**Assignment 17.6**

**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.

**2. MR Unit testing is based on**

There are three main MRUnit classes that drive the tests: MapDriver, ReduceDriver and MapReduceDriver. They are generic classes whose type parameters depend on input and output types of mapper, reducer and whole map/reduce job, respectively.

* MapDriver class

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

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.

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.

* MapReduceDriver class

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

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.

**3.How testing is useful in industry**

Testing is important because software bugs could be expensive or even dangerous. Software bugs can potentially cause monetary and human loss, history is full of such examples.

* In April 2015, Bloomberg terminal in London crashed due to software glitch affected more than 300,000 traders on financial markets. It forced the government to postpone a 3bn pound debt sale.
* Nissan cars have to recall over 1 million cars from the market due to software failure in the airbag sensory detectors. There has been reported two accident due to this software failure.
* Starbucks was forced to close about 60 percent of stores in the U.S and Canada due to software failure in its POS system. At one point store served coffee for free as they unable to process the transaction.
* Some of the Amazon’s third party retailers saw their product price is reduced to 1p due to a software glitch. They were left with heavy losses.

In the today’s competitive market only the quality product stays longtime firmly, so to make sure the produce the good quality product the testing of application is key factor in SDLC.

As it not possible makes it software application is defect free but testing will be necessary.

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

Hadoop maintains some built-in counters for every job, which report various metrics for your job. For example, there are counters for the number of bytes and records processed, which allows us to confirm that the expected amount of input was consumed and the expected amount of output was produced.

Built in counters are of three types:

* **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.

* **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.

* **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).

**5.Raw comparator VS Writable Comparator**

Two ways you may compare your keys is by implementing the [org.apache.hadoop.io.WritableComparable](http://hadoop.apache.org/common/docs/current/api/org/apache/hadoop/io/WritableComparable.html" \t "_blank) interface or by implementing the RawComparator interface. In the former approach, you will compare (deserialized) objects, but in the latter approach, you will compare the keys using their corresponding raw bytes.

For the two objects we are comparing, there are two corresponding byte arrays, the starting positions of the objects in the byte arrays, and the length of the bytes they occupy. Please note that the byte arrays themselves represent other things and not only the objects we are comparing.

**Partitioner, Sort comparator, Group comparator**

**Partitioner:**

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.

**Sort Comparator:**

Sort order for keys is found as follows:

1. 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.)
2. Otherwise, keys must be a subclass of WritableComparable, and the registered comparator for the key class is used.
3. 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.

**Group comparator:**

|  |  |
| --- | --- |
|  | **Input**:  symbol time price  a 1 10  a 2 20  b 3 30  **Map output**: create composite key\values like so:  symbol-time time-price  **a-1**         1-10  **a-2**         2-20  **b-3**         3-30  The Partitioner will route the a-1 and a-2 keys to the same reducer despite the keys being different. It will also route the b-3 to a separate reducer.  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 group comparator will ensure the reducer gets:  (a,{**1-10,2-20**}) |