Cloud Computing **Emanuele Giona**
A.Y. 2018/2019

Word Count using Apache Spark on GCP

1 Introduction

Apache Spark [1] is a unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning and graph processing. It supports cluster execution by leveraging a number of resource managers and distributed filesystems, including YARN [2] and HDFS [3], both part of the same Apache Hadoop project [4].

Manually setting up a cluster can be overwhelming, therefore several solutions provided by Google, Amazon, Microsoft, and others are popular in the Cloud Computing industry; Google Cloud Platform (GCP) [5] is therefore Google's offer to solve these problems. In particular, Google Cloud Dataproc [6] goes a step further by also providing a fully-functional, application-ready cluster with Apache Spark and its whole stack already installed and configured.

In order to better understand the features of Apache Spark and the underlying GCP Dataproc system, this project consists of cluster setup, the deployment of a simple application like Word Count, and the execution of an experiment in order to understand the compliance of the system to a set of requirements.

2 Requirements

In this section, the project requirements are presented.

2.1 Infrastructure Requirements

This set of requirements aims to evaluate Google Cloud Dataproc clusters' load balancing, elasticity, and resiliency.

- IR1: load balancing among application instances running in the same datacenter;
- IR2: load balancing among application instances running in different datacenters (*Optional*);
- IR3: VM deployment in multiple availability zones;
- IR4: horizontal auto-scaling (both scale-in and scale-out);
- IR5: self-healing properties.

2.2 Application Requirements

This set of requirements aims to evaluate the application's ability to exploit an elastic cluster and to provide a quick benchmarking tool.

- AR1: elastic application;
- AR2: automatic workload generation for application elasticity and infrastructure scalability testing purposes.

3 Word Count

A common problem in Computer Science is to count how many occurrences of each token are present in a file(s); this task is widely known as Word Count. It can be efficiently solved exploiting simple algorithms in situations where the file(s) can fit in a machine's RAM or disk(s), but challenges arise when the file(s) is either too big or data is continuously collected at such speeds that there isn't even time for storing it.

For this project, I have chosen the latter case of word count, monitoring a directory and counting the words in newly added files to it; the solution can be found using a streaming algorithm, processing data in temporal batches. The code shown in Listing 1 implements the aforementioned solution exploiting PySpark, the Python package for the Apache Spark framework. It is a slightly modified version of a streaming example[1], enabling Spark's dynamic allocation in order to comply with the AR1 requirement (lines 12-15 - see Section 2.2), and tuning the temporal batch to 5 seconds (line 18).

```
1  import sys
2
3  from pyspark import SparkContext, SparkConf
4  from pyspark.streaming import StreamingContext
5
6  if __name__ == "__main__":
7      if len(sys.argv) != 2:
8          print("Usage: hdfs_wordcount.py <directory>",
9                file=sys.stderr)
10         sys.exit(-1)
11
12     conf = SparkConf()\
13         .set("spark.shuffle.service.enabled", "true")\
14         .set("spark.dynamicAllocation.enabled", "true")\
15         .set("spark.dynamicAllocation.maxExecutors", "3")
16     sc = SparkContext(appName="PythonStreamingHDFSWordCount",
17                       conf=conf)
18     ssc = StreamingContext(sc, 5)
19
20     lines = ssc.textFileStream(sys.argv[1])
21     counts = lines.flatMap(lambda line: line.split(" "))\
22                   .map(lambda x: (x, 1))\
23                   .reduceByKey(lambda a, b: a+b)
24     counts.pprint()
25
26     ssc.start()
27     ssc.awaitTermination()
```

Listing 1: hdfs_wordcount.py (from Spark Streaming examples)

Due to the AR2 requirement (see Section 2.2), I have developed a simple benchmarking suite, consisting of two Python scripts: `file_generator.py` and `benchmark.py`.

The `file_generator.py` script implements a single file generator, making use of a predefined text sample and repeating it as many times as required. The text file is named after the current timestamp, created in a temporary directory first and the text sample repetitions are appended

[1]URL: https://github.com/apache/spark/blob/master/examples/src/main/python/streaming/hdfs_wordcount.py

to it for performance purposes; upon reaching the desired length, the file is moved into the target directory. Full code is shown in Listing 2.

```
1  import sys
2  from os.path import join, exists
3  from shutil import move
4  from time import time
5
6
7  # Set of words to be used by the word generator
8  TEXT = """<omitted>"""\
9      .replace(",", "")\
10     .replace(".", "")\
11     .replace("\n", "")
12
13
14 def generate_file(length: int = 1,
15                    tmp_dir: str = "tmp/",
16                    tgt_dir: str = "data/",
17                    quiet: bool = False) -> None:
18     """
19     Creates a an arbitrary length file in a temporary directory,
20     then moves it to the target directory.
21
22     :param length: number of repetitions of TEXT to be
23                    contained in the generated file
24     :param tmp_dir: path to the temporary directory
25     :param tgt_dir: path to the target directory
26     :param quiet: if True, no output to the standard output
27     """
28
29     timestamp = str(time()).replace(".", "")
30     filename = "text_{timestamp}.txt".format(timestamp=timestamp)
31     tmp_path = join(tmp_dir, filename)
32     tgt_path = join(tgt_dir, filename)
33
34     # file creation
35     try:
36         if not quiet:
37             print("Creating file...")
38
39         with open(tmp_path, "w", encoding="utf-8"):
40             pass
41
42         with open(tmp_path, "a", encoding="utf-8") as file:
43             for i in range(0, length):
44                 file.write(TEXT)
45                 file.write(" ")
46                 file.flush()
47
```

```
48          if not quiet:
49              print("Done.")
50
51      except Exception as e:
52          if not quiet:
53              print("Error during file creation ({msg}).")
54                    .format(msg=str(e)))
55
56          return
57
58      # file moving
59      try:
60          if not quiet:
61              print("Moving file...")
62          move(tmp_path, tgt_path)
63
64          if not quiet:
65              print("Done.")
66
67      except Exception as e:
68          if not quiet:
69              print("Error during file moving ({msg}).")
70                    .format(msg=str(e)))
71
72          return
73
74
75  if __name__ == "__main__":
76      if len(sys.argv) != 4:
77          print("Usage: file_generator.py <length> <tmp_dir> <tgt_dir>",
78                file=sys.stderr)
79          sys.exit(-1)
80
81      file_len = int(sys.argv[1])
82      tmp = sys.argv[2]
83      tgt = sys.argv[3]
84
85      assert file_len > 0, "Length should be at least 1"
86      assert exists(tmp), "Temporary directory does not exist"
87      assert exists(tgt), "Target directory does not exist"
88
89      generate_file(length=file_len,
90                    tmp_dir=tmp,
91                    tgt_dir=tgt)
```

Listing 2: file_generator.py

The `benchmark.py` script instead spawns a dynamic number of threads, following a specific predetermined pattern (lines 18-22), each of them executing the `generate_file()` function defined in the `file_generator.py` script. The number of threads (and therefore the number of files the Spark application has to process) increases over time until 100, then decreasing with the same speed. The higher the number of threads, the longer the files they generate (up to

500 - line 13), hence testing the capabilities of the cluster to face an unexpected increase in the workload. Full code is shown in Listing 3.

```
import sys
from os.path import exists
from threading import Thread
from typing import List
from time import sleep
from datetime import datetime

from file_generator import generate_file


# files have dynamic length (in terms of repetitions of the
# TEXT sample)
FILE_LENGTHS = [length for length in range(10, 501)]


# application load pattern: from 1 to 100 threads writing
# files simultaneously
LOAD_PATTERN = [generators for generators in range(1, 110, 10)]
inverse_pattern = LOAD_PATTERN.copy()
inverse_pattern.reverse()
LOAD_PATTERN = LOAD_PATTERN + inverse_pattern
LOAD_PATTERN = LOAD_PATTERN


def get_time():
    """
    Gets the current time
    :return: Current time, as String
    """

    return str(datetime.now()).split(".")[0]


def spawn_threads(lengths: List[int],
                  tmp_dir: str,
                  tgt_dir: str,
                  quiet: bool = False) -> None:
    """
    Spawns a number of threads, each writing a file with the length
    specified in the 'lengths' argument.
    Threads execute the 'generate_file' function from
    'file_generator.py'.

    :param lengths: list of file lengths; the number of threads
                    spawned is equal to the length of this argument
    :param tmp_dir: path to the temporary directory
    :param tgt_dir: path to the target directory
```

```
48      :param quiet: if True, no output to the standard output
49      """
50
51      threads = [Thread(target=generate_file,
52                        kwargs={"length": file_len,
53                                "tmp_dir": tmp_dir,
54                                "tgt_dir": tgt_dir,
55                                "quiet": quiet}
56                        )
57                 for file_len in lengths]
58
59      if not quiet:
60          print("\nSpawning {num} threads..."
61                .format(num=len(lengths)))
62
63      for thread in threads:
64          thread.start()
65
66      if not quiet:
67          print("Waiting for execution...")
68
69      for thread in threads:
70          thread.join()
71
72      if not quiet:
73          print("Execution completed.")
74
75
76  if __name__ == "__main__":
77      if len(sys.argv) != 3:
78          print("Usage: benchmark.py <tmp_dir> <tgt_dir>",
79                file=sys.stderr)
80          sys.exit(-1)
81
82      tmp = sys.argv[1]
83      tgt = sys.argv[2]
84
85      assert exists(tmp), "Temporary directory does not exist"
86      assert exists(tgt), "Target directory does not exist"
87
88      print("\nStarting benchmarking... ({time})"
89            .format(time=get_time()))
90
91      for threads_number in LOAD_PATTERN:
92          index = min(len(FILE_LENGTHS)-1, int(threads_number*5))
93          spawn_threads(lengths=[FILE_LENGTHS[index]] * threads_number,
94                        tmp_dir=tmp,
95                        tgt_dir=tgt,
96                        quiet=True)
97          sleep(2)
```

```
98
99      print("\nBenchmarking completed. ({time})"
100             .format(time=get_time()))
```

Listing 3: benchmark.py

4 Dataproc Cluster Setup

In this section, all the required steps to setup a Google Cloud Dataproc cluster are presented in detail.

4.1 Bucket

From the left-side menu, find the **Storage** sub-menu and click the *Browser* option, as shown in Figure 1.

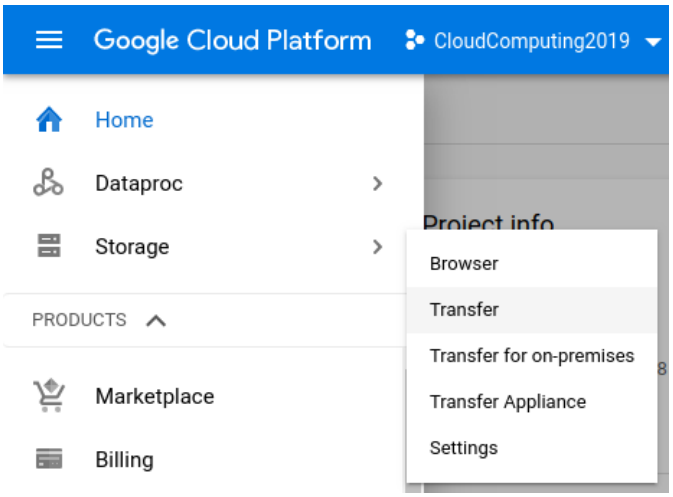
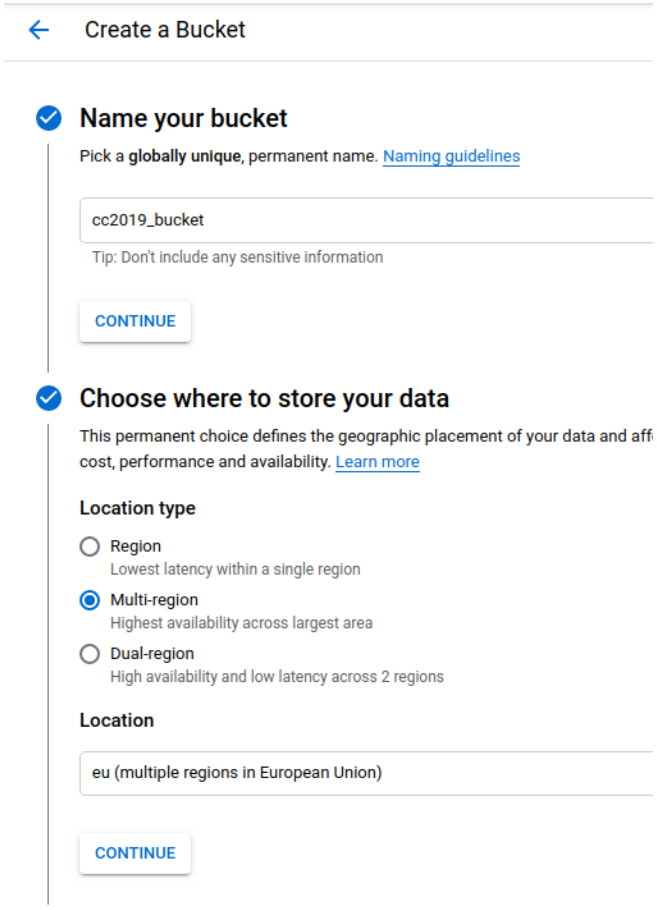


Figure 1: Bucket creation (step 1)

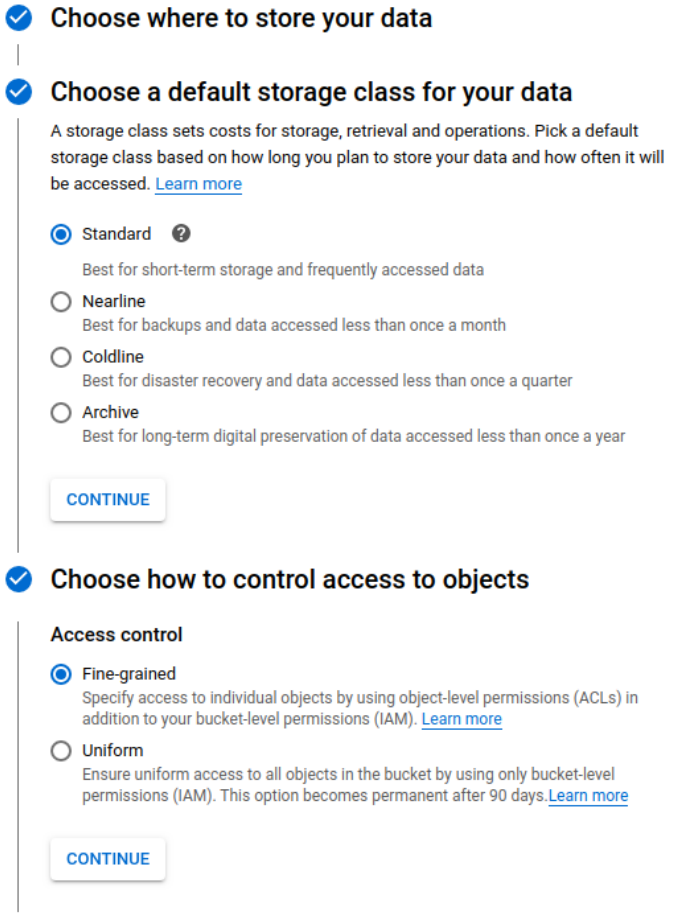
Upon clicking on the **Create** button, a new form will show up, asking for name, regional location, storage class, and access control options for the bucket (see Figure 2).

In my case, I have chosen *Multi-region* in order to exploit a wider geographical distribution for the data nodes, and therefore potentially achieve a higher resilience to local outages and malfunctionings. The rest of the options have been left with the default values, since they were the optimal choices for my use case.

After a brief provisioning time, the bucket is fully accessible via the Google Cloud Console, and I proceeded to create the following folders: `config`, `src`, `data`, `tmp` (see Figure 3). Through the same Console, I also uploaded the various source files of the application in the most appropriate directories (see Sections 3, 4.2).



(a) Step 2



(b) Step 3

Figure 2: Bucket creation (steps 2 & 3)

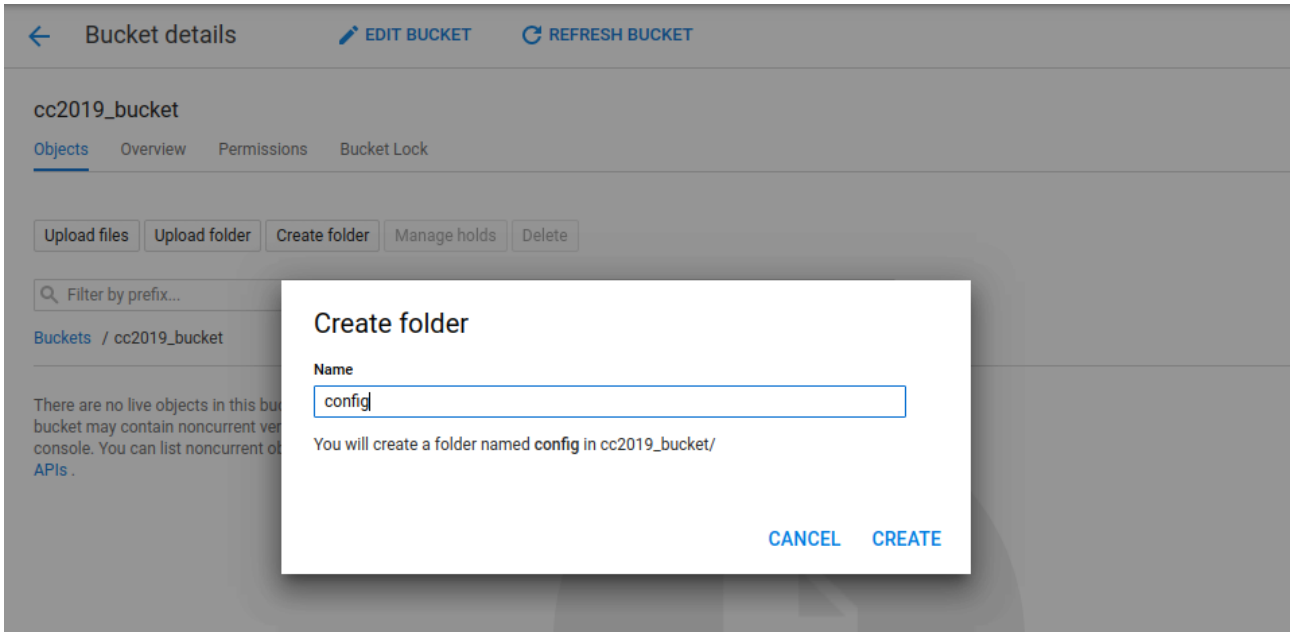


Figure 3: Bucket file browser

4.2 Policy

In order to comply with the IR4 requirement (see Section 2.1), the following custom autoscaling policy has been developed.

```
workerConfig:
  minInstances: 2
  maxInstances: 3
secondaryWorkerConfig:
  maxInstances: 0
basicAlgorithm:
  cooldownPeriod: 3m
  yarnConfig:
    scaleUpFactor: 1.0
    scaleDownFactor: 1.0
    gracefulDecommissionTimeout: 10m
```

Listing 4: autoscaling_policy.yaml

This policy allows for the use of a maximum of 3 primary workers, with 2 being the minimum number of their instances (more on this in Section 4.3); additionally it does not make use of any preemptible VM. The scaling algorithm is referred to as `basicAlgorithm` and it's the only one available for Google Cloud Dataproc clusters so far: `cooldownPeriod` specifies the time interval after which a new autoscaling action can be applied again once a scaling action has already occurred. The rest of parameters are used in configuring the YARN resource manager, with `scaleUpFactor` and `scaleDownFactor` specifying the target memory percentage[2] once a scale-up or scale-down action is performed, respectively. The `gracefulDecommissionTimeout` parameter instead specifies the timeout duration before a YARN node can be decommissioned, waiting for potentially on-going jobs to complete.

More parameters of `yarnConfig` include the cluster size increase and decrease percentage threshold for scale-up and scale-down actions to happen: in case the autoscaler recommendation does not meet the threshold, the action is not performed.

In order to seamlessly use the custom autoscaling policy in any new cluster (and actually for any other operation), Google Cloud Platform allows the use of Cloud Shell, a remote Unix-like terminal with the `gcloud` CLI application already installed and configured. After having launched Cloud Shell, running the following commands import a YAML file containing a valid GCP policy and assign the `simple_policy` name to it.

```
mkdir bucket
gcsfuse cc2019_bucket bucket
cd bucket/config
gcloud dataproc autoscaling-policies \
import simple_policy --source autoscaling_policy.yaml
```

Listing 5: Cloud Shell commands to import an autoscaling policy

The previous commands assume the bucket is named `cc2019_bucket` and a file named `autoscaling_policy.yaml` has been uploaded into the `config` directory of the bucket (see Section 4.1).

[2] *i.e.* `scaleUpFactor: 1.0` requires that the entirety of the pending memory has to be allocated as result of the next scale-up action

4.3 Cluster

From the left-side menu, find the **Dataproc** sub-menu and click the *Cluster* option, as shown in Figure 4.

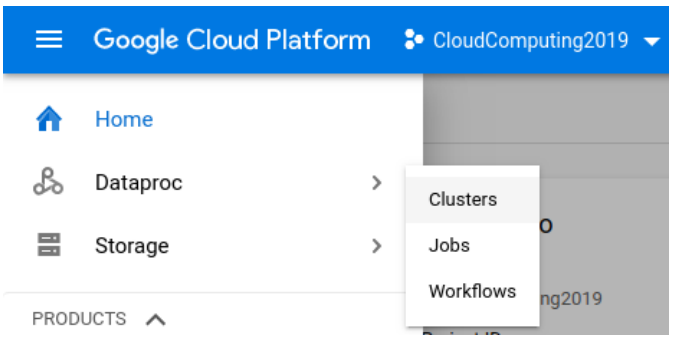


Figure 4: Cluster creation (step 1)

Upon clicking on the **Create** button, a new form will show up asking for cluster name, region, zone and mode of deployment (see Figure 5a).

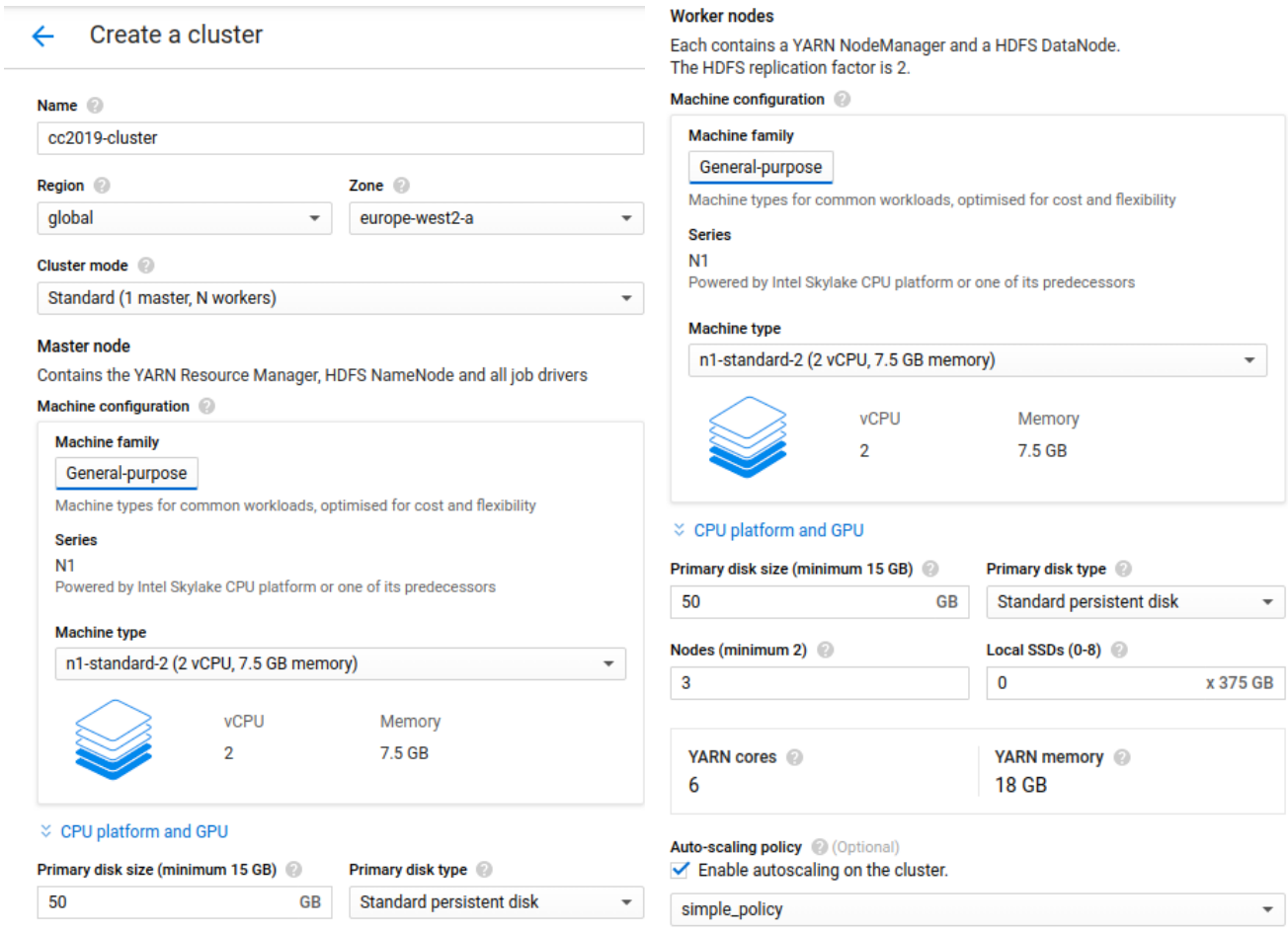
In GCP, a **region** is a specific geographical location where resources can be hosted; each region contains one or more **zones**. VM instances and persistent disks are *zonal resources* as they can only be accessed by other resources in the same zone; static external IP addresses are instead *regional resources*, and they can be used by any other resource in the same region, regardless of zone.

In this case, I have chosen the *Global* region, which allows me to drop the region restrictions, potentially being able to access any resource provided by Google Cloud Platform. Unfortunately, there is no option to define a cluster across multiple zones, and thus I have selected *europe-west2-a* as zone.

For the Word Count application, I have chosen a standard 1 master node and N worker nodes cluster; the master node has been equipped with 2 vCPUs, 7.5 GB of RAM, and a 50 GB HDD as persistent storage option (see Figure 5a). All the **three** worker nodes have been equipped with the same hardware configuration (see Figure 5b).

In the GCP Console, the *shown* YARN cores only refer to the vCPUs allocated to worker nodes, therefore the value of 6; in the free plan I am using, the YARN cores are *limited to 8* and also the number of IP addresses that can be allocated to a cluster is *limited to 4*. The YARN cores limit actually also counts the vCPUs allocated to the master node on top of the worker node ones, hence why the hardware settings in place are actually maximizing the free plan constraints, despite being sub par.

Leaving the networking settings with the default values, as shown in Figure 6, I chose to use the same bucket created in Section 4.1 also to stage the cluster's jobs; additionally I selected the latest Cloud Dataproc Image (version 1.4), due to the PySpark requirement of Apache Spark 2.4.



(a) Step 2

(b) Step 3

Figure 5: Cluster creation (steps 2 & 3)

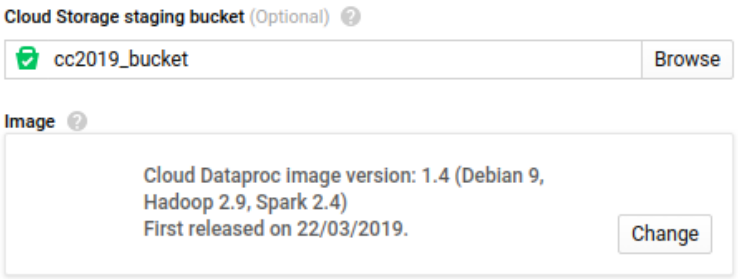


Figure 6: Cluster creation (step 4)

5 Experiment

In order to verify the compliance to the requirements (see Section 2), the application and the infrastructure have undergone the following experiment:

1. Launching a single PySpark job with no load (namely `job1`);
2. Adding workload to `job1`;
3. Launching two more PySpark jobs (namely `job2` and `job3`);

4. Adding workload to `job2` and `job3`;

5. Stopping workload generation at different times;

6. Stopping PySpark jobs at different times.

In order to launch a Spark job on the cluster, the Cloud Shell has been used, launching the following command:

```
gcloud dataproc jobs submit pyspark src/hdfs_wordcount.py \
--cluster=cc2019-cluster --id=job1 -- gs://cc2019_bucket/data/job1
```

Listing 6: Cloud Shell command to launch a PySpark job

In Listing 6, it is assumed the current directory is `bucket` (previously created in Listing 5), the Word Count file is located in `src/hdfs_wordcount.py`, the cluster's name is `cc2019-cluster`, and the directory to monitor is `gs://cc2019_bucket/data/job1`.

The application's workload is also generated through the Cloud Shell, launching the following command:

```
python3 src/benchmark.py tmp/ data/job1/ &
```

Listing 7: Cloud Shell command to launch `benchmark.py`

This command allows to run the `benchmark.py` script in background, generating files in the `tmp` directory, then moving them into the `data/job1` directory. As in Listing 6, it is assumed the current directory is `bucket`. Files moved into `data/job1` have the effect of actually appearing at the remote `gs://cc2019_bucket/data/job1` URI, due to mounting the Cloud Bucket into the `bucket` directory via the `gcsfuse` tool in Listing 5, and therefore being fed to the Word Count job previously launched.

Notice that the PySpark jobs have to be manually stopped due to the fact that the Word Count version currently deployed (see Section 3) is a streaming application and is *not aware* of the lack of input data implementing a timeout before termination.

In order to forcefully stop providing workload to a job, the BASH `kill -9 <PID>` command has been used on the process running the `benchmark.py` script associated to that job, while the manual halting of PySpark jobs has been achieved using the Google Cloud Console, accessed through the **Dataproc** sub-menu, then clicking on *Jobs*.

Figures 7, 8, 9, and 10 have been captured from Google's cloud computing systems management service Stackdriver at the end of the experiment, then labelled with the following interesting points in time:

1. 13:33 - `job1` submitted, benchmark for `job1` started;

2. 13:35 - `job2` submitted, benchmark for `job2` started;

3. 13:38 - `job3` submitted;

4. 13:42 - benchmark for `job3` started, scale-up action;

5. 14:00 - benchmarks for `job3` and `job2` stopped, `job3` stopped;

6. 14:25 - benchmark for `job1` stopped, `job1` stopped.

It is easy to see the effect of the `job1` and `job2` on the cluster, although it is not enough to trigger a scale-up action, since the `basicAlgorithm` scaling only applies with thresholds on YARN's pending memory, which are not met yet. As soon as `job3` is submitted, YARN's available memory has been depleted (see time 3 (13:38) in Figures 7, 9), but without workload, the following warning message is being printed into the job's standard output:

```
YarnScheduler WARNING: Initial job has not accepted any resources;
check your cluster UI to ensure that workers are registered and
have sufficient resources
```

Listing 8: Cloud Shell command to launch `benchmarking.py`

As soon as the workload is provided and the memory demands exceed the cluster's capacity, a new VM is added to the cluster as result of a scale-up action, fulfilling the application's needs (see time 4 (13:42) in Figures 7, 8, 9).

Similarly, when at time 5 (14:00) `job3` is stopped (and also its associated benchmark and `job2`'s), the current capacity is excessive and a scale-down action is performed in order to prevent quota wastage.

Figure 10 is included and labelled as well only for completeness, since HDFS capacity isn't really the focus for a Spark Streaming application of the likes of the deployed version of Word Count.

Figure 7: YARN memory (GB)

Figure 8: YARN nodes

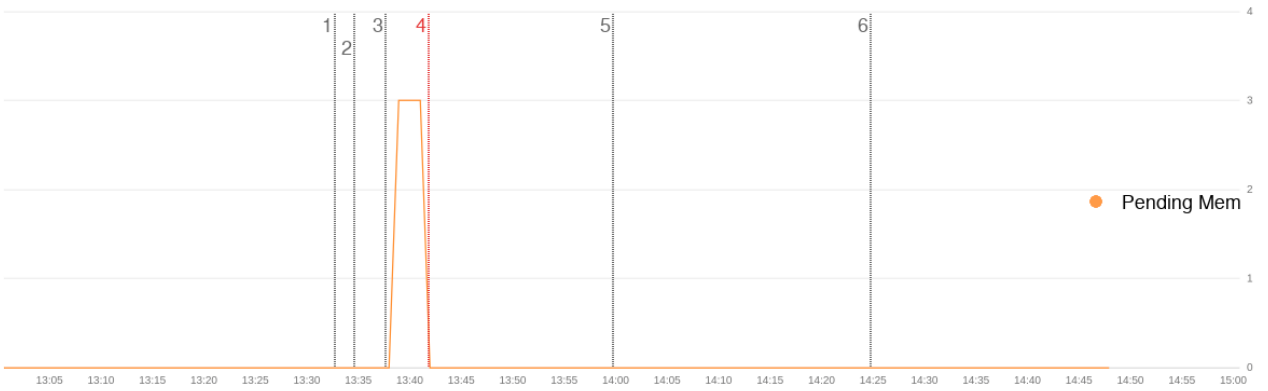


Figure 9: YARN pending memory (GB)

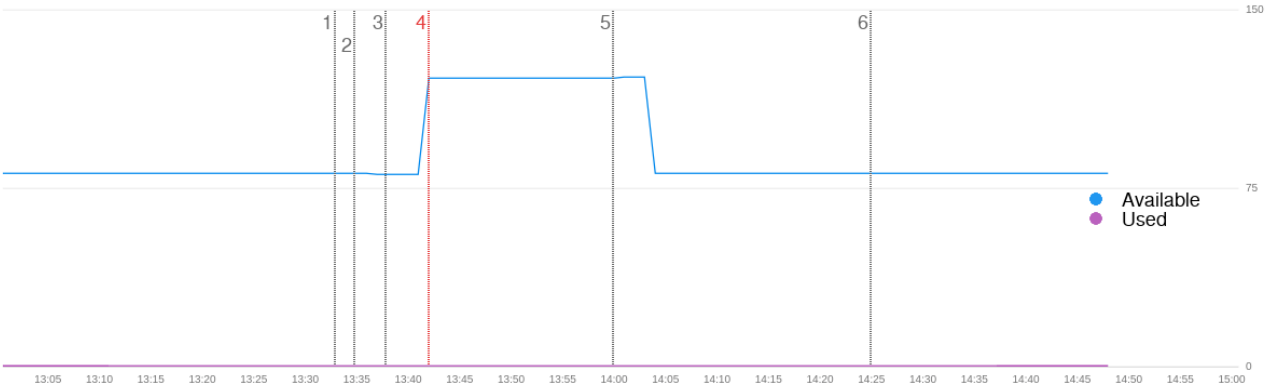


Figure 10: HDFS capacity (GB)

6 Conclusions

In light of the experiment and annex measurement carried out in Section 5, the presented Word Count application (Section 3), implemented using the Apache Spark Streaming framework and deployed on a Google Cloud Dataproc cluster (Section 4) equipped with a custom autoscaling policy defined in Section 4.2, scores the following compliances with respect to the requirements described in Section 2:

- IR1: compliant
  Apache Spark handles load balancing by design, and Google Cloud Dataproc comes with Spark, HDFS, and YARN pre-installed, therefore satisfying the requirement.

- IR2: compliant
  Google Cloud Platform Zones consist of multiple datacenters; applications instances are deployed to VMs that can be hosted at different physical datacenters.

- IR3: not compliant - Dataproc free plan restrictions
  Despite the selection of the *Global* region in Section 4.3, Google Cloud Dataproc does not allow the deployement of VMs in multiple zones. My best efforts towards this requirement are limited to tuning a high-availability bucket as shown in Section 4.1, since deploying different clusters and then networking them together for using each other's VMs would have been impossible due to the free plan restrictions.

- IR4: compliant
  The deployed cluster configuration exploits horizontal autoscaling thanks to a custom policy as described in Section 4.2 and demonstrated in Section 5.

- IR5: compliant
  Apache Spark provides fault tolerance by design, and Google Cloud Dataproc comes with Spark, HDFS, and YARN pre-installed, therefore satisfying the requirement.

- AR1: compliant
  Apache Spark applications are elastic by design, but additional care has been taken by enabling dynamic allocation for Spark Streaming, as shown in Section 3.

- AR2: compliant
  Custom benchmarking suite has been documented in Section 3; the experiment and measurements in Section 5 have been carried out using the provided workload generation tool.

References

[1] Apache Spark - Unified Analytics Engine for Big Data
URL: https://spark.apache.org/

[2] Apache Hadoop YARN - Resource Negotiator
https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html

[3] Apache Hadoop HDFS - Hadoop Distributed File System
URL: https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html

[4] Apache Hadoop
URL: https://hadoop.apache.org/

[5] Google Cloud Platform (GCP)
URL: https://cloud.google.com/

[6] Dataproc - Apache Spark-ready environment running on top of GCP
URL: https://cloud.google.com/dataproc/