Parallel Computing - Final assignment - n-gram generation with Hadoop

Ubaldo Puocci
ubaldo.puocci@stud.unifi.it

Abstract

In this work we will show how to implement the generation of n-grams in a parallel and distributed way using the Hadoop framework and what optimizations can be used to further improve our approach.

1. Introduction

An n-gram is a contiguous sequence of n items from a given sample of text or speech. In our case, the given text from which we compute our n-grams is made out of the top 50 plain text version of the most downloaded books on Project Gutenberg as of the 27th of January 2020. The general idea of the algorithm is the following:

1. Load each text file
2. Clean the text file:
   (a) Remove all punctuation, non-text and non-number characters;
   (b) Remove all numbers;
   (c) If the file is being cleaned for the word n-grams generation, also remove the stopwords in the text;
3. Generate word or letter n-grams and print them to a file.

2. Implementation of the algorithm

The n-grams generation per se is achieved with two simple methods that take the same type and number of arguments: `List<String> ngrams(int n, String str)` and `List<List<String>> ngrams(int n, String str)`. The former generates n-grams of letters, while the latter generates n-grams of words. For example, regarding the first function, if we have $n = 3$ and `str=parallel`, then the output would be:

- par, ara, ral, all, lle, lel.

On the contrary, if we have $n = 2$ and `str=hi hadoop hello hadoop` for the second method, the output would be:

- hi hadoop;
- hadoop hello;
- hello hadoop.

3. Hadoop and HDFS

The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. It is designed to scale up from single servers to thousands of machines, each offering local computation and storage. Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of computers, each of which may be prone to failures.

HDFS stands for Hadoop Distributed File System and is where all our input and output files are stored. As the name suggests, it is best suited for storing large amounts of data in a distributed manner. That is not our case, since we are running a

local installation with no data replication in place because we are running on only one DataNode.

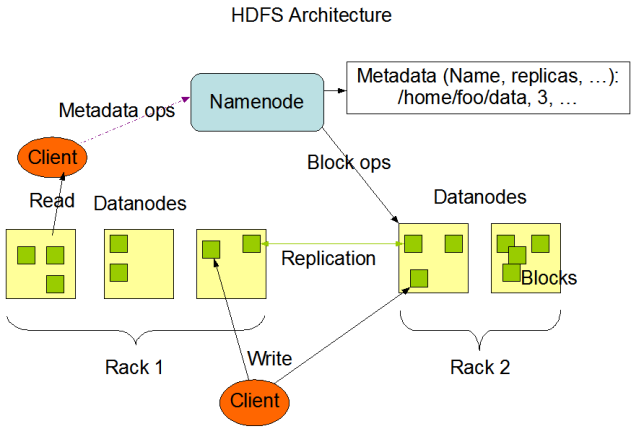

Figure 1. HDFS architecture

In HDFS, we have mainly three important directories related to our work:

- the directory where the inputs are stored;
- the two directories where the outputs are stored.

4. MapReduce and its implementation

Hadoop MapReduce is a software framework for easily writing applications which process vast amounts of data in-parallel on large clusters in a reliable, fault-tolerant manner.

A MapReduce job usually splits the input dataset into independent chunks which are processed by the Mapper(s) in a completely parallel manner. The output is then given as an input to a Reducer which usually writes its output to the HDFS. The user needs to specify only the location of the inputs and outputs, and the necessary Map and Reduce classes that need to extend or implement specified interfaces and/or abstract classes. In our work, all of our Mapper classes extend the `Mapper<K, V, K1, V1>` class, and our Reducer extends the `Reducer<K1, V1, K2, V2>` class.

The MapReduce framework operates exclusively on `<key, value>` pairs, that is, the framework views the input to the job as a set of `<key, value>` pairs and produces a set of `<key, value>` pairs as the output of the job.

As we aimed for simplicity and because we had to compute n-grams of both letters and words, we designed two different MapReduce jobs consisting of two Mappers and one (shared) Reducer each.

4.1. Letters n-gram generation

The job that generates n-grams of letters consists of two Mappers and one Reducer. The first Mapper takes as input the `<Object, Text>` pair that represent a logical portion of the file being analyzed by that particular Mapper and its contents. The first thing that our job does, is parse and clean all the files from unnecessary junk: all splits of data are analyzed and all characters that are either a number or a special character are removed. Then, as an additional layer of parsing and to ensure that we do not miss any special character, only the strings that pass the `\\w+` regex are kept. Now the Mapper is ready to produce its output: a set of `<Text, 1>` pair is created, each one representing a word in the text.

The second Mapper takes as input each `<Text, 1>` pair and computes each letter n-gram for the key. This creates a list of strings that are then individually added as `<Text, 1>` pair to the output. Finally, the Reducer can be executed. Its job is to simply aggregate the output of our latest Mapper and to output a `<Text, IntWritable>` pair that represents an histogram-like set of values:

Text	IntWritable
...	...
borr	134
bors	156
bort	13
...	...

Table 1. Example subset of the first MapReduce job output

4.2. Words n-gram generation

Similarly to our first job, the one that generates n-grams of words also consists of two Mappers and one Reducer. The underlying idea is very similar, but there are several key differences.

First, the first Mapper takes as input the same <Object, Text> pair as the first one to parse and clean all files from unnecessary junk, but instead of eliminating only characters, it can optionally also remove stopwords from the texts. Since we do not want to split the text into individual words here, our output will still be a <Text, 1> pair, but the key is now referencing a whole portion of the input data.

The second Mapper of our job is now ready to take its input as <Text, 1> pairs. What it does is, given a text as key value, compute all possible n-gram of words and put them into a multi-dimensional list of strings. This list is then iterated upon and each of its element is creating a <Text, 1> output pair, where the key is now our final computed n-gram. Finally, we execute exactly the same Reducer as the one executed in the first job to get a <Text, IntWritable> pair that represents, like before, an histogram-like set of values:

Text	IntWritable
...	...
abandon moscow	5
abandon need	1
abandoned attempt	2
...	...

Table 2. Example subset of the first MapReduce job output

4.3. MapReduce optimizations: the Combiner

One of the main ways our work has been further optimized is through the use of a Combiner class. A Combiner is a Reducer-like class that can aggregate the <key, value> pairs of each Mapper before giving them as input to the Reducer. In our specific cases we can easily suppose that the use of a Reducer is recommended in our first job: in fact, we can have some of the words and letters n-gram (depending after which Mapper the Combiner gets executed) already aggregated Mapper-wise before going through the Reducer. On the contrary, a Reducer is mostly useless in our second job when executed after the first mapper because it is highly unlikely that two portion of the same (or different) books are completely equal and thus eligible for aggregation. A combiner is surely most helpful after the second Mapper executed in our second job, for the same reason aforementioned for our first job.

5. Some results

Here are shown the top 10 words and letters n-grams with $n = 2$ for the former and $n = 4$ for the latter.

n-gram	count
that	55286
ther	44057
with	44041
tion	32421
here	28394
ould	27464
ight	24295
thin	24282
hing	21273
have	21014

Table 3. Top 10 letters n-grams, $n = 4$

n-gram	count
prince andrew	902
tm electronic	691
electronic works	652
one another	583
said mr	578
archive foundation	569
old man	547
literary archive	539
electronic work	517
could see	512

Table 4. Top 10 words n-grams, $n = 2$

6. Conclusion and further optimizations

What we have seen in this work is how we managed to implement a typical workload into a MapReduce job running on Hadoop, providing a parallel approach due to the native framework design. Further improvements can be made regarding the whole infrastructure, the most important one being the possibility to deploy it on a AWS

Cluster to take advantage of the much higher computational power.

References

[1] *Apache Hadoop 3.1.3*. URL: https://hadoop.apache.org/docs/r3.1.3/.