

Distributed Systems

Cloud Computing Assignment

Mohammad Mahdi Islami
810195548

سوال اول:

تعریفی راجع به Hadoop و Spark بنویسید و تفاوت های آن دو را بیان نمایید. سپس برای انجام سوالات بعدی آن ها را بر روی Ubuntu نصب نمایید.

Hadoop

Hadoop یک فریم ورک پردازش توزیع شده open-source است که پردازش و ذخیره سازی داده ها را برای برنامه های Big Data در خوشه های مقیاس پذیر سرورهای رایانه مدیریت می کند. این مرکز اکوسیستم فناوری های کلان داده است که اساساً برای پشتیبانی از اقدامات تحلیلی پیشرفته از جمله پیش بینی ، داده کاوی و یادگیری ماشین استفاده می شود. سیستم های Hadoop می توانند اشکال مختلف داده های ساختاریافته و غیر ساختاری را مدیریت کنند و انعطاف پذیری بیشتری نسبت به پایگاه داده های رابطه ای و انبارهای داده برای جمع آوری ، پردازش ، تجزیه و تحلیل و مدیریت داده ها به کاربران می دهند.

Hadoop به عنوان یک پروژه یاهو در سال ۲۰۰۶ شروع به کار کرد و بعداً به یک پروژه open-source آپاچی تبدیل شد. هدوپ در واقع یک فرم کلی از پردازش توزیع شده است که چندین مولفه دارد:

- Hadoop Distributed File System (HDFS)

این بخش فایلها را در قالب Hadoop بومی ذخیره می کند و آنها را در یک خوشه، موازی می کند.

- YARN

برنامه ای است که زمان اجرای برنامه را هماهنگ می کند.

- MapReduce

الگوریتمی که در واقع داده ها را به طور موازی پردازش می کند.

Hadoop بر پایه ی جاوا شکل گرفته است و از طریق بسیاری از زبان های برنامه نویسی برای نوشتن کد MapReduce ، از جمله پایتون و Thrift قابل دسترسی است.

علاوه بر این بخش های پایه، Hadoop همچنین شامل:

Sqoop است که داده های رابطه ای را به HDFS منتقل می کند. Hive ، یک رابط مانند SQL است که به کاربران اجازه می دهد تا از طریق HDFS نمایش داده شوند و از Mahout، برای یادگیری ماشین استفاده می شود.

Spark

Spark پروژه جدیدتری است که در ابتدا در سال ۲۰۱۲ در AMPLab در UC Berkeley توسعه یافت. یک پروژه سطح بالا Apache است که برای پردازش داده ها به صورت موازی در یک خوشه متمرکز استفاده می شود، اما بزرگترین تفاوت این است که در حافظه کار می کند.

در حالی که Hadoop فایل ها را از HDFS می خواند و می نویسد ، Spark با استفاده از مفهومی معروف به RDD (مجموعه داده های توزیع شده قابل انعطاف)، داده ها را در RAM پردازش می کند. Spark می تواند در حالت مستقل ، با خوشه Hadoop به عنوان منبع اجرا شود.

Spark در Spark Core ، موتوری که برنامه ریزی ها، بهینه سازی ها و عملکرد RDD را هدایت می کند، ساخته شده و همچنین Spark را به HDFS، متصل می کند. چندین کتابخانه وجود دارد که در Spark Core استفاده می شوند، از جمله:

- Spark SQL، که امکان اجرای دستورات SQL-مانند را روی مجموعه داده های توزیع شده فراهم می کند.
- MLlib برای یادگیری ماشین.
- GraphX برای مسایل تحلیل گراف و نمودار.

Spark چندین API دارد. رابط اصلی به زبان Scala نوشته شده است و بر اساس استفاده زیاد توسط دانشمندان داده ، رابط Python و R نیز اضافه شده است. جاوا گزینه دیگری برای کد زدن در Spark است.

Databricks، اکنون بر توسعه Spark نظارت می کند و Spark را به مشتریان ارائه می دهد.

سوال دوم:

ابتدا HDFS را تعریف نمایید؟ اگر بخواهیم از پوشه روت hdfs یک ls بگیریم از چه دستوری استفاده می‌نماییم؟

```
$hadoop fs -ls /
```

سوال سوم:

دستورات لازم برای ساخت یک پوشه با نام myfolder در hdfs و لود کردن فایل ۱ mygraph در آن پوشه را بنویسید.

```
$hdfs dfs -mkdir /user
```

با اجرای این دستور یک پوشه به نام user در فضای hdfs ساخته می‌شود.

پ.ن: چون طبق دستورالعمل Quick Start عمل کردم نام پوشه را به user تغییر دادم.

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/mojtaba/hadoop-3.2.0/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/mojtaba/hadoop-3.2.0/share/hadoop/yarn/lib/giraph-examples-1.2.0-for-hadoop-3.2.0-jar-with-dependencies.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

```
$hdfs dfs -copyFromLocal mygraph.txt /user/mygraph.txt
```

این دستور آدرس مستقیم فایل mygraph.txt را به عنوان پارامتر اول و آدرس مقصد در hdfs را می‌گیرد و آن را به hdfs منتقل می‌کند.
دستورات دیگری مانند استفاده از put- نیز وجود دارد.

```
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/home/mojtaba/hadoop-3.2.0/share/hadoop/common/lib/slf4j-log4j12-1.7.25.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/home/mojtaba/hadoop-3.2.0/share/hadoop/yarn/lib/giraph-examples-1.2.0-for-hadoop-3.2.0-jar-with-dependencies.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [org.slf4j.impl.Log4jLoggerFactory]
```

سوال چهارم:

در Hadoop با استفاده از زبان java و تکنیک MapReduce برنامه‌ای بنویسید که تمام گره‌هایی که تفاضل مجموع وزن‌های یال‌های ورودی از مجموع وزن‌های یال‌های خروجی، فرد می‌باشد را گزارش نماید. (پیشنهاد می‌گردد ابتدا مثال word count را برای Hadoop مطالعه فرمایید.)

کلیه فایل‌های مورد استفاده برای هدوپ شامل کد، خروجی، و کلاس‌ها در آدرس زیر هستند:

810195548/Hadoop-Java/

در این قسمت گزارش، مراحل اجرای برنامه ی دلخواه جاوا با تکنیک Map-Reduce در آپاچی هدوپ توضیح داده شده است.

در این قسمت یک برنامه به زبان جاوا با فریم‌ورکی که کتابخانه‌های هدوپ در اختیار ما می‌گذارند نوشته شده است که از طریق لینک زیر قابل مشاهده است.

Task3.java

در این کد در کلاس EdgeWeightMapper که از کلاس اصلی Mapper ارث می‌برد نودهای مبدا و مقصد و وزن‌های آن‌ها جدا می‌شوند وزن برای یال خروجی منفی و برای یال ورودی با علامت مثبت لحاظ می‌شود.

```
public static class EdgeWeightMapper
    extends Mapper<Object, Text, Text, IntWritable>{

    private Text word = new Text();
    private Text word2 = new Text();

    public void map(Object key, Text value, Context context
        ) throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());
        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            word2.set(itr.nextToken());
            int weight = Integer.parseInt(itr.nextToken());
            context.write(word, new IntWritable(-weight));
            context.write(word2, new IntWritable(weight));
        }
    }
}
```

در کلاس EdgeWeightReducer که از کلاس اصلی Reducer خود هدوپ ارث‌بری می‌کند وزن‌هایشان جمع شده و در نهایت شرط اصلی یعنی $sum \% 2 \neq 0$ چک میشود.

```
public static class EdgeWeightReducer
    extends Reducer<Text,IntWritable,Text,IntWritable> {
    private IntWritable result = new IntWritable();

    public void reduce(Text key, Iterable<IntWritable> values,
        Context context
    ) throws IOException, InterruptedException {
        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        if ( sum % 2 != 0 ) {
            result.set(sum);
            context.write(key, result);
        }
    }
}
```

هدوپ با تکنیک map-reduce یک فایل خروجی برای برنامه‌ی ما تولید کرده که شامل 4988 خط یا نود است که دو ستون دارد که با Tab از هم جدا شده‌اند. ستون اول یک راس از گراف و ستون تفاضل مجموع وزن یالهای ورودی از وزن یالهای خروجی هر راسی است که این مقدار برایش فرد است.

```
09:17,033 INFO mapreduce.Job: map 0% reduce 0%
10:04,044 INFO mapreduce.Job: map 67% reduce 0%
10:09,588 INFO mapreduce.Job: map 100% reduce 0%
10:46,766 INFO mapreduce.Job: map 100% reduce 100%
10:46,786 INFO mapreduce.Job: Job job_1607969913540_0001 comple
```

خروجی از لینک زیر قابل مشاهده است:

[Part-r-00000](#)

سوال پنجم:

در spark با استفاده از زبان Scala سوال فوق را تکرار نمایید.
(پیشنهاد میگردد ابتدا مثال word count را برای Spark مطالعه فرمایید)

کلیه فایل های مورد استفاده برای اسپارک شامل کد، خروجی، و کلاس ها در آدرس زیر هستند:

810195548/Spark-Scala/

در این قسمت گزارش، مراحل اجرای برنامه ی دلخواه اسکالا در اسپارک توضیح داده شده است.

پس از مطالعه برنامه ی Word Count در Quick Start اسپارک برنامه ای به زبان اسکالا نوشتیم که ابتدا فایل را از hdfs را به عنوان آرگومان دریافت میکند سپس یک متغیر ایجاد میکنیم که فایل ورودی که شامل سه ستون است را ابتدا بدین صورت جدا مپ کند:

Source -weight

و یک متغیر دیگر که بدین صورت مپ کند:

Target weight

فایل نوشته شده از طریق لینک زیر قابل مشاهده است:

Task2.scala

سپس این دو مپ را به یکدیگر الحاق میکنیم و با دستور ReduceByKey نود های تکرار شده را یکی میکنیم.
سپس مقادیر weight را جمع میکنیم بدین صورت تفاضل مجموع یالهای ورودی از خروجی به دست می آید.

```
1  import org.apache.spark.SparkContext
2  import org.apache.spark.SparkContext._
3  import org.apache.spark.SparkConf
4
5  object Task2 {
6    def main(args: Array[String]) {
7      val sc = new SparkContext(new SparkConf().setAppName("Task2"))
8      val file = sc.textFile("hdfs://localhost:9000" + args(0))
9      val tgt = file.map(line=> line.split("\t") match { case Array(x,y,z) => (y.toInt,z.toInt) } )
10     val source = file.map(line=> line.split("\t") match { case Array(x,y,z) => (x.toInt,-z.toInt) } )
11     val output = tgt.++(source)
12     output.cache()
13     val edge = output.reduceByKey((a, b) => a + b).filter(_._2 % 2 != 0).map(x=>x._1+"\t"+x._2)
14     edge.collect()
15     edge.saveAsTextFile("hdfs://localhost:9000" + args(1))
16   }
17 }
18
```

اسپارک ۴ فایل خروجی برای ما تولید میکند که شامل نود های که در شرط فرد بود تفاضل صدق میکنند در ستون اول و مقدار این تفاضل در ستون دوم است که از طریق لینک های زیر قابل مشاهده هستند.

[Part-r-00000](#)

[Part-r-00001](#)

[Part-r-00002](#)

[Part-r-00003](#)

سوال ششم:

چه تفاوت‌هایی میان نتایج سوال‌های چهارم و پنجم ملاحظه می‌نمایید؟ یافته‌های خود را شرح دهید .

■ از نظر زمان باید گفت که اسپارک سریع تر از هِدوپ عمل کرد و خروجی را سریعتر به ما تحویل می‌دهد.

■ اسپارک ۴ فایل خروجی برای ما تولید کرده نشان می‌دهد به صورت توزیع شده این پردازش را برای ما انجام داده و نتایج این پردازش‌های موازی به صورت ۴ فایل جداگانه است. تعداد نود هایی که هر فایل محاسبه کرده (تعداد خط خروجی) برابر است با :

$$1247 + 1265 + 1222 + 1254$$

که برابر با ۴۹۸۸ خط است.

■ هِدوپ یک فایل برای ما ایجاد کرده است البته ما هِدوپ را به صورت localhost اجرا کردیم و در فایل worker تنها localhost قرار دارد بنابراین همه ی پردازش را خود سیستم انجام می‌دهد و و بنابراین یک فایل خروجی در نهایت داریم. فایل نهایی هم ۴۹۸۸ خط است که با مقادیر اسپارک مطابقت دارد.

■ در جواب ها تغییری وجود ندارد و هر دو تعداد برابری نود را شناسایی کرده اند و مقداری هم که برای تفاضل حساب کرده اند یکسان است برای مثال داریم برای دو نود 6480 و ۴۶۶۷ در نتایج هِدوپ و اسپارک داریم:

4665	555	931	265
4666	-805	755	-255
4667	-749	6827	37
4668	731	4667	-749
4670	-515	8727	-435
4671	-965	9647	1493
4676	-759	3139	699
4678	220	8842	812

280	557	6477	1007
5176	81	6480	551
6480	551	6485	181
4480	-1081	6493	93
1004	2717	650	749

که مقادیر تفاضل محاسبه شده برای هر دو برابر و فرد است و نتایج درست محاسبه شده است.

مراحل اجرای برنامه ی دلخواه جاوا با تکنیک Map-Reduce در آپاچی هدوپ

ابتدا برای اجرای صحیح هدوپ باید فایل های زیر را مطابق تنظیمات سیستم خود مثل آدرس جاواهوم و ... کانفیگ کنیم:

`mapred-site.xml` `yarn-site.xml` `Hadoop.env` `core-site.xml`

برای راه اندازی هدوپ ابتدا لازم است نودهای هدوپ را با قرار گرفتن در پوشه ی `sbin` دایرکتوری هدوپ با اجرای دستور `./start-all.sh` راه اندازی کنیم. سپس با دستور `jps` از صحت اجرای نودها اطمینان حاصل میکنیم.

سپس کدی که به زبان جاوا و با تکنیک Map Reduce نوشتیم را با `classpath` هدوپ که شامل کتابخانه های آن است کامپایل کرده و سپس کلاس های آن را به فایل با فرمت JAR تبدیل کنیم. بنابراین با اجرای دستور زیر این مراحل را انجام می دهیم:

`$javac Task3.java -cp $(hadoop classpath)`

```
Task3.class          'Task3$EdgeWeightReducer.class'
'Task3$EdgeWeightMapper.class'  _Task3.java
```

سپس با دستور زیر کلاس ها را به جار تبدیل میکنیم:

`$jar cvfe Task3.jar Task3 Task3.class /`
`Task3\Task3$EdgeWeightReducer.class /`
`Task3\Task3$EdgeWeightMapper.class`

```
Task3.class          'Task3$EdgeWeightReducer.class'  Task3.java
Task3$EdgeWeightMapper.class'  Task3.jar
```

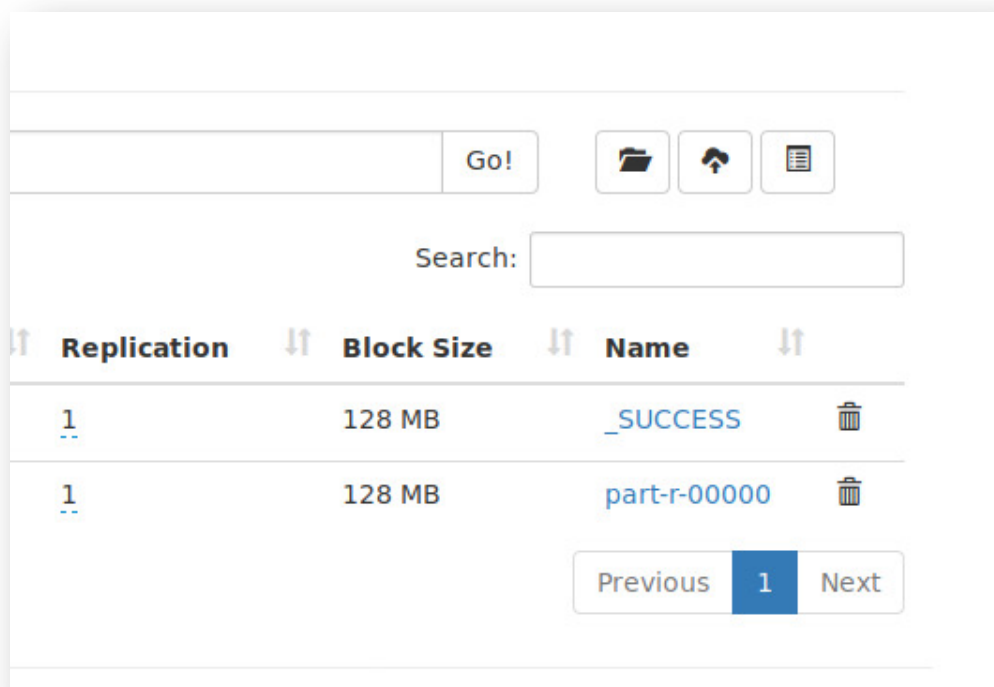
\$hadoop jar Task3.jar /user/mygraph.txt /user/out.txt

سپس با دادن آدرس فایل جار Task3.jar ، فایل ورودی mygraph.txt و فایلی که خروجی را می‌خواهیم برای ما ایجاد کند را به این دستور می‌دهیم تا کد ما را اجرا کند. سپس خروجی مانند شکل زیر در ترمینال چاپ میشود که نشان میدهد عملیات Map و Reduce مرحله به مرحله کامل شده‌اند.



```
09:17,033 INFO mapreduce.Job: map 0% reduce 0%
10:04,044 INFO mapreduce.Job: map 67% reduce 0%
10:09,588 INFO mapreduce.Job: map 100% reduce 0%
10:46,766 INFO mapreduce.Job: map 100% reduce 100%
10:46,786 INFO mapreduce.Job: Job job_1607969913540_0001 comple
```

```
-12-14 22:10:46,786 INFO mapreduce.Job: Job job_1607969913540_0001 completed successfully
-12-14 22:10:47,030 INFO mapreduce.Job: Counters: 54
  File System Counters
    FILE: Number of bytes read=27335
    FILE: Number of bytes written=499417
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=13699213
    HDFS: Number of bytes written=22632
    HDFS: Number of read operations=8
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=2
    HDFS: Number of bytes read erasure-coded=0
  Job Counters
    Launched map tasks=1
    Launched reduce tasks=1
    Data-local map tasks=1
    Total time spent by all maps in occupied slots (ms)=47079
    Total time spent by all reduces in occupied slots (ms)=33052
    Total time spent by all map tasks (ms)=47079
    Total time spent by all reduce tasks (ms)=33052
    Total vcore-milliseconds taken by all map tasks=47079
    Total vcore-milliseconds taken by all reduce tasks=33052
    Total megabyte-milliseconds taken by all map tasks=48208896
    Total megabyte-milliseconds taken by all reduce tasks=33845248
  Map-Reduce Framework
    Map input records=1000000
    Map output records=2000000
    Map output bytes=17778888
    Map output materialized bytes=27335
    Input split bytes=103
    Combine input records=2000000
    Combine output records=2511
    Reduce input groups=2511
    Reduce shuffle bytes=27335
    Reduce input records=2511
```

در اینجا کد ما بدون دادن خطا توانسته است اجرا شود. حال به Web UI هدوپ به آدرس localhost:9870 میرویم تا خروجی را در hdfs مشاهده کنیم.



The screenshot shows the Hadoop Web UI interface. At the top, there is a search bar with a 'Go!' button and three icons: a folder, a refresh, and a list. Below the search bar is a table with columns: Replication, Block Size, Name, and an action icon. The table contains two rows. The first row shows a replication of 1, a block size of 128 MB, and a name of _SUCCESS. The second row shows a replication of 1, a block size of 128 MB, and a name of part-r-00000. At the bottom of the table, there are buttons for 'Previous', '1' (selected), and 'Next'.

Replication	Block Size	Name	
1	128 MB	_SUCCESS	
1	128 MB	part-r-00000	

Previous 1 Next

در اینجا یعنی همان آدرسی که در دستور اجرای برنامه به عنوان محل ذخیره خروجی دادیم، خروجی تحت عنوان part-r-00000 تولید میشود که شامل جواب های ماست و از لینک زیر قابل مشاهده است.

[Part-r-00000](#)

هدوپ یک فایل خروجی برای برنامه ی ما تولید کرده است که شامل 4988 خط یا نود است که تفاضل مجموع وزن یالهای ورودی از وزن یالهای خروجی اشان فرد است.

مراحل اجرای برنامه ی دلخواه اسکالا در اسپارک

در اینجا پس از دانلود یک mirror مناسب (من spark 3.0.1 با scala 2.12 استفاده کردم) کدی که به زبان اسکالا نوشتیم را با استفاده از ابزار sbt خود اسکالا کامپایل میکنیم تا فایل JAR مورد استفاده توسط اسپارک را تولید کند.

پوشه ای که برای کامپایل برنامه اسکالای خود با دستور sbt package استفاده کنیم باید ساختاری بدین صورت داشته باشد:

```
./build.sbt
./src
./src/main
./src/main/scala
./src/main/scala/Task2.scala
```

```
[info] welcome to sbt 1.4.4 (Private Build Java 1.8.0_222)
[info] loading project definition from /home/mojtaba/scala/Task2/project
[info] loading settings for project task2 from build.sbt ...
[info] set current project to Task2 (in build file:/home/mojtaba/scala/Task2/)
[info] compiling 1 Scala source to /home/mojtaba/scala/Task2/target/scala-2.12/classes ...
[success] Total time: 48 s, completed Dec 14, 2020 11:14:05 PM
```





ابتدا به پوشه ی spark/bin رفته سپس فایلی که در hdfs هدوپ آپلود شده است و فایل JAR ای که ساخته ایم را با استفاده از دستور زیر اجرا میکنیم و خروجی را نیز در hdfs ذخیره می کنیم.

```
./bin/spark-submit --class "Task2" target/scala-2.12/task2_2.12-1.0.jar /user/mygraph.txt /user/out1
```

سپس برنامه اجرا می‌شود و خروجی در آدرسی که داده شده در hdfs ساخته می‌شود.

```
2020-12-14 23:17:19,804 INFO mapred.SparkHadoopMapRedUtil: attempt_20201214231710_0008_m_000001_0: Committed
2020-12-14 23:17:19,806 INFO executor.Executor: Finished task 1.0 in stage 3.0 (TID 9). 1588 bytes result sent to driver
2020-12-14 23:17:19,807 INFO scheduler.TaskSetManager: Finished task 1.0 in stage 3.0 (TID 9) in 6151 ms on 192.168.43.39 (executor driver) (4/4)
2020-12-14 23:17:19,808 INFO scheduler.TaskSchedulerImpl: Removed TaskSet 3.0, whose tasks have all completed, from pool
2020-12-14 23:17:19,809 INFO scheduler.DAGScheduler: ResultStage 3 (runJob at SparkHadoopWriter.scala:78) finished in 6.193 s
2020-12-14 23:17:19,811 INFO scheduler.DAGScheduler: Job 1 is finished. Cancelling potential speculative or zombie tasks for this job
2020-12-14 23:17:19,812 INFO scheduler.TaskSchedulerImpl: Killing all running tasks in stage 3: Stage finished
2020-12-14 23:17:19,812 INFO scheduler.DAGScheduler: Job 1 finished: runJob at SparkHadoopWriter.scala:78, took 6.210504 s
2020-12-14 23:17:20,052 INFO io.SparkHadoopWriter: Job job_20201214231710_0008 committed.
2020-12-14 23:17:20,060 INFO spark.SparkContext: Invoking stop() from shutdown hook
2020-12-14 23:17:20,070 INFO server.AbstractConnector: Stopped Spark@50305a(HTTP/1.1,[http/1.1]){0.0.0.0:4040}
2020-12-14 23:17:20,072 INFO ui.SparkUI: Stopped Spark web UI at http://192.168.43.39:4040
2020-12-14 23:17:20,097 INFO spark.MapOutputTrackerMasterEndpoint: MapOutputTrackerMasterEndpoint stopped!
2020-12-14 23:17:20,235 INFO memory.MemoryStore: MemoryStore cleared
2020-12-14 23:17:20,236 INFO storage.BlockManager: BlockManager stopped
2020-12-14 23:17:20,246 INFO storage.BlockManagerMaster: BlockManagerMaster stopped
2020-12-14 23:17:20,250 INFO scheduler.OutputCommitCoordinator$OutputCommitCoordinatorEndpoint: OutputCommitCoordinator stopped!
2020-12-14 23:17:20,258 INFO spark.SparkContext: Successfully stopped SparkContext
2020-12-14 23:17:20,258 INFO util.ShutdownHookManager: Shutdown hook called
2020-12-14 23:17:20,259 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-1502a703-6498-402f-91ed-88d0b9807de3
2020-12-14 23:17:20,261 INFO util.ShutdownHookManager: Deleting directory /tmp/spark-5229f757-801e-4c01-a5eb-b45f52a13fdb
```

در Web Ui ی هدوپ میبینیم:

Search: <input type="text"/>			
Operation	↕	Block Size	↕ Name ↕
		128 MB	<u>_SUCCESS</u> 
		128 MB	part-00000 
		128 MB	part-00001 
		128 MB	part-00002 
		128 MB	part-00003 