1. 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.

Typically some of the operations of Hadoop counters are:

1. Number of mapper and reducer launched.
2. The number of bytes was read and written
3. The number of tasks was launched and successfully ran
4. The amount of CPU and memory consumed is appropriate or not for your job and cluster nodes

By default MapReduce provides us with many built-in counters to track all this details, and also provides us the freedom to create our own counters. In the case if we want to have track any kind of statistics about the records written as logic in mapper and reducers. Then custom counters come into the picture.

Another use of custom counters is in the debugging process – where it can also be used to determine the number of BAD records.

2. MR Unit testing is based on

MRUnit is testing framework which provides support structure to test map reduce jobs. It provides mocking support which can be helpful in testing Mapper, Reducer, Mapper+Reducer and Driver.

3. How testing is useful in industry

1. Validation of Structured and Unstructured Data: Data needs to be classified as the structured and unstructured parts.

(i) Structured Data: It is the data which can be stored in the form of tables (rows and columns) without any processing for example database, call details and excel sheets.

(ii) Unstructured Data: It is the data which does not have a predefined data model or structure for example data in the form of weblogs, audio, tweets, and comments.

Adequate time needs to be spent over the validation of the data at an initial stage, and it is the point where we encounter an abundance of bad data from various sources.

2. Ace Test Environment: Efficient test environment ensures that data from multiple sources is of acceptable quality for accurate analysis. Although replicating the complete set of big data into the test environment is next to impossible, so a small subset of the data is created for the test environment to verify the behavior. Careful planning is required to exercise all paths with subsets of data in a manner that fully verifies the application.

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

1.Defining Mapreduce Task Counters

Task counters gather information about tasks over the course of their execution, and the results are aggregated over all the tasks in a job. For example, the MAP\_INPUT\_RECORDS counter 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.

2.Defining File system counters

File system counters track 2 main details , number of bytes read by the file system and number of bytes written.

BYTES\_READ counter is tracked by File Input Format

Bytes read (BYTES\_READ) :The number of bytes read by map tasks via the FileInputFormat.

BYTES\_WRITTEN counter is tracked by File Output Format

Bytes written (BYTES\_WRITTEN) :The number of bytes written by map tasks (for map-only jobs) or reduce tasks via the FileOutputFormat.

3.Defining Job Counters

Job counters are maintained by the jobtracker (or application master in YARN), so they don’t need to be sent across the network, unlike all other counters, including user-defined ones. They measure job-level statistics, not values that change while a task is running. For example, TOTAL\_LAUNCHED\_MAPS counts the number of map tasks that were launched over the course of a job (including ones that failed).

1. Raw comparator VS Writable Comparator

Raw comparator:

If you still 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.

Writable Comparator:

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.

WritableComparable interface is just a subinterface of the Writable and java.lang.Comparable interfaces. For implementing a WritableComparable we must have compareTo method apart from readFields and write methods, as shown below:

public interface WritableComparable extends Writable, Comparable

{

void readFields(DataInput in);

void write(DataOutput out);

int compareTo(WritableComparable o)

}

1. 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. The number of partitioners is equal to the number of reducers. That means a partitioner will divide the data according to the number of reducers. Therefore, the data passed from a single partitioner is processed by a single Reducer.

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 partitioner task accepts the key-value pairs from the map task as its input. Partition implies dividing the data into segments. According to the given conditional criteria of partitions, the input key-value paired data can be divided

Sort comparator:

Sort order for keys is found as follows:

If the property mapred.output.key.comparator.class is set, either explicitly or by calling setSortComparatorClass() on Job, then an instance of that class is used. (In the old API the equivalent method is setOutputKeyComparatorClass() on JobConf.)

Otherwise, keys must be a subclass of WritableComparable, and the registered comparator for the key class is used.

If there is no registered comparator, then a RawComparator is used that deserializes the byte streams being compared into objects and delegates to the WritableComparable’s compareTo() method.

A secondary sort problem relates to sorting values associated with a key in the reduce phase. Sometimes, it is called value-to-key conversion. The secondary sorting technique will enable us to sort the values (in ascending or descending order) passed to each reducer.

Group comparator:

**Group Comparator** decides which map output keys **will be united(grouped) into one key,** and of course all collections of values will be grouped too. Usually it takes a first key as the only one for summary collection.