MapReduce With Columnar Storage

Peitsa Lähteenmäki

***Abstract*—The MapReduce programming paradigm has achieved more popularity over the last few years as an option to distributed database systems in large scale data processing. Though is has been sometimes criticized for hindering performance when used with certain tasks, and working well with only a few. Several reasons for this have been suggested with improvements to increase its overall effectiveness.**

In this article I take a look at these issues with MapReduce concentrating especially on one of the implementations of it: Hadoop. Special attention is also given to methods used with columnar databases and how those can be used to improve the performance of MapReduce tasks in Hadoop.

***Index Terms*—Column oriented, MapReduce, Hadoop.**

I. Introduction

As the amount of data collected on different systems increases year to year, the need for an effective method for its processing has become increasingly more important. One of such methods is the relatively new MapReduce programming paradigm [6]. Often compared to more traditional distributed database systems in terms of its effectivity, it is still functionally very different from them [2]. Where traditional database systems are usually based on the relational database model, MapReduce has very little in common with databases at all.

In the form MapReduce was originally described it was only directed towards certain limited tasks. Yet it was soon realized that it could be employed with many other tasks as well. With a more varied use base it became quite clear that MapReduce was not well suited for every task. In some cases this can be caused by the ineffective use of the paradigm, but there are certainly cases to which it is not suited for. Although even in such situations some steps that can be taken to improve its performance, yet even with large scale optimizations there are cases to which MapReduce is inherently ill suited for.

One way to implement programs, that take advantage of MapReduce, is to employ a programming framework which was designed for it. As the paradigm in itself does not provide anything expect the definition, some sort of a program/framework is required to take advantage of it. In fact this is the only way to use MapReduce in a way that makes sense, as otherwise one would have no way to actually run any of the MapReduce based program code. There currently quite many different MapReduce frameworks available, but in this article I will only concentrate on one of them: Hadoop. Though it has been sometimes criticised for being rather slow it has still become quite popular.

Most of this paper is directed towards introducing ways of making different programs, implemented with the Hadoop framework, run more efficiently. Most of the methods described here can also be used with other systems, and in fact some of them were originally designed for systems very different from MapReduce. This is especially true with the columnar methods described in chapter IV. Originally they were used with different database systems employing a columnar storage format, but were later transformed into a form compatible with MapReduce [4] [5].

I start by providing a short introduction to the MapReduce programming paradigm in chapter II. After this there is an explanation of the functionality of Hadoop, in relation to the topics discussed in this paper, in chapter III. Finally in chapter IV I introduce a few methods from columnar databases and how those can be used with Hadoop to improve performance.

II. MapReduce

MapReduce is a simple programming paradigm which attempts to make distributed processing of data easier. As originally described by Dean and Ghemawat [6] MapReduce achieves this functionality through the use of two functions: map and reduce.

The map function is defined in the same manner as its counterpart in functional languages. That is, it takes an input value and 'maps' that value to an output. It should be noted that both the input and output values can also be lists. The reduce function does not have an exactly similar counterpart in functional languages. Its purpose is to simply combine the values provided by the map function to an output value.

Simply defining these functions, of course, does not give an advantage as such. The power of this paradigm comes from the fact that distributing these functions across multiple processing nodes can be done with a single programming framework. That is, the framework simply takes the (user made) definitions for the map and reduce functions and automatically handles the distribution of them across the processing nodes. Because of this, by using this paradigm the distribution of the task performed is trivial and all programming effort can be directed to actual development.

Usually these functions are used together in such a manner that the map function produces a set of values from an input set to the reduce function. Reduce will then combine these intermediade values to an actual output result. It is not required that both functions would contain any actual functionality, and indeed one of them (or both) can be empty. Although this somewhat defeats the purpose of using MapReduce at all because the problem can, in such cases, be solved by using only parts of MapReduce. Though this could be useful in cases where MapReduce is only used as an easy way to distribute a process across multiple nodes.

Let us take a look at a simple example: counting the frequency of words in a piece of text. Algorithm 1 shows a simple way to count the frequency of words in a single input string and algorithm 2 shows a way how to combine these

results. The Mapreduce framework, on which this example would be run on, would start by dividing a single large input text into several smaller parts. Each map node in the system would then receive one such a part to process by the map function. After all of the map tasks finish executing, their results would then be combined in the reduce function, which would be run on the reduce nodes.

Algorithm 1 FREQUENCYMAP(s)

Require: A string value s.
Ensure: The frequency of words in s.

```
1: freq ⇐ empty array
2: for all word in s do
3:     if freq[word] ≠ 0 then
4:         freq[word] + +
5:     else
6:         freq[word] ← 1
7:     end if
8: end for
9: return freq
```

Algorithm 2 FREQUENCYREDUCE($frequencies$)

Require: Array of arrays $frequencies$.
Ensure: The combination of frequencies of words in $frequencies$.

```
1: freq ⇐ frequencies[0]
2: for all f in frequencies do
3:     for all word in f do
4:         if freq[word] ≠ 0 then
5:             freq[word] + +
6:         else
7:             freq[word] ← 1
8:         end if
9:     end for
10: end for
11: return freq
```

The paradigm in itself does not give any more specific description of a 'valid' MapReduce framework. Because of this many of the frameworks developed differ quite significantly from one another. And in fact many of the methods designed for different MapReduce based programs rely on single underlying framework, and might work very differently on others. Also there are no guarantees that implementing such a method on a different platform is even possible. This is usually why improvements on MapReduce are introduced with a preselected framework in mind.

III. HADOOP

One of the more prominent MapReduce frameworks currently in use is Hadoop. Although sometimes criticized for being too low level and rather slow it has still achieved quite a large user base [2]. This can probably be attributed to its ease of use and low amount of initial configuration. Also it is quite easy and cost effective to add nodes to an already existing Hadoop system, thus making it easily extendable.

A. Data Distribution

In Hadoop each of the nodes in the system has some space for the files related to task currently running on it, i.e. the input and output files. Usually all of the used input data is stored on specific data nodes. These nodes will then handle the distribution of data across other nodes as needed. If a node does not already have the required data to perform a task, the input file is copied over from a data node according to the division made by the framework. In order to avoid useless traffic only the part of the file, which is actually used in the task, is copied over. The method of dividing the data for the map tasks can be defined by the user.

If possible Hadoop will always try to perform the execution of map tasks in such a way that minimal amount of file transfers are required. I.e. a map node will get such a task which can be performed based on the files already on it. In a case were a node has all of the required data but is busy executing another task, Hadoop will designate the task to another node. This might of course induce file transmission, but it is still more viable than waiting for the other task to finish.

B. Input and Output

By default Hadoop accepts input through files, though it is possible to configure other data sources (such as SQL queries to a database). One file must always be specified to be used as input for the system. The actual content of the file can be defined in any way the user wants to; Hadoop itself contains a few default ways to read the input, e.g. comma separated values. Of course if a custom file format is used, then the user must also define the way the file is to be read. Basically this consists of a file reader and a partitioner which divides the read data for the map tasks. Usually the input files would be read one line at a time and each line would then constitute a single map task, though, if a custom format is used, any kind of division is possible.

Besides the input also the case of output should be considered. Like with the input files the output produced by the reduce part also resides in files on the system. This of course is the case only when the reduce function produces an actual output. One could easily make a reduce function which would not produce any output at all to the file system, but instead would save the result with, for example, an SQL query.

No matter the method used to read the input, the assumption is that the input file format is textual. This increases the amount of processing required by the reading phase as the text needs to be converted to the actual values used by the map task. For example, the character representation of an integer needs to be converted to the actual number. An intuitive way of improving the performance would be to use a binary based file format. This way the values would not need to be converted from text, but could be instead used exactly as read. Using a binary file is indeed very possible and does seem to improve the performance significantly [4] [5].

C. De-serialization

Transforming a value from the file system to the system memory consists of two separate parts: reading and de-serialization. The reading part is self explanatory, just reading the binary from the file. De-serialization on the other hand means the transformation of this binary to an actual value to be used in a program. This differentiation might seem somewhat artificial but it is needed in order to explain some of the concepts in this paper.

One of the larger hindrances for Hadoop is the de-serialization of values from the underlying file system. Even when the binary file format is used the de-serialization process takes a long time. This is especially true if the value is something more complex than a simple integer or a string. This problem can be partly shown to be caused by the programming language used for the development of Hadoop - Java [4]. How ever this problem only manifests in certain cases i.e. when large amounts of data are accessed in a short time period. Because of this it can be avoided by changing the way data is loaded from the file system.

Figure 1. shows how differently typed data affects the de-serialization cost of the data. As you can see in the case where the value is a complex object (that is, a map) the cost is highest. Also notable is the effect of the programming language: C++ is significantly faster with simple types when compared to Java.

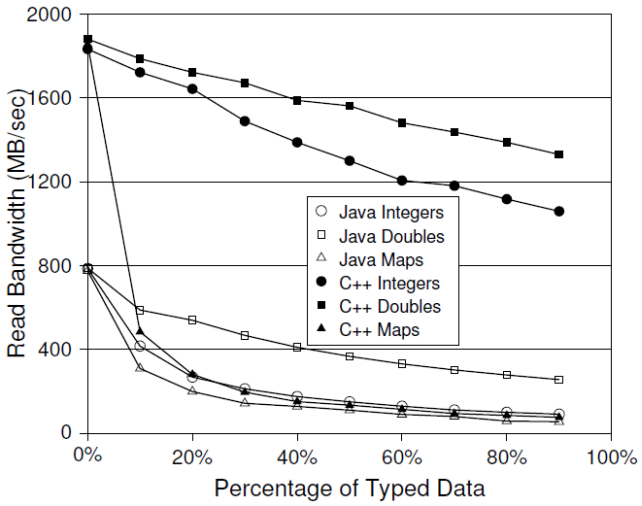

Fig. 1. Costs of de-serialization for different data types [4].

Jiang et. al. [3] have also identified quite a few different factors that have a significant effect on the performance of Hadoop. Yet in each case they also found a way how mitigate the effects of each factor. Because of this it can be argued that most problems with performance in Hadoop are not necessarily inherit to the system, but can instead be avoided by proper design.

IV. Columnar Methods

Most of the current major database systems follow the 'classical' row-wise storage format. That is, they store the data in the tables as rows into the underlying file system. This way the actual positioning of the data in the file is very similar to the structure of the database table it represents.

Lately another kind of storage format has achieved more popularity. Instead of storing the values as rows, one would use columns. A database table would then be represented by a multitude of different files, each containing values from a single column in the table. When values are then added to the table, instead of adding a row to the underlying files, the system splits the row according to its columns and saves each value to a separate file destined for the values of this column.

A. Single Column File

This simple way of implementing a columnar storage of data can be used with Hadoop as well. It can be simply achieved by configuring the underlying file structure used by Hadoop to store values in this manner [4]. As stated in chapter III each program implemented with Hadoop can specify the way it will read input and produce output. Thus one would only need to configure the readers and writers to achieve this simple columnar storage.

Configuring Hadoop to use such a storing method is rather straightforward, but with it rises a few problems. First, a single map task in Hadoop will only take a single input file. Yet, if such a storage is used, a single file contains all of the values of a single column, and so only one map task can be run per column. This problem can be solved simply by splitting the single large column file into multiple smaller ones, each containing a part of the original values [5].

The second problem rises when a map task would need to access multiple columns. In a case where there are more than one data node in a Hadoop system, it is not guaranteed that all nodes contain all of the data. If this is the case a significant amount of time might be taken by the transportation of the required files to the nodes. This problem can be avoided by making sure that the splits made to the column files separated them at the same point. That is, all of the column file splits have an equal amount of rows. This way it is more easy to assure that all of the required data (the rows processed by the map task) are in the node when needed [4]. Though this requires that the default file placement and replication methods of Hadoop are to be overwritten in such a manner that the column files reside on the same nodes.

Another way to tackle the same problem is to create groups of those columns most likely to be used together. One such group would consists of all the columns, another from a few and another from the ones left over. This method takes more storage space, but (assuming that the predictions are correct) decreases the probability of large scale column file transportation [5]. The underlying file system of Hadoop does allow one to choose how single files are stored in the system and thus it is possible to configure it to group files together in this manner.

B. Materialization

Besides the location of the data, also the way how it is read from the file system should be considered. In a case where a map task needs multiple columns it is not likely that all values

from all the columns are accessed. Assume a case where a value in column A determines (e.g. with an if-clause) if a value from column B is needed at all. Of course both of the files containing these columns need to be on the node, but probably not all values in the column B file are needed. In such a case reading all of the values in column B file is pointless as it inflicts unnecessary IO costs. To prevent this from happening values from column B can only be read as needed, this method is known as late materialization [1].

Opposed to late materialization is early materialization. With it values in a column are all read before they are actually used. This can easily lead to unneeded reads, as described above. Late materialization prolongs the actual read operations until the map task actually needs to access the values. In a case where IO costs are somewhat high, early materialization is only useful in very rare cases. In fact, especially when the amount of columns is high, there are very few reasons not to use late materialization [4] [5].

With Hadoop it is not very easy to implement late materialization in the manner described above. In the default case, the actual column files have been already read at the point the map function is started. This is caused by the reader component in the system which provides the map function with the values it uses even before it starts. Because of this any efforts to avoid read operations in the map phase are too late. In order to void problems like this the reader component can also be provided by the user.

With a custom reader there are two distinct ways to provide a late materialization strategy. First the reader reads all of the input required by the map task, but does not de-serialize it. That is, all of the data is read from disk but the map task is only provided with the raw binary. While this method does not avoid read operations it can still improve the run time in cases where the de-serialized values would be complex (e.g. maps, objects, etc...) [4]. Second the map function is only provided with the values of few columns and it will then use those values to provide the reduce function with information of the required data. The actual processing of the data would then also happen in the reduce phase [5].

The second method can be explained more easily through an example: assume that our MapReduce task is equivalent to the SQL query:

```
SELECT A, B, C, D
FROM table
WHERE A < 1000
```

In this case the map function would be provided with the values of column A. It would then use these to process the filtering where-condition to find out the indexes of the valid data tuples. These indexes are then given to the reduce function which will read the column files of B, C and D at these points to access the values. After it will then simply combine them to produce the requested result. This way all of the unnecessary reads to columns B, C and D can be avoided.

Figure 2. shows the effect of lazy- (CIF-SL) and early materialization (CIF) on the processing time. The figure represent the case of de-serializing a map typed value which explains the large difference between the two cases.

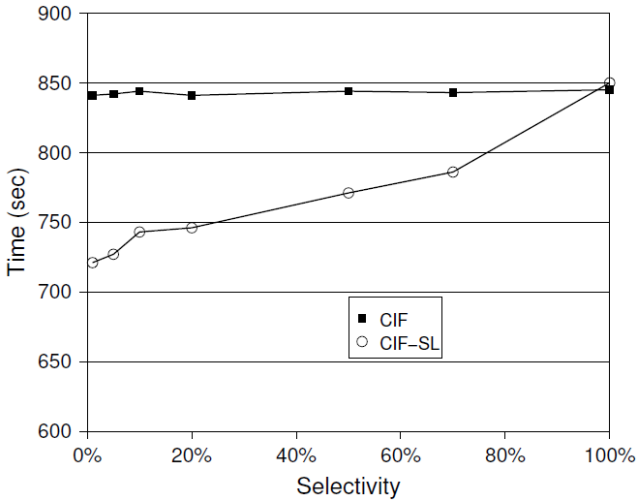

Fig. 2. The difference of lazy- and early materialization [4].

C. Compression

Another way to increase the performance of Hadoop is to add some sort of a compression method to the files used. In most cases this increases the processing time on the CPU but decreases the IO costs. Of course it is very implementation specific which one of these is preferred over the other, with Hadoop it would seem that compression does offer better runtime efficiency [4] [5].

Using compression with Hadoop is somewhat straightforward: by applying the required compression and decompression stages to the output and input phases respectively the desired compression method can be used. Though it must be taken into consideration that only certain compression methods are suitable to be used with Hadoop. As most of these methods require the possibility for accessing files at random, only algorithms allowing partial compression can be used. That is, the data must decompressable at random points without the need to decompress the whole file in order to read a part of it.

D. Results

The methods described in this chapter affect the performance in different ways. Some improve the read times while some decrease the cost of processing the data. I will not show results of performance of each of these methods, but instead concentrate on a case where all of these have been used together. Figure 3 shows such a case.

As shown the performance of the purely text based file format (TXT) is (as expected) the worst. This is mainly due to the large de-serialization costs associated with it. The binary based purely columnar format (SEQ) avoids this problem mostly thus improving the performance significantly. The CIF and RCFile formats employ all of the methods described in this chapter. As shown they also provide the best results. The large differences to TXT and SEQ can mostly be explained by the amount of data both of these avoid reading.

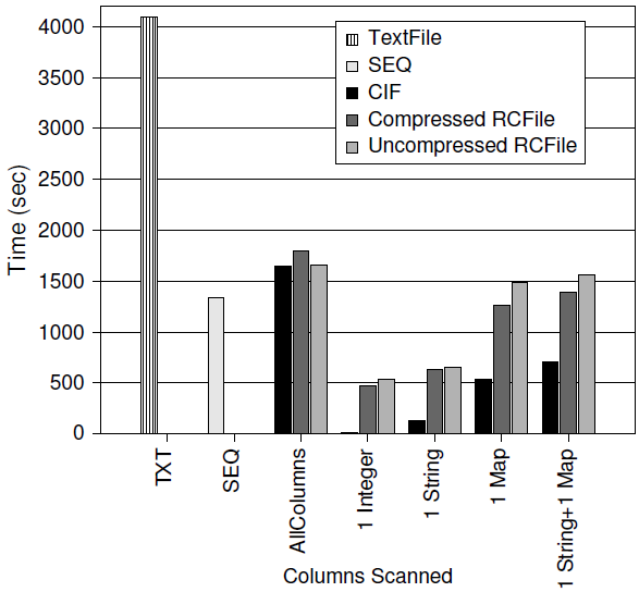

Fig. 3. Performance evaluation of different file formats [4].

V. Conclusion

In this article I have taken a look at the MapReduce paradigm and one of its implementations: Hadoop. Although some critique have been directed towards them, in regards of their performance as compared to distributed database systems, they are still a viable choice for distributed data processing. This is especially true in cases where some optimization has been put into the underlying file system and the way it is used by Hadoop.

One effective way to optimize Hadoop is to use a columnar storage method in the underlying file system. By applying some of the methods used with many columnar database systems, quite substantial increases in performance can be achieved. In most cases even using only one of the methods described in this paper the performance could be improved, but it makes more sense to use more than one. Tests performed on systems configured to work this way seem to indicate that quite a significant performance boost can be achieved: system configured to use all of these methods can achieve on average three times faster data processing rates [4].

References

[1] D. J. Abadi, D. S. Myers, D. J. DeWitt and S. R. Madden, *Materialization Strategies in a Column-Oriented DBMS* Proceedings of ICDE, 2007.

[2] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, M. Stonebraker, *A Comparison of Approaches to Large-Scale Data Analysis* Proceedings of the 2009 ACM SIGMOD International Conference on Management of data, pp. 165-178, 2009.

[3] D. Jiang, B. C. Ooi, L. Shi and S. Wu, *The Performance of MapReduce: An In-depth Study* PVLDB vol. 3, no. 1, pp. 472-483, 2010.

[4] A. Floratou, J. M. Patel, E. J. Shekita and S. Tata, *Colum-Oriented Storage Techniques for MapReduce.* International Conference on Very Large Data Bases Proceedings vol. 4, no. 7, pp. 419-429, Apr. 2011.

[5] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi and S. Wu, *Llama: Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework.* Proceedings of the ACM SIGMOD International Conference on Management of Data pp.961-972, Jun. 2011.

[6] J. Dean and S. Ghemawat, *MapReduce: Simplified Data Processing on Large Clusters* Sixth Symposium on Operating System Design and Implementation, Dec. 2004.