Similarity Joins in MapReduce

Benjamin Coors, Kristian Hunt, and Alain Kaeslin

KTH Royal Institute of Technology
{coors,khunt,kaeslin}@kth.se

Abstract. This paper studies how similarity joins can be implemented in the MapReduce paradigm. Similarity joins refer to the problem of finding similar records. We describe an approach on how to efficiently implement similarity joins in the MapReduce paradigm and apply this approach to the problem of finding different spellings of a person's name in a database of authors of scientific papers. Our implementation tokenizes the author name strings into n-grams and uses the Jaccard index set similarity measure. We show that we receive the best results for 3 and 4-grams and that the number of comparisons needed is lower by a factor of at least hundred than in the naive approach of comparing each record with all others.

1 Introduction

Detection of similar records or similarity joining is a problem that occurs in many applications such as plagiarism detection, recommender systems or mining of social networking data. With the continuous increase of data those type of applications are processing, this process becomes a big computational burden. Often such tasks can become too expensive to be handled by a single computer. This has lead to an increased interest in technologies for parallel, distributed data processing on computer clusters such as MapReduce (see Dean and Ghemawat [3]).

The join operator is algebraically defined as a Cartesian product followed by the selection operator that specifies the join condition. Joins have been thoroughly studied and efficient algorithms such as the sort-merge join or hash join exist. They can efficiently reduce the runtime, since they do not require the Cartesian product to be explicitly constructed. This is achieved by exploiting sorting and hashing properties respectively. These algorithms work for join conditions such as equality. However, if the join condition is a similarity threshold, sort-merge or hash join algorithms cannot be applied. This is due to the fact, that no sorting exists where similar records always appear in order and similar records do not map to the same hash. Therefore, other means to efficiently implement similarity joins are required.

In this project, we have implemented a MapReduce-based algorithm for set-similarity joins on strings. For testing we use a database of approximately 250'000 author names obtained from Kaggle Inc [4]. The database contains noise because "there are many authors who publish under several variations of their own name", typos, different orderings of first- and last names etc. The goal is to detect different writings of the same person's name. Formally, this is self joining the author relation A on the author's name attribute $A.n$, given a set-similarity function sim(), a similarity threshold τ and a tokenisation method tokenise() which transforms strings into token sets such that $A \bowtie_{\text{sim(tokenise(A.n),tokenise(A'.n))} \geq \tau} A$.

2 Method

Section 2.1 discusses prefix filter, a principle used to enable fast implementations of set-similarity joins. Section 2.2 introduces the MapReduce-based algorithm we have implemented. Finally, Section 2.3 discusses how the algorithm introduced can be used on the author database.

2.1 Prefix Filter

Since building the Cartesian product of all records and subsequently applying a selection operator based on a set-similarity function sim() to perform a join operation is computationally expensive, filters are usually used to decrease the number of candidate pairs to which the selection operator is applied. One commonly used filter is the prefix filter (see Chaudhuri, Ganti, and Kaushik [2]). The intuition behind the prefix filter is, that the similarity between two sets cannot reach a given threshold if a specific subset (called prefix) of the two sets has an empty intersection (see Augsten and Bohlen [1]).

Definition 1. *Given an ordering $\mathcal{O}$ of the universe $\mathcal{U}$ from which all set elements are drawn, the k-prefix of any set S is, the subset of S consisting of the first k elements of S as per the ordering $\mathcal{O}$.*

Example 1. The 3-prefix of the set {𓅟, 𓆓, 𓁨, 𓄿, 𓆛, 𓃭} given the ordering 𓁨 < 𓃭 < 𓄿 < 𓅟 < 𓆓 < 𓆛 is {𓁨, 𓃭, 𓄿}.

Theorem 1. ***Prefix Filtering Principle.*** *Given two sets A and B, and an ordering $\mathcal{O}$ on the elements of both sets, and a threshold τ, if $\mathrm{sim}(A, B) \geq \tau$, then the $(|A| - \alpha + 1)$-prefix A_p of A and the $(|B| - \alpha + 1)$-prefix B_p of B with respect to the given order share at least one element, $A_p \cap B_p \neq \emptyset$. The constant α is dependant on the particular similarity function* sim() *and threshold τ used.*

Based on the definition and theorem above, an efficient similarity join algorithm can be implemented by applying the selection operator only on records having at least one prefix token in common. Since sharing a prefix token is a necessary, but not sufficient condition for similar records, the filter does not produce false positives. To produce the least amount of candidate pairs, the increasing token-frequency order is commonly used as ordering $\mathcal{O}$.

2.2 MapReduce Algorithm

The MapReduce-based algorithm presented by Vernica, Carey, and Li [6] consists of three stages, out of which some stages consists of several `map-reduce` phases.

Stage 1 This stage determines the increasing token-frequency order over all the input records. This is achieved by two `map-reduce` phases. The first one tokenizes the records using the tokenise() function and counts the occurrence of all tokens in all records. This phase is essentially equal to the famous word frequency MapReduce program (see Dean and Ghemawat [3]). The second phase sorts the tokens according to their frequency.

Stage 2 This stage, called the kernel, performs the prefix filtering to form all candidate pairs and afterwards does similarity checking on them. This stage consists of a single `map-reduce` phase.

The mapper computes the prefix A_p as per the token order computed by the previous stage for every record A. Every prefix token of the record is used as a key. Therefore, $(|A| - \alpha + 1)$ `(key, value)`-pairs are emitted per record. Consecutively, the values get grouped by prefix tokens during the MapReduce sorting step, therefore all the values passed to a reduce call are potentially similar records, since they share at least one common token in their prefixes. The reducer calculates the similarity using the function sim() for every candidate pair and emits the pair if the similarity is above the threshold τ.

Stage 3 Since two records A and B can possibly share more than one token in their prefixes A_p and B_p, the output from the previous stage can contain duplicates. In this stage they are removed from the output. This stage again consist of a single `map-reduce` phase.

2.3 Application to Author Names

When applying the method described to the author names problem, a number of choices have to be made. For example, it might be necessary to perform data pre-processing. As the author names database contains a lot of noise, any non-alphabetical characters were removed in the tokenization process. Furthermore, the author names were in inconsistent capitalization and therefore the case is ignored in further steps.

Another aspect to be decided is how to perform the tokenization itself. The processed strings can be split into words. In the case of author names this would mean splitting the author name into first, middle and last name. However, this would have the disadvantage of making it more likely to not detect misspelt names since even small differences in strings result in high distances between them. For this reason, we chose to use n-grams. Because the first, middle and last name of the author can be written in different orders and possibly be shortened or left out, the author names were first tokenized into words before these were further split into n-grams. This means that n-grams do not span across words and do not take the location of word separators into consideration. The size of the n-gram has a large influence on the number of comparisons and results as well as the accuracy of the results. This will be discussed further in section 3.

A number of options are available for the set-similarity function. The most common choice to measure the similarity between two sets A and B is the Jaccard index $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$. This is the similarity function used by Vernica, Carey, and Li [6] and was also chosen for this problem. For the prefix filter the Jaccard index requires an α of $\lceil \tau |X| \rceil$, where X is either the set A or B and where τ is a suitable similarity threshold. To find this threshold for the Jaccard index, different values were evaluated in terms of quality of the results and performance.

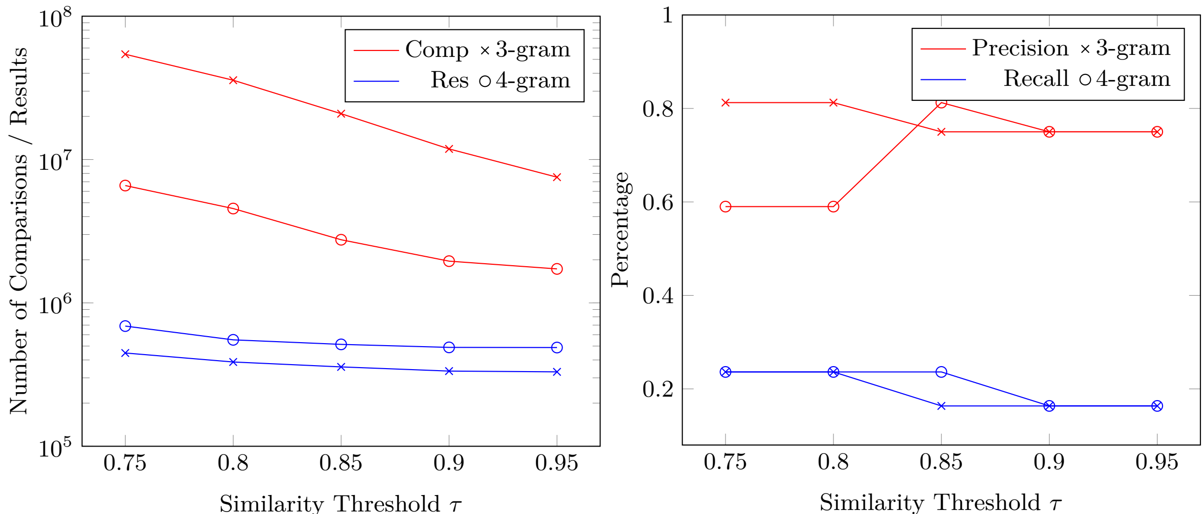
3 Results

When running the algorithm with different similarity thresholds τ, we can observe that both the number of comparisons over name pairs and number of resulting name matches decrease with a larger τ. It can be observed in Figure 1 that the decrease is steeper for the number of comparisons while the number of retrieved results stays relatively similar.

The algorithm was run with both 3-grams and 4-grams. In general, the choice of n-gram size is a tradeoff between computation time and quality of results. As can be seen, the number of comparisons is significantly higher for 3-grams. This is the reason why using 2-grams would be computationally infeasible. Unlike the number of comparisons, the number of retrieved results increases with higher n-gram. However, the quality of results is lower which makes higher n-grams such as 5-grams unattractive to use.

The lower quality of results can be observed in the precision and recall lines for the results of the author "Edsger Wybe Dijkstra". While the recall is the same for 3 and 4-grams, the precision is lower for 4-grams, indicating a lower percentage of correct matches in the result. This shows that higher n-grams do not retrieve more accurate results even though the number of results increases.

The number of comparisons also shows the advantage of the presented approach over the naive solution of comparing every author name with all others. For the given dataset of about $n = 250,000$ author names, the number of comparisons does not exceed 0.2% of the $\frac{n(n-1)}{2}$ comparisons needed in the naive approach.



(a) Number of comparisons over name pairs and number of resulting name matches over different similarity thresholds τ.

(b) Precision and Recall for the author 'Edsger Wybe Dijkstra' for different similarity thresholds τ.

Fig. 1: Relationship of the similarity threshold τ with different performance measures.

4 Issues and Further Work

Generally, joining two very large data sets together does not fit into the MapReduce paradigm gracefully, since MapReduce is very good at processing large data sets by looking at every record in isolation (Miner and Shook [5]). This is particularly problematic in stage 2 of the method implemented, where the cross product of all records sharing at least one prefix token is built. Although, this cross product will in most cases be significantly smaller than the cross product of the complete relation, it still might not fit into memory. In that case, sub-partitioning the data into blocks that fit in memory is necessary (see Vernica, Carey, and Li [6]).

One issue when tokenizing author names into 3 or 4-grams work best is that names which are shorter than 3 or 4 letters, respectively, are being discarded in the tokenization process. This can, in particular, be a problem with Asian names as second and last names are more often shorter than this lower bound. For example, the name "Adeline Yen Mah" will be tokenized into {Adel, deli, elin, line}. We therefore suggest an alternative tokenization which would instead produce {Adel, deli, elin, line, Yen, Mah} for further analysis.

5 Summary

In this project the efficient implementation of set similarity joins was studied using the MapReduce paradigm. A three-stage method proposed by Vernica, Carey, and Li [6] was adapted to the problem of detecting different writings of the same author's name and implemented using the Hadoop framework. We discussed and motivated the choices we took to apply the given method to this problem.

In an experimental evaluation, we analyzed the impact of the tokenization function and similarity threshold τ on the computational burden and quality of the result. We conclude that using higher order n-grams reduces the computational effort but negatively impacts the quality. Compared to the naive $O(n^2)$ approach to similarity joining, our approach reduces the number of comparisons needed by a factor of at least hundred.

References

[1] N. Augsten and M. Bohlen. *Similarity Joins in Relational Database Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2013.

[2] S. Chaudhuri, V. Ganti, and R. Kaushik. "A Primitive Operator for Similarity Joins in Data Cleaning". In: *Data Engineering, 2006. ICDE '06. Proceedings of the 22nd International Conference on*. 2006, pp. 5–5.

[3] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters". In: *Communications of the ACM* 51.1 (2008), pp. 107–113.

[4] Kaggle Inc. *KDD Cup 2013 - Author-Paper Identification Challenge (Track 1)*. [Online; accessed 29-April-2015]. URL: `https://www.kaggle.com/c/kdd-cup-2013-author-paper-identification-challenge`.

[5] D. Miner and A. Shook. *MapReduce Design Patterns: Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, 2012.

[6] Rares Vernica, Michael J Carey, and Chen Li. "Efficient parallel set-similarity joins using MapReduce". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 495–506.