**Job Tracker –**

* JT runs on a separate node and not usually on a DN.
* JT is an essential Daemon for MR execution in MRv1. in MRv2 It is replaced by ResourceManager or ApplicationMaster.
* JT receives requests for MR execution from client.
* JT talks to NN to determine the location of data.
* JT finds the best TT nodes to execute tasks based on the data locality (proximity of the data) and the available slots to execute a task on a given node.
* JT monitors the individual TTs and the submits back the overall status of the job back to the client.
* JT process is critical to the Hadoop cluster in terms of MR execution.
* When the JT is down, HDFS will still be functional but the MR execution cannot be started and the existing MR jobs will be halted.

**Task Tracker –**

* TT runs on DN. Mostly on all DNs.
* TT is replaced by Node Manager in MRv2.
* Mapper and Reducer tasks are executed on DNs administered by TTs.
* TTs will be assigned Mapper and Reducer tasks to execute by JT.
* TT will be in constant communication with the JT signaling the progress of the task in execution.
* TT failure is not considered fatal. When a TT becomes unresponsive, JT will assign the task executed by the TT to another node.

**Map Reduce Process:**

* **Map:** Map job works on data located on data nodes, this is known as data locality. Map output is stored on local disk and not on HDFS and retained till reducer picks it up. When no reducer is specified then Map output will be written to HDFS.
* **Shuffle and Sort:** Merge: Combine output of each map job, Sort output of map job based on key and Finally partitioning based on key value.
* **Reduce:** No. of reducer are to be decided independently. And its output is written to HDFS for reliability.
* **Split size and Block Size**: Block are HDFS concept, while split is Map reduce concept

**Map Reduce Mechanism:**

Map reduce code is scalable, that means it can run on 1000 computers without a single line of change in code. Visit below link for a sample map reduce code. <https://github.com/dsSanjeet/mapreduce-examples>

**Combiner and Tool Runner:**

Combiner classes extends Reducer class. They are local reducers. Combiner can be applied if nature of problem is Commutative and Associative.

Commutative A+B = B+A.

Associative (A+B) +C = A+ (B+C)

It reduces effort between mapper and reducer.

**Tool Runner:** when it comes to write driver program (contain main method of program) for the MapReduce Job, it’s always preferable to use ToolRunner class & Tool interface. These are just simple implementations but slightly confusing to understand as both contain the “**run”** methods.

**Serialization Deserializaiton:**

Serialization is a mechanism of converting the state of an object into a byte stream. Deserialization is the reverse process where the byte stream is used to recreate the actual Java object in memory. This mechanism is used to persist the object.

**Features of SerDe Framework:** Compact, Fast, Extensible, Interoperable. Java’s SerDe has shortcomings so Hadoop SerDe framework was designed.

**Avro**

Apache Avro is a language neutral SerDe system.

It supports many languages like Java, C++, Ruby, Python, C etc. Avro Schema is written in JSON.

I/O Files

**Input File Formats**

Sequence Files: Flat Files consisting of Binary Key Value pairs.

* SequencFileInputFormat
* SequenceFileAsTextInputFormat
* SeqenceFileAsBinaryInputFormat

**Output File Format**

* TextOutputFromat
* SequenceFileOutputFormat
* MapFile
* MultipleOutputFormat

MapReduce Behind the scene

* Job gets submitted by WaitForCompletion().
* Mapreduce.frameowrk.name property is defined in Mapred-site.xml.It controls job run. It can be is set as Local, classic or YARN.

YARN (MRv2)

In MRv2 the Job Tracker got split into following:

* Resource Manager (Job Scheduling)
* Application Master (Task Monitoring)

Task Tracker has been replaced with the **NodeManager**, a YARN service that manages resources and deployment on a node.

Advantages of YARN are: Scalability. More than 1 YARN can co-exist and Better Memory utilization.

**Entities of YARN are:**

* Client
* YARN Resource Manager
* YARN Node Manager
* MapReduce Application Master
* YARN Child
* Distributed File System

Failure Scenarios in YARN (MRv2):

**Task Failure:** Due to code issues, infinite loop etc.

AM marks the job as failed after mapred.task.timeout.period The failed task will be redirected on another machine and number of attempts retried on another machine will be decided by mapred.map.maxattempts (4). mapred.reduce.maxattempts (4). The acceptable % failure is defined in following properties.

mapred.map.failures.maxpercent

mapred.reduce.failure.maxpercent

**Application Master Failure:** In case of AM failure task need not be resubmitted they can be recovered. But by default this recovery property is not switched on. Following property is used to switch it on.

yarn.app.mapreduce.am.job.recovery.enable

When AM fails RM stops getting heart beat from AM. RM then starts AM on new container. Then execution of job is continued if recovery option is set turned on. Number of attempts is determined by yarn.resourcemanager.am.max-retries

**Node Manager Failure:** When nodemanager fails it stops sending heartbeat to RM. It waits and then marks it as failed.

All remaining task as resend to new Node manager.

If specific node manager fails often and cross a threshold, then it is blacklisted and taken off the available pool.

**Resource Manager Failure:** It is serious failure. In YARN there is checkpoint mechanism to deal with it. New resource manager instance is brought up by admin an it recovers from last saved state, so rerun of all jobs is not required.

Job Scheduling in MRv2.

**FIFO** Scheduler: Allocates resources based on arrival time.

**FAIR Scheduler:** Allocates resources to weighted pools, with fair sharing within each pool. When configuring the scheduling policy of a pool, Domain Resource Fairness (DRF) is a type of fair scheduler.

**Capacity Scheduler:** Allocates resources to pools, with FIFO scheduling within each pool.

**Shuffle and Sort:**

**Shuffling** is the process by which it transfers mappers intermediate output to the reducer. Reducer gets 1 or more keys and associated values on the basis of reducers. The intermediated key – value generated by mapper is **sorted** automatically by key. In Sort phase merging and sorting of map output takes place.

Memory Buffer, Spill on local disk

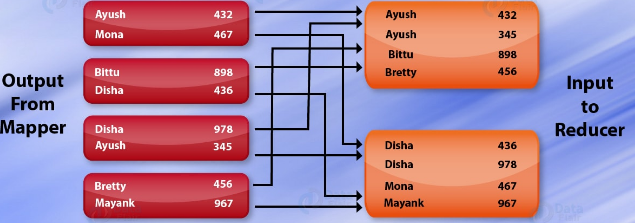
io.sort.mb (100mb)

mapred.local.dir

mapred.compress.map.output

mapred.map.output.compression.codec

io.sort.factor (10)



**Performance Tuning features:**

**Speculative Execution:** Apache Hadoop does not fix or diagnose slow-running tasks. Instead, it tries to detect when a task is running slower than expected and launches another, an equivalent task as a backup (the backup task is called as speculative task). This process is called speculative execution in Hadoop.

It is not a reliability feature. It is just optimization feature. As soon as original or speculative task finishes other task is killed. Below properties should be set to true.

mapred.map.tasks.speculative.execution

mapred.reduce.tasks.speculative.execution

**Task JVM Reuse:** MapReduce tasks are executed by JVM processes/threads, which are forked by the TaskTracker. The creation of a JVM, which includes the initialization of execution environments, is costly, especially when the number of tasks is large. In the default configuration, the number of JVMs needed to finish a job should be equal to the number of the tasks. In other words, the default setting uses one JVM to execute one task. When the execution of a task completes, its JVM will be killed by the TaskTracker.

JVM Reuse is an optimization of reusing JVMs for multiple tasks. If it is enabled, multiple tasks can be executed sequentially with one JVM. These small tasks are also called as uber task

mapred.job.reuse.jvm.num.tasks is used to enable JVM reuse.

**Skipping Bad records:** To enable this feature we need to set following properties in our desired range. mapred.map.max.attempts

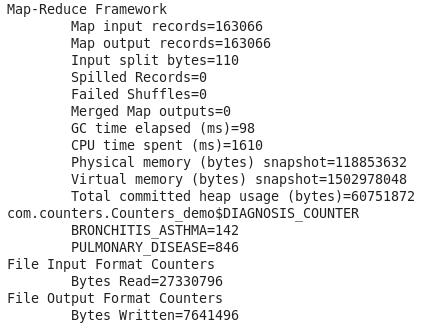
mapred.reduce.max.attempts

**Counters**

**Job Counters:** Number of reduce and Map tasks that ran. The time spent on running tasks and how many maps got advantage of data locality features.

**Task Counters:**

* **FileOutputFromat Counter**
* **FileSystemCounter**
* **FileInputFormat Counter**
* **MapReduce Framework**



**User Defined Counter:** MapReduce allows user code to define a set of counters, which are then incremented as desired in the mapper or reducer.

In Map class we add following code to implement counter context.getCounter(“bad words counter”, “BadRecords”).increment(1);

Command to run MR job on Hadoop:

First compile your java code

Then create the jar of from your class file.

*jar -cvf mycode.jar -C /home/commandline/exec/*

Finally run the job

*hadoop jar mycode.jar /inp /out*

In case of python

*python mostpop.py -r hadoop --hadoop-streaming-jar usr/hdp/hadoop-streaming.jar u.data.*

Sample python MR code.

*from mrjob.job import MRJob*

*from mrjob.step import MRStep*

*class RatingsBreakdown(MRJob):*

*def steps(self):*

*return [*

*MRStep(mapper=self.mapper\_get\_ratings,*

*reducer=self.reducer\_count\_ratings),*

*MRStep(reducer=self.reducer\_sorted\_output)*

*]*

*def mapper\_get\_ratings(self, \_, line):*

*(userID, movieID, rating, timestamp) = line.split('\t')*

*yield movieID, 1*

*def reducer\_count\_ratings(self, key, values):*

*yield str(sum(values)).zfill(5), key*

*def reducer\_sorted\_output(self, count, movies):*

*for movie in movies:*

*yield movie, count*

*if \_\_name\_\_ == '\_\_main\_\_':*

*RatingsBreakdown.run()*