**ASSIGNMENT 17.6**

**Explain the following terms in detail –**

**What are the uses of counters**

Hadoop MapReduce Counter provides a way to measure the progress or the number of operations that occur within MapReduce programs. Basically, MapReduce framework provides a number of built-in counters to measure basic I/O operations, such as FILE\_BYTES\_READ/WRITTEN and Map/Combine/Reduce input/output records. These counters are very useful especially when you evaluate some MapReduce programs. Besides, the MapReduce Counter allows users to employ your own counters. Since MapReduce Counters are automatically aggregated over Map and Reduce phases, it is one of the easiest way to investigate internal behaviours of MapReduce programs.

To create user defined counter and increment it, use: context.getCounter(, ).increment();

**MR Unit testing is based on**

MRUnit testing framework is based on JUnit and it can test Map Reduce programs written on 0.20, 0.23.x, 1.0.x , 2.x version of Hadoop.

With MRUnit, you can craft test input, push it through your mapper and/or reducer, and verify it’s output all in a JUnit test.  As do other JUnit tests, this allows you to debug your code using the JUnit test as a driver.  A map/reduce pair can be tested using MRUnit’s MapReduceDriver.  A combiner can be tested using MapReduceDriver as well.  A PipelineMapReduceDriver allows you to test a workflow of map/reduce jobs.  Currently, partitioners do not have a test driver under MRUnit.  MRUnit allows you to do TDD and write light-weight unit tests which accommodate Hadoop’s specific architecture and constructs.

**How testing is useful in industry :**

Testing could be simply put as "Software Testing", since the field it is involved is IT. Any product/ software created requires testing before it is sold to customers, because Quality matters. Testing a product/ software not only assures that the product is 100% stable/working, but it also ensures the level of Quality.

Software testing is a process of executing a program or application with the intent of finding the software bugs. It can also be stated as the process of validating and verifyingthat a software program or application or product meets the business and technical requirements that guided it’s design and development, Works as expected and Can be implemented with the same characteristic.

Software Testing can be broadly divided into two fields:

Manual testing and Automation testing

* 1. **Manual Testing** is the process of manually testing software for defects. It requires a tester to play the role of an end user and use most of all features of the application to ensure correct behaviour.
  2. **Automation** **Testing** uses special software (separate from the software being tested) to control the execution of tests and the comparison of actual outcomes with predicted outcomes.

Eg: Tools for Automation Testing - Selenium, QTP, etc...

Testing is useful to measure software quality.

* To evaluate the quality of the current product or process
* To improve quality of a product /process by continuous monitoring
* Take decisions based on analysis.

**Mapreduce Task Counters,File system counters,Job Counter**

There are some built-in counters which exist per job. Below are built-in counter groups-

* **MapReduce Task Counters** - Collects task specific information (e.g., number of input records) during its execution time.
* **FileSystem Counters** - Collects information like number of bytes read or written by a task
* **FileInputFormat Counters** - Collects information of number of bytes read through FileInputFormat
* **FileOutputFormat Counters** - Collects information of number of bytes written through FileOutputFormat
* **Job Counters -** These counters are used by JobTracker. Statistics collected by them include e.g., number of task launched for a job.

**Raw comparator VS Writable Comparator :**

A [Writable](https://hadoop.apache.org/docs/r2.6.1/api/org/apache/hadoop/io/Writable.html) which is also [Comparable](http://download.oracle.com/javase/7/docs/api/java/lang/Comparable.html?is-external=true" \o "class or interface in java.lang).WritableComparables can be compared to each other, typically via Comparators. Any type which is to be used as a key in the Hadoop Map-Reduce framework should implement this interface.

|  |  |
| --- | --- |
|  | **Use of RawComparator :**  If you want to optimize time taken by Map Reduce Job, then you have to use RawComparator.  Intermediate key value pairs have been passed from Mapper to Reducer. Before these values reach Reducer from Mapper, shuffle and sorting steps will be performed.  Sorting is improved because the RawComparator will compare the keys by byte. If we did not use RawComparator, the intermediary keys would have to be completely de-serialized to perform a comparison. We do not directly implement RawComparator. Instead we extend WritableComparator, which internally implements RawComparator. |

**Partitioner, Sort comparator, Group comparator :**

**Partitioner :**

A partitioner works like a condition in processing an input dataset. The partition phase takes place after the Map phase and before the Reduce phase.

A partitioner partitions the key-value pairs of intermediate Map-outputs. It partitions the data using a user-defined condition, which works like a hash function. The total number of partitions is same as the number of Reducer tasks for the job. Let us take an example to understand how the partitioner works.

**Sort comparator :**

Say, your key is (Attribute1, Attribute2).Now you could use the Sort Comparator, to first sort by Attribute1 and then by Attribute2.

For example:

Key= (2008, 32) // year, temperature.

Now, if you want to sort by year and then by temperature, you could use the Sort Comparator, as follows:

public static class KeyComparator extends WritableComparator {

protected KeyComparator() {

super(CompositeKey.class, true);

}

@Override

public int compare (WritableComparable w1, WritableComparable w2) {

CompositeKey ip1 = (CompositeKey) w1;

CompositeKey ip2 = (CompositeKey) w2;

int result = CompositeKey.compare(ip1.getYear(), ip2.getYear());

if (result != 0) {

return result;

}

return CompositeKey.compare(ip1.getTemperature(), ip2.getTemperature());

}

}

**Group comparator :**

Once the composites key\value arrive at the reducer instead of the reducer getting

(**a-1**,{1-10})

(**a-2**,{2-20})

the above will happen due to the unique key values following composition.

the group comparator will ensure the reducer gets:

(a,{**1-10,2-20**})

In a single reduce method call.