Distributed Similarity Joins on Big Textual Data: Toward a Robust Cost-Based Framework

Fabian Fier
Supervised by Johann-Christoph Freytag
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
fier@informatik.hu-berlin.de

ABSTRACT

Motivated by increasing dataset sizes, various MapReduce-based similarity join algorithms have emerged. In our past work (to appear), we compared nine of the most prominent algorithms experimentally. Surprisingly, we found that their runtimes become inhibitively long for only moderately large datasets. There are two main reasons. First, data grouping and replication between Map and Reduce relies on input data characteristics such as word distribution. A skewed distribution as it is common for textual data leads to data groups which reveal very unequal computation costs, leading to Straggling Reducer issues. Second, each Reduce instance only has limited main memory. Data spilling also leads to Straggling Reducers. In order to leverage parallelization, all approaches we investigated rely on high replication and hit this memory limit even with relatively small input data. In this work, we propose an initial approach toward a join framework to overcome both of these issues. It includes a cost-based grouping and replication strategy which is robust against large data sizes and various data characteristics such as skew. Furthermore, we propose an addition to the MapReduce programming paradigm. It unblocks the Reduce execution by running Reducers on partial intermediate datasets, allowing for arbitrarily large data sets between Map and Reduce.

1. INTRODUCTION

Similarity joins are an important operation for user recommendations, near-duplicate detection, or plagiarism detection. They compute similar pairs of objects, such as strings, sets, multisets, or more complex structures. Similarity is expressed by similarity (or distance) functions such as Jaccard, Cosine, or Edit. A naive approach to compute a similarity self-join is to build the cross product over an input dataset and filter out all non-similar pairs. This approach has a quadratic runtime. In the literature, there are various non-distributed non-parallelized approaches for similarity joins based on a two-phase approach [1, 2, 3, 10, 14]. They compute a set of candidate pairs which is usually orders of magnitudes smaller than the cross product. Subsequently, they verify if the candidates are similar. We refer to them as *filter-and-verification* approaches. Motivated by increasing dataset sizes, MapReduce-based distributed approaches have emerged [5, 12, 13]. We conducted an extensive experimental study on nine current MapReduce-based set similarity join algorithms on textual data (to appear). There are two key findings. First, we compared the runtime of the MapReduce join algorithms to the runtime of competing non-distributed algorithms from the recent experimental survey of Mann et al. [11]. The runtime of MapReduce join algorithms on small datasets is inferior to the runtime of non-distributed approaches. This is not surprising due to the MapReduce overhead. The second finding is that none of the approaches can compute the join on larger (or even arbitrarily large) datasets. The runtimes increase so drastically that we terminated the executions after a long timeout.

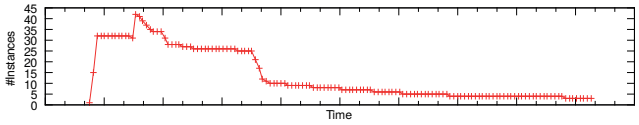

Figure 1: Straggling Reducer Issue.

We identified two main reasons for these runtime issues on large datasets. First, for every MapReduce-based similarity join algorithm we investigated we found non-optimal input datasets that lead to only a few running join Reduce instances while all other instances were left idle. That is, we often observed *Straggling Reducers*. Figure 1 shows the compute instance usage of a non-optimal join execution on a cluster of 48 compute instances. After roughly half the execution time, only a few instances are used. The instance usage is directly connected to data grouping and replication between Map and Reduce. All algorithms under investigation exploit and thus rely on certain data characteristics for replication and grouping. The most relevant characteristics are the global token frequency of the input dataset and the number of tokens in each record of a dataset. *Stop words*, which occur in a majority of records of a dataset, cause skewed data groups within most join approaches we investigated. As second cause, we identified memory overload within Reduce instances. All approaches heavily replicate data to leverage parallelization. The original MapReduce

Proceedings of the VLDB 2017 PhD Workshop, August 28, 2017. Munich, Germany.

programming paradigm as introduced by Dean et al. [4] requires the Reduce instances to wait for the Map steps to finish before the intermediate data groups are sorted and grouped by key. When the Reduce buffers are filled, data is spilled to disk, often causing high runtime penalties. The use of Combiners is not possible for similarity joins, because Reducers are stateful. This limitation is inherent to standard MapReduce.

In this paper, we propose an initial approach toward a robust framework to compute distributed similarity joins. It overcomes the Straggling Reducer issues and the input dataset size limitation we experienced in our past experiments. Our approach is twofold. First, we find a grouping and replication strategy which distributes compute load evenly over the existing compute instances. This is challenging since it is not sufficient to generate data groups of equal size. The runtime of a join computation within one group is dependent on characteristics of the data in the group such as record lengths. Second, we enable MapReduce to handle large intermediate datasets by proposing an extension for MapReduce which unblocks the Reduce execution based on statistical information gathered in a preprocessing step.

The idea of load balancing in MapReduce based on statistics is not new. The TopCluster algorithm [8] is an online approach which includes cardinality estimations at runtime. Our approach on the other hand needs exact data statistics in order to unblock the Reduce execution. These statistics have to be collected before the join execution. Our approach is comparable to the one by Kolb et al. [9], which involves a preprocessing MR job to collect data statistics and a join job which uses the statistics for an optimal data grouping and replication. We extend this approach by using the knowlege of the group sizes to unblock the Reduce execution. Furthermore, we tailor the grouping and replication to the specific problem of set similarity joins.

The contributions of this paper are as follows:

- We propose a first approach toward a robust distributed similarity join framework.
- We define a robust grouping and replication strategy leading to evenly distributed compute loads amongst the available compute nodes.
- We extend the MapReduce programming paradigm to unblock Reduce execution to handle (potentially arbitrarily) large datasets.

The structure of the paper is as follows. In Section 2, we give an overview on the similarity join problem, algorithmic approaches, and motivate the need for research with the runtime issues we experienced in our past experiments. In Section 3, we introduce our approach for a robust join framework and its interaction with an extension of MapReduce to unblock Reduce execution. In Section 4, we conclude our work and give an outlook on future work.

2. BACKGROUND

Without loss of generality, we use the set similarity self-join as a running example. Our framework can be applied to other filter-and-verification-based similarity joins as well. The set similarity join computes all pairs of similar sets (s_1, s_2) within a set of records S. A similarity function $sim(s_1, s_2)$ expresses the similarity between two records. For sets, there are similarity functions such as Jaccard, Cosine, or Dice. The user chooses a threshold t above which two sets are considered being similar. Formally, given a set S, a similarity function $sim(s_1, s_2)$, and a similarity threshold t, the set similarity join computes the set $\{(s_1, s_2) \in S \times S | sim(s_1, s_2) \geq t, s_1 \neq s_2\}$.

A naive approach computes the similarity on all pairs (s_1, s_2). Since it has a quadratic runtime, it is not feasible even for small datasets. In the literature, filter-and-verification approaches emerged. Their basic idea is to generate an (inverted) index over all input records. For each postings list, they compute the cross product (half of it in the self-join case to be exact) and the union of all these cross products. Each distinct record ID pair in the union is a candidate pair, because the two records contain at least one common token. These candidate pairs are further verified to compute the end result. Sophisticated filtering techniques keep the indexes and the number of candidate pairs small. The most prominent filter is the *prefix filter* [1, 2, 3]. Given a record length, a similarity function, and a similarity threshold, the prefix length is the minimum number of tokens which need to be indexed to guarantee an overlap of at least one common token if it is similar to another record.

Motivated by increasing dataset sizes, MapReduce-based versions of the filter-and-verification approach emerged [5, 12, 13]. The main idea is identical to the non-distributed approaches. It is to compute an inverted index, to compute the cross product on each postings list, and to verify the resulting candidate pairs. The inverted index is built as follows. A Map step computes key-value pairs with a token or a more complex signature as key. The MapReduce framework groups key-value pairs with the same key to one Reduce instance. This instance computes the cross product on the postings list. Depending on the value of the key-value pair (all tokens of the input record vs. only the record ID), the verification takes place within the Reduce, or there are further MapReduce steps to join the original records to the candidate pairs for the verification.

The key generation of all algorithms known to us relies on characteristics of the input data. In the most basic algorithm [7], each token in the input record is used as key. Obviously, the number of record groups is equal to the number of distinct tokens in the input dataset. The size of each record group depends on the global frequency of its key token. The data replication is dependent on the record lengths. For sufficiently large datasets with stop words (tokens which occur in almost every record) and/ or many long records, the Straggling Reducer effect occurs. More sophisticated approaches use a prefix filter, which reduces the number of tokens for replication to a prefix, which is shorter than the record length, but still dependent on it. The use of such filters shifts the Straggling Reducer issue to larger datasets and/ or datasets with longer records, but does not solve it for arbitrarily large datasets.

We expect the input of the similarity join to be text, which is integer-tokenized by a preprocessing step. The tokenization may include changing letter cases, stemming, or stop word removal. Depending on the preprocessing, the properties of input datasets vary by token distribution (stop words, infrequent tokens), dictionary size, and record size. The token distribution of textual data is usually Zipfian, which means that there are few very frequent tokens. This is a challenge for approaches relying on token distribution.

3. APPROACH

In Figure 2, we illustrate the dataflow of our framework. The first step computes exact data statistics. It computes record length frequencies and global token frequencies. These statistics can be computed in linear time and are highly parallelizable. Furthermore, it estimates runtime costs for the join execution, based on data samples with differing average record lengths. The second step computes the actual join. Every Map instance obtains the statistics from the first step via a setup function which is called once before the input data is read. Based on these statistics, it determines a suitable data grouping and replication and assigns keys to its output accordingly. Each join Reducer also obtains the statistics via the setup function. Using the statistics, it can compute the exact size of each group and start computing the join on this group once all data for it has completely arrived. This can happen before all Mappers have finished their execution. Note that this requires a change in the original MapReduce. The Reduce-side shuffling periodically counts the occurrences of each key in its input. It triggers the execution of the first-order function once one of the groups is complete. The Reducer can run any existing state-of-the art non-distributed similarity join.

In the following, we describe how to find a suitable grouping and replication based on the statistics. We use the Jaccard similarity function as an example, because it is the most commonly used function in the literature. Our framework is also applicable to any other set-based similarity function. Jaccard is defined by the intersection divided by the union of two records $\frac{|a \cap b|}{|a \cup b|}$. Note that records with differing lengths can be similar. Figure 3 shows this length relationship for a similarity threshold of 0.7. For each record length on the y axis, it shows on the x axis, which record lengths have to be considered as join candidates. Let us assume the input has a length distribution as depicted in Figure 4. In order to obtain data groups which can be self-joined independently, we group together all records with the same length and replicate each group to all larger length groups it can be similar to. The resulting groups would be $\{1\}, \{2\}, \{3,4\}, \{4,5,6,7\}, \{5,6,7,8\}, \{6,7,8,9,10\}$ etc. Note that these groups have very uneven cardinalities, for example $|\{1\}| = 8.000$, $|\{6,7,8,9,10\}| = 688.000$ etc.

In order to distribute the cardinalities evenly, we propose to apply a hash-based grouping and replication on these groups. Figure 5 shows an example for a hashing factor of 4. A hashing function assigns each record to one of 4 groups. Each record is distributed 4 times, so that it joins each other record in exactly one of the squares in the figure. Note that there is a tradeoff related to the hashing factor. If it is very low, there are only few large groups and the replication is low. If it is high, there are many small groups and the replication is high.

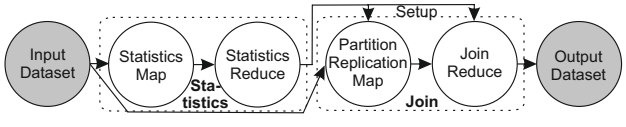

Figure 2: Dataflow Graph of our Execution Framework.

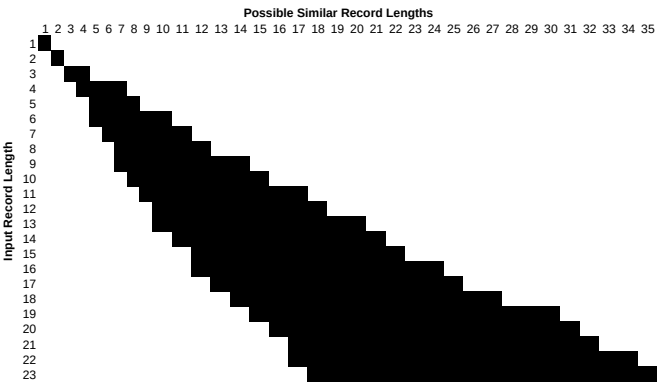

Figure 3: Possible Similar Record Lengths for Jaccard and Similarity Threshold 0.7.

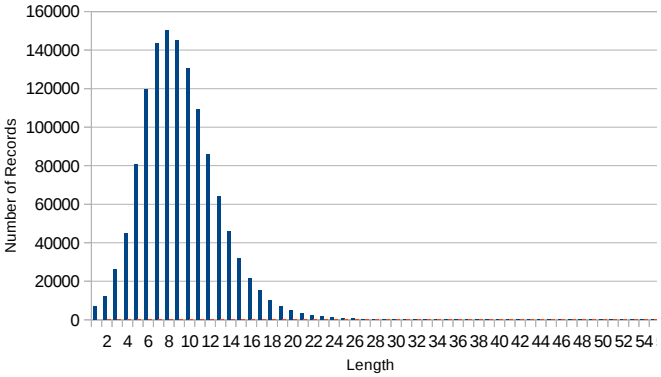

Figure 4: Example Record Length Distribution.

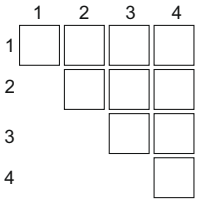

Figure 5: Hash-Based Grouping and Replication with hashing factor h=4.

An even data distribution is not sufficient to prevent Straggling Reducer effects. The costs of joining a partition with long records is higher than the cost of joining a partition with equally many short records. Let us assume that the Reducer of the join step has a quadratic runtime, which represents the worst case. The runtime costs of computing a self-join on one group of records with cardinality $groupSize$ and with an average record length of $avgRecLen$ can be estimated with Equation 1, assuming that the tokens in the records are sorted by a global token order allowing for a merge-join. In Figure 6, we show a plot of this cost estimation function. It shows that the costs grow exponentially with regard to the number of records in the group. The power of the increase grows exponentially with regard to the average length of the records in the group. In order

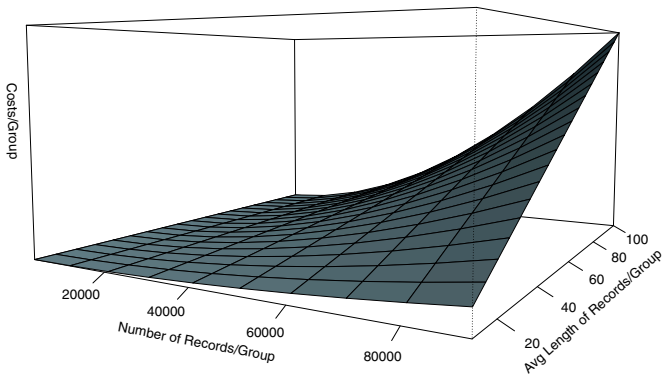

Figure 6: Cost Estimation for one data group.

to avoid a Straggling Reducer effect, our aim is to find a data grouping and replication which at least limits the maximum compute costs over all groups or ideally imposes equal computation costs for each data group. In Figure 6, equal computation costs would occur if all groups would exhibit a combination of number of records and average record lengths on an intersection of the graph with a horizontal plane.

$$\binom{groupSize}{2} * 2 * avgRecLen \tag{1}$$

Our idea is to optimize the overall computation costs with the hashing factor h as variable (Equation 2) and the constraint that the computation cost of each group may not be larger than the maximum cost threshold m, which ensures that no Reducer gets overloaded.

$$\min_{h \in \mathbb{N}^+} \sum_{group} costs(group, h), costs(group, h) \leq m \tag{2}$$

The group-wise costs within this equation could either be estimated by Equation 1 or it might use runtimes on sampled data from the statistics MapReduce step.

4. CONCLUSIONS, FUTURE WORK

In this paper, we introduced a first approach toward a distributed similarity join framework which is robust against arbitrary input dataset sizes and data characteristics such as skew. We plan to detail it out, implement it and run experiments with it. One crucial detail is to ensure that there is a sufficient number of record groups which is complete. If a Reduce instance collects only non-complete groups, straggling will still occur. Another open detail is the choice of the hash function for the join. Grouping and replication strategies from existing MapReduce-based similarity join approaches could be integrated in the proposed strategy. Especially signature creating approaches like MassJoin [5] and sophisticated grouping strategies like MRGroupJoin [6] using the pigeonhole principle are promising.

In future experiments, we are especially interested in the tradeoff between replication and group size. Furthermore, it is interesting if it pays off to use empirical runtime statistics for the join costs or simply estimate the runtime analytically.

5. ACKNOWLEDGMENTS

This work was supported by the Humboldt Elsevier Advanced Data and Text (HEADT) Center.

6. REFERENCES

[1] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *Proceedings of the 32nd international conference on Very large data bases*, pages 918–929. VLDB Endowment, 2006.

[2] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *Proceedings of the 16th international conference on World Wide Web*, pages 131–140. ACM, 2007.

[3] 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*, pages 5–5. IEEE, 2006.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, pages 137–150, 2004.

[5] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *Data Engineering (ICDE), 2014 IEEE 30th International Conference on*, pages 340–351. IEEE, 2014.

[6] D. Deng, G. Li, H. Wen, and J. Feng. An efficient partition based method for exact set similarity joins. *Proceedings of the VLDB Endowment*, 9(4):360–371, 2015.

[7] T. Elsayed, J. Lin, and D. W. Oard. Pairwise document similarity in large collections with mapreduce. In *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers*, pages 265–268. Association for Computational Linguistics, 2008.

[8] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533. IEEE, 2012.

[9] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 618–629. IEEE, 2012.

[10] G. Li, D. Deng, J. Wang, and J. Feng. Pass-join: A partition-based method for similarity joins. *Proceedings of the VLDB Endowment*, 5(3):253–264, 2011.

[11] W. Mann, N. Augsten, and P. Bouros. An empirical evaluation of set similarity join techniques. *Proceedings of the VLDB Endowment*, 9(9):636–647, 2016.

[12] A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proceedings of the VLDB Endowment*, 5(8):704–715, 2012.

[13] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 495–506. ACM, 2010.

[14] C. Xiao, W. Wang, X. Lin, J. X. Yu, and G. Wang. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems (TODS)*, 36(3):15, 2011.