EECS 247
Feb 2021

UCI

Indexing Data with MapReduce

An Example with Hadoop

prepared by
Arya Daroui

1. Abstract

With the proliferation of the Internet there has come massive amounts of data that different organizations and entities store for different services. As these datatabases grow, the more cumbersome it becomes to actually find and read the data stored within. A common solution to this is to index important subdata beforehand for easier lookup. Fortunately, the nature of indexing lends itself to being processed by MapReduce very well. We explore the benefits of indexing with a example video data using MapReduce with Hadoop.

2. Introduction

If you have ever performed a fresh installation of your operating system, you may notice that your computer is sluggish for a period of time after its first launch. In the background, the operating system is indexing your files so that if you search for a specific file in the future, it can return results much more quickly than scanning every single file in the system for every search. While scanning through data takes time and CPU cycles away from other tasks, consider the storage capacity of the typical home computer versus the storage of a commercial database. It should become clear that scanning through the many terabytes of data a single server rack can contain becomes problematic for throughput.

First, let us focus on how some of this data may be stored, so that we can learn how to index it. There are many structures and patterns for storing data objects, but the benefits of indexing is apparent in a hierarchal-graphical model. Consider a data object with various attributes, as shown in Figure 1. This is a logical way to organize various kinds of data, like user accounts or product details, but for this example, the data objects are videos on a streaming service, and the attributes are tags for a video's category or genre.

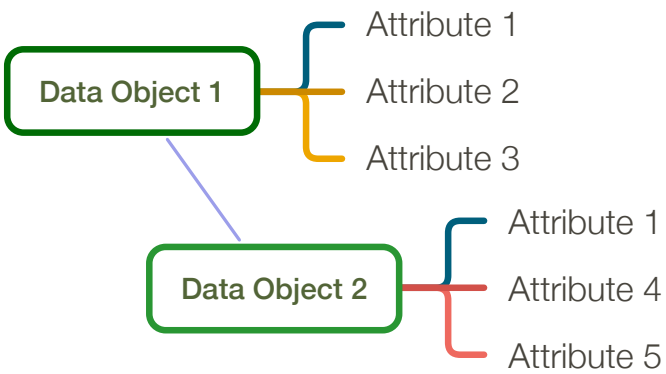

Figure 1. A Generic hierarchal-graphical database.

As new videos are uploaded to the streaming service, it is easy to add new nodes with its various attributes, and potentially even link certain nodes together for something like a playlist. But consider what would happen if a user submits a request to search for a specific video tag. Our server would have to go through the costly process of scanning every single node to determine if it has that tag attribute before returning to the user.

A simple solution that may come to mind is to preprocess the data and organize it by attribute—which is exactly what indexing is. We can create a sibling dataset that inverts the structure of the previous hierarchal model; the attributes are the root nodes, and the branches are all the objects with that attribute, as shown in Figure 2. Now, when a request comes in for a particular tag attribute, we can immediately jump to that node and return its children.

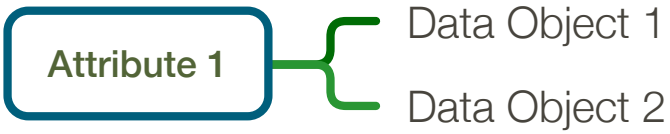

Figure 2. Inverting the hierarchal model, i.e. indexing.

3. Methodology

The heart of this example is the MapReduce algorithm. Indexing lends itself very well to MapReduce because the action of finding videos is highly parallelizable in that it does not have any interlinked data dependencies. In other words, each mapper can work independently on a partition of video objects and not have to worry about relevant data being stored on objects the other mapper is working on. Similarly, after sorting/shuffling, the reducers can go through each tag and add video branches to them independently.

3.1 MapReduce

Let us quickly look at the input and output at each stage of the MapReduce program so we can get an idea for how it works and our reason for using it. Consider the input data below.

```
vid0 tech entertainment history
vid1 health education entertainment
```

Our mapper takes each line and prints the video and its tag for each tag.

```
tech vid0
entertainment vid0
history vid0
health vid1
education vid1
entertainment vid1
```

The shuffler/sorter organizes the data so that each type does not get split between reducers.

```
education vid1
entertainment vid0
entertainment vid1
health vid1
```

```
history vid0
tech vid0
```

The reducer collapses all the videos to a single common tag, giving us our final output.

```
tech vid0
education vid1
entertainment vid0 vid 1
history vid0
health vid1
```

3.2 Setup

While I did find a publicly available YouTube dataset with tags, it only contained 3,813 videos, which was far too small. To get the most out of MapReduce, we want the input data to be large enough that there can be at least multiple mappers and hopefully multiple reducers. To do this, I created synthetic data with 5,000,000 videos, each with three tags. I was able to get this synthetic data to 172 MB, which is important because since Hadoop 2.4, the default input block size is 128 MB; so we should expect to have two mappers working on this job. Unless otherwise specified, the default number of reduce tasks is set to 1, which we will keep because we want a single output file. However, if we deployed this job across multiple server nodes, we would likely want more reducers.

For the experiment, I used the same Python mapper and reducer programs in three different scenarios for comparison. First, I used piping for `std I/O`. Piping the programs is pseudo-parallel, because while the programs all start at the same time, they wait for input from the previous stages. Next, I piped the mapper reducer programs again, but this time I sorted in-between. This is not important in the final output because the single reduce program is agnostic to order, but it provides a better comparison because MapReduce inherently sorts. Finally, the Hadoop MapReduce job which used the same mapper and reducer programs as before. These were all run on my quad-core i7 Macintosh.

To get the Python programs to run with Hadoop, we must use Hadoop's Streaming API. The shell commands for both Hadoop's API and what was used for the generic shell execution are all in the Appendix, along with their various outputs. The final output is omitted because it is a 162 MB text file that my computer struggles to even open.

4. Data and Discussion

In Table 1, we can see the performance of the experiment outlined previously. Again, note that the piped programs run pseudo-parallel, so each stage has the same start time; therefore, the total time is the end time of the last stage. Additionally, while Hadoop's Streaming API does log job metrics, it does not log the exact time stages start and stop, so those have been omitted from the table as well.

Looking over the log from the Hadoop job, we had two mappers and one reducer task running, which our expectations we outlined in our methodology. However, what is effectively just a single additonal mapping task significantly increased performance. The Hadoop MapReduce job finished 8 seconds faster, than the normal piped job, and 55 seconds faster than the piped job with sorting—keep in mind that the MapReduce job has mandatory sorting even though it does not need it for this example. Living in an age where web services try to cut down response time for their services to the milliseconds, an 8 second increase in index performance for a 200 MB input is excellent.

Table 1. Indexing 5 million videos with piping, piping with sorting, and with MapReduce. Times mark the end time for each stage; processes are parallel so they share the same start time.

	Pipe (s)	**Pipe & Sort** (s)	**MapReduce** (s)
Head	10.4	9.1	-
Mapper	10.4	9.1	-
Sort	-	65.2	-
Reducer	47.6	94.7	-
Total time	*47.6*	*94.7*	*39.6*

5. Conclusion

What this report has shown so far is the problem of data access in large databases, a solution in preprocessing and indexing important data beforehand, and implementing this solution in various configurations for comparison. The main configurations being a pseudo-parallel mapping and reducing execution and a true Hadoop MapReduce, multi-task execution that yielded greater results. This was performed on a consumer-grade computer, so our remaining assumption is that a database would be stored on servers with multi-node processing capability already, and because indexing is now shown to be very MapReduce friendly, the job would be even more efficient with more mapper and reducer tasks.

Appendix

Code

makedata.py

```
from random import sample

tags = ('sports', 'gaming', 'news', 'history', 'tech', 'finance', 'fashion', 'health', 'education', '
    entertainment')
numVideos = 5000000

with open("videotagssmall.txt", 'w') as file:
    for vidIndex in range(numVideos):
        tagIndices = sample(range(0,10), 3)
        file.write(' '.join(['vid', str(vidIndex), tags[tagIndices[0]], tags[tagIndices[1]], tags[tagIndices
            [2]], '\n']))
```

mapper.py

```
import sys

for line in sys.stdin:
    line = line.strip()
    items = line.split()

    video = items.pop(0)
    for tag in items:
        print( "%s\t%s" % (tag, video) )
```

reducer.py

```
import sys

tags = {}

for line in sys.stdin:
    line = line.strip()
    tag, video = line.split('\t',1)

    if tag in tags:
        tags[tag].append(video)
    else:
        tags[tag] = [video]

for key in tags:
    print(key, end='\t')
    for video in tags[key]:
        print(video, end='\t')
    print('\n', end='')
```

Shell commands

Parallel piping

The `sort` and its pipe can be removed if desired.

```
time head -n5000000 videotagsbig.txt | python mapper.py | sort | python reducer.py
```

Hadoop Streaming

References local python files and input/output files stored on the HDFS partition.

```
hadoop jar /usr/local/Cellar/hadoop/3.3.0/libexec/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-file "mapper.py" \
-mapper "/Users/aryadaroui/miniconda3/bin/python3 mapper.py" \
-file "reducer.py" \
-reducer "/Users/aryadaroui/miniconda3/bin/python3 reducer.py" \
-input videotagsbig.txt \
-output Hadoopoutput
```

Shell screenshots

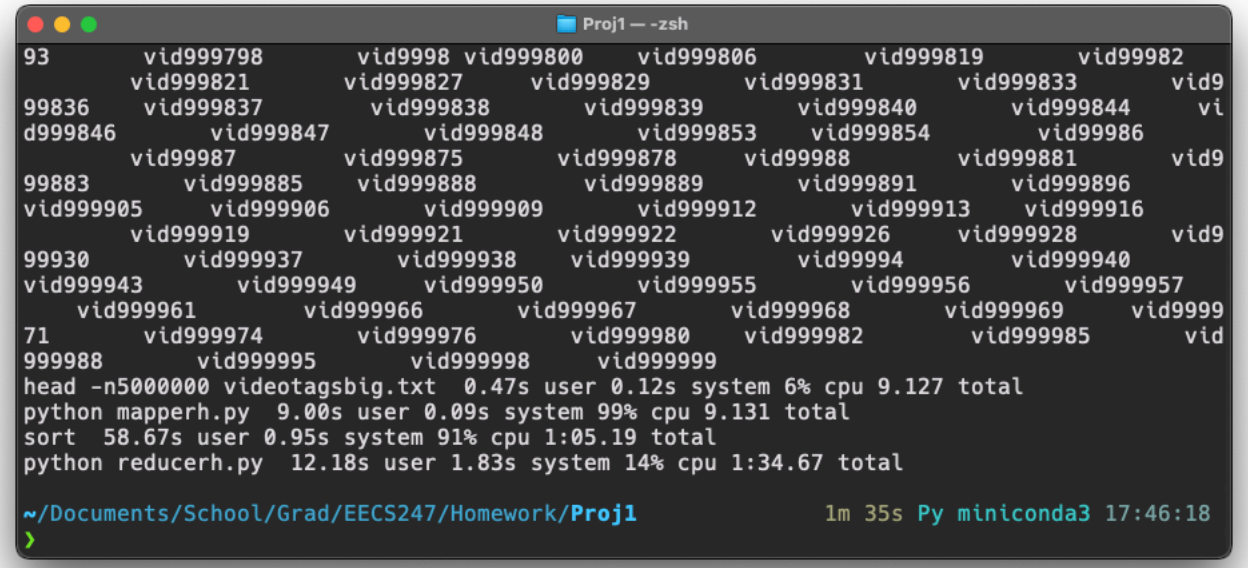

Figure 3. Timed output of parallel piping with sorting.

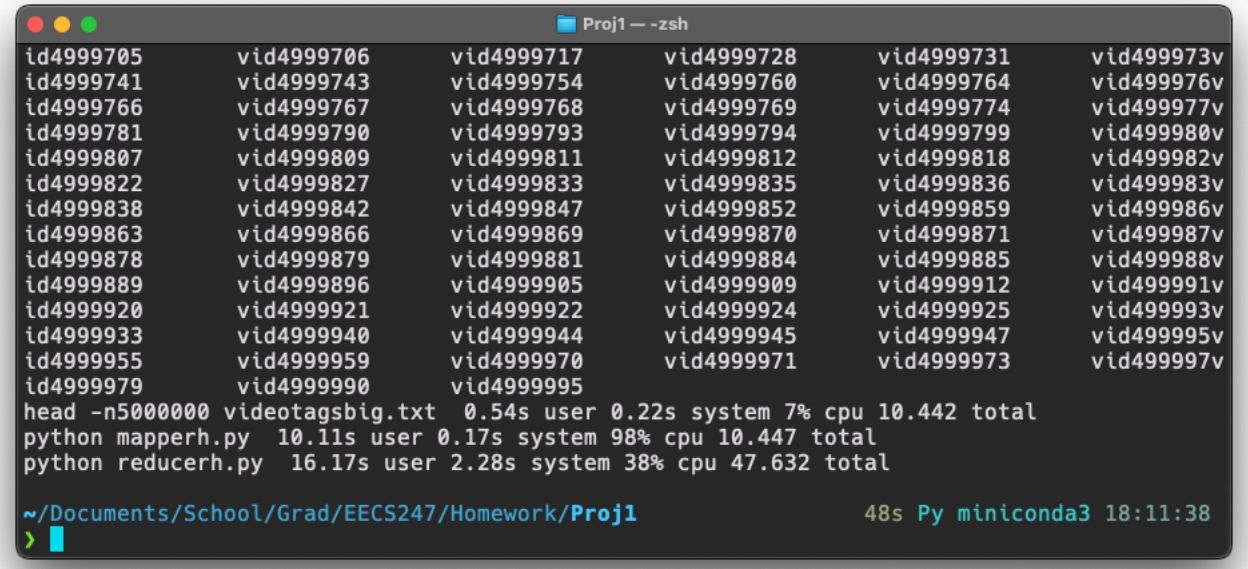

Figure 4. Timed output of parallel piping withpout sorting.

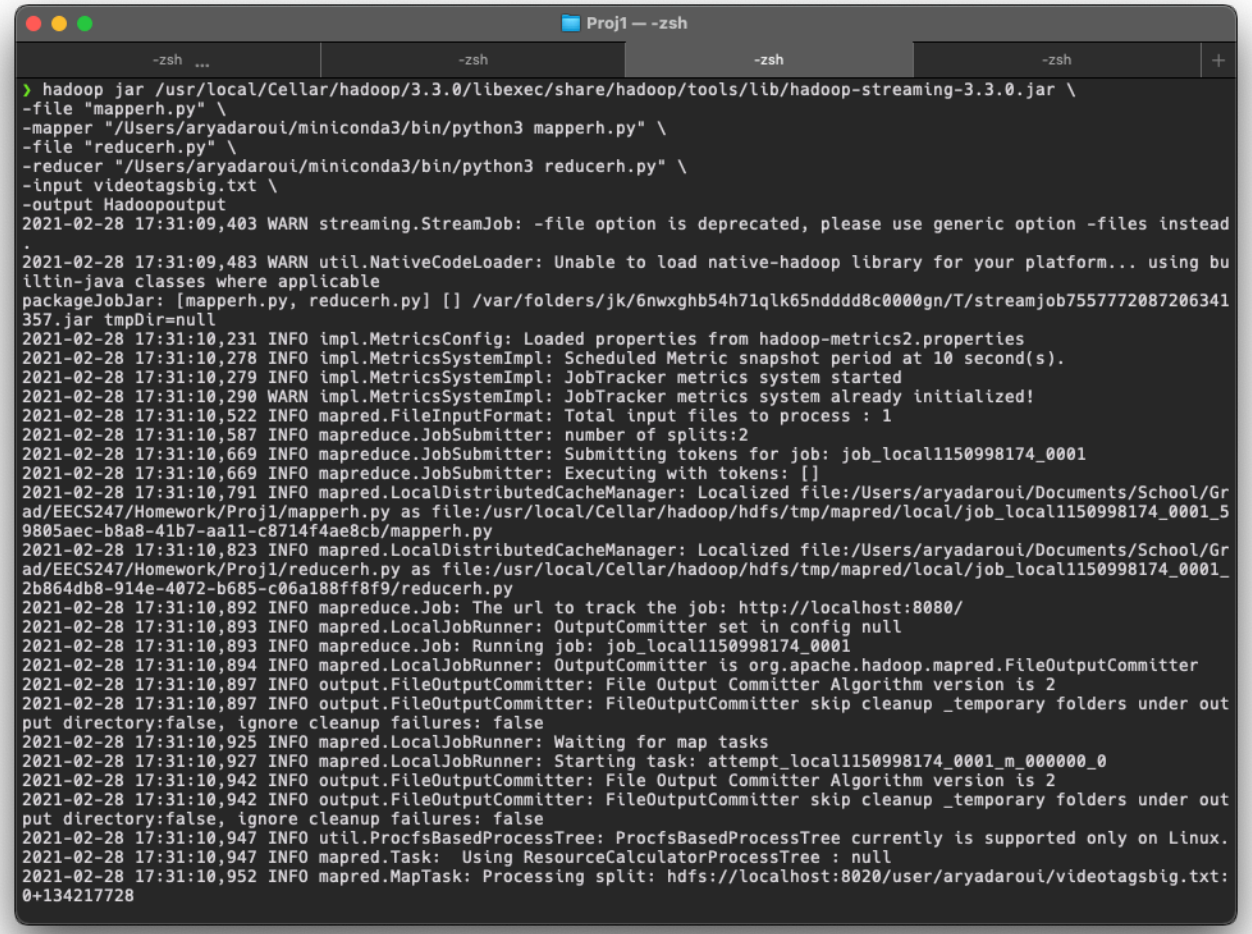

Figure 5. Start of the Hadoop streaming job's output.

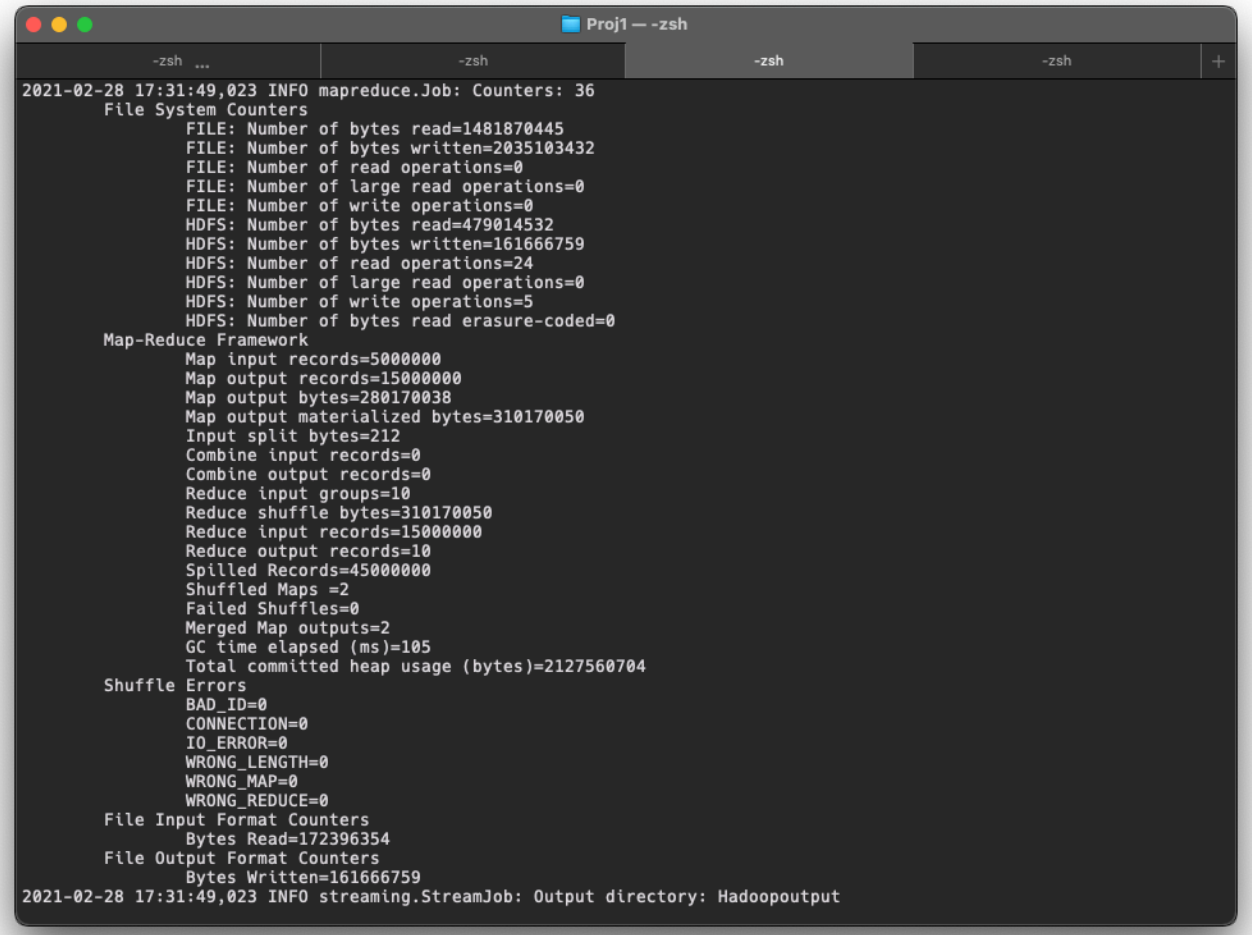

Figure 6. End of the Hadoop streaming job's output.