  function because all the mapper has to do is, it has to compute the hash value for or the partitioner value for the mapper key and then, put the record into different buckets, and the copy phase is a network heavy phase. So, the copy phase is a network is in fact it is the only network operation that is done or the large scale network operation that is done while executing a map reduce frame work, a map reduce job, ok. So, we will see how to speed up this copy phase more.

(Refer Slide Time: 09:23)

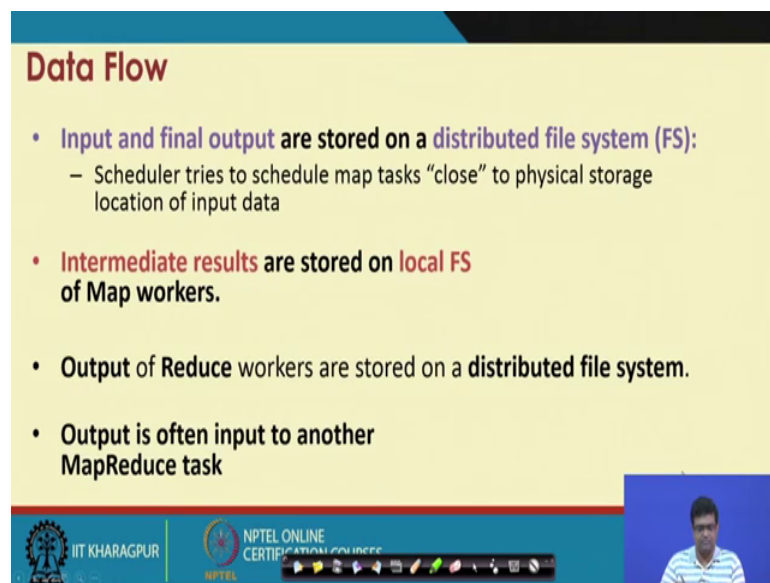


Now, note that this is a zoomed in view on what happens in each mapper site. So, as the mapper function is executing, it is producing more and more mapper output records. So, the mapper output records are being portioned and sorted and it is cleared that this mapper output record I mean it may so happen that the all the mapper outputs record from a given mapper will not fit into the mapper memory. So, these have to be written to the disk. Now in which disk is this written, so we will come to that but once this is and this happens in many many phases and then, there is a local merge on disk inside the mapper task which is executed before the copy phase is executed, ok.

So, basically the mapper outputs records are written to the local file system by the mapper.

Now, when the copy phase starts, the reducer calls a fetch on the mapper task to give it the appropriate partition of the data. So, now it receives many such partitions which are sorted in themselves. So, as they receive these sorted data, they merge and store it or the reducer mergers and stores it into its local file system because again the reducer input record need not be need not be there or it may be that it feels from the memory. So, this merge also actually is executed on disk. However, the key thing here is that both on the mapper side and on the reducer side, it is a one pass operation. So, one pass through the data is able to achieve this whole either the partitioning of the data on the mapper side or the merging of the data on the reducer side, this is very important for fast execution of map reduce task.

(Refer Slide Time: 12:29)



**Data Flow**

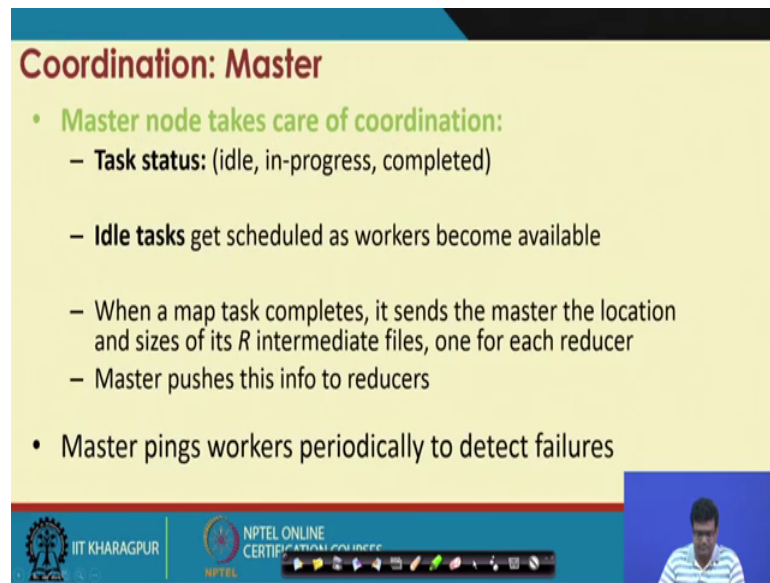
- **Input and final output are stored on a distributed file system (FS):**
  - Scheduler tries to schedule map tasks “close” to physical storage location of input data
- **Intermediate results are stored on local FS of Map workers.**
- **Output of Reduce workers are stored on a distributed file system.**
- **Output is often input to another MapReduce task**

The slide also features logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATION COURSES, along with a small video inset of a presenter in the bottom right corner.

So, as we have discussed the input and output and the final output are final output of the reducer task are stored on the distributed file system. So, these are fault tolerant by design.

However the intermediate results are stored on the local file system of the map and reduce workers and the output of one map reduce task is often an input to the other map reduce task ok.

(Refer Slide Time: 13:11)



**Coordination: Master**

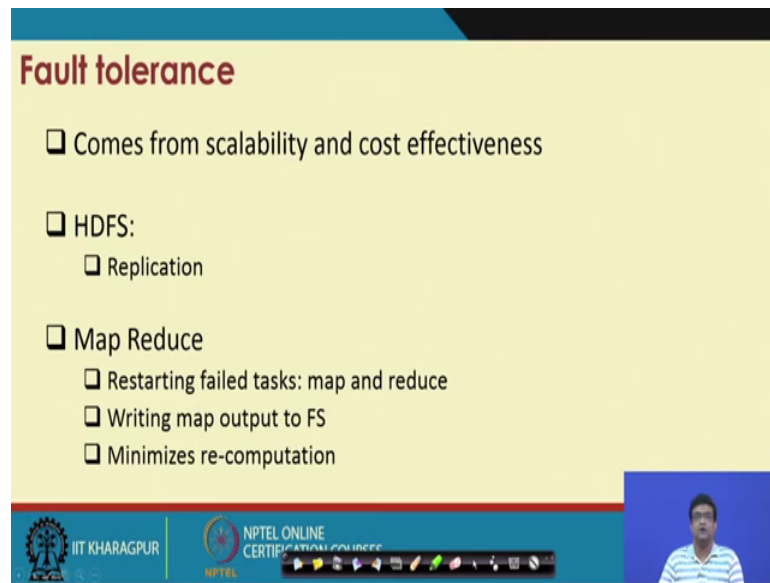
- **Master node takes care of coordination:**
  - **Task status:** (idle, in-progress, completed)
  - **Idle tasks** get scheduled as workers become available
  - When a map task completes, it sends the master the location and sizes of its  $R$  intermediate files, one for each reducer
  - Master pushes this info to reducers
- Master pings workers periodically to detect failures

The slide is part of an NPTEL presentation. At the bottom, there are logos for IIT KHARAGPUR and NPTEL ONLINE CERTIFICATE PROGRAMME IN ENGINEERING. A small video inset shows a person in a striped shirt.

Now, how does the master; so, as we have discussed the master node takes care of the coordination between the different mapper and the reducer task ok. So, basically what happens is for every task that the master node response the task is either in idle state or in progress or completed state ok. So, initially all tasks are idle and as more and more workers become available and the tasks are ready to be executed their status gets the idle task gets scheduled and they become in progressed tasks.

So, the other thing is when a map task completes it intimates the master that it has completed and it sends the master the location and the sizes of its intermediate files and then the master gives this info to the reducer which then calls talks to the mapper tasks for a fetch from the of the intermediate result. And of course, the master pings the workers periodically to detect if there are any failures.

(Refer Slide Time: 14:49)



**Fault tolerance**

- Comes from scalability and cost effectiveness
- HDFS:
  - Replication
- Map Reduce
  - Restarting failed tasks: map and reduce
  - Writing map output to FS
  - Minimizes re-computation

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE

Now, comes the important issue of fault tolerance because as we have seen in the big data world, the fault tolerance is a very important requirement. So, there are roughly two components. So, a fault tolerance is important. The first is HDFS and we have seen how it achieves fault tolerance through replication. So, we can assume that whatever data is stored on the HDFS is you know tolerant to failure.

Now, on the map reduce front while a job is being executed one still needs a fault tolerance. So, the fault tolerance is achieved by restarting. So, the master restarts the map and reduce tasks which have failed and it can write map output to the to the file system to the distributed file system to minimize recomputation at certain times.



(Refer Slide Time: 16:09)

**Failures**

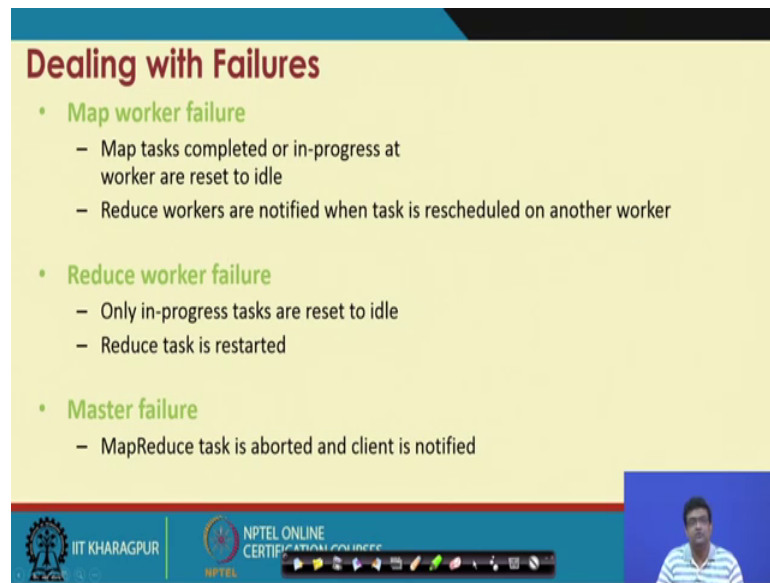
- Task failure
  - Task has failed - report error to node manager, appmaster, client.
  - Task not responsive, JVM failure - Node manager restarts tasks.
- Application Master failure
  - Application master sends heartbeats to resource manager.
  - If not received, the resource manager retrieves job history of the run tasks.
- Node manager failure

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE

So, we go into detail of a task failure. So, detail of failure the first, so as you have seen there are 3 main components of any map reduce task, one is the tasks either the mapper task or the reducer tasks which get executed on the nodes. So, the 2nd important component is the application master which coordinates and controls between the task and finally, there is the node manager which basically coordinates between the different nodes of the cluster.

So, all these 3 can fail. So, if the task fails, it is known to the node manager and the app master and both. So, the app master periodically pings the tasks and if they do not get the response from the task, then they assume that the task has failed. Similarly, application master sends heartbeat to the resource manager or the node manager. So, if the resource manager fails, then the application master knows that or if the application master fails, the resource manager knows that the application master has failed. And then, it restarts the entire job, whereas if only tasks fail, only those tasks need to be restarted by the node manager.

(Refer Slide Time: 18:07)



**Dealing with Failures**

- **Map worker failure**
  - Map tasks completed or in-progress at worker are reset to idle
  - Reduce workers are notified when task is rescheduled on another worker
- **Reduce worker failure**
  - Only in-progress tasks are reset to idle
  - Reduce task is restarted
- **Master failure**
  - MapReduce task is aborted and client is notified

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE

So, now we see that how task failures are dealt with. So, the first is what happens if a map worker fails? So, map tasks completed or in progress at worker are reset to idle. So, first the app master resets a failed map task to idle and then, the reduce workers are notified and when the task is rescheduled, the reduce worker gets the input from this rescheduled tasks.

If a reduce worker fails, then only in progress tasks are reset to idle, ok. Completed task need not be reset to idle. So, completed tasks are remained completed and then, the reduce start is simply restarted.

In the first case even the completed mapper tasks need to be restarted because the input of that mapper tasks, task needs to be read by the reducer. In case the master fails as we have discussed, the resource manager will abort the job and a new job will be started.

(Refer Slide Time: 19:52)

**How many Map and Reduce jobs?**

- $M$  map tasks,  $R$  reduce tasks
- **Rule of a thumb:**
  - Make  $M$  much larger than the number of nodes in the cluster
  - One DFS chunk per map is common
  - Improves dynamic load balancing and speeds up recovery from worker failures
- **Usually  $R$  is smaller than  $M$** 
  - Because output is spread across  $R$  files

*splits →*  
*blocks -*

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE

So, that completes our discussion on how various aspects of the map reduce framework is implemented. Now, we discussed some issues which need to resolved for every map reduce program. So, the first issue is how many mapper and reducer jobs should we use.

Now, let us say there are  $M$  mapper jobs and  $R$  reducers jobs. So, typically in order to take care of or take advantage of the maximum amount of parallelism, the number of mapper tasks can be 1 per DFS chunk. So, in this case also called sometimes called splits or just blocks. So, in DFS terminology, this will be called blocks and in the map reduce terminology, this will be called splits. So, the number of map tasks typically can be the same as the number of splits of data or number of blocks of data that are input to the mapper. If there is a large number of mapper tasks, this actually improves the overall performance, it improve the load balancing because then each mapper tasks is not working on a huge amount of data and that helps the scheduling of different tasks and also, it speeds up the recovery from the worker failure. In fact, it is if the mapper task is very small, it is unlikely that the failure will occur to a mapper task.

Now, the other thing is the number of reducer tasks. So, the number of reducer task typically depends on the job. So, for example in the word count program, the number of reducer task is upper bounded by the total number of words which are present. So, in general or typically the number of reducer tasks needs to be provided to a map reduce

job, otherwise it can be taken to be default value something like which is set by the cluster. So, something like 10 or 15 as depending on the number of nodes in the cluster.

(Refer Slide Time: 22:41)

### Task Granularity & Pipelining

- **Fine granularity tasks:** map tasks  $\gg$  machines
  - Minimizes time for fault recovery
  - Can do pipeline shuffling with map execution
  - Better dynamic load balancing

Process	Time
User Program	MapReduce() ... wait ...
Master	Assign tasks to worker machines...
Worker 1	Map 1, Map 3
Worker 2	Map 2
Worker 3	Read 1.1, Read 1.3, Read 1.2, Reduce 1
Worker 4	Read 2.1, Read 2.2, Read 2.3, Reduce 2

IIT KHARAGPUR NPTEL ONLINE CERTIFICATION COURSES

Now, we discuss the idea of a pipelining, ok. So, as we know that typically we will have a large number of mapper tasks, ok. So, what the app master does is it pipelines the mapper task. So, for example, you can see that there are 3 mapper task in this case 2 mapper tasks are scheduled on worker 1. So, the first mapper task ended in a very short time and the second mapper task was scheduled on worker 2. So, the third mapper task was again scheduled on worker1. So, it is very common to have a pipelined execution that is one after the other execution of mapper task on the same workers, and then, one can do pipeline shuffling for map execution for better dynamic load balancing and then, one can, so once the and the reducer tasks can be scheduled on other machines which read from different mapper tasks as I have seen here.

So, read 1 dot 1 is basically the read of mapper 1s output which starts when the mapper 1 ends, then read 1 dot 3 is the read from mapper 3s output to reducer 1s output which starts when the mapper 3 ends and then, finally when the mapper 2 ends, then we can execute read 2 dot 2.

(Refer Slide Time: 24:59)

**Refinements: Backup Tasks**

- **Problem**
  - Slow workers significantly lengthen the job completion time:
    - Other jobs on the machine
    - Bad disks
    - Weird things
- **Solution**
  - Near end of phase, spawn backup copies of tasks
    - Whichever one finishes first “wins”
- **Effect**
  - Dramatically shortens job completion time

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSE | Rajaraman, J. Ullman

Now, many a times what happens is that out of a large number of mapper tasks or even reducer task, sometimes a few of the tasks are very slow. So, these are not failures, but these mapper tasks execute in a very slow manner. So, there comes a stage when most of the mapper tasks are over and this could be because of many reasons like that particular machine is having a hardware problem or bad disk or some operating system problem or something like that,.

So in this case, sometimes the overall job execution time becomes unnecessarily high and the workers and for example, the reduced workers are also waiting for this map, slow mappers to end, ok. So, while maybe other machines are free to run the job the map job in a fast manner. So in this case the solution becomes that near the end of the mapper tasks what the master does is it schedules backup tasks for the in progress or for the in progress map task or reduce tasks. So, and whichever so now for the same map task or the reduce task, they are actually getting executed on two machines simultaneously. So, this shortens the total job completion time.

Then, the protocol is that whichever of these competing backup tasks ends first, the output of that backup task is taken and then, the other backup tasks for the same task are ignored, ok. So, this is another improvement, ok.

(Refer Slide Time: 27:23)

### Refinement: Combiners

- Often a Map task will produce many pairs of the form  $(k, v_1), (k, v_2), \dots$  for the same key  $k$ 
  - E.g., popular words in the word count example
- Can save network time by pre-aggregating values in the mapper:
  - $\text{combine}(k, \text{list}(v_i)) \rightarrow v_2$
  - Combiner is usually same as the reduce function
- Works only if reduce function is commutative and associative

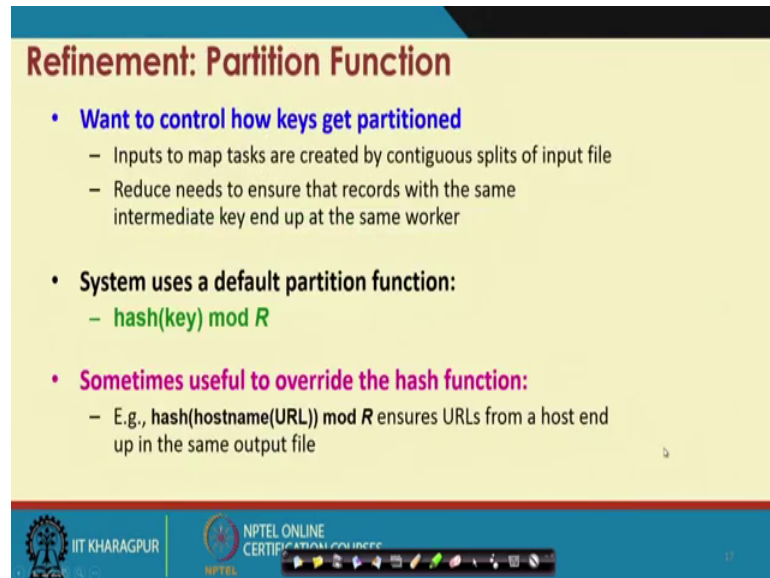
The diagram shows three Map Tasks (Map Task 1, Map Task 2, Map Task 3) each producing intermediate key-value pairs. These are then sent to two Reduce Tasks (Reduce Task 1, Reduce Task 2). Handwritten red annotations on the right side of the slide show a sequence of operations: W1,1, W1,1, W1,2, W1,1, W1,5, indicating the aggregation of values for a specific key (W1) across different mappers and reducers.

Another improvement is that is one of combiner. So, as we have seen the mapper produces many outputs of the form key, value. Now, it may so happen that for the same key, a single mapper produces many many records. So, then in theory it is possible to combine all these records into one record and thus, saving and thus reducing the number of intermediate records by a lot. So, this is the task of the combiners. So, combiners are can be thought of as something like mapper side reducer or reducers which work on the mapper output records in the mapper task itself as to produce combine records.

Now, one has to be careful while using combiner. So, we have used the combiner in case of the word count program that is because the. So, if we have let us say word W11 W11, then we can combine this to form W12 and give this to the output, give this to the reducer which will again add. So, let us say reduce this came from mapper 1 mapper 2 gave W13 and so on and so forth. So, the reducer can combine this to get an output W15. So, the count of the word W1 becomes 5. However, in certain mappers this may not be the case. So, for example, sorry in certain reducers this may not be possible. So, for example, if your reducer is not commutative or associative, then this may not be possible. So, one need to be careful while using the combiner. So, only if your reducer is commutative and associative, can you use the combiner. So, an example of a reducer where you cannot use combiners is for example if your reducer is computing an average of the input numbers, so as you can see the average of averages is not the same as the

average of the original list, or the entire list of numbers. So, in that case the combiners will not work, ok.

(Refer Slide Time: 30:37)



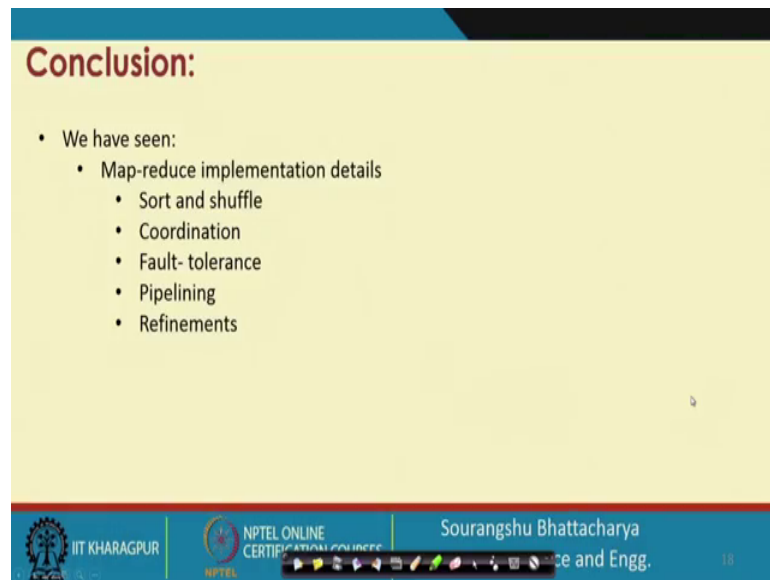
**Refinement: Partition Function**

- **Want to control how keys get partitioned**
  - Inputs to map tasks are created by contiguous splits of input file
  - Reduce needs to ensure that records with the same intermediate key end up at the same worker
- **System uses a default partition function:**
  - $\text{hash}(\text{key}) \bmod R$
- **Sometimes useful to override the hash function:**
  - E.g.,  $\text{hash}(\text{hostname}(\text{URL})) \bmod R$  ensures URLs from a host end up in the same output file

IIT KHARAGPUR | NPTEL ONLINE CERTIFICATION COURSES | NPTEL

Finally, as we have discussed that the bucketing in case of mappers is done by default using the hash of the mapper output key; however you can use your custom partitioners which are very important sometimes if you want to take care of the skew in the key distribution. So, in that case instead of using simple hash partitioner, you can use something like a histogram partitioner where basically the hashing function is used in a manner such that the approximate number of reducer input records for each part. So, the the input is partitioned such that in addition to satisfying the fact that one key should go to one reducer, the number of records in going to each reducer is made roughly equal.

(Refer Slide Time: 31:56)



**Conclusion:**

- We have seen:
  - Map-reduce implementation details
    - Sort and shuffle
    - Coordination
    - Fault-tolerance
    - Pipelining
    - Refinements

The slide footer contains the IIT Kharagpur logo, NPTEL ONLINE CERTIFICATION COURSE logo, the name Sourangshu Bhattacharya, and the text 'ce and Engg.' along with a slide number '18'.

So, with this, so in conclusion we have seen some implementation details of map reduce specifically sort and shuffle, coordination between the map tasks, fault-tolerance pipelining and some refinements.

So, this concludes our lecture on the Hadoop system and these are the references for this lecture.

Thank you.