Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics

Abstract

Set similarity join is an essential operation in big data analytics, e.g., data integration and data cleaning, that finds similar pairs from two collections of sets. To cope with the increasing scale of the data, distributed algorithms are called for to support large-scale set similarity joins. Multiple techniques have been proposed to perform similarity joins using MapReduce in recent years. These techniques, however, usually produce huge amounts of duplicates in order to perform parallel processing successfully as MapReduce is a shared-nothing framework. The large number of duplicates incurs on both large shuffle cost and unnecessary computation cost, which significantly decrease the performance. Moreover, these approaches do not provide a load balancing guarantee, which results in a skewness problem and negatively affects the scalability properties of these techniques. To address these problems, this paper proposes a duplicate-free framework, called FS-Join, to perform set similarity joins efficiently by utilizing an innovative vertical partitioning technique. FS-Join employs three powerful filtering methods to prune dissimilar string pairs without computing their similarity scores. Experimental results on four real datasets show that FS-Join outperforms the state-of-the-art methods by one order of magnitude on average, which demonstrates the good scalability and performance qualities of the proposed technique.

Keywords: Similarity Join, FP-tree, MapReduce.

1 Introduction

Similarity join is an essential operation that finds all pairs of records from two data collections whose similarity scores are no less than a given threshold using a similarity function, e.g., Jaccard similarity [15]. Similarity joins are widely used in a variety of applications including data integration [6], data cleaning [7], duplicate detection [18], record linkage [17] and entity resolution [8]. Most of the existing similarity join algorithms are in-memory approaches [14], [13], [3], [1], [1], [18], [11]. The era of big data, however, poses new challenges for large-scale string similarity joins and calls for new efficient and scalable algorithms. As MapReduce has become the most popular framework for big data analysis, in this paper, we study efficient and scalable string similarity joins using MapReduce. One straightforward method is to enumerate all string pairs from two string collections and use MapReduce to process all the generated pairs. However, this method is inefficient and dose not scale to large data

sets. To improve the performance, signature-based algorithms have been proposed, which use a filter-and-verification framework [15], [12], [5]. In the filter phase, the tokens/terms in each string are ordered based on a global order (e.g., lexicographical ordering or frequency-based ordering), then some of the tokens are chosen (sometimes combined with other information, e.g., length and position information) as signatures. Each signature token is treated as a key, and the input string containing the key is the corresponding value, e.g., $(B, \{B, C, I, J, K\})$ is a (key, value) pair in Figure 1. Usually, each string has more than one signature, e.g., s2 has two signature tokens B and C, and generates multiple copies of the input string. By using signature tokens as keys, strings sharing a common signature will be shuffled to the same reduce node. From the perspective of data partition, the string dataset is partitioned horizontally into several partitions, in which there are many duplicates. Two strings are considered to be a candidate pair if and only if their signatures share a common token1. In the verification phase, the candidates are verified by computing their accurate similarity scores based on similarity functions, e.g., Jaccard or Cosine similarity, to produce the final results (if the similarity score of a string pair is no less than the given threshold value, then it is a result). Figure 1 shows an example of the similarity join in existing MapReduce algorithms [15], [5]. Even though such signature-based methods improve the performance, they still have several limitations.

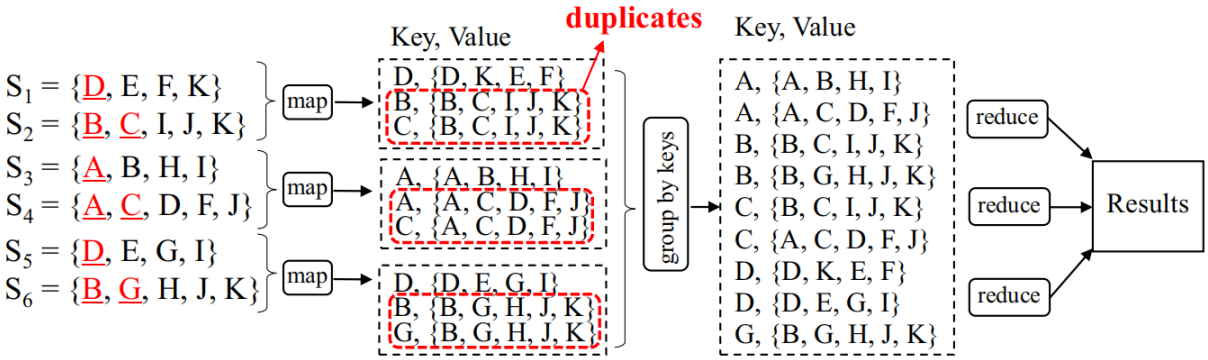

Figure 1. Example of existing similarity joins using MapReduce. Each string is tokenized as a set, and the tokens within a string are ordered based on the the lexicographical order. The underscored tokens are selected as signatures. Each signature token is regarded as a key, and the corresponding string is regarded as its value.

2 Related works

2.1 In-memory string similarity joins

There are many studies on in-memory similarity join algorithms [14], [13], [3], [1], [2], [18], which can be categorized into two kinds: set-based string similarity join (SSJoin) and character-based string similarity join (EDJoin). SSJoin considers the string as a set of tokens and uses set-based similarity functions, such as Jaccard, Dice and Cosine, while EDJoin considers the string as a character sequence and uses edit distance as its similarity function. This work focuses on SSJoin. To efficiently answer string similarity joins, most of the existing works follow a filter-and-verification framework [19], [16], [4], [10]. In the filter stage, the dissimilar string pairs are pruned based on some efficient filtering methods, e.g.,

prefix- filter, and the candidates are generated. In the verification stage, the candidates are verified by computing the accurate similarity score. More precisely, [3] proposed prefix filtering method by utilizing prefix tokens as signatures. [18] improved the prefix filtering by integrating the position information and length information. [2] built inverted indices for all the strings, where keys are tokens in the signatures and values are the corresponding strings containing the tokens. [19], [18] proposed more optimization techniques for the inverted list. [16] extended the fixed-length based prefix filtering and proposed a variable-length prefix filtering. For the same string, [16] extracts more tokens than fixed-length based prefix filtering methods and constructs multiple inverted indices incrementally. [4] explored various global orderings and proposed a multiple prefix filtering based algorithm. The multiple prefix filters are applied in an incrementally manner. [10] designed a tree based index for prefixes of multiple attributes and proposed one cost model to guide the index construction. Different from these studies, in this paper we focuses on how to support large-scale similarity joins using MapReduce.

2.2 MapReduce-based string similarity joins

To support string similarity joins over big data, many recent contributions focus on implementing algorithms on Map-Reduce [15], [12], [5]. More precisely, [15] proposed a signature-based algorithm, called RIDPairsPPJoin, which follows the filter-and-verification framework. In the filter phase, it creates signatures for each string by choosing a set of prefix tokens. Each prefix token is considered as a key, and a string including the key is regarded as the corresponding value. The strings with the same signatures are transmitted into one group for further verification. In the verification phase, an inverted index for each group of strings is created to accelerate processing, and filtering methods are applied to further prune dissimilar pairs. However, RIDPairsPPJoin has two limitations: (1) a string may be duplicated multiple times, since multiple tokens are selected as signatures, which incur high shuffle cost and redundant computations; and (2) the values in each Reduce task are a list of strings containing the same key, which have different sizes. Therefore, there is no guarantee of load balancing in the Reduce nodes. It is reported that RIDPairsPPJoin dose not scalable well [12]. [12] proposed a new algorithm V-Smart-Join for similarity joins on multisets and vectors. V-Smart-Join performs similarity joins in two phases: Join and Similarity. In the Join phase, it contains several MapReduce jobs and provides different implementations, including Online-Aggregation, Lookup and Sharding. In the Similarity phase, the partial results are aggregated to get final candidates pairs. According to [12], Online-Aggregation has the best scalability among the three implementations. In the Join phase, each token of every strings is outputted as a key. This is like building an inverted index for all tokens in the data set on HDFS. V-Smart-Join has the same issues (duplications and load balancing problems) as RIDPairsPPJoin. In addition, it has two extra limitations: (1) the cost of enumerating each pair of rids in each inverted list is expensive; and (2) no filtering is applied during the join process, which results in a high number of false positives, extra shuffle cost and unnecessary computation. [5] proposed a similarity join method, called MassJoin, for long strings based on their centralized algorithm in [9]. In [5], each string s in S is partitioned to even segments and all of them as its signatures; for each string t in T, it must generate many signatures to ensure there is a common signature with s that with length in $\lceil |t| \times \theta \rceil \leq |s| \leq \lfloor |t|/\theta \rfloor$. If $\theta = 0.8$ and $|t| = 100$, the length range is [80,125]. So, for each integer from 80 to 125, string t will

generate signatures separately. It also has the same issues as RIDPairsPPJoin. To solve the duplication and load balancing issues, in this paper, we propose a new framework for similarity join, called FS-Join. FS-Join applies a vertical partitioning method to partition each strings into disjoint segments. All the segments in a same partition constitute a fragment. The fragments have same (or similar) sizes. In the Map phase, FS-Join uses the partition ids as keys and corresponding segments as values. Since segments are disjoint, there is no duplication. In addition, in the Reduce phase, all the segments belonging to the same fragment are shuffled into a same Reduce node. Due to the same sizes of fragments, FS-Join guarantees proper load balancing. Table I summarizes the comparison of FS-Join and the state-of-the-art methods, i.e., RIDPairsPPJoin [15], VSmart-Join [12] and MassJoin [5].

3 Method

3.1 Overview

Figure 2 shows the architecture of FS-Join, which contains three phases: Ordering, Filtering and Verification.

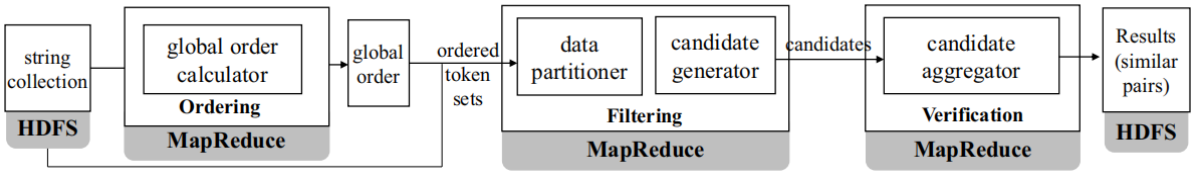

Figure 2. architecture of FS-Join

The first step of FS-Join is to obtain a global order. FS-Join adopts the ordering method proposed in [15], that is (1) first calculating the frequency for each token, then (2) sorting the tokens in an ascending order in terms of token's frequency. The global order calculator of FS-Join uses one MapReduce job to compute the global order for the string collection in HDFS. The details of ordering phase are omitted here due to the space limitation, we recommend readers to find the detailed ordering algorithm in [15].

In the filtering phase, FS-Join uses one MapReduce job to generate candidate string pairs by pruning dissimilar pairs without computing their accurate similarity scores. This is the core operation of FS-Join and has two phases: partitioning phase and join phase. More precisely, the data partitioner first receives the global order returned from the previous ordering step and selects a number of pivots from ordered sequences. Next, it sorts all the tokens in each received strings according to the global order. Then, the data partitioner partitions each string into several segments based on a set of carefully chosen pivots. Finally, FS-Join treats the partition id as the key and the segment in the corresponding partition as the value in the Map phase. The segments belonging to the same partition are shuffled to the same reduce node. In each reducer, the candidate generator of FS-Join generates candidates by using a prefix-based inverted index and several filtering methods, e.g., length filtering, segment-aware filtering. The output of the i-th reducer is a list of (p, c) pairs, where $p = (s, t)$ is a string pair while c is the number of common tokens between s and t in the i-th partition, i.e., the i-th segment of s and t.

Finally, FS-Join uses one MapReduce job to verify the candidates generated in the filter phase. If the final number of common tokens of a string pair is greater than a threshold value, then it is an answer. Otherwise, it is not. The details will be discussed in 3.3.

3.2 Vertical Partitioning

This paper proposes a novel partitioning method, called vertical partitioning, which is used by FS-Join to support efficient string similarity joins. The high level idea of the vertical partitioning is that (1) the vector space is very sparse when a string dataset is transformed into a vector space model; and (2) when splitting the vector space into disjoint fragments, each fragment only contains small subset of the strings.

Selecting a proper global ordering and the pivots are two important aspects of vertical partitioning for string datasets. Performing parallel similarity join using MapReduce, the workload balancing and the shuffle cost should be taken into consideration to get high performance. In this paper, the authors adopt the ascending order of term frequency as the global ordering. Taking this global ordering, we can identify the most popular tokens and use this information to implement an effective load balancing technique. Regarding the selection of pivots, there are two problems that should be considered: (1) how to select the pivots, and (2) how many pivots should be selected.

This paper proposed a pivot selection method that uses the tokens' distribution. This method aims to generate fragments with the same number to tokens. Thus, we select the pivots that can partition the total term frequency of tokens in the global orderfing evenly. The pivots set can be formalized as below($N_p = |\mathcal{P}|$):

$$\mathcal{P} = \{\mathcal{O}_i | i = k \times |\mathcal{O}|/N_p, 0 < k \leq N_p\}$$

Assume that all the computing nodes have the same memory. Let N_c be the number of nodes used to perform joins (the same to the number of fragments), D be the size of a given dataset, M be the memory size. The number of pivots is defined to be $N_c - 1(D/M \leq N_c)$.

Algorithm 1: FS-Join Algorithm

```
1  //Ordering phase
2  SetUp(context)
3  O ← Load Global Ordering;
4  P ← PiovtsSelection(O, N);

5  //Filtering phase
6  Map(rid, string)
7  segments ← VerticalPartition(string, O, P);
8  foreach segment ∈ segments do
9      context.write(partitionID, <segment, segInfo>);

10 Reduce(partitionID, list(<segment, segInfo>))
11 Candidates ← PerformJoin(<segment, segInfo>);
12 foreach candidate ∈ Candidates do
13     context.write(RidPair, Count);

14 //Verification phase
15 Map(RidPair, Count)
16 context.write(RidPair, Count);
17 Reduce(RidPair, list(Count))
18 sim ← Verification(list(Count));
19 if sim ≥ θ then
20     context.write(RidPair, sim);
```

3.3 FS-Join Algorithm

In this section, we describe two key phases of FS-Join: generating candidates and verifying candidates. To perform similarity joins using MapReduce on sharednothing clusters, data duplication is a common technique utilized to improve parallel processing [15], [5]. Since similarity join is a pairwise-comparison problem, it inevitably generates vast volumes of data duplicates and high shuffle cost. These two factors negatively impact the overall performance. More precisely, the existing MapReduce-based string similarity join algorithms [15], [5] have two limitations: (1) generating many duplicates, which causes high shuffle and computation costs; and (2) incurring on load balancing problems. To solve these problems, we propose a new framework for similarity joins that uses the vertical partitioning.

We first use a MapReduce job to implement vertical partitioning in the Map phase and perform the join in the Reduce phase, as shown in Figure 1. The Algorithm 1 is given. In the MapReduce framework, there is a setup method before the map task. We use setup to load the global ordering $\mathcal{O}$ outputted by the last MapReduce Job. Then, we select N_p pivots P using the global ordering $\mathcal{O}$ (Line 3 - 4). The pivots split the sorted token set $\mathcal{O}$ into $N_p + 1$ segments $Seg_k (0 \leq k \leq N_p + 1)$. For each record S_i in a mapper, we sort the tokens of S_i using $\mathcal{O}$ and split the sorted token set into several segments Seg_i^k using $\mathcal{P}$ (Line 7). The integer k (partitionID) is the sequence number of $\mathcal{F}_k$ that Seg_i^k belongs to. For each segment Seg_i^k, the mapper outputs the $(k, <Seg_i^k, segInfo>)$ pair (Line 9). The segments Seg_i^k $(0 \leq i \leq |S|)$ from different strings with key k belong to $\mathcal{F}_k$ and are shuffled to the same

Reduce node. In the Reduce, the segments from the same $\mathcal{F}_k$ will be processed to compute the number of common tokens of each pair of segments. Doing this like performing joins in relational databases. So, this can be done by loop join, index join, directly (Line 11). During the join process, the position aware filtering technique is applied to prune false positives. Finally, the record id pairs(key) and the number of common tokens(value) will be outputted (Line 13). In this MapReduce job, all the strings are spitted into several segments and shuffled to reduce nodes without data duplication. Then, we get the partial results, the number of common tokens of different segments from all possible similar pairs. Observe that, [15], [5] choose the tokens in signatures as the keys, which results in large number of duplicates, as shown in Figure1. However, FS-Join uses partition ids as keys, which has no duplicates as shown in Figure 3.

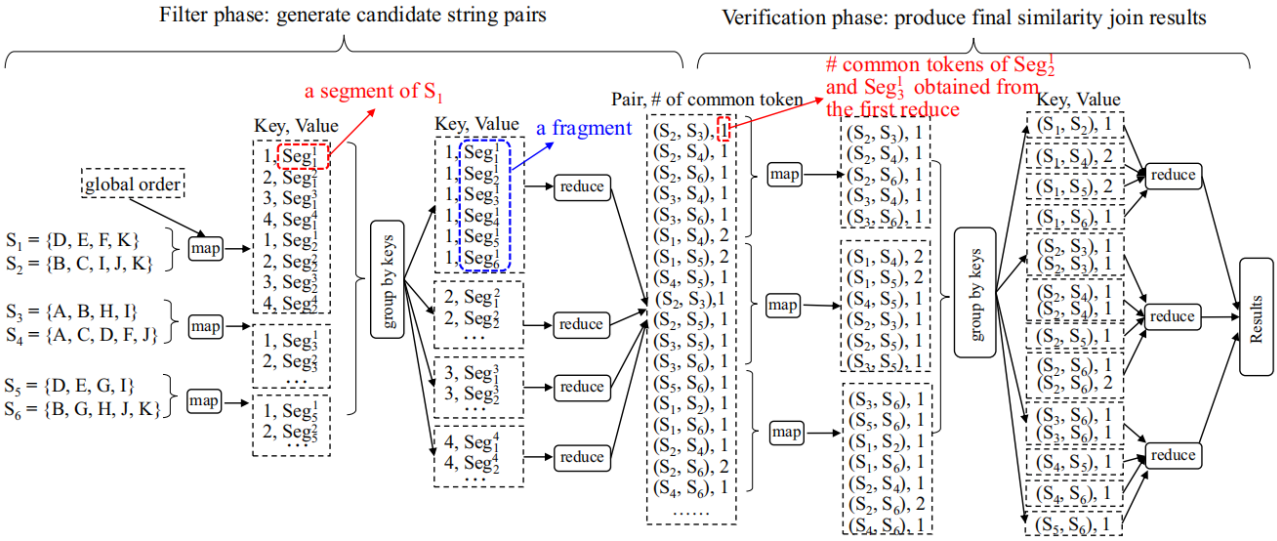

Figure 3. Computation framework of FS-Join

4 Implementation details

4.1 Comparing with the released source codes

There is no related source codes are available, i achieve the code of FS-Join by partitioning them into several sections. For example,I implement the filtering stage of string connection through the filter class. First, map each token in the string through the map data structure, and use the pivot selection function to divide the segments, and finally integrate it into each node for further processing——verify. In the verification phase, the main implementation content is to use the *Jaccard* function on each node to calculate the similarity between strings. If it meets the given threshold, the expected pair will be output.

4.2 Experimental environment setup

All the experiments are implemented on a 16-machine (one master and 15 slaves) cluster running Apach Hadoop 2.7 and OpenJDK 1.8.0, considering the representative algorithms all adopt Hadoop platform. Each machine in the cluster is a CentOS 6.5 server that has two Xeon E5-2680 2.8GHz of

10 cores CPU, 64GBs of RAM, 1TB hard disk and 1GBit Ethernet interconnect. To maximize the utilization of our available resources, we optimize resource allocation in Hadoop, as the configure shown in Table.1.

Table 1. Experimental environment setup

Parameter	Value	Description
mapreduce.map.java.opts	4096MB	map task max memory
mapreduce.reduce.java.opts	20240MB	reduce task max mem.
yarn.scheduler.maximum-allocation-mb	4000MB	job max memory
yarn.scheduler.minimum-allocation-mb	5120MB	job min memory
yarn.nodemanager.resource.memory-mb	41440MB	node max available pgysical memory
yarn.nodemanager.resource.cpu-vcores	20	YARN available virtual CPU count

Since reducers need to buffer data, we allocate 4x more memory to reduce tasks than map tasks. By default we set the number of reducers same with that of nodes to maximally avoid resource bottlenecks, as the lowest processing node depend the overall run time.

4.3 Dataset

We conduct experiments on four real-world datasets Facebook[1], Kosarak[2], Enron[3], Accidents[4]. Facebook contains application usage records for 297K users among 8,1000 applications. Kosarak collects anonymous click-stream data of articles each user has browsed from a on-line news portal. Each record represents a set of articles that the user has browsed. Enron is an e-mail collection where the e-mail bodies are divided into tokens based on spaces. Accidents is set of traffic accidents containing detailed information on the different circumstances. In our experiments, we randomly selected almost 3 million records from each of the four datasets to construct two collections as input,ensuring that there are no duplicate sets with the same identifier in different collections. The detailed information of all datasets are shown in Table.2.

	$\|\mathcal{S}\|$	avg, mix, mxn $\|s_i\|$			$\|\mathcal{S}'\|$	avg, min, max $\|\mathcal{S}'(a_i)\|$		
Facebook	297K	19.6	1	774	13604	428.8	1	253963
Kosarak	300K	8.1	1	2498	31976	76.0	1	182107
Enron	300K	140.8	1	3162	791165	53.4	1	239119
Accidents	300K	33.7	1	51	458	22131.3	1	299969

Table 2. Dataset details

[1]http://odysseas.calit2.uci.edu/doku.php/public:online$_s ocial_n etworks$
[2]http://fimi.uantwerpen.be/data/kosarak
[3]http://www.cs.cmu.edu/enron
[4]http://fimi.uantwerpen.be/data/accidents.dat

4.4 Main contributions

In this research report, I achieve FS-Join's code, and evaluate FS-Join's capbilities by running string similarity joins on three real-world datasets. I first compare our algorithms with the PP-Join algorithm under different thresholds on three datasets. The results show that FS-Join outperforms the state-of-the-art methods in all the datasets. Then we conduct more experiments to evaluate different features of FS-Join.

- I achieve the not open source code of FS-Join.
- I evaluate the runtime of our algorithms with the variations of data scale in the cluster.

5 Results and analysis

Table.3 presents the running time of string similarity join queries (self-join). From the experimental results, we can observe that FS-Join performs and scales significantly better than PP-Join techniques in enron30w dataset. But it fails to perform well in fb30w and accidents30w datasets. In order to study the scalability of FS-Join, we conducted experiments to test its execution time with the variation of data scale.

Table.4 shows the execution time of FS-Join and PP-Join on two datasets under different data sizes. In this experiment, we used four different scales for each dataset, 1w, 5w, 10w and 20w. Since the similarity join is a pairwise-comparison problem, the number of candidate pairs increases quadratically on the size of dataset. From Table.4, we can observe that when the data size increases by 2X, the time FS-Join costs will increase less than 33% in most cases(under the same threshold). Overall, FS-Join has good scalability with the number of computing nodes in the cluster.

Table 3. Runtime

dataset	enron30w(ks)	fb30w(ks)	accidents30w(ks)
PP-Join	4.038	0.229	6.228
FS-Join	2.075	1.134	29.439

Table 4. Scalability

	PP-Join(s)	FS-Join(s)
enron1w	56.34	53.84
enron5w	99.50	63.22
enron10w	572.44	287.20
enron20w	2140.261	1005.84

6 Conclusion and future work

The string similarity join is a key data processing operator in many Big Data applications. In this paper we proposed FS-Join, a highly scalable MapReduce-based string similarity join algorithm based on a novel partitioning technique (Vertical Partitioning). Moreover, this paper proposed highly effective optimization techniques and introduced new filtering approaches. I thoroughly analyze the performance of FS-Join theoretically and experimentally. The results show that it does not always perform so well in comparative experiments with PP-JOIN under different data sets. After analysis, it is because after filtering using the Join algorithm, the size of the candidate set is still redundant.

In the future, I plan to further reduce the candidate set to improve the calculation efficiency of FS-Join, such as using the length filtering strategy to filter the data in the segments in each node before performing subsequent operations, or using prefix tree data The structure replaces the inverted list and reduces the number of traversals.

References

[1] Arvind Arasu, Venkatesh Ganti, and Raghav Kaushik. Efficient exact set-similarity joins. In Proceedings of the 32nd international conference on Very large data bases, pages 918–929, 2006.

[2] Roberto J. Bayardo, Yiming Ma, and Ramakrishnan Srikant. Scaling up all pairs similarity search. In Proceedings of the 16th International Conference on World Wide Web, WWW '07, page 131–140, New York, NY, USA, 2007. Association for Computing Machinery.

[3] Surajit Chaudhuri, Venkatesh Ganti, and Raghav Kaushik. A primitive operator for similarity joins in data cleaning. 22nd International Conference on Data Engineering (ICDE'06), pages 5–5, 2006.

[4] Chunqing LI Chuitian RONG, Yasin N. SILVA. String similarity join with different similarity thresholds based on novel indexing techniques. Frontiers of Computer Science, 11(2):307, 2017.

[5] Dong Deng, Guoliang Li, Shuang Hao, Jiannan Wang, and Jianhua Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. 2014 IEEE 30th International Conference on Data Engineering, pages 340–351, 2014.

[6] Mauricio A. Hernández and Salvatore J. Stolfo. The merge/purge problem for large databases. In SIGMOD '95: Proceedings of the 1995 ACM SIGMOD international conference on Management of data, pages 127–138, New York, NY, USA, 1995. ACM Press.

[7] Mauricio A. Hernández and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. Data Mining and Knowledge Discovery, 2:9–37, 1998.

[8] Lars Kolb, Andreas Thor, and Erhard Rahm. Load balancing for mapreduce-based entity resolution. 2012 IEEE 28th International Conference on Data Engineering, pages 618–629, 2011.

[9] Guoliang Li, Dong Deng, Jiannan Wang, and Jianhua Feng. Pass-join: A partition-based method for similarity joins. Proc. VLDB Endow., 5(3):253–264, nov 2011.

[10] Guoliang Li, Jian He, Dong Deng, and Jian Li. Efficient similarity join and search on multi-attribute data. In Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD '15, page 1137–1151, New York, NY, USA, 2015. Association for Computing Machinery.

[11] Jiaheng Lu, Chunbin Lin, Wei Wang, Chen Li, and Haiyong Wang. String similarity measures and joins with synonyms. In Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, page 373–384, New York, NY, USA, 2013. Association for Computing Machinery.

[12] Ahmed Metwally and Christos Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. Proc. VLDB Endow., 5(8):704–715, apr 2012.

[13] Sunita Sarawagi and Alok Kirpal. Efficient set joins on similarity predicates. In Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, SIGMOD '04, page 743–754, New York, NY, USA, 2004. Association for Computing Machinery.

[14] Wenbo Tao, Dong Deng, and Michael Stonebraker. Approximate string joins with abbreviations. Proc. VLDB Endow., 11(1):53–65, sep 2017.

[15] 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, SIGMOD '10, page 495–506, New York, NY, USA, 2010. Association for Computing Machinery.

[16] Jiannan Wang, Guoliang Li, and Jianhua Feng. Can we beat the prefix filtering? an adaptive framework for similarity join and search. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12, page 85–96, New York, NY, USA, 2012. Association for Computing Machinery.

[17] William E. Winkler. The state of record linkage and current research problems. 1999.

[18] Chuan Xiao, Wei Wang, Xuemin Lin, and Jeffrey Xu Yu. Efficient similarity joins for near duplicate detection. In Proceedings of the 17th International Conference on World Wide Web, WWW '08, page 131–140, New York, NY, USA, 2008. Association for Computing Machinery.

[19] Junfeng Zhou, Ziyang Chen, and Jingrong Zhang. Sejoin: An optimized algorithm towards efficient approximate string searches. In Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval, SIGIR '11, page 1249–1250, New York, NY, USA, 2011. Association for Computing Machinery.