CS744 Assignment 1

Due: Sep 25, 2019 (extended)

Overview

This assignment is designed to support your in-class understanding of how data analytics stacks work and get some hands-on experience in using them. You will need to deploy Apache Hadoop as the underlying file system and Apache Spark as the execution engine. You will then develop several small applications based on them. You will produce a short report detailing your observations and takeaways.

Learning Outcomes

After completing this programming assignment, you should be able to:

- Deploy and configure Apache Spark and HDFS.
- Write Spark simple applications and launch them in the cluster.
- Describe how Apache Spark and HDFS work, and interact with each other.

Environment Setup

You will complete your assignment in CloudLab. You can refer to Assignment 0 (http://pages.cs.wisc.edu/~shivaram/cs744-fa19/assignment-zero.html) to learn how to use CloudLab. We suggest you create experiments in form of groups and work together. An experiment expires within two days, which is very quick. So, set a time frame that all your group members can sit together and focus on the project, or make sure to extend the experiment when it is necessary.

In this assignment, we provide you a CloudLab profile called “cs744-fa19-assignment1” under “UWMadison744-F19” project for you to start your experiment. The profile is a simple 3-node cluster with Ubuntu 16 installed on each machine. You get full control of the machines once the experiment is created, so feel free to download any missing packages you need in the assignment.

As the first step, you should run following commands on every VM:

1. sudo apt-get update –fix-missing
2. sudo apt-get install openjdk-8-jdk
3. enable the SSH service among the nodes in the cluster. To do this, you have to generate a private/public key pair using: `ssh-keygen -t rsa` on the master node. You should designate a VM to act as a master/slave (say node-0) while the others are assigned as slaves only. Then, manually copy the public key of node-0 to the `authorized_keys` file in all the nodes(including node-0) under `~/.ssh/` . To get the content of the public key, do:

   ```
   cat ~/.ssh/id_rsa.pub
   ```

When you copy the content, make sure you do not append any newlines. Otherwise, it will not work.

Once you have done this you can copy files from the master node (i.e. node-0) to the other nodes using tools like parallel-ssh (http://code.google.com/p/parallel-ssh). To use parallel-ssh you will need to create a file with the hostnames of all the machines. You can test your parallel-ssh with a command like

```
parallel-ssh -i -h slaves -O StrictHostKeyChecking=no hostname
```

Part 0: Mounting disks

Your home directory in the CloudLab machine is relatively small and can only hold 16GB of data. We have also enabled another mount point to contain around 25GB of space on each node which should be sufficient to complete this assignment.

However you need to create this mount point using the following commands (on each node).

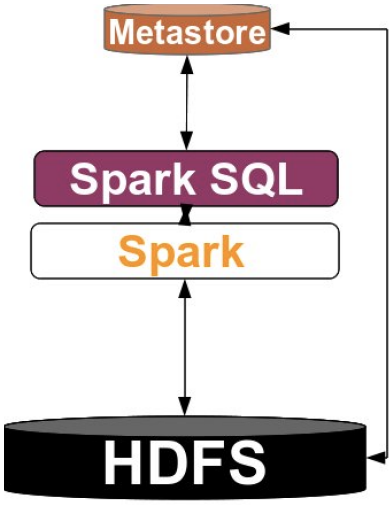
```
> sudo mkfs.ext4 /dev/xvda4
# This formats the partition to be of type ext4
> sudo mkdir -p /mnt/data
# Create a directory where the filesystem will be mounted
> sudo mount /dev/xvda4 /mnt/data
# Mount the partition at the particular directory
```

After you complete the above steps you can verify this is correct by running

```
> shivaram@node1:~$ df -h | grep data
/dev/xvda4                                        24G   44M   23G   1% /mnt/data
```

Now you can use /mnt/data to store files in HDFS or to store shuffle data in Spark (see below)

Part 1: Software Deployment



Deployment stack

Apache Hadoop (https://hadoop.apache.org)

Apache Hadoop is a collection of open-source software utilities that provides simple distributed programming models for processing of large data sets. It mainly consists of the Hadoop Distribited File System (HDFS), Hadoop MapReduce and Hadoop YARN. In this assignment, we will only use HDFS. HDFS consists of a NameNode process running on the master instance and a set of DataNode processes running on slave instances. The NameNode records metadata and handles requests. The DataNode stores actual data.

You can find the detailed deployment instructions in this link (https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-common/ClusterSetup.html) or you can follow our simplified version:

First, let's download Hadoop on every machine in the cluster. Note that you can do this on the master node and then use parallel-ssh (http://code.google.com/p/parallel-ssh/) or parallel-scp to run the same command or copy data to all VMs.

```
wget http://apache.mirrors.hoobly.com/hadoop/common/hadoop-3.1.2/hadoop-3.1.2.tar.gz
tar zvxf hadoop-3.1.2.tar.gz
```

There are a few configuration files we need to edit. They are originally empty so users have to manually set them. Add the following contents in the `<property>` field in `hadoop-3.1.2/etc/hadoop/core-site.xml`:

```
<configuration>
<property>
<name>fs.default.name</name>
<value>hdfs://namenode_IP:9000</value>
</property>
</configuration>
```

where `namenode_IP` refers to the IP address of the master node. This configuration indicates where the NameNode will be listening for connections.

Also you need to add the following in `hadoop-3.1.2/etc/hadoop/hdfs-site.xml`. Make sure you specify the path by yourself (You should create folders by yourself if needed. For example, create `hadoop-3.1.2/data/namenode/` and set it to be the path for namenode dir). These directories indicate where data for the NameNode and DataNode will be stored respectively. The path in the xml file should be absolute. Note that the same path needs to exist on all the slave machines which will be running DataNodes.

```
<configuration>
<property>
<name>dfs.namenode.name.dir</name>
<value>/path/to/namenode/dir/</value>
</property>
<property>
<name>dfs.datanode.data.dir</name>
<value>/path/to/datanode/dir</value>
</property>
</configuration>
```

You also need to manually specify `JAVA_HOME` in `hadoop-3.1.2/etc/hadoop/hadoop-env.sh`. You can get the path with the command: `update-alternatives --display java`. Take the value of the current link and remove the trailing `/bin/java`. For example, a possible link can be `/usr/lib/jvm/java-8-openjdk-amd64/jre`. Then, set the `JAVA_HOME` by replacing `export JAVA_HOME=${JAVA_HOME}` with `export JAVA_HOME=/actual/path`.

Copy the config files with these changes to all the machines.

We also need to edit `hadoop-3.1.2/etc/hadoop/workers` to add the IP address of all the datanodes. In our case, we need to add the IP addresses for all the nodes in the cluster, so every node can store data.

Now, we start to format the namenode and start the namymenode daemon. Firstly, add `hadoop-3.1.2/bin` and `hadoop-3.1.2/sbin` to $PATH. Then, do:

```
hdfs namenode -format
start-dfs.sh
```

This will also start all the datanode daemons:

To check the HDFS status, go to (port 9870 is for hadoop version 3 and higher):

```
<namenode_IP>:9870/dfshealth.html
```

You can also use command `jps` to check whether HDFS is up, there should be a `NameNode` process is running on your master VM, and a `DataNode` process is running on each of your VMs.

Now, the HDFS is setup. Type the following to see the available commands in HDFS.

```
hdfs dfs -help
```

Apache Spark (https://spark.apache.org/docs/latest/index.html)

Apache Spark is a powerful open-source unified analytics engine for big data processing, which is built upon its core idea of Resilient Distributed Datasets (RDDs). Spark standalone consists of a set of daemons: a Master daemon, and a set of Worker daemons. Spark applications are coordinated by a SparkContext object which will connect to the Master, responsible for allocating resources across applications. Once connected, Spark acquires Executors on every Worker node in the cluster, which are processes that run computations and store data for your applications. Finally, the application's tasks are handled to Executors for execution. We will use Spark in standalone mode, which means it doesn't need to rely on resource management systems like YARN.

Instructions on building a Spark cluster can be found in Spark's official document (https://spark.apache.org/docs/latest/spark-standalone.html). Or you can follow our instructions:

Firstly, download and decompress the Spark binary on each node in the cluster:

```
wget http://mirror.metrocast.net/apache/spark/spark-2.4.4/spark-2.4.4-bin-hadoop2.7.tgz
tar zvxf spark-2.4.4-bin-hadoop2.7.tgz
```

Similar to HDFS you will need to modify `spark-2.4.4-bin-hadoop2.7/conf/slaves` to include the IP address of all the slave machines.

To start the Spark standalone cluster you can then run the following command on the master node:

```
spark-2.4.4-bin-hadoop2.7/sbin/start-all.sh
```

You can go to `<master_node_IP>:8080` to check the status of the Spark cluster.

To check that the cluster is up and running you can use `jps` to check that a `Master` process is running on your master VM, and a `Worker` process is running on each of your slave VMs.

To stop all nodes in the cluster, do

```
spark-2.4.4-bin-hadoop2.7/sbin/stop-all.sh
```

Next, setup the properties for the memory and CPU used by Spark applications. Set Spark driver memory to 32GB and executor memory to 32GB. Set executor cores to be 10 and number of cpus per task to be 1. Document about setting properties is here (https://spark.apache.org/docs/2.0.0/configuration.html#spark-properties).

Part 2: A simple Spark application

In this part, you will implement a simple Spark application. We have provided some sample data collected by IOT devices at http://pages.cs.wisc.edu/~shivaram/cs744-fa18/assets/export.csv (http://pages.cs.wisc.edu/~shivaram/cs744-fa18/assets/export.csv). You need to sort the data firstly by the country code alphabetically (the third column) then by the timestamp (the last column). Here is an example:

Input:

… cca2	**… device_id**	**… timestamp**
… US	… 1	… 1
… IN	… 2	… 2

... cca2	... device_id	... timestamp
... US	... 3	... 2
... CN	... 4	... 4
... US	... 5	... 3
... IN	... 6	... 1

Output:

... cca2	... device_id	... timestamp
... CN	... 4	... 4
... IN	... 6	... 1
... IN	... 2	... 2
... US	... 1	... 1
... US	... 3	... 2
... US	... 5	... 3

You should first load the data into HDFS. Then, write a Spark program in Java/Python/Scala to sort the data. Examples of self-contained applications in all of those languages are given here (https://spark.apache.org/docs/latest/quick-start.html#self-contained-applications).

We suggest you also go through the RDD programming guide (https://spark.apache.org/docs/latest/rdd-programming-guide.html). Resilient Distributed Datasets (RDD) is the main abstraction in Spark which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. In our case you will be creating RDD object from the data that you load from HDFS. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations (not necessary to do for this part of the assignment, but will need to do it in part 3). Finally, RDDs automatically recover from node failures.

An example of couple commands if you are using PySpark (Python API that supports Spark) that should be handy.

```
from pyspark import SparkContext, SparkConf

# The first thing that Spark application does is creating an object SparkContext.
conf = SparkConf().setAppName(appName).setMaster(master)
sc = SparkContext(conf=conf)

# You can read the data from file into RDD object as a collection of lines
lines = sc.textFile("data.txt")
```

After loading data you can apply RDD operations on it. Read more about transformations and actions here (https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-operations).

```
lineLengths = lines.map(lambda s: len(s))
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

In order to run your Spark application you need to submit it using spark-submit script from Spark's bin directory. More details on submitting applications could be found here (https://spark.apache.org/docs/latest/submitting-applications.html).

Finally, your application should output the results into HDFS in form of csv. It should take in two arguments, the first a path to the input file and the second the path to the output file. Note that if two data tuples have the same country code and timestamp, the order of them does not matter.

Part 3: PageRank

In this part, you will need to implement the PageRank (https://en.wikipedia.org/wiki/PageRank#Simplified_algorithm) algorithm, which is an algorithm used by search engines like Google to evaluate the quality of links to a webpage. The algorithm can be summarized as follows:

1. Set initial rank of each page to be 1.
2. On each iteration, each page contributes to its neighbors by rank(p)/ # of neighbors.
3. Update each page's rank to be 0.15 + 0.85 * (sum of contributions).
4. Go to next iteration.

In this assignment, we will run the algorithm on two data sets. Berkeley-Stanford web graph (https://snap.stanford.edu/data/web-BerkStan.html) is a smaller data set to help you test your algorithm and enwiki-20180601-pages-articles (we have already put it to path `/proj/uwmadison744-f19-PG0/data-part3/enwiki-pages-articles/`) is a larger one to help you better understand the performance of Spark. Each line in the data set consists of a page and one of its neighbors. You need to copy them to HDFS first. In this assignment, always run the algorithm for a total of 10 iterations.

Task 1. Write a Scala/Python/Java Spark application that implements the PageRank algorithm.

Task 2. Add appropriate custom RDD partitioning and see what changes. Every RDD has an optional Partitioner object. Any shuffle operation on an RDD with a Partitioner will respec that Partitioner. Any shuffle operation on two RDDs (e.g., join) will take on the Partitioner of one of them, if one is set. You can control the partitioner by using the partitionBy(…) command. Also pay close attention to what operations preserve the partitioner and what dont (see comment on flatMap and flatMapValues (http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD.flatMap))

Task 3. Persist the appropriate RDD as in-memory objects and see what changes. Read about RDD persistence (https://spark.apache.org/docs/latest/rdd-programming-guide.html#rdd-persistence).

Task 4. Kill a Worker process and see the changes. You should trigger the failure to a desired worker VM when the application reaches 25% and 75% of its lifetime:

1. Clear the memory cache using `sudo sh -c "sync; echo 3 > /proc/sys/vm/drop_caches"` .
2. Kill the Worker process.

With respect to Task 1-4, in your report you should report the application completion time. Present / reason about the difference in performance or your own findings, if any. Take a look at the lineage graphs of applications from Spark UI, or investigate into the log to find the amount of network/storage read/write bandwidth and number of tasks the number of tasks for every execution may help you better understand the performance issues.

Deliverables

You should submit a tar.gz file to Canvas, which consists of a brief report (filename: groupx.pdf) and the code of each task (you will be put into your groups on canvas so only 1 person should need to submit it). Put the code of each part and each task into separate folders give them meaningful names. Code should be commented well (that will be worth some percentage of your grade for the assignment, grader will be looking at your code). Also put a README file for each task and provide the instructions about how to run your code. Include a `run.sh` script for each part of the assignment that can re-execute your code on a similar CloudLab cluster assuming that Hadoop and Spark are present in the same location.

Acknowledgements

This assignment uses insights from Professor Aditya Akella's assignment 1 of CS744 Fall 2017 fall and Professor Mosharaf Chowdhury's assignment 1 of ECE598 Fall 2017.