

UCI

# Indexing Data with MapReduce

---

## An Example with Hadoop



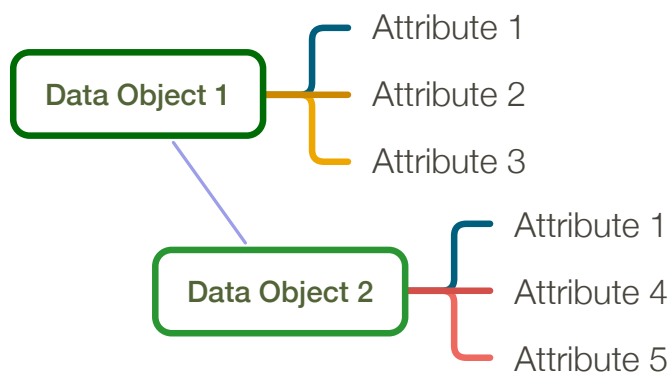
## 1. Abstract

With the proliferation of the Internet there has come massive amounts of data that different organizations and entities store for different services. As these databases 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 an 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.

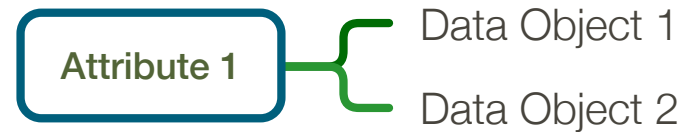


**Figure 1.** A Generic hierarchal-graphical database.

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 additional 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)
<b>Head</b>	10.4	9.1	-
<b>Mapper</b>	10.4	9.1	-
<b>Sort</b>	-	65.2	-
<b>Reducer</b>	47.6	94.7	-
<b>Total time</b>	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

```

1 from random import sample
2
3 tags = ('sports', 'gaming', 'news', 'history', 'tech', 'finance', 'fashion', 'health', 'education', '
    entertainment')
4 numVideos = 5000000
5
6 with open("videotagssmall.txt", 'w') as file:
7     for vidIndex in range(numVideos):
8         tagIndices = sample(range(0,10), 3)
9         file.write(' '.join(['vid', str(vidIndex), tags[tagIndices[0]], tags[tagIndices[1]], tags[tagIndices
            [2]], '\n']))

```

mapper.py

```

1 import sys
2
3 for line in sys.stdin:
4     line = line.strip()
5     items = line.split()
6
7     video = items.pop(0)
8     for tag in items:
9         print( "%s\t%s" % (tag, video) )

```

reducer.py

```

1 import sys
2
3 tags = {}
4
5 for line in sys.stdin:
6     line = line.strip()
7     tag, video = line.split('\t',1)
8
9     if tag in tags:
10         tags[tag].append(video)
11     else:
12         tags[tag] = [video]
13
14 for key in tags:
15     print(key, end='\t')
16     for video in tags[key]:
17         print(video, end='\t')
18     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.

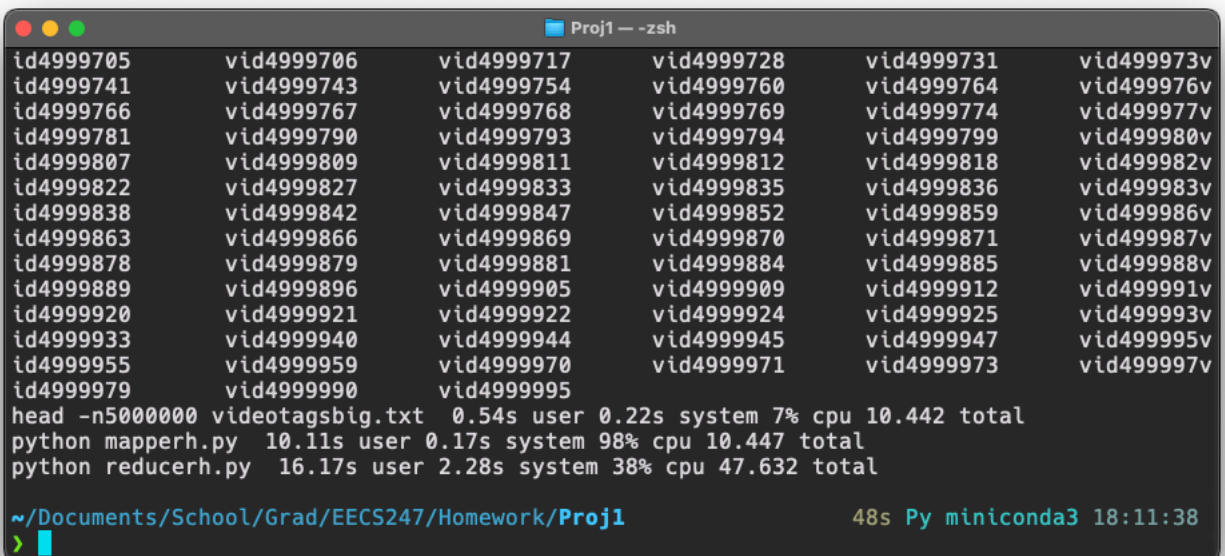
```
hadoop jar /usr/local/Cellar/hadoop/3.3.0/libexec/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-file "mapper.py" \
-mapper "/Users/aryadaroui/miniconda3/bin/python3 mapper.py" \
-file "reducer.py" \
-reducer "/Users/aryadaroui/miniconda3/bin/python3 reducer.py" \
-input videotagsbig.txt \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

### Shell screenshots

```
Proj1 - zsh
93      vid999798      vid9998      vid999800      vid999806      vid999819      vid99982
      vid999821      vid999827      vid999829      vid999831      vid999833      vid9
99836      vid999837      vid999838      vid999839      vid999840      vid999844      vi
d999846      vid999847      vid999848      vid999853      vid999854      vid99986
      vid99987      vid999875      vid999878      vid99988      vid999881      vid9
99883      vid999885      vid999888      vid999889      vid999891      vid999896
vid999905      vid999906      vid999909      vid999912      vid999913      vid999916
      vid999919      vid999921      vid999922      vid999926      vid999928      vid9
99930      vid999937      vid999938      vid999939      vid99994      vid999940      vid9
vid999943      vid999949      vid999950      vid999955      vid999956      vid999957
      vid999961      vid999966      vid999967      vid999968      vid999969      vid9999
71      vid999974      vid999976      vid999980      vid999982      vid999985      vid
999988      vid999995      vid999998      vid999999
head -n5000000 videotagsbig.txt 0.47s user 0.12s system 6% cpu 9.127 total
python mapperh.py 9.00s user 0.09s system 99% cpu 9.131 total
sort 58.67s user 0.95s system 91% cpu 1:05.19 total
python reducerh.py 12.18s user 1.83s system 14% cpu 1:34.67 total

~/Documents/School/Grad/EECS247/Homework/Proj1 1m 35s Py miniconda3 17:46:18
>
```

**Figure 3.** Timed output of parallel piping with sorting.



```
Proj1 -- -zsh
id4999705      vid4999706      vid4999717      vid4999728      vid4999731      vid499973v
id4999741      vid4999743      vid4999754      vid4999760      vid4999764      vid499976v
id4999766      vid4999767      vid4999768      vid4999769      vid4999774      vid499977v
id4999781      vid4999790      vid4999793      vid4999794      vid4999799      vid499980v
id4999807      vid4999809      vid4999811      vid4999812      vid4999818      vid499982v
id4999822      vid4999827      vid4999833      vid4999835      vid4999836      vid499983v
id4999838      vid4999842      vid4999847      vid4999852      vid4999859      vid499986v
id4999863      vid4999866      vid4999869      vid4999870      vid4999871      vid499987v
id4999878      vid4999879      vid4999881      vid4999884      vid4999885      vid499988v
id4999889      vid4999896      vid4999905      vid4999909      vid4999912      vid499991v
id4999920      vid4999921      vid4999922      vid4999924      vid4999925      vid499993v
id4999933      vid4999940      vid4999944      vid4999945      vid4999947      vid499995v
id4999955      vid4999959      vid4999970      vid4999971      vid4999973      vid499997v
id4999979      vid4999990      vid4999995
head -n5000000 videotagsbig.txt 0.54s user 0.22s system 7% cpu 10.442 total
python mapperh.py 10.11s user 0.17s system 98% cpu 10.447 total
python reducerh.py 16.17s user 2.28s system 38% cpu 47.632 total

~/Documents/School/Grad/EECS247/Homework/Proj1 48s Py miniconda3 18:11:38
>
```

**Figure 4.** Timed output of parallel piping without sorting.

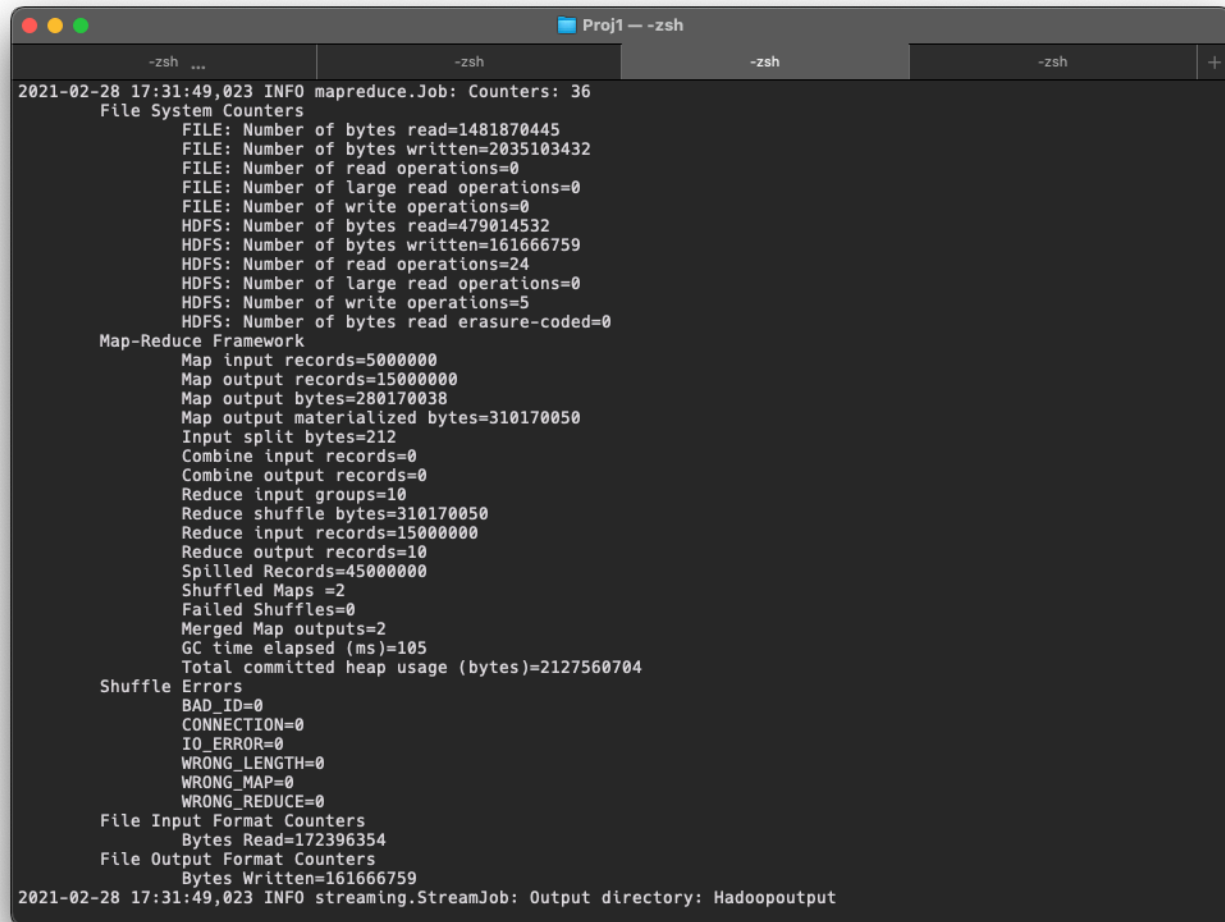


```

> hadoop jar /usr/local/Cellar/hadoop/3.3.0/libexec/share/hadoop/tools/lib/hadoop-streaming-3.3.0.jar \
-file "mapperh.py" \
-mapper "/Users/aryadaroui/miniconda3/bin/python3 mapperh.py" \
-file "reducerh.py" \
-reducer "/Users/aryadaroui/miniconda3/bin/python3 reducerh.py" \
-input videotagsbig.txt \
-output Hadoopoutput
2021-02-28 17:31:09,403 WARN streaming.StreamJob: -file option is deprecated, please use generic option -files instead
2021-02-28 17:31:09,483 WARN util.NativeCodeLoader: Unable to load native-hadoop library for your platform... using bu
iltin-java classes where applicable
packageJobJar: [mapperh.py, reducerh.py] [] /var/folders/jk/6nwqghb54h7lqlk65ndddd8c0000gn/T/streamjob7557772087206341
357.jar tmpDir=null
2021-02-28 17:31:10,231 INFO impl.MetricsConfig: Loaded properties from hadoop-metrics2.properties
2021-02-28 17:31:10,278 INFO impl.MetricsSystemImpl: Scheduled Metric snapshot period at 10 second(s).
2021-02-28 17:31:10,279 INFO impl.MetricsSystemImpl: JobTracker metrics system started
2021-02-28 17:31:10,290 WARN impl.MetricsSystemImpl: JobTracker metrics system already initialized!
2021-02-28 17:31:10,522 INFO mapred.FileInputFormat: Total input files to process : 1
2021-02-28 17:31:10,587 INFO mapreduce.JobSubmitter: number of splits:2
2021-02-28 17:31:10,669 INFO mapreduce.JobSubmitter: Submitting tokens for job: job_local1150998174_0001
2021-02-28 17:31:10,669 INFO mapreduce.JobSubmitter: Executing with tokens: []
2021-02-28 17:31:10,791 INFO mapred.LocalDistributedCacheManager: Localized file:/Users/aryadaroui/Documents/School/Gr
ad/EECS247/Homework/Proj1/mapperh.py as file:/usr/local/Cellar/hadoop/hdfs/tmp/mapred/local/job_local1150998174_0001_5
9805aec-b8a8-41b7-aa11-c8714f4ae8cb/mapperh.py
2021-02-28 17:31:10,823 INFO mapred.LocalDistributedCacheManager: Localized file:/Users/aryadaroui/Documents/School/Gr
ad/EECS247/Homework/Proj1/reducerh.py as file:/usr/local/Cellar/hadoop/hdfs/tmp/mapred/local/job_local1150998174_0001_
2b864db8-914e-4072-b685-c06a188ff8f9/reducerh.py
2021-02-28 17:31:10,892 INFO mapreduce.Job: The url to track the job: http://localhost:8080/
2021-02-28 17:31:10,893 INFO mapred.LocalJobRunner: OutputCommitter set in config null
2021-02-28 17:31:10,893 INFO mapreduce.Job: Running job: job_local1150998174_0001
2021-02-28 17:31:10,894 INFO mapred.LocalJobRunner: OutputCommitter is org.apache.hadoop.mapred.FileOutputCommitter
2021-02-28 17:31:10,897 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2021-02-28 17:31:10,897 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under out
put directory:false, ignore cleanup failures: false
2021-02-28 17:31:10,925 INFO mapred.LocalJobRunner: Waiting for map tasks
2021-02-28 17:31:10,927 INFO mapred.LocalJobRunner: Starting task: attempt_local1150998174_0001_m_000000_0
2021-02-28 17:31:10,942 INFO output.FileOutputCommitter: File Output Committer Algorithm version is 2
2021-02-28 17:31:10,942 INFO output.FileOutputCommitter: FileOutputCommitter skip cleanup _temporary folders under out
put directory:false, ignore cleanup failures: false
2021-02-28 17:31:10,947 INFO util.ProcfsBasedProcessTree: ProcfsBasedProcessTree currently is supported only on Linux.
2021-02-28 17:31:10,947 INFO mapred.Task: Using ResourceCalculatorProcessTree : null
2021-02-28 17:31:10,952 INFO mapred.MapTask: Processing split: hdfs://localhost:8020/user/aryadaroui/videotagsbig.txt:
0+134217728

```

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



```
2021-02-28 17:31:49,023 INFO mapreduce.Job: Counters: 36
  File System Counters
    FILE: Number of bytes read=1481870445
    FILE: Number of bytes written=2035103432
    FILE: Number of read operations=0
    FILE: Number of large read operations=0
    FILE: Number of write operations=0
    HDFS: Number of bytes read=479014532
    HDFS: Number of bytes written=161666759
    HDFS: Number of read operations=24
    HDFS: Number of large read operations=0
    HDFS: Number of write operations=5
    HDFS: Number of bytes read erasure-coded=0
  Map-Reduce Framework
    Map input records=5000000
    Map output records=15000000
    Map output bytes=280170038
    Map output materialized bytes=310170050
    Input split bytes=212
    Combine input records=0
    Combine output records=0
    Reduce input groups=10
    Reduce shuffle bytes=310170050
    Reduce input records=15000000
    Reduce output records=10
    Spilled Records=4500000
    Shuffled Maps =2
    Failed Shuffles=0
    Merged Map outputs=2
    GC time elapsed (ms)=105
    Total committed heap usage (bytes)=2127560704
  Shuffle Errors
    BAD_ID=0
    CONNECTION=0
    IO_ERROR=0
    WRONG_LENGTH=0
    WRONG_MAP=0
    WRONG_REDUCE=0
  File Input Format Counters
    Bytes Read=172396354
  File Output Format Counters
    Bytes Written=161666759
2021-02-28 17:31:49,023 INFO streaming.StreamJob: Output directory: Hadoopoutput
```

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