Large Datasets for Scientific Applications - Project

Authors:

Armin Haskovic
Armin.Haskovic.3165@student.uu.se

Julia Lundgren
Julia.Lundgren.6470@student.uu.se

Joseph Rogers
Joseph.Rogers.7397@student.uu.se

Adam Schön
Adam.Schon.6129@student.uu.se

Uppsala University

June 4, 2017

Abstract

During this project music metadata from the Million Song Dataset have been analyzed through four different MapReduce jobs on a Hadoop Multi Node Cluster. Four different analyzes that were performed by different MapReduce jobs were carried out. The time it took to analyze the data were measured and compared when using one, two, three and four nodes. The algorithm was run on different sized data sets, namely 162MB, 1GB, 10GB, 25GB and 50GB. The result shows only a slight time difference, most likely because of the time it takes for Hadoop to read/write compared to the computational time of the algorithm.

1 Background

With data set growing ever larger older data processing algorithms have become almost useless when handling them. Instead new modern big data algorithms have emerged in the industry. These can be used to do large scale analysis on data sets. This kind analysis is use in many fields such as DNA decoding, Machine Optimization, Financial trading and more. [1]

Large scale analysis of musical trends can provide valuable insights for both researchers and commercial enterprises. Understanding what types of music people commonly listen to can be very useful when suggesting new content to them. Seeing large musical trends change over time can help companies focus their efforts on specific things. It can also signal them to stop worrying about certain groups as much if popularity becomes too low.

Music services such Spotify, Vevo and Rhapsody use music intelligence to give suggestions to their users of new content they might enjoy. News websites that have music editorials such as BBC, eMusic and MTV also benefit from music intelligence by able to see what's popular in common culture. According to The Echo Nest's website `http://the.echonest.com/`, they provide music intelligence to companies with total customer counts of over 100 million users a month.

2 Data format

Files from the dataset are provided using the .h5 format. There is one file for each song listed in the database. We converted all files to the .csv format. This was done so that they could be easily parsed using stdin with Python. This was convenient since all the files were now human readable and we could see each piece of discrete metadata ourselves. However, the data was unstructured with no easy way to index into specific parts of it. If we wanted information about a specific record, then we were faced with the daunting possibility of having to potentially scan the entirety of the database in order to find it.

Apart from the convenience of using files with .csv format, the size of the files drastically reduced. Files in the .h5 format were in the region of 100-400kb per file whilst the files in .csv format were in the region of 4kb per file. This is partly because we filter out the unnecessary values that we won't use so less information is stored in the .csv files, but also, using hdf5 for small files introduces a lot of overhead which makes the files larger than needed [2]. As we don't need the extra information that is stored in the original files, which lets us store smaller files so we won't need extra space, but also not limiting ourselves to using Java for convenience, we chose to go with the csv format.

Working with a lot of small files creates problems for the hadoop file system (hdfs) and also while using mapreduce. As we have one million songs where each song is stored in one file these problems become apparent. In hdfs, each file which is stored also contains metadata in the namenodes memory. Having lots of small files in hdfs would then use a lot of memory (about 150 bytes per file) which isn't feasible with millions of files [3]. For mapreduce, each file uses one mapper so if there are small files, each mapper only processes a small amount of information. Because of the large amount of small files used, the time to process instead becomes a lot larger than if one would process bigger files [3]. This is why we combined the csv files to larger files. Instead of storing each file in hdfs, we combined around 30000 files into each file to make it easier to store and process it with mapreduce.

3 Computational experiments

We processed the Million Song Database using MapReduce on a Hadoop Multi-Node cluster. We chose to use Hadoop because our desired analysis lent themselves well to batch type jobs where all of the data would need to be analyzed in order to give a conclusive answer. We chose four different experiments to see how music has changed over time. Each of these experiments correlates to a specific MapReduce job that we ran on our Hadoop Cluster. Different experiments varied in their level of computational intensity.

- Average Tempo of Songs by Year.
- Average Duration Songs by Year.
- Artist with highest total hotness by Year.
- Hottest time signature by Year.

The first two experiments for average tempo and average duration represent relatively simple MapReduce jobs where we just need to keep track of global averages on a year by year basis. Experiment 3 dealing with hotness is more complicated because it is relating multiple pieces of metadata too each other. That means that a record which might be parsed late into the job may still have relevance to a record we encountered earlier. Because of this we have to maintain a much more complicated runtime data-structure that slowly builds up over time as the job executes. At the end of the reduce phase this data-structure is parsed for the relevant information that we then output.

This data structure can be thought of as a 3D-array. The first dimension is for every year present in the database. The 2nd dimension represents each artist found within a particular year in the database. The 3rd and final dimension holds the hotness value for that particular artist that year. In this way, the depth of dimensions 1 and 3 is static by the depth of dimension 2 may very year to year because the total number of artists that we have records for each year may change over time. This is especially noticeable for the early years before about 1950. In the actual python code this structure was represented as an array of dictionaries for each year. Artist names were keys in the dictionary that returned hotness values for that year. Experiment 4 is similar to experiment 3. It requires use of the same type of data structure where each time signature for each year is continuously having its overall hotness updated. Eventually we choose the time signature with the highest total hotness for each year.

These four basic experiments were replicated on a variety of hardware setups to the effects on their performance scaling. Types of adjustments to the setup included...

- Adding Hadoop nodes to measure horizontal scaling.
- Increasing the size of the database.
- Changing the complexity of the Mapper/Reducer

4 Results and discussion

In this section we present the results of our experiments with some comments on them. We have a total of four graphs as well as a set of tables containing all of our numerical results. First is the results of our analysis of the data.

As can be seen in Figure 1 there is a slight increase in the mean song tempo until the mid-eighties and after that the mean tempo keeps a constant value. In Figure 2 we can see that the average song duration had a distinct increase in the late sixties/early seventies and have stayed rather constant since. Figure 3 is showing the summed hotness value for the artist with the highest value each year. The graph is showing increasingly higher values, which is mainly because there are more songs in the dataset from recent years. All three figures have a deviating point as a last value. This is because there are only a few songs available for the year 2011.

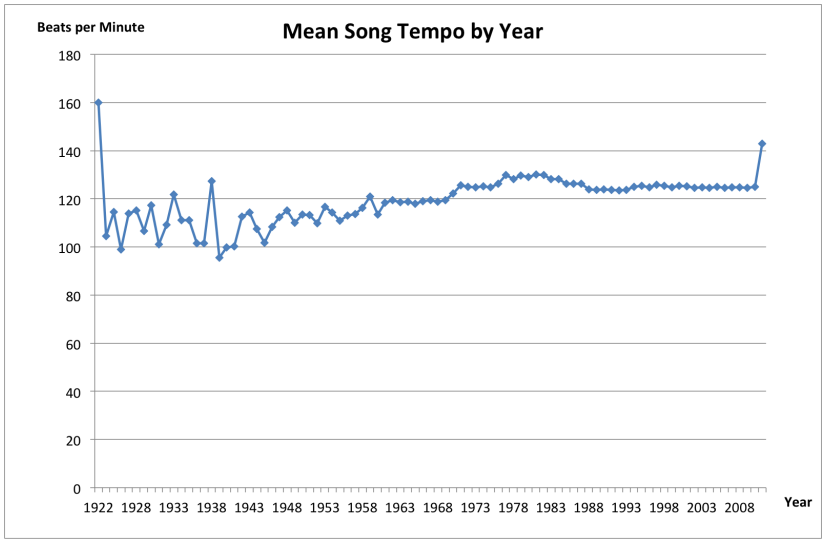

Figure 1: The mean song tempo by year for the Million song database.

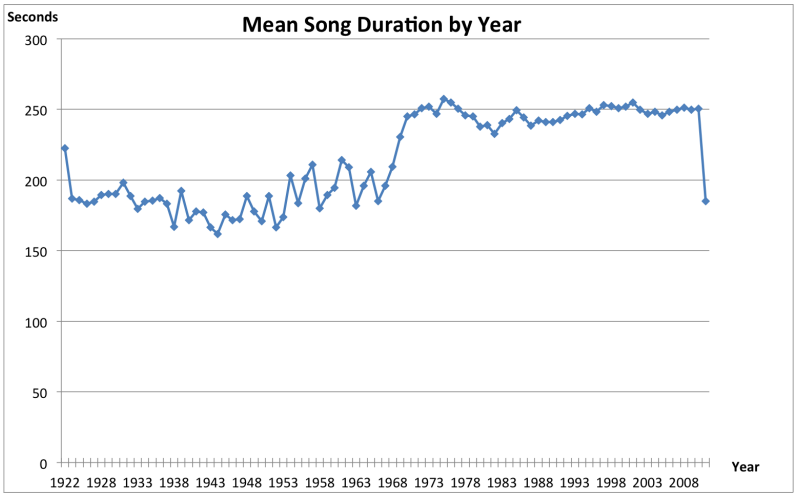

Figure 2: The mean song duration by year for the Million song database.

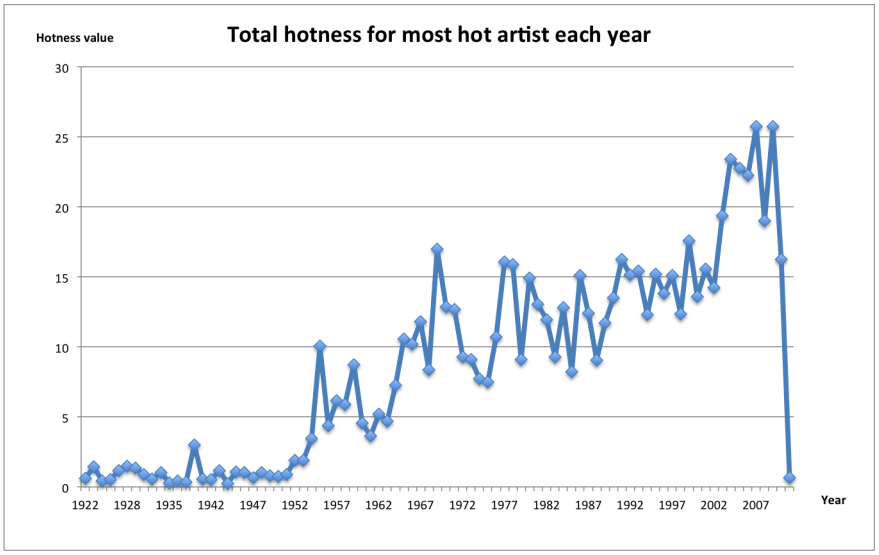

Figure 3: The artist with the highest total hotness by year for the Million song database.

Below you can see the speedup of the 50GB database as we add nodes to it for our four different MapReduce programs.4 We define the speedup for different node counts N to be equal to the execution time of the MapReduce jobs with one node divided by the parallel execution time of the job with multiple nodes.

$$S(N) = T_(1)/T_(N) \tag{1}$$

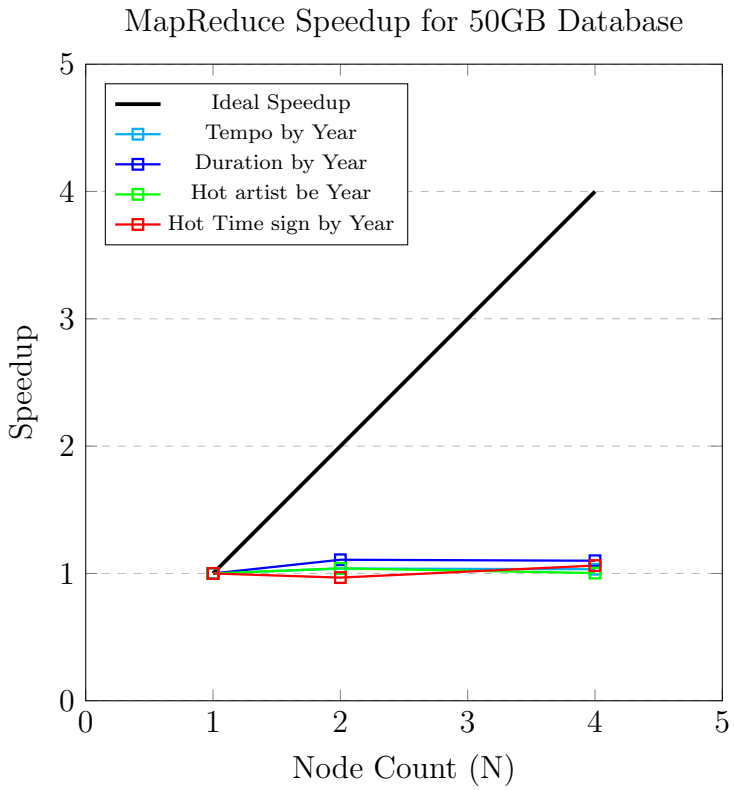

Figure 4: The speedup of MapReduce operations on the 50GB database for different reducers.

As can be seen in figure 4 the speedup when adding nodes is nowhere close the ideal speedup. This is most likely because the computational part of our analysis - which is mainly effected when adding more nodes - is not big enough to make a difference here. The main part of our analysis is therefor focused on reading/writing our files. If we would have used a analysis which would have required more computational power the results might have looked different and the increase of nodes would have shown to be more effective.

The Hadoop system is more efficient in terms of high throughput and not optimized for I/O performance, which is a known weakness for HDFS. This is because HDFS is using a write-once-read-many-times model, which is good for high throughput access but also leads to a poor write and read performance. During a MapReduce task the shuffling phase becomes a significant bottleneck in terms of performance. HDFS also suffers from an inefficient file accessing performance and since we stored our data in files its likely our results got affected by that. There are

methods for dealing with these problems but unfortunately we didn't have time to try any of them. One method that has been proven to reduce the I/O operations on the disk during the mapping phase in a MapReduce job is caching input data with a storage in-memory. Another method is to use a combiner that executes during the mapping phase. This ensures that the combiner is used and it reduces the amount of required shuffling of the intermediate results. To get around the file accessing performance problem it could have been a good idea to store the data in a database instead. One approach that has been shown successful is to use a NoSQL database both for input data and caching intermediate data. [4]

Table 1: Tempo

Nodes	162MB	1GB	10GB	25GB	50GB
1	28.133s	58.836s	7m49.301s	23m37.681s	41m21.558s
2	27.2s	1m8.914s	8m5.384s	19m47.774s	39m48.385s
4	24.830s	54.772s	7m24.795s	18m57.915s	40m1.965s

Table 2: Duration

Nodes	162MB	1GB	10GB	25GB	50GB
1	20.820s	49.702s	6m52.126s	18m20.154s	40m26.978s
2	21.823s	47.837s	8m22.170s	17m54.964s	36m30.859s
4	20.844s	50.737s	7m7.550s	19m26.725s	36m47.246s

Table 3: Hot Artist by Year

Nodes	162MB	1GB	10GB	25GB	50GB
1	39.743s	2m16.847s	19m27.602s	51m3.536s	100m40.709s
2	41.715s	2m8.965s	19m36.936s	49m46.972s	96m54.780s
4	41.703s	2m16.026s	19m51.064s	49m12.808s	100m20.346s

Table 4: Hot Time Sign by Year.

Nodes	162MB	1GB	10GB	25GB	50GB
1	15.179s	0m31.907s	4m29.196s	10m0.934s	21m21.791s
2	14.685	0m30.972s	4m25.498s	10m24.364s	22m5.457s
4	14.826s	0m29.965s	4m11.720s	10m13.343s	20m7.453s

References

[1] *Use of big data analytics*, https://www.ap-institute.com/big-data-articles/how-is-big-data-used-in-practice-10-use-cases-everyone-should-read.

[2] *Things that can affect performance*, https://support.hdfgroup.org/HDF5/faq/perfissues.html, Accessed: 2017-05-30.

[3] *The small files problem*, http://blog.cloudera.com/blog/2009/02/the-small-files-problem/, Accessed: 2017-05-30.

[4] *Hadoop mapreduce performance enhancement using in-node combiners e*, https://arxiv.org/pdf/1511.04861.pdf.