

Big Data Concepts

28 December 2023 09:18

Before we proceed set the date and time in all the nodes so as to synchronize with your local time

```
$ timedatectl
$ cat /etc/timezone
$ timedatectl list-timezones
$ timedatectl set-timezone Asia/Calcutta
$ timedatectl
```

```
root@mainserver1:~# timedatectl set-timezone Asia/Calcutta
root@mainserver1:~# timedatectl
                Local time: Thu 2023-12-28 09:29:27 IST
                Universal time: Thu 2023-12-28 03:59:27 UTC
                RTC time: Thu 2023-12-28 03:59:28
                Time zone: Asia/Calcutta (IST, +0530)
System clock synchronized: yes
                NTP service: active
                RTC in local TZ: no
```

MapReduce Character Count Example

```
$ mkdir mycounts
```

```
$ cd mycounts
```

```
$ nano WC_Mapper.java
package com.javatpoint;
```

```
import java.io.IOException;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reporter;
public class WC_Mapper extends MapReduceBase implements Mapper<LongWritable,Text,Text,IntWritable>{
    public void map(LongWritable key, Text value,OutputCollector<Text,IntWritable> output,
        Reporter reporter) throws IOException{
        String line = value.toString();
        String tokenizer[] = line.split("");
        for(String SingleChar : tokenizer)
        {
            Text charKey = new Text(SingleChar);
            IntWritable One = new IntWritable(1);
            output.collect(charKey, One);
        }
    }
}
```

<save and exit>

```
$ nano WC_Recuder.java
package com.javatpoint;
import java.io.IOException;
import java.util.Iterator;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
```

```

public class WC_Reducer extends MapReduceBase implements Reducer<Text,IntWritable,Text,IntWritable> {
public void reduce(Text key, Iterator<IntWritable> values,OutputCollector<Text,IntWritable> output,
Reporter reporter) throws IOException {
int sum=0;
while (values.hasNext()) {
sum+=values.next().get();
}
output.collect(key,new IntWritable(sum));
}
}
}

```

<save and exit>

\$ nano WC_Runner.java

```

package com.javatpoint;

import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
public class WC_Runner {
    public static void main(String[] args) throws IOException{
        JobConf conf = new JobConf(WC_Runner.class);
        conf.setJobName("CharCount");
        conf.setOutputKeyClass(Text.class);
        conf.setOutputValueClass(IntWritable.class);
        conf.setMapperClass(WC_Mapper.class);
        conf.setCombinerClass(WC_Reducer.class);
        conf.setReducerClass(WC_Reducer.class);
        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);
        FileInputFormat.setInputPaths(conf,new Path(args[0]));
        FileOutputFormat.setOutputPath(conf,new Path(args[1]));
        JobClient.runJob(conf);
    }
}

```

<save and exit>

\$ javac -d . WC_Mapper.java WC_Reducer.java WC_Runner.java

\$ tree

hadoop@mainserver1:~/mycounts\$ tree

```

.
├── com
│   └── javatpoint
│       ├── WC_Mapper.class
│       ├── WC_Reducer.class
│       └── WC_Runner.class
├── WC_Mapper.java
├── WC_Reducer.java
└── WC_Runner.java

```

2 directories, 6 files

\$ jar -cf demo-charcount.jar *

\$ jar -tf demo-charcount.jar

```
hadoop@mainserver1:~/mycounts$ jar -tf demo-charcount.jar
META-INF/
META-INF/MANIFEST.MF
com/
com/javatpoint/
com/javatpoint/WC_Mapper.class
com/javatpoint/WC_Reducer.class
com/javatpoint/WC_Runner.class
WC_Mapper.java
WC_Reducer.java
WC_Runner.java
```

```
$ nano myname.txt
Harsha K T
<save and exit>
```

```
$ hadoop dfs -mkdir /mycounting
$ hadoop dfs -put myname.txt /mycounting
$ hadoop dfs -ls /mycounting
$ hadoop jar ./demo-charcount.jar com.javatpoint.WC_Runner /mycounting/myname.txt /mycounting/output
$ hadoop dfs -cat /mycounting/output/part-00000
```

Output: (you need to find this at the end of the output)

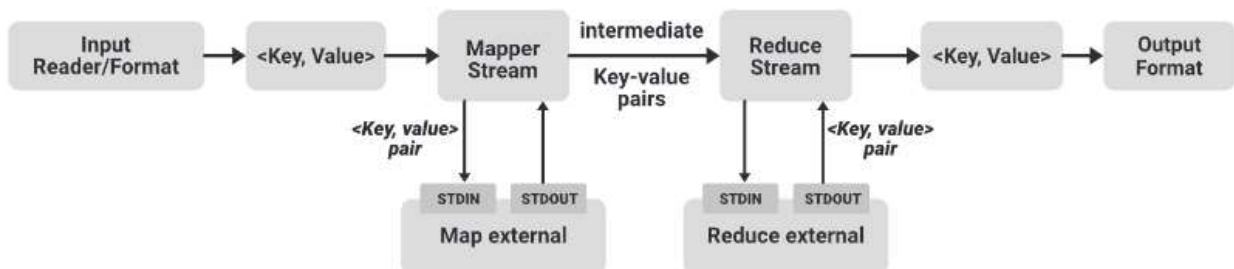
```
23/12/28 11:36:26 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... u
sing builtin-java classes where applicable
      2
H      1
K      1
T      1
a      2
h      1
r      1
s      1
```

MapReduce Streaming

It is a utility or feature that comes with a Hadoop Distribution that allows developer or programmers to write the Map-Reduce program using different programming languages (eg:- Python, Perl, C++, Ruby ..).

We can use any language that can read from the standard input(STDIN) (eg:- keyboard) and write using standard output (STDOUT). The Hadoop framework is completely written in java but programs for Hadoop are not necessarily need to core in java programming language.

Hadoop Streaming



In the above example image, we can see that the flow shown in a dotted block is a basic MapReduce job. In that, we have an Input Reader which is responsible for reading the input data and produces the list of key-value pairs. We can read data in .csv format, in delimiter format, from a database table, image data(.jpg, .png), audio data etc. The only requirement to read all these types of data is that we have to create a particular input format for that data with these input readers. The input reader contains the complete logic about the data it is reading. Suppose we want to read an image then we have to specify the logic in the input reader so that it can read that image data and finally it will generate key-value pairs for that image data.

If we are reading an image data then we can generate key-value pair for each pixel where the key will be the location of the pixel and the value will be its color value from (0-255) for a colored image. Now this list of key-value pairs is fed to the Map phase and Mapper will work on each of these key-value pair of each pixel and generate some intermediate key-value pairs which are then fed to the Reducer after doing shuffling and sorting then the final output produced by the reducer will be written to the HDFS. These are how a simple Map-Reduce job works.

Demo on Hadoop Streaming

```
$ nano student_marks
```

```
Ankush 80
JidNyasa 90
Harshayee 40
Nikita 100
Nikleshwar 95
Niranjan 97
Nisha 88
Nitish 70
Rudolph 98
Shritej 66
chitra-devi 100
Utkarsh 100
pratik 77
shambu 87
rajshree 99
Prasad 35
<save and exit>
```

```
$ hadoop dfs -put student_marks /mycounting
```

```
$ marks='cut -d " " -f2'; $HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar -D
mapreduce.job.name='Experiment' -input /mycounting/student_marks -output /mycounting/student_out -mapper "$marks" -reducer 'cat'
```

```
$ hadoop dfs -cat /mycounting/student_out/part-00000
```

Final output of the above command

```
WARNING: All illegal access operations will be denied in a future release
OpenJDK 64-Bit Server VM warning: You have loaded library /usr/local/hadoop/lib/native/libhadoop.so.1.0.0
which might have disabled stack guard. The VM will try to fix the stack guard now.
It's highly recommended that you fix the library with 'execstack -c <libfile>', or link it with '-z noexec
stack'.
23/12/28 12:58:01 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... u
sing builtin-java classes where applicable

100
100
100
35
40
66
70
77
80
87
88
90
95
97
98
99
```

MapReduce Distributed Cache

- **Distributed Cache:** Distributed Cache is a mechanism in Hadoop that allows the sharing of small amounts of read-only data among all the nodes in a Hadoop cluster. This is particularly useful when tasks running on the nodes need access to common reference data or other resources.
- In a Hadoop job, you might have a set of files or archives that are needed by all the tasks on the nodes. Instead of copying these files to every node, which could be inefficient and consume a lot of storage, Distributed Cache allows you to cache these files on each node.
- During the job setup, files are added to the Distributed Cache, and Hadoop ensures that they are available on each node

before the map or reduce tasks start.

- MapReduce is a programming model for distributed data processing, and Distributed Cache is a feature in Hadoop that allows the efficient sharing of read-only data among the nodes in a Hadoop cluster, reducing the need for redundant data storage and improving performance.

MapReduce Partitioners and Combiners

These are two important concepts that help us to optimize the performance of MapReduce Jobs.

1. Partitioners
2. Combiners

- Partitioners determine how the output of the map tasks is distributed to the reduce tasks, while Combiners provide a way to perform a local reduce on the output of the map tasks to reduce the amount of data transferred over the network. Both are important for optimizing the performance of MapReduce jobs.
- **Partitioners:**
 - In a MapReduce job, the output of the map tasks is grouped by key before being sent to the reduce tasks. The Partitioner is responsible for determining which reduce task will receive a particular key's data. The goal is to evenly distribute the data across the reduce tasks to ensure a balanced workload.

Partitioners process:-

1. The Map function emits key-value pairs.
2. The Partitioner takes the key of each pair and decides which partition (i.e., reduce task) the key-value pair will be sent to.
3. All key-value pairs with the same key are sent to the same partition.
4. Each partition is processed independently by a separate reduce task.

The default behavior of the Partitioner is to hash the key and use the result to determine the partition. However, you can implement a custom Partitioner to control how keys are distributed among the reduce tasks.

Combiners:

- A Combiner is an optional optimization in MapReduce that performs a local reduce on the output of the map tasks before the data is transferred to the reduce tasks. The purpose of the Combiner is to reduce the amount of data that needs to be transferred over the network between the map and reduce tasks.
- The Combiner is similar to a mini-reduce operation that runs on the output of each map task. It combines the values associated with the same key to produce a smaller set of intermediate key-value pairs. Combiners are particularly useful when the output of the map tasks is large, and the data can be effectively reduced before being sent to the reduce tasks.
- It's important to note that the Combiner should be designed to be both associative and commutative so that it can be applied multiple times in any order without changing the result. This property ensures that the final result is not affected by the order in which the Combiners are applied.

MapReduce Input Formats:-

Apache Hadoop and MapReduce, an InputFormat is responsible for defining how input data is split, read, and processed by the individual map tasks in a MapReduce job. Different InputFormats are available to handle various types of input data sources.

The choice of InputFormat depends on the nature of your input data and how it should be processed by the map tasks in your MapReduce job.

Hadoop provides flexibility through these InputFormats to handle a wide range of data sources and formats.

1. TextInputFormat:

- Reads plain text files where each line is treated as a separate record.
- Each record is represented as a key-value pair, where the key is the byte offset of the line in the file, and the value is the content of the line.

2. KeyValueTextInputFormat:

- Similar to TextInputFormat, but it interprets each line as a tab-delimited key-value pair.
- The key and value are separated by a tab character.

3. SequenceFileInputFormat:

- Reads Hadoop SequenceFiles, which are binary files that store key-value pairs.

- Supports compressed and uncompressed SequenceFiles.

4. **TextInputFormat with Custom Record Delimiter:**

- Similar to TextInputFormat, but allows you to specify a custom delimiter for record boundaries.

5. **CombineFileInputFormat:**

- Optimizes small file processing by combining multiple small files into a single input split.
- Helps reduce the overhead associated with processing a large number of small files.

6. **NLineInputFormat:**

- Splits input files into fixed-size chunks, where each chunk contains a specified number of lines.
- Useful when the input records are of variable length.

7. **FileInputFormat:**

- An abstract class that serves as the base class for various input formats.
- Developers can extend this class to create custom InputFormats tailored to specific data sources.

8. **DBInputFormat:**

- Reads data from relational database tables.
- Allows MapReduce jobs to process data stored in database tables.

9. **AvroKeyInputFormat and AvroKeyValueInputFormat:**

- Reads Avro data, which is a binary serialization format.
- AvroKeyInputFormat reads keys of Avro records, while AvroKeyValueInputFormat reads both keys and values.

10. **Custom InputFormats:**

- Developers can create custom InputFormats by implementing the InputFormat interface.
- This is useful for handling specialized data sources or formats not covered by the built-in InputFormats.

MapReduce Output Formats:-

- OutputFormats determine how the output of the reduce tasks is written to the storage system.

OutputFormats:

1. **TextOutputFormat:**

1. Writes the output as plain text files.
2. Each key-value pair is written as a line, with the key and value separated by a tab character (or other delimiter).

2. **SequenceFileOutputFormat:**

1. Writes the output as Hadoop SequenceFiles, which are binary files that store key-value pairs.
2. Supports compressed and uncompressed SequenceFiles.

3. **KeyValueTextOutputFormat:**

1. Similar to TextOutputFormat, but writes key-value pairs in a tab-delimited format.

4. **MultipleOutputs:**

1. Allows a MapReduce job to write output to multiple named outputs.
2. Useful when you want to write different types of data to different directories or files.

5. **NullOutputFormat:**

- Discards the output of the job.
- This can be useful in certain scenarios where you are only interested in the map and reduce functions
- and do not need to store the results.

6. **DBOutputFormat:**

- Writes output directly to relational database tables.
- Useful when you want to store the results of a MapReduce job in a database.

7. **AvroKeyOutputFormat and AvroKeyValueOutputFormat:**

- Writes output in Avro format, a binary serialization format.
- AvroKeyOutputFormat writes only the keys of Avro records,

while AvroKeyValueOutputFormat writes both keys and values.

1. **Custom OutputFormats:**

- Developers can create custom OutputFormats by implementing the OutputFormat interface.
- This is useful when the desired output format is not covered by the built-in OutputFormats

2. **NullOutputFormat:**

- Discards the output of the job.
- This can be useful in certain scenarios where you are only interested in the map and reduce functions
- and do not need to store the results.

3. **DBOutputFormat:**

- Writes output directly to relational database tables.
- Useful when you want to store the results of a MapReduce job in a database.

4. **AvroKeyOutputFormat and AvroKeyValueOutputFormat:**

- Writes output in Avro format, a binary serialization format.
- AvroKeyOutputFormat writes only the keys of Avro records,

while AvroKeyValueOutputFormat writes both keys and values.

1. **Custom OutputFormats:**

- Developers can create custom OutputFormats by implementing the OutputFormat interface.
- This is useful when the desired output format is not covered by the built-in OutputFormats

The choice of OutputFormat depends on how you want to store and organize the results of your MapReduce job. For example, if you want human-readable output, TextOutputFormat might be suitable.

If you need a compact binary format, SequenceFileOutputFormat or Avro formats might be more appropriate.

When designing a MapReduce job, it's crucial to choose the appropriate InputFormat and OutputFormat based on the characteristics of your input and the desired format for your output.

Output formats are similar to input formats

Mapreduce and Python - Example

```
$ cd
```

```
$ nano mapper.py
#!/usr/bin/env python
```

```
import sys
```

```
for line in sys.stdin:
```

```
    # Remove leading and trailing whitespaces
```

```
    line = line.strip()
```

```
    # Split the line into words
```

```
    words = line.split()
```

```
    # Emit key-value pairs (word, 1) for each word
```

```
    for word in words:
```

```
        print(f"{word}\t1")
```

```
$ nano reducer.py
#!/usr/bin/env python
```

```
import sys
```

```
word_count = {}
```

```
for line in sys.stdin:
```

```
    # Remove leading and trailing whitespaces
```

```
    line = line.strip()
```

```
    # Split the line into key-value pairs
```

```
    word, count = line.split("\t", 1)
```

```
    # Convert count to an integer
```

```
    count = int(count)
```

```
    # Update word count
```

```
    word_count[word] = word_count.get(word, 0) + count
```

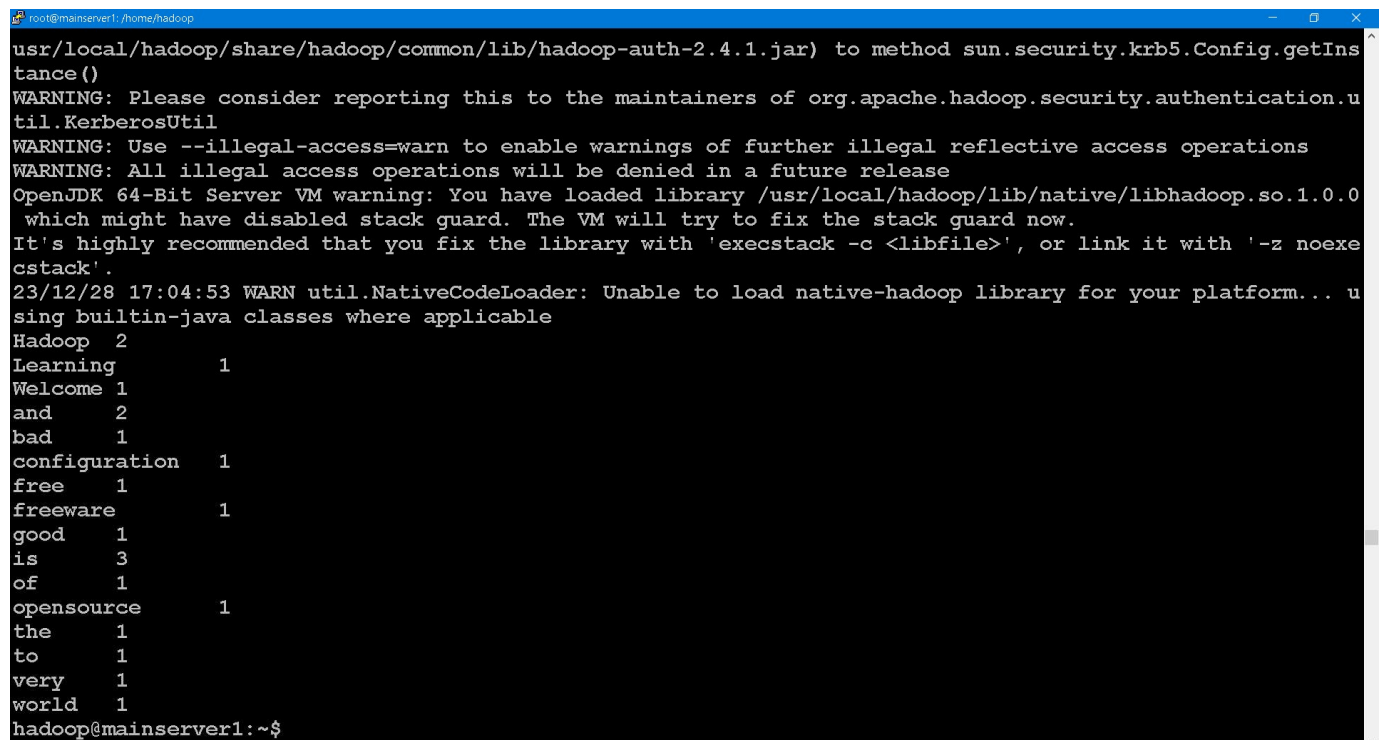
```
# Emit the final word count results
```

```
for word, count in word_count.items():
```

```
    print(f"{word}\t{count}")
```

```
$HADOOP_HOME/bin/hadoop jar $HADOOP_HOME/share/hadoop/tools/lib/hadoop-streaming-2.4.1.jar -files mapper.py,reducer.py -input /testing/mydata.txt -output /testing/mydata_out -mapper ./mapper.py -reducer ./reducer.py
```

```
$ hadoop dfs -cat /testing/mydata_out/part-00000
```

A screenshot of a terminal window with a blue title bar. The terminal shows the execution of a Hadoop jar command and its output. The output includes several warning messages from the Hadoop security and JVM libraries, followed by a list of words and their counts. The words are: Hadoop (2), Learning (1), Welcome (1), and (2), bad (1), configuration (1), free (1), freeware (1), good (1), is (3), of (1), opensource (1), the (1), to (1), very (1), and world (1). The prompt at the bottom is 'hadoop@mainserver1:~\$'.

```
usr/local/hadoop/share/hadoop/common/lib/hadoop-auth-2.4.1.jar) to method sun.security.krb5.Config.getInst
tance()
WARNING: Please consider reporting this to the maintainers of org.apache.hadoop.security.authentication.u
til.KerberosUtil
WARNING: Use --illegal-access=warn to enable warnings of further illegal reflective access operations
WARNING: All illegal access operations will be denied in a future release
OpenJDK 64-Bit Server VM warning: You have loaded library /usr/local/hadoop/lib/native/libhadoop.so.1.0.0
which might have disabled stack guard. The VM will try to fix the stack guard now.
It's highly recommended that you fix the library with 'execstack -c <libfile>', or link it with '-z noexe
cstack'.
23/12/28 17:04:53 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... u
sing builtin-java classes where applicable
Hadoop 2
Learning      1
Welcome 1
and          2
bad          1
configuration 1
free         1
freeware     1
good         1
is           3
of           1
opensource   1
the          1
to           1
very         1
world        1
hadoop@mainserver1:~$
```

Final output