Similarity Join on Hadoop

An Evaluation of the Hadoop Map/Reduce Environment

Andrina Mascher, Tim Felgentreff

Map/Reduce Algorithms on Hadoop, Research Group Information Systems,
Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany,
{andrina.mascher, tim.felgentreff}@student.hpi.uni-potsdam.de

Abstract. In this paper we present our findings for a similarity join algorithm for use in a Map/Reduce setup on Apache's Hadoop and the conclusions we have been able to draw about the Hadoop implementation of Map/Reduce and Map/Reduce in general. Wikipedia pages were chosen as test input to compare, their similarity measured by co-occurrences of words with the Jaccard coefficient.

1 Introduction

Efficient processing of large input data is reached through parallelization on a distributed system. Our cluster is managed by Hadoop which uses the Map/Reduce paradigm through which a complex task is divided into many smaller ones. The project was implemented in Java due to Hadoop.

2 Algorithm

For a large set of pages from Wikipedia we calculate the Jaccard coefficient and display only those pairs with a coefficient exceeding a given limit. Our Algorithm is influenced by a similar paper on finding co-editors with Map/Reduce [**?**]. We also did not calculate the Jaccard coefficient for every possible pair of pages, which would have meant calculating intersection and union for every pair, a very time consuming task exactly in $\mathcal{O}(NumberOfPages^2 * NumberOfWords^2)$. We only consider pairs of pages that share at least one word and are therefore potentially similar. Furthermore, the coefficient of those pages that differ greatly is calculated faster than very similar pages. We finally result in $\mathcal{O}(2 * NumberOfPages^2 * NumberOfWords)$ because some words lead to some pairs of pages they occur in and some pairs of pages share some words. Eventually we achieve $\mathcal{O}(n^3)$ instead of $\mathcal{O}(n^4)$.

The words themselves are discarded quite early because only the cardinalities of the intersections and unions are needed. In order to compute similarity we do not want all words taken into account, but rather filter only relevant words to achieve higher accuracy when comparing texts.

2.1 Supporting Algorithms

Jaccard

The Jaccard coefficient measures the similarity of two sets $\mathcal{A}$ and $\mathcal{B}$ by calculating $|\mathcal{A} \cap \mathcal{B}| \div |\mathcal{A} \cup \mathcal{B}|$. This results in a value between 0 and 1 with 1 meaning identical. When comparing texts we want to consider multiple occurences of words and therefore use intersection and union as multiset-operations opposing to the original context by Paul Jaccard (1868–1944) [**?**].

Porter Stemmer

To increase findings of possibly similar pages we stem the words and use lowercase letters only. It is then possible to group words which will result in higher recall [**?**]. In order to be able to stem a word we eliminate numbers and special characters and stem only those words with more than four letters. We chose the popular Porter Stemming algorithm, because it is very well adjusted for English texts [**?**].

2.2 Phase 1 - Pairing

Map Relevant words in a page are found through markups such as headings and links or preferably through tf/idf. Every relevant word is stemmed with the Porter Stemmer algorithm and added to a hashtable from which further values can be calculated. The output value for a relevant word is a class CountTriple that consists of the given page, the number of occurences of this word in this page and the count of all relevant words in this page.

Input: page, {word}

Output: relevant word, (page, occurences, wordcount)

Reduce For a given word all corresponding pages are combined in the value-set as CountTriples. Since all pairs of pages are potentially similar they are added to the output. Each pair is added only once in a defined order since they will be used as key later on.

Input: word, {(page, occurences, wordcount)}

Output: (page1, occurences, wordcount), (page2, occurences, wordcount)

2.3 Phase 2 - Jaccard

Map This Map job receives two CountTriples with two different pages that refer to the same word, although the word itself was discarded before. Their multiset-intersection is computed by selecting the minimum of the count of this specific word. The sum of their overall wordcounts will be used to calculate the multiset-union later on.

Input: (page1, occurences, wordcount), (page2, occurences, wordcount)

Output: (page1, page2), (minimum, sum)

Reduce A pair of pages might share several words, whose occurences are represented in the value-set. All minima sum up to the overall intersection $\mathcal{I}$. The sum $\mathcal{S}$ is the same in all elements. The Jaccard coefficient of both pages is eventually calculated by $\mathcal{I} \div (\mathcal{S} - \mathcal{I})$. The denominator now resembles the multiset-union. If the coefficient exceeds a predefined limit the pair of pages will be added to the output.

Input: (page1, page2), {(minimum, sum)}

Output: (page1, page2), jaccard

2.4 Phase 3 - Output

Map For a given pair of pages there will be two sets of output due to summetry of similarity.

Input: (page1, page2), jaccard

Output: page1, (page2, jaccard) and page2, (page1, jaccard)

Reduce The last Reduce jobs collect all similar pages for a given page with their Jaccard coefficients. This step is trivial, only if required the output could be sorted or visualised.

Input: page1, {(page2, jaccard)}

Output: page1, {(page2, jaccard)}

3 Benchmarking

We benchmarked the results on a heterogeneous cluster of 10 machines with moderate hardware.

As we can gather from Fig. 1 the run-time complexity of the algorithm matches a graph in $\mathcal{O}(n^3)$, as we had anticipated. However, due to the distributed nature of Hadoop, running over small data sets is slower than running a naive implementation locally. The gain comes with the massive parallelization of Hadoop, and thus is more and more apparent on large inputs.

The reasons we found for this are mainly in the way Hadoop distributes data between the Map and Reduce phases. Especially for small jobs with low per-node run-time the overhead of sending, in the worst-case scenario all of our data, over

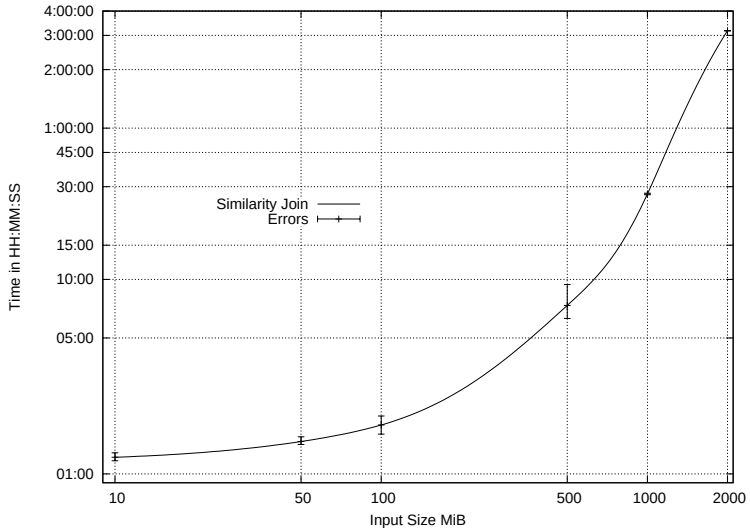

Fig. 1: Run-time development over growing input sizes

TCP/IP three times over outmatches any gain through parallel processing. This assertion appeared to hold when using larger input-sets. It proved to be correct when we changed the algorithm from running four Map/Reduce phases to only running three.

By decreasing the number of phases our running time went down by an almost constant fraction for each point of measurement, indicating that indeed a large amount of our input is send over the network in-between jobs.

4 Lessons learned

4.1 Writing to HDFS

During our initial testing, we tried to use a default Jaccard coefficient of 0.5 to declare two pages as similar. On larger data sets, however, this led to huge amounts of data being written to the HDFS, at one time jobs running on input sizes of 10GB wrote in the term of execution almost 100GB into the HDFS.

We found, that writing to HDFS was often the bottleneck here, with Maps going straight to 100% and Reduce jobs hanging at 98/99% waiting for huge amounts of data to be written to the distributed file-system.

It seems hard-drive speed is a crucial limiting factor in the Hadoop cluster we used. We worked around the problem to get better benchmarking results by raising the Jaccard index to 0.8 which, for the last Map/Reduce run, led to a much smaller amount of data being written. Still remaining was the first run as unavoidably write-intensive by pairing each page with all its candidates for similarity. The amount of data written here is very dependent on the input

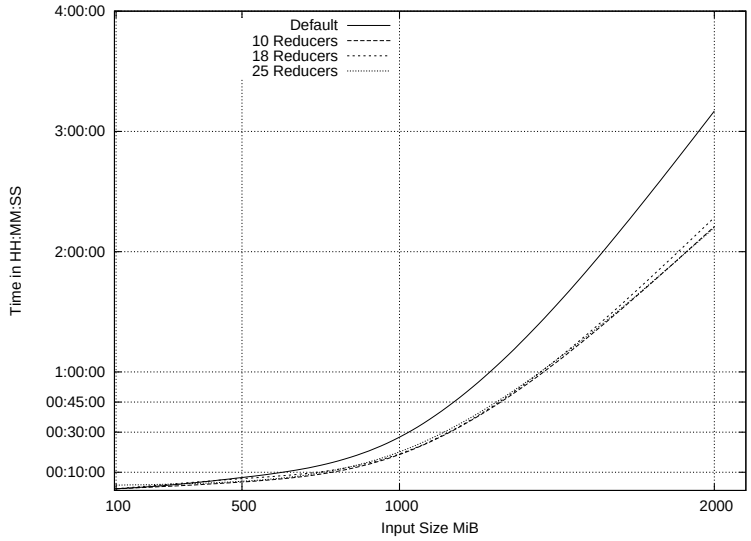

Fig. 2: Processing Speed in Relation to Number of Reducers

data, however, for our Wikipedia articles we found that doubling the input size increased the size of possible candidates almost tenfold.

4.2 Running more Reducers

It appears that the performance of the cluster is very dependent on the number of Reducers running (Fig. 2). While the number of Mappers is simply determined by the block-size in the HDFS and thus simply dependent on the locality of the data, the Hadoop framework seemed to have a hard time determining the right amount of Reduce jobs for the cluster. Left to its own devices we observed that even a growth in input data was only slowly met with a growth in Reduce jobs, so that with up to 10GB of input only one or sometimes two Reducers were started.

The heuristics Hadoop uses for this decision might be good for homogeneous clusters, however, as we had a couple of machines running much faster than others, advising the framework to employ more Reducers generally led to better performance. Fine-tuning was still required, though, and we found the best advice to be roughly equivalent to the number of nodes available. As this is only advice to the framework the real number of Reducers sometimes went up to 1.5 times that number. This way, if necessary, faster nodes could start additional jobs while the others were still processing.

With growing input sizes, using even more Reducers (up to a factor of 2.5) seemed to become even more efficient, as the time impact caused by the network distribution diminishes compared to pure calculation time.

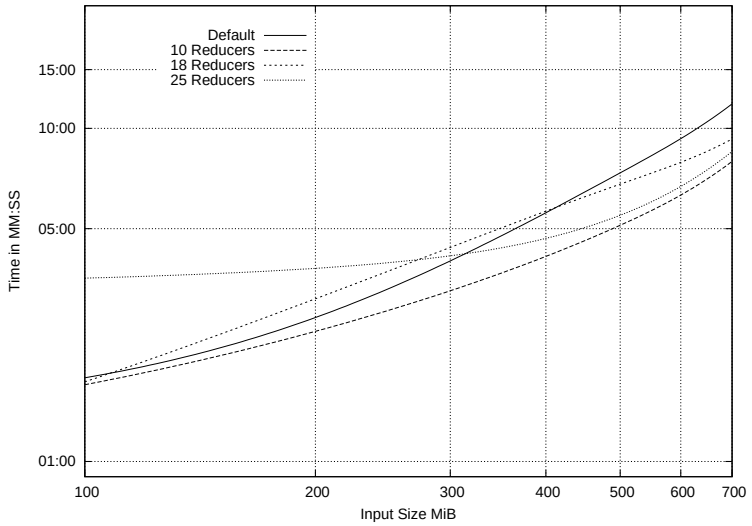

Fig. 3: Overhead when Running more Reducers

4.3 Network Overhead on Data Redistribution with many Reducers

Interesting to note for small datasets is the time penalty that comes with running more Reducers. Fig. 3 is a close-up in log-scale for the smaller datasets of up to 700MB. With small datasets, not all nodes run a Map job, however, if we force the framework to run at least one Reduce job on every machine it needs to distribute data to the nodes that do not have any output from the Maps at all.

5 Conclusions

These observations lead us to stress the importance of configuring the Hadoop cluster according to the requirements. Runtime greatly depends on the combination of Reducers and expected input sizes. At least on heterogeneous clusters Hadoop is unable to properly determine the optimal configurations by itself. More homogeneity might mean more versatility here so relying on Hadoop to breathe some life into a varied bunch of older machines means one has to spend more time thinking about configuration.

Acknowledgments

We thank Alexander Albrecht for his advice on the algorithm and his help in finding the optimal way at tackling the problem. We would also like to thank Professor Neumann for enabling us to attend this seminar.