**ASS-17.6**

**What are the uses of counters**

**1.Counters are used to measure the progress of job.**

**2.It provides statistical Information regarding job progress and is helpful in knowing the completion status of jobs**

For example

File Input Format Counters

Bytes Read=33485841275

File Output Format Counters

Bytes Written=90

When the job is complete, its statistics

(known as counters) are printed out. These are very useful for confirming that the job did what you expected. For example, for this job, we can see that 1.2 billion records were analyzed (“Map input records”), read from around 34 GB of compressed files on HDFS

**3.Custom Counters can be used for Debugging**

For example If we have a temperature dataset and if the value is above 150 C is not possible

So we will create a custom counter of temperature which will count the number of wrong data It can be used as a measure for the accuracy and quality of data

**MR Unit testing is based on**

MR Unit Testing is based on **Junit Testing.**

Junit Testing is an automation tool used for testing.

It works on feeding the test data for the code .It Implements JUnit promotes the idea of "first testing then coding", which emphasizes on setting up the test data for

a piece of code that can be tested first and then implemented. This approach is like "test a little, code a little, test a little, code a little."

**Junit Testing working :**

Step1:we will create a class for test the given class

Step2: use test annotation to test the method of class by giving values

Example

1. **public** **class** Calculation {
3. **public** **static** **int** findMax(**int** arr[]){
4. **int** max=0;
5. **for**(**int** i=1;i<arr.length;i++){
6. **if**(max<arr[i])
7. max=arr[i];
8. }
9. **return** max;
10. }
11. **}**

**Junit Testing class for performing test on find Max Method of calculation**

1. **import** **static** org.junit.Assert.\*;
2. **import** com.javatpoint.logic.\*;
3. **import** org.junit.Test;
5. **public** **class** TestLogic {
7. @Test
8. **public** **void** testFindMax(){
9. assertEquals(4,Calculation.findMax(**new** **int**[]{1,3,4,2}));
10. assertEquals(-1,Calculation.findMax(**new** **int**[]{-12,-1,-3,-4,-2}));
11. }
12. }

Features of Unit Testing

* JUnit is an open source framework, which is used for writing and running tests.
* Provides annotations to identify test methods.
* Provides assertions for testing expected results.
* Provides test runners for running tests.
* JUnit tests allow you to write codes faster, which increases quality.
* JUnit is elegantly simple. It is less complex and takes less time.
* JUnit tests can be run automatically and they check their own results and provide immediate feedback. There's no need to manually comb through a report of test results.
* JUnit tests can be organized into test suites containing test cases and even other test suites.
* JUnit shows test progress in a bar that is green if the test is running smoothly, and it turns red when a test fails.

**MapDriver class**

1. It is a harness that allows to test a Mapper instance.

2. We provide the input (k, v)\* pairs that should be sent to the Mapper, and outputs you expect to be sent by the Mapper to the collector for those inputs.

3. By calling runTest(), the harness will deliver the input to the Mapper and will check its outputs against the expected results. **ReduceDriver class**

• Harness that allows you to test a Reducer instance.

• You provide a key and a set of intermediate values for that key that represent inputs that should be sent to the Reducer (as if they came from a Mapper), and outputs you expect to be sent by the Reducer to the collector.

• By calling runTest(), the harness will deliver the input to the Reducer and will check its outputs against the expected results. 15 MRUnit Testing MapReduceDriver class

• Harness that allows you to test a Mapper and a Reducer instance together (along with an optional combiner).

• You provide the input key and value that should be sent to the Mapper, and outputs you expect to be sent by the Reducer to the collector for those inputs.

• By calling runTest(), the harness will deliver the input to the Mapper, feed the intermediate results to the Reducer (without checking them), and will check the Reducer's outputs against the expected results. • This is designed to handle the (k, v)\* -> (k, v)\* case from the Mapper/Reducer pair, representing a single unit test

. • If a combiner is specified, then it will be run exactly once after the Mapper and before the Reducer

**How testing is useful in industry**

1.Testing is used for Debugging a Product

2.It is used for producing quality product

3.It is used for Testing the application and to monitor the performance of our application

4.It is used for upgrading or enhancing the software to improve the market value, brand name etc;

**Task Counters**

1.Task counters gather information about tasks over the course of their execution, and the results are aggregated over all the tasks in a job.

2.The MAP\_INPUT\_RECORDS counter, for example, counts the input records read by each map task and aggregates over all map tasks in a job, so that the final figure is the total number of input records for the whole job.

Task counters are maintained by each task attempt, and periodically sent to the application master so they can be globally aggregated.

**Job Counters**

1.Job counters are maintained by the application master, so they don’t need to be

sent across the network, unlike all other counters, including user-defined ones. 2.They measure job-level statistics, not values that change while a task is running. Example

**TOTAL\_LAUNCHED\_MAPS counts the number of map tasks that were launched over the course of a job (including tasks that failed).**

**Raw comparator VS Writable Comparator**

**Raw comparator**

This interface permits implementors to compare records read from a stream without deserializing them into objects, thereby avoiding any overhead of object creation.

For example, the comparator for IntWritables implements the raw compare() method by reading an integer from each of the byte arrays b1 and b2 and comparing them directly from the given start positions (s1 and s2) and lengths (l1 and l2).

**Writable Comparator**

WritableComparator is a general-purpose implementation of RawComparator for

WritableComparable classes. It provides two main functions.

First, it provides a default implementation of the raw compare() method that deserializes the objects to be compared from the stream and invokes the object compare() method. Second, it acts as a factory for RawComparator instances (that Writable implementations have registered).

**Partitioner**

It Decides the flow of data into reducer by default the mapreduce framework uses a hash modulo partitioner which uses the hashcode of key for partitioning and instead we can use a custom partitioner and we can send the mapper output to the reducer of our wish

**Sort comparator**

It is the class which is responsible for shuffle and sorting of keys from Mapper .It by default extends the raw comparator class and sort the keys in ascending order.Insteasd we can also use setcomparator method and define a class that implements raw comparator and can be sorted in descending order

**Group comparator**

It is used for grouping the keys in reducer based on which reducer runs

For example If we uses a groupcomparator class and we can make it to group by a part of the key based on which the keys will be recognized by reducer