Problem Statement

Explain the following terms in detail

-- What are the uses of counters?

-- MR Unit testing is based on

-- How testing is useful in industry

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

-- Raw comparator VS Writable Comparator

-- Partitioner, Sort comparator, Group comparator

Q1.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 behaviors of MapReduce programs.

**Two types of counters:**

**1. Hadoop Built-In counters:**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.

**2. User Defined Counters**

In addition to built-in counters, user can define his own counters using similar functionalities provided by programming languages. For example, in [Java](http://www.guru99.com/java-tutorial.html)'enum' are used to define user defined counters.

Q2. MR Unit testing is based on

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.

Q3. How testing is useful in industry

Previously testing had been regarded as an intermediate stage that could be quickly rushed through before handing over a piece of software to the client. Software development agencies were not interested in devoting the resources to complete these processes thoroughly. Testing was simply a tick in the box.

The inevitable result of this policy was software that was incomplete, with functionality in massive variance to the clients’ actual needs, or both. In some cases the software delivered was totally unusable. Around 80-90% of software projects goes over budget or has to be cut short.

A number of high profile, publicly significant IT projects have massively overrun in cost or time, with inevitable damage to both the IT firm, client and development industry.

In some several cases, IT projects have had to be abandoned completely. The inadequacy of the testing regime has led to complete project failure. The consequences of these failures for the industry are huge: consultancies and development houses rely upon trust, reputation and proven project management excellence to win and retain business.

A strong and well-executed testing plan is not the silver bullet solution to a successful project. But few IT projects achieve the required outcome without it. Testing provides evidence whether an IT system will actually function as specified, be efficient to use, operate with minimum errors and demonstrate performance. It will identify whether an IT system is robust or whether it needs further work before it is rolled out. Testing will also reduce development costs by identifying errors early in the development cycle, where they can be corrected with a lower overall impact on the project.

Q4. Mapreduce Task Counters, File system counters, Job Counter

**Hadoop Built-In counters:**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.

Q5. Raw comparator VS Writable Comparator

Implementing the [org.apache.hadoop.io.RawComparator](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/RawComparator.html" \t "_blank) interface will definitely help speed up your Map/Reduce (MR) Jobs. As you may recall, a MR Job is composed of receiving and sending key-value pairs. The process looks like the following.

* (K1,V1) –> Map –> (K2,V2)
* (K2,List[V2]) –> Reduce –> (K3,V3)

The key-value pairs (K2,V2) are called the intermediary key-value pairs. They are passed from the mapper to the reducer. Before these intermediary key-value pairs reach the reducer, a shuffle and sort step is performed. The shuffle is the assignment of the intermediary keys (K2) to reducers and the sort is the sorting of these keys. In this blog, by implementing the RawComparator to compare the intermediary keys, this extra effort will greatly improve sorting. 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 deserialized to perform a comparison.

Below is a snippet of the RawComparator. As you notice, it does not directly implement RawComparator. Rather, it extends WritableComparator (which implements RawComparator). We could have directly implemented RawComparator, but by extending WritableComparator, depending on the complexity of our intermediary key, we may use some of the utility methods of WritableComparator.

|  |  |
| --- | --- |
| 1  1 | public class IndexPairComparator extends WritableComparator {      protected IndexPairComparator() {          super(IndexPair.class);      }        @Override      public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {          int i1 = readInt(b1, s1);          int i2 = readInt(b2, s2);            int comp = (i1 < i2) ? -1 : (i1 == i2) ? 0 : 1;          if(0 != comp)              return comp;            int j1 = readInt(b1, s1+4);          int j2 = readInt(b2, s2+4);          comp = (j1 < j2) ? -1 : (j1 == j2) ? 0 : 1;            return comp;      }  }   implementing raw byte comparison of intermediary keys will improve the speed of your MR Jobs as opposed to relying on a comparison based on deserialized objects |

Q6. Partitioner, Sort comparator, Group comparator

Partitioner:

Partitioner controls the partitioning of the keys of the intermediate map-outputs. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the same as the number of reduce tasks for the job. Hence this controls which of the m reduce tasks the intermediate key (and hence the record) is sent for reduction.

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

**GroupComparator**: 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.]]**

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |