```
(look,1)
```

The commands for checking output in **part-00001** file are:

```
$ cat part-00001
(walk, 1)
(or, 1)
(talk, 1)
(only, 1)
(love, 1)
(care, 1)
(share, 1)
```

Go through the following section to know more about the 'spark-submit' command.

Spark-submit Syntax

```
spark-submit [options] <app jar | python file> [app arguments]
```

Options

The table given below describes a list of **options**:-

S.No	Option	Description
1	--master	spark://host:port, mesos://host:port, yarn, or local.
2	--deploy-mode	Whether to launch the driver program locally ("client") or on one of the worker machines inside the cluster ("cluster") (Default: client).
3	--class	Your application's main class (for Java / Scala apps).
4	--name	A name of your application.
5	--jars	Comma-separated list of local jars to include on the driver and executor classpaths.
6	--packages	Comma-separated list of maven coordinates of jars to include on the driver and executor classpaths.
7	--repositories	Comma-separated list of additional remote repositories to search for the maven coordinates given with --packages.

8	--py-files	Comma-separated list of .zip, .egg, or .py files to place on the PYTHON PATH for Python apps.
9	--files	Comma-separated list of files to be placed in the working directory of each executor.
10	--conf (prop=val)	Arbitrary Spark configuration property.
11	--properties-file	Path to a file from which to load extra properties. If not specified, this will look for conf/spark-defaults.
12	--driver-memory	Memory for driver (e.g. 1000M, 2G) (Default: 512M).
13	--driver-java-options	Extra Java options to pass to the driver.
14	--driver-library-path	Extra library path entries to pass to the driver.
15	--driver-class-path	Extra class path entries to pass to the driver. Note that jars added with --jars are automatically included in the classpath.
16	--executor-memory	Memory per executor (e.g. 1000M, 2G) (Default: 1G).
17	--proxy-user	User to impersonate when submitting the application.
18	--help, -h	Show this help message and exit.
19	--verbose, -v	Print additional debug output.
20	--version	Print the version of current Spark.
21	--driver-cores NUM	Cores for driver (Default: 1).
22	--supervise	If given, restarts the driver on failure.
23	--kill	If given, kills the driver specified.
24	--status	If given, requests the status of the driver specified.
25	--total-executor-cores	Total cores for all executors.
26	--executor-cores	Number of cores per executor. (Default: 1 in YARN mode, or all available cores on the worker in standalone mode).

6. ADVANCED SPARK PROGRAMMING

Spark contains two different types of shared variables- one is **broadcast variables** and second is **accumulators**.

- **Broadcast variables:** used to efficiently, distribute large values.
- **Accumulators:** used to aggregate the information of particular collection.

Broadcast Variables

Broadcast variables allow the programmer to keep a read-only variable cached on each machine rather than shipping a copy of it with tasks. They can be used, for example, to give every node, a copy of a large input dataset, in an efficient manner. Spark also attempts to distribute broadcast variables using efficient broadcast algorithms to reduce communication cost.

Spark actions are executed through a set of stages, separated by distributed "shuffle" operations. Spark automatically broadcasts the common data needed by tasks within each stage.

The data broadcasted this way is cached in serialized form and is deserialized before running each task. This means that explicitly creating broadcast variables, is only useful when tasks across multiple stages need the same data or when caching the data in deserialized form is important.

Broadcast variables are created from a variable **v** by calling **SparkContext.broadcast(v)**. The broadcast variable is a wrapper around **v**, and its value can be accessed by calling the **value** method. The code given below shows this:

```
scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))
```

Output:

```
broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0)
```

After the broadcast variable is created, it should be used instead of the value **v** in any functions run on the cluster, so that **v** is not shipped to the nodes more than once. In addition, the object **v** should not be modified after its broadcast, in order to ensure that all nodes get the same value of the broadcast variable.

Accumulators

Accumulators are variables that are only "added" to through an associative operation and can therefore, be efficiently supported in parallel. They can be used to implement counters (as in MapReduce) or sums. Spark natively supports accumulators of numeric types, and programmers can add support for new types. If accumulators are created with a name, they will be displayed in **Spark's UI**. This can be useful for understanding the progress of running stages (NOTE: this is not yet supported in Python).

An accumulator is created from an initial value **v** by calling **SparkContext.accumulator(v)**. Tasks running on the cluster can then add to it using the **add** method or the **+=** operator (in Scala and Python). However, they cannot read its value. Only the driver program can read the accumulator's value, using its **value** method.

The code given below shows an accumulator being used to add up the elements of an array:

```
scala> val accum = sc.accumulator(0)


scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum += x)
```

If you want to see the output of above code then use the following command:

```
scala> accum.value
```

Output

```
res2: Int = 10
```

Numeric RDD Operations

Spark allows you to do different operations on numeric data, using one of the predefined API methods. Spark's numeric operations are implemented with a streaming algorithm that allows building the model, one element at a time.

These operations are computed and returned as a **StatusCounter** object by calling **status()** method.

The following is a list of numeric methods available in **StatusCounter**.

S.No	Method & Meaning
1	**count()** Number of elements in the RDD.
2	**Mean()** Average of the elements in the RDD.
3	**Sum()** Total value of the elements in the RDD.
4	**Max()** Maximum value among all elements in the RDD.

5	**Min()** Minimum value among all elements in the RDD.
6	**Variance()** Variance of the elements.
7	**Stdev()** Standard deviation.

If you want to use only one of these methods, you can call the corresponding method directly on RDD.