MassJoin: A MapReduce-based Method for Scalable String Similarity Joins

Dong Deng[†], Guoliang Li[†], Shuang Hao[†], Jiannan Wang[†], Jianhua Feng[†], Wen-Syan Li[‡]
[†]*Department of Computer Science, Tsinghua University, Beijing, China*
[‡]*SAP Labs, Shanghai, China*
{dd11, haos13, wjn08}@mails.thu.edu.cn; {liguoliang,fengjh}@thu.edu.cn; wen-syan.li@sap.com

***Abstract*—String similarity join is an essential operation in data integration. The era of big data calls for scalable algorithms to support large-scale string similarity joins. In this paper, we study scalable string similarity joins using MapReduce. We propose a MapReduce-based framework, called MASSJOIN, which supports both set-based similarity functions and character-based similarity functions. We extend the existing partition-based signature scheme to support set-based similarity functions. We utilize the signatures to generate key-value pairs. To reduce the transmission cost, we merge key-value pairs to significantly reduce the number of key-value pairs, from cubic to linear complexity, while not sacrificing the pruning power. To improve the performance, we incorporate "light-weight" filter units into the key-value pairs which can be utilized to prune large number of dissimilar pairs without significantly increasing the transmission cost. Experimental results on real-world datasets show that our method significantly outperformed state-of-the-art approaches.**

I. INTRODUCTION

Data integration has received significant attention in the last three decades, because it can combine heterogenous data from different sources and provide a unified view of these data. The string similarity join, which, given two collections of strings, finds all similar string pairs, is an essential operation in data integration. The similarity between two strings is usually quantified by similarity functions. There are two main types of similarity functions (see Section II): set-based similarity functions (e.g., Jaccard) and character-based similarity functions (e.g., Edit distance). String similarity joins have many real-world applications, e.g., entity resolution, copy detection, and document clustering.

Most of existing similarity-join methods used in-memory algorithms. The era of big data poses new challenges for large-scale string similarity joins and calls for new scalable algorithms. MapReduce provides a programming model for processing large-scale data, and in this paper we study scalable string similarity joins using MapReduce. A naive method is to enumerate all string pairs from the two collections and use MapReduce to process these string pairs. However this method is rather expensive for large string sets. To address this problem, Vernica et al. [15] proposed a prefix filtering based method, which used a filter-and-verification framework. In the filter step, they selected some tokens from each string and generated a set of candidate pairs who share a common token. In the verification step, they verified the candidate pairs to generate the final answers. One big limitation of this method is low pruning power. As a single token is very short and usually has low selectivity, many dissimilar pairs will share a same token and cannot be pruned.

To address this limitation, we propose a MapReduce-based framework, called MASSJOIN, for scalable string similarity joins, which supports both set-based similarity functions and character-based similarity functions. We also adopt a filter-and-verification framework. In the filter step, we generate the signatures for each string and prove that two strings are similar only if they share a common signature. We utilize this property to generate the candidate pairs. In the verification step, we verify the candidate pairs to generate the final results. One big challenge is to generate high-quality signatures. PASSJOIN [12] proposed a high-quality partition-based signature scheme for the edit distance function. We extend the partition-based signature scheme to support set-based similarity functions. It is worth noting that the extension is nontrivial (see Section III), because the partition number is fixed for edit distance while the partition numbers for set-based similarity functions are not.

To use MapReduce, we take the signatures as keys and the strings as values to generate key-value pairs. Then we use the key-value pairs to compute the candidate pairs that share a same key (see Section IV). However this method may generate large numbers of key-value pairs and leads to huge transmission cost. For example, considering the Jaccard function, this method generates $\mathcal{O}(\ell^3)$ key-value pairs for a string with length ℓ. To address this issue, we merge key-value pairs to significantly reduce the number of key-value pairs but without sacrificing the pruning power (see Section V). For example, we can reduce the number from $\mathcal{O}(\ell^3)$ to $\mathcal{O}(\ell)$.

To improve the performance, we incorporate "light-wight" filter units into the key-value pairs which can be utilized to prune large numbers of dissimilar pairs without significantly increasing the transmission cost (see Section VI).

In summary, we make the following contributions.

- We extend the partition-based signature scheme to support set-based similarity functions. We propose a scalable MapReduce-based framework to support both set-based and character-based similarity functions.
- We devise a merge-based algorithm to significantly reduce the number of key-value pairs without sacrificing the pruning power.
- We develop a light-weight filter unit based method to prune large numbers of dissimilar pairs while not significantly increasing the transmission cost.
- We have implemented our method and experimental results on real-world datasets show that our method significantly outperforms state-of-the-art approaches.

The structure of the rest paper is as follows. We formulate our problem and review related works in Section II. We extend existing partition-based signature scheme to support set-based similarity functions in Section III. Section IV presents our MASSJOIN framework. We devise a merge-based method to reduce the number of key-value pairs in Section V and develop a light-weight filter unit based method to reduce the number of candidate pairs in Section VI. We show the experimental results in Section VII and give the conclusion in Section VIII.

II. PRELIMINARY

A. Problem Definition

The similarity join problem finds all similar string pairs from two given string collections, where the similarity between two strings is usually quantified by similarity functions. Given a similarity function SIM and a similarity threshold δ, two strings r and s are similar if and only if $\text{SIM}(r,s) \geq \delta$. There are two main types of similarity functions, set-based similarity functions and character-based similarity functions.

Set-based similarity functions first tokenize each string into a set of tokens, where a token can be either a word or a n-gram. In the paper we suppose each token is a tokenized word. For simplicity, strings and token sets are interchangeably used if there is no ambiguous. There are three well-known set-based similarity functions, Jaccard Similarity, Dice Similarity, and Cosine Similarity, defined as below.

$$\text{JAC}(r,s)=\frac{|r\cap s|}{|r\cup s|},\ \text{COS}(r,s)=\frac{|r\cap s|}{\sqrt{|r|\cdot|s|}},\ \text{DICE}(r,s)=\frac{2|r\cap s|}{|r|+|s|},$$

where $|\cdot|$ denotes the size of a set. For example, suppose r ="`Barack H Obama II`" and s ="`Barack Obama II`". Then we have $|r|=4, |s|=3, |r\cap s|=3$, and $|r\cup s|=4$. Thus, $\text{JAC}(r,s)=\frac{3}{4}$, $\text{COS}(r,s)=\frac{3}{\sqrt{12}}$ and $\text{DICE}(r,s)=\frac{6}{7}$.

Character-based similarity functions are defined based on the number of character operations to transform one string to another. Edit Distance (ED) is a well-known character-based similarity function. The ED of two strings is the minimum number of edit operations needed to transform one string to another, where the permitted edit operations include insertion, deletion and substitution. For example, given r ="`micheal`" and s ="`michael`", we have $\text{ED}(r,s)=2$.

Next we formulate the similarity-join problem.

Definition 1 (Similarity Joins): Given two string collections $\mathcal{R}$ and $\mathcal{S}$, a similarity function SIM (or a distance function DIS) and a similarity threshold δ (or a distance threshold τ), a similarity join finds all string pairs $\langle r\in\mathcal{R}, s\in\mathcal{S}\rangle$ such that $\text{SIM}\ (r,s)\geq\delta$ (or $\text{DIS}\ (r,s)\leq\tau$).

For example, consider the two string sets in Table I. Suppose the Jaccard threshold is $\delta=0.8$. The similarity join finds a similar pair $\langle r_2, s_2\rangle$ with $\text{JAC}(r_2,s_2)=0.8$.

B. Related Work

Partition-based Method: PASSJOIN [12], [11] is the state-of-the-art similarity-join algorithm for edit distance. It won the champion in the recent similarity-join competition organized by EDBT 2013[1]. PASSJOIN employs a filter-and-verification framework. In the filter step, it generates signatures for strings. If two strings are similar, they must share a common signature. It prunes large numbers of dissimilar pairs based on this idea and the survived pairs are called candidate pairs. In the verification step, it verifies the candidate pairs to generate the final answers.

TABLE I. TWO STRING COLLECTIONS: $\mathcal{R}$ AND $\mathcal{S}$

	id	$string$	$size$
$\mathcal{R}$	r_1	conference on service information management	5
	r_2	policy on service information management	5
	r_3	the policy on service	4
$\mathcal{S}$	s_1	the conference on information management	5
	s_2	service information management policy	4
	s_3	conference on information management policy	5

One big challenge in PASSJOIN is to generate high-quality signatures for two strings. Suppose the edit-distance threshold is τ. Given two strings r and s, it partitions r into $\tau+1$ disjoint segments, and proves that if s is similar to r, s must contain a substring that matches one of r's segments, based on the pigeon-hole theory. Thus for string r, it generates $\tau+1$ signatures. For string s, it selects some substrings of s as its signatures.

However PASSJOIN cannot support set-based similarity functions, because the number of segments (e.g., $\tau+1$) is not fixed for these functions. We extend PASSJOIN to support set-based similarity functions and propose a scalable framework MASSJOIN for large-scale string similarity join using MapReduce. Moreover, to reduce the transmission cost, we merge the selected substrings and decrease the number of signatures from $\mathcal{O}(l^3)$ to $\mathcal{O}(l)$ for set-based similarity functions, where l is the string length, and from $\mathcal{O}(\tau^3)$ to $\mathcal{O}(\min(\tau^2,l))$ for character-based similarity functions.

Similarity Join Using Map-Reduce: There are a lot of works on implementing database operators using Map-Reduce framework [10]. Vernica et al. [15] proposed a similarity join method using MapReduce which utilized the prefix filtering to support set-based similarity functions. They selected a subset of tokens as signatures and proved that two strings are similar only if their signatures share common tokens. For each string, they used each token in its prefix as a key and the string as a value to generate the key-value pairs. As a single token is very short and usually has low selectivity, these methods have two disadvantages. First, many dissimilar pairs may share the same token, and they will generate many false positives and thus lead to poor pruning power. Second, a single token may lead to a skewed problem and there may be large numbers of strings sharing the same key. Thus the reducer with such keys will have a heavy workload. Metwally et al. [13] proposed a 2-stage algorithm V-SMART-Join for similarity joins on sets, multisets and vectors. They also used a single token as a key and had the same issues as [15]. Afrati et al. [1] proposed multiple algorithms to perform similarity joins in a single MapReduce stage. They analyzed the map, reduce and communication cost. However for long strings, it is rather expensive to transfer the strings using a single MapReduce stage. Kim et al. [9] addressed the top-k similarity join problem using MapReduce,

[1] http://www2.informatik.hu-berlin.de/~wandelt/searchjoincompetition2013

which is different from our problem. Deng et al [6] proposed to use Map-Reduce similarity joins to solve the webtable column understanding problem which is also different from our problem.

In-Memory Similarity Joins: There are many studies on in-memory similarity join algorithms [21], [4], [16], [20], [2], [8], [7], [18], [17], [22], [23], [3], [14], [11], [22]. Bayardo et al. [3] first proposed the prefix-filter framework for similarity joins. Xiao et al. [23] improved the prefix-filter framework by introducing position filtering. ED-Join [21] extended prefix filtering to support edit distance and used location-based mismatch techniques to shorten prefixes and content-based mismatch to filter dissimilar pairs. Wang et al. [18] proposed an adaptive prefix-filter framework to dynamically select prefixes with different lengths. Wang et al. [16] proposed Trie-Join which utilizes a trie structure to recursively perform prefix filter. Arasu et al. [2] proposed PartEnum which partitions strings into pieces and enumerates deletion neighborhoods on the pieces as signature to do similarity joins. Wang et al. [20] proposed VChunkJoin to use qchunks with different lengths as signatures and employ the prefix-filter framework to do similarity joins. Wang et al. [17], [19] devised a new kind of similarity functions and proposed efficient algorithm on the new similarity functions. Qin et al. [14] proposed an asymmetric signature for both similarity join and search problems. Chaudhuri et al. [4] proposed a primitive operator for similarity joins. Xiao et al. [22] studied top-k similarity joins using adaptive prefix filtering. Jacox et al. [8] studied similarity joins under metric space. Different from these studies, in this paper we focuses on how to support large-scale similarity joins using MapReduce.

MapReduce: MapReduce [5] is a famous framework proposed by Google to facilitate processing large-scale data in parallel. The MapReduce program runs on a large cluster with multiple nodes. The input data files are divided into small file splits and distributed in a distributed file system (DFS). MapReduce includes a map phase, a shuffle phase and a reduce phase to process the data. Each map node reads the file splits on the node, processes the input $\langle key, value\rangle$ pairs, and emits intermediate $\langle key, value\rangle$ pairs. The intermediate $\langle key, value\rangle$ pairs are shuffled based on the keys and transferred to the reduce nodes. All the intermediate $\langle key, value\rangle$ pairs with same key must be shuffled to the same reduce node. Each reduce node receives a key-value pair $\langle key, list(value)\rangle$, where $list(value)$ is a list of values sharing the same key, processes the pair, and writes its output to DFS.

III. Signatures for Set-based Similarity Functions

To avoid enumerating all string pairs from the two given string sets, we adopt a filter-and-verification framework. We extend the partition-based signature scheme designed for edit distance [12] to support set-based similarity functions in this section. Notice that the extension is non-trivial, because PassJoin relies on a given edit-distance threshold and generates a fixed number of signatures. However for set-based similarity functions, the number depends on the string lengths. We need to deduce the number of signatures and devise new algorithms to generate signatures. To address this problem, we first discuss how to generate signatures for two strings in Section III-A and then extend the method to support two string sets in Section III-B. Finally, we give the signature complexity in Section III-C.

A. Signatures for Two Strings r and s

Given two token sets r and s, and a jaccard threshold δ, we sort each token set based on a universal order, e.g., alphabetical order, and obtain two sorted token lists. For simplicity, r and s are referred to their corresponding sorted token lists if there are no ambiguous. If r and s are similar w.r.t Jaccard threshold δ, i.e., $\text{JAC}(r,s) = \frac{|r\cap s|}{|r\cup s|} = \frac{|r\cap s|}{|r|+|s|-|r\cap s|} \geq \delta$, we can deduce $|r\cap s| \geq \frac{\delta}{1+\delta}(|r|+|s|)$. The total number of different tokens between r and s is $|r-s|+|s-r|$. Since $|r-s|$ and $|s-r|$ are two integers, $|r-s| \leq |r| - |r\cap s| \leq \lfloor \frac{|r|-\delta|s|}{1+\delta} \rfloor$ and $|s-r| \leq |s| - |r\cap s| \leq \lfloor \frac{|s|-\delta|r|}{1+\delta} \rfloor$. Let $U = \lfloor \frac{|r|-\delta|s|}{1+\delta} \rfloor + \lfloor \frac{|s|-\delta|r|}{1+\delta} \rfloor$ which should be an upper bound of $|r-s|+|s-r|$. We split r into $U+1$ disjoint segments. Based on the pigeon-hole theory, if s is similar to r, s must contain a substring which matches a segment of r. We use the segments of r as r's signatures and select some substrings of s as s's signatures. Then if r and s are similar, they must share a common signature. Next we formally discuss how to generate the signatures for r and s.

Signatures for r: There are multiple ways to partition r into $U+1$ segments. Here we use an even partition scheme as an example, where all $U+1$ segments have nearly the same length. Formally, given a string r with length $\ell = |r|$, let $k = \ell - \lfloor \frac{\ell}{U+1} \rfloor * (U+1)$. The last k segments of string r have a length of $\lceil \frac{\ell}{U+1} \rceil$ and the first $U+1-k$ segments have a length of $\lfloor \frac{\ell}{U+1} \rfloor$. Let $l(U+1, i, \ell)$, abbreviated as l_i^ℓ, denote the length of the i-th segment of r. Obviously $l_i^\ell = \lfloor \frac{\ell}{U+1} \rfloor$ if $i \leq U+1-k$; otherwise $\lceil \frac{\ell}{U+1} \rceil$. Let $p(U+1, i, \ell)$, abbreviated as p_i^ℓ, denote the start position of the i-th segment, i.e., $p_1^\ell = 1$ and $p_i^\ell = \sum_{j=1}^{j<i} l_j^\ell + 1$. The i-th segment of r is the substring of r starting from p_i^ℓ and with length l_i^ℓ, denoted by $r[p_i^\ell, l_i^\ell]$. Thus the signatures of r are triples $\langle r[p_i^\ell, l_i^\ell], i, \ell\rangle$ for $1 \leq i \leq U+1$.

Signatures for s: If s is similar to r, it requires to have a signature matching one of signatures of r, e.g., $\langle r[p_i^\ell, l_i^\ell], i, \ell\rangle$. Thus the signature of s should be in the form of $\langle s[x_i^\ell, l_i^\ell], i, \ell\rangle$ such that $\langle r[p_i^\ell, l_i^\ell], i, \ell\rangle = \langle s[x_i^\ell, l_i^\ell], i, \ell\rangle$.

Intuitively, the start position of the i-th substring of s, e.g., x_i^ℓ, can be any integer in $[1, |s| - l_i^\ell + 1]$. However this method will generate large numbers of signatures. To address this issue, we propose two signature generation methods, a *position-aware* method and a *multi-match-aware* method.

Position-aware method. Consider the i-th signature of r with start position p_i^ℓ, and a signature of s that matches the i-th signature of r with start position x_i^ℓ. If $x_i^\ell < p_i^\ell$, we can deduce that $p_i^\ell - x_i^\ell \leq |r-s|$, because r and s are sorted by a universal order, and if $r[p_i^\ell, l_i^\ell] = s[x_i^\ell, l_i^\ell]$, among the first $p_i^\ell - 1$ tokens of r and the first $x_i^\ell - 1$ tokens of s, there are at least $p_i^\ell - x_i^\ell$ tokens that appear in r and do not appear in s. Since there are totally $|r-s|$ such tokens, $p_i^\ell - x_i^\ell \leq |r-s| \leq \lfloor \frac{|r|-\delta|s|}{1+\delta} \rfloor$. Let $U_{r-s} = \lfloor \frac{|r|-\delta|s|}{1+\delta} \rfloor$, we have $p_i^\ell - x_i^\ell \leq U_{r-s}$ and $x_i^\ell \geq p_i^\ell - U_{r-s}$. Similarly, if $x_i^\ell \geq p_i^\ell$, we have $x_i^\ell - p_i^\ell \leq |s-r| \leq \lfloor \frac{|s|-\delta|r|}{1+\delta} \rfloor$.

Let $U_{s-r} = \lfloor \frac{|s|-\delta|r|}{1+\delta} \rfloor$, we have $x_i^\ell \le p_i^\ell + U_{s-r}$. Thus the position-aware method sets $x_i^\ell \in [p_i^\ell - U_{r-s}, p_i^\ell + U_{s-r}]$ which will not involve false negatives as stated in Lemma 1.

Lemma 1: The *position-aware* signature generation method will not involve false negatives.[2]

Proof Sketch: We prove it by contradiction. Suppose r and s are similar and s has a substring with start position x_i^ℓ matching the i-th segment of r and $x_i^\ell < p_i^\ell - U_{r-s}$. Obviously $|r-s| \ge p_i^\ell - x_i^\ell > U_{r-s}$ which contradicts with that $U_{r-s} \le |r-s|$. Similarly we can prove $x_i^\ell \le p_i^\ell + U_{s-r}$. ■

Multi-match-aware method. Still consider the i-th signature of r with start position p_i^ℓ, and a signature of s that matches the i-th signature of r with start position x_i^ℓ. We have $x_i^\ell \ge p_i^\ell - (i-1)$, because if $x_i^\ell < p_i^\ell - (i-1)$, $|r[1, p_i^\ell - 1] - s[1, x_i^\ell - 1]| > i-1$ and there are at least i different tokens in r and s before the i-th segment based on the length difference. In other words, if they are similar, after the i-th segment, there are at most $U+1-i-1$ mismatch tokens. Since there are $U+1-i$ segments, there must be a matching signature after the i-th segment based on the pigeon-hole theory. Obviously we can skip this one and use the latter one. Similarly, if we consider the reversed strings of r and s, we have $|s| - x_i^\ell \ge |r| - p_i^\ell - (U+1-i)$. We can deduce that $x_i^\ell \le p_i^\ell + |s| - |r| + (U+1-i)$. Thus the *multi-match-aware* method sets $x_i^\ell \in [p_i^\ell - (i-1), p_i^\ell + |s| - \ell + (U+1-i)]$ where $\ell = |r|$, and this method will not involve any false negative as stated in Lemma 3.

Lemma 2: The *multi-match-aware* signature generation method will not involve false negatives.

Proof Sketch: Consider any string r with length ℓ. The start position of its i-th segment is p_i^ℓ. If the first $i-1$ segments of r contain more than $i-1$ different tokens from s, regardless of the i-th segment, they must share another common segment in the last $U+1-i$ segments and we can drop the i-th segment. Meanwhile, if r and s are similar and s contains a substring with start position x_i^ℓ matching the i-th segment of r, the number of different tokens in the first $i-1$ segments of r cannot be smaller $p_i^\ell - x_i^\ell$. Thus we have $p_i^\ell - x_i^\ell \le i-1$, i.e., $x_i^\ell \ge p_i^\ell - (i-1)$. Symmetrically, we can get $x_i^\ell \le p_i^\ell + |s| - \ell + (U+1-i)$. ■

Interestingly, we can use the two methods simultaneously and propose a hybrid method which sets $x_i^\ell \in [\max(p_i^\ell - (i-1), p_i^\ell - U_{r-s}), \min(p_i^\ell + |s| - \ell + (U+1-i), p_i^\ell + U_{s-r})]$. This hybrid method will not involve false negatives as stated in Lemma 3, because the two methods are independent.

Lemma 3: The hybrid signature generation method will not involve false negatives.

Proof Sketch: Based on Lemma 1, we have for any two strings r and s, if s has a substring with start position x_i^ℓ matching the i-th segment of r and $x_i^\ell \notin [p_i^\ell - U_{r-s}, p_i^\ell + U_{s-r}]$, $|r-s| > U_{r-s}$ or $|s-r| > U_{s-r}$. That is to say r and s cannot be similar. Thus for any two similar strings, all there matching signatures must be within the range generated by the *position-aware* method. Similarly this claim is also true for the *multi-match-aware* method. Thus the hybrid signature generation method will not involve false negatives. ■

[2]For space constraints, we omit detailed proofs of lemmas in this paper

TABLE II. BOUNDS FOR A SIMILAR STRING PAIR $\langle r, s \rangle$ WITHIN THRESHOLD δ

	JAC	COS	DICE	ED
U_{r-s}	$\lfloor \frac{\|r\|-\delta\|s\|}{1+\delta} \rfloor$	$\lfloor \|r\| - \delta\sqrt{\|r\|\|s\|} \rfloor$	$\lfloor \|r\| - \frac{\delta(\|r\|+\|s\|)}{2} \rfloor$	-
U_{s-r}	$\lfloor \frac{\|s\|-\delta\|r\|}{1+\delta} \rfloor$	$\lfloor \|s\| - \delta\sqrt{\|r\|\|s\|} \rfloor$	$\lfloor \|s\| - \frac{\delta(\|r\|+\|s\|)}{2} \rfloor$	-
U	$U_{r-s} + U_{s-r}$			-
$\overline{U}$	$\lfloor \frac{1-\delta}{\delta}\|r\| \rfloor$	$\lfloor \frac{1-\delta^2}{\delta^2}\|r\| \rfloor$	$\lfloor \frac{2(1-\delta)}{\delta}\|r\| \rfloor$	τ
$\triangle$	$\lfloor \frac{1-\delta}{\delta}\|r\| \rfloor + 1$	$\lfloor \frac{1-\delta^2}{\delta^2}\|r\| \rfloor + 1$	$\lfloor \frac{2(1-\delta)}{\delta}\|r\| \rfloor + 1$	τ+1
L_o	$\lceil \delta\|s\| \rceil$	$\lceil \delta^2\|s\| \rceil$	$\lceil \frac{\delta}{2-\delta}\|s\| \rceil$	$\|s\|-\tau$
L_u	$\lfloor \frac{\|s\|}{\delta} \rfloor$	$\lfloor \frac{\|s\|}{\delta^2} \rfloor$	$\lfloor \frac{2-\delta}{\delta}\|s\| \rfloor$	$\|s\|+\tau$
g_δ	$\frac{1-\delta}{\delta}$	$\frac{1-\delta^2}{\delta^2}$	$\frac{2(1-\delta)}{\delta}$	-
f_δ	$\frac{(1+\delta^3)(1-\delta)^3}{\delta^3}$	$\frac{(1+\delta^6)(1-\delta^2)^3}{\delta^6}$	$\frac{16(1-\delta)^3(3\delta^2-6\delta+4}{(2-\delta)^3\delta^3}$	-

Thus the signatures of s with respect to r are $\langle s[x_i^\ell, l_i^\ell], i, \ell \rangle$ for $1 \le i \le U+1$ and $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$, where

$$\perp_i^\ell = \max(p_i^\ell - (i-1), p_i^\ell - U_{r-s}),$$

$$\top_i^\ell = \min(p_i^\ell + |s| - \ell + (U+1-i), p_i^\ell + U_{s-r}).$$

Example 1: Consider two strings r_1 and s_1 in Table I. Suppose the JAC threshold is $\delta = 0.8$. To generate the signatures for r_1, we compute $\ell = |r| = 5$, $|s| = 5$, $U = 0$, $p_1^\ell = 1$ and $l_1^\ell = 5$, and obtain the signature $\langle r_1[1,5], 1, 5 \rangle$ for r_1. To generate the signatures for s_1, we compute $U_{r-s} = 0$, $U_{s-r} = 0$, $\perp_1^\ell = 1$, and $\top_1^\ell = 1$, and obtain the signature $\langle s_1[1,5], 1, 5 \rangle$ for s_1.

B. Signatures for Two String Sets

When we join two datasets $\mathcal{R}$ and $\mathcal{S}$, many strings in $\mathcal{R}$ may be similar to many strings in $\mathcal{S}$, and we need to generalize our method to support two string sets.

Based on the length pruning, if a string s is similar to r, i.e., $\frac{|r \cap s|}{|r \cup s|} \ge \delta$, we can deduce $|s| \le |r \cup s| \le \frac{|r \cap s|}{\delta} \le \frac{|r|}{\delta}$. Similarly we can deduce $|s| \ge |r \cap s| \ge \delta|r \cup s| \ge \delta|r|$. Thus the upper bound of lengths of the possible similar strings to r is $\lfloor |r|/\delta \rfloor$ and the lower bound is $\lceil \delta|r| \rceil$. Obviously for string r we need to consider the strings with lengths in the range and generate valid signatures of r for any possible similar string s with length in $[\lceil \delta|r| \rceil, \lfloor |r|/\delta \rfloor]$. To this end, we derive an upper bound of the number of different tokens to r for all possible similar strings that are similar to r, denoted by $\overline{U}$. Obviously $\overline{U} = \max\{|r-s| + |s-r| \big| s \text{ is similar to } r\}$. As $|r-s| + |s-r| \le |r| + |s| - 2|r \cap s| \le \lfloor \frac{1-\delta}{1+\delta}(|r| + |s|) \rfloor \le \lfloor \frac{1-\delta}{1+\delta}(|r| + \frac{|r|}{\delta}) \rfloor = \lfloor \frac{1-\delta}{\delta}|r| \rfloor$, we can deduce a bound $\overline{U} = \lfloor \frac{1-\delta}{\delta}|r| \rfloor$. Thus if a string s is similar to r, we have $|r-s| + |s-r| \le \overline{U}$. Notice that we can deduce a much tighter bound of $\overline{U}$ for Jaccard as stated in Lemma 4.

Lemma 4: We can deduce a tighter upper bound $\overline{U} = \lfloor \frac{|r| - \delta\lfloor |r|/\delta \rfloor}{1+\delta} \rfloor + \lfloor \frac{\lfloor |r|/\delta \rfloor - \delta|r|}{1+\delta} \rfloor$.

Based on the upper bound $\overline{U}$, we can devise a partition scheme for two string sets as below.

Signatures for $r \in \mathcal{R}$: We partition r into $\triangle = \overline{U} + 1$ segments as discussed in Section III-A. The signatures of r are triples $\langle r[p_i^\ell, l_i^\ell], i, \ell \rangle$ for $1 \le i \le \triangle$. Similarly, we can deduce the bounds and signatures for other functions as shown in Tables II-IV.

Signatures for $s \in \mathcal{S}$: s may be similar to many strings in $\mathcal{R}$. Based on the length pruning, the upper bound of lengths of the possible similar strings to s is $L_u = \lfloor |s|/\delta \rfloor$ and the lower bound is $L_o = \lceil \delta |s| \rceil$. Thus for s, we need to consider strings with length $\ell \in [L_o, L_u]$ and generate signatures $s[x_i^\ell, l_i^\ell]$ for each $L_o \le \ell \le L_u$ and $1 \le i \le \triangle$.

In conclusion, the signatures generated are shown in Table IV. Note that to make the formula consistent for different similarity functions, we set $U_{r-s} = \ell - |s| + (\triangle - i)$ and $U_{s-r} = i - 1$ for ED.

Example 2: Consider the two string collections $\mathcal{R}$ and $\mathcal{S}$ in Table I. Suppose the Jaccard Similarity threshold is $\delta = 0.8$. For $r_1 \in \mathcal{R}$, we can deduce $\ell = 5$, $\overline{U} = 1$, $\triangle = 2$, $l_1^\ell = 2$, $l_2^\ell = 3$, $p_1^\ell = 1$ and $p_2^\ell = 3$. Thus, we need to generate two signatures $\langle r_1[1,2], 1, 5 \rangle$ and $\langle r_1[3,3], 2, 5 \rangle$ for r_1. For string $s_1 \in \mathcal{S}$, we can deduce $L_o = 4$ and $L_u = 6$, i.e., $4 \le \ell \le 6$. For $\ell = 4$, we can deduce that $\triangle = 2$, $l_1^\ell = 2$, $l_2^\ell = 2$, $U_{r-s} = 0$, $U_{s-r} = 1$, $\perp_1^\ell = 1$, $\top_1^\ell = 2$, $\perp_2^\ell = 3$ and $\top_2^\ell = 4$, thus it generates four signatures $\langle s_1[1,2], 1, 4 \rangle$, $\langle s_1[2,2], 1, 4 \rangle$, $\langle s_1[3,2], 2, 4 \rangle$ and $\langle s_1[4,2], 2, 4 \rangle$. Similarly, for $\ell = 5$, we generate two signatures $\langle s_1[1,2], 1, 5 \rangle$ and $\langle s_1[3,3], 2, 5 \rangle$. For $\ell = 6$, we generate two signatures $\langle s_1[1,3], 1, 6 \rangle$ and $\langle s_1[3,3], 2, 6 \rangle$. In total, we need to generate eight signatures for s_1.

C. Signature Complexity

For string $r \in \mathcal{S}$, as we generate one signature for $1 \le i \le \triangle$, it totally has $\triangle$ signatures. For any string s, the total number of its generated signatures is $\mathcal{O}((L_u - |s|)^3 + (|s| - L_o)^3)$ as shown in Lemma 5.

Lemma 5: The total number of signatures for any string s is $\mathcal{O}((L_u - |s|)^3 + (|s| - L_o)^3)$.

For example, suppose we use JAC similarity and the similarity threshold is δ, we have the total number of generated signatures for string s is $\mathcal{O}(\frac{(1+\delta^3)(1-\delta)^3}{\delta^3}|s|^3)$.

IV. THE MASSJOIN FRAMEWORK

In this section we present a scalable MapReduce-based string similarity join algorithm, called MASSJOIN, which can support both set-based similarity functions and character-based similarity functions. For simplicity, we use Jaccard as an example in this section. MapReduce contains two main stages: the filter stage and the verification stage. We will introduce the two stages respectively in Section IV-A and Section IV-B. Then we give the complexity in Section IV-C. Finally we discuss how to support self joins in Section IV-D.

A. Filter Stage

In this filter phase, we generate candidate pairs using the filter techniques in Section III: if two strings r and s are similar, r and s must share a signature. In the map phase, we use the signatures as keys and the strings as values. Thus for $r \in \mathcal{R}$, the key-value pairs are $\langle \langle r[p_i^\ell, l_i^\ell], i, \ell \rangle, r \rangle$ for $1 \le i \le \triangle$. For $s \in \mathcal{S}$, the key-value pairs are $\langle \langle s[x_i^\ell, l_i^\ell], i, \ell \rangle, s \rangle$ for $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$, $1 \le i \le \triangle$ and $L_o \le \ell \le L_u$. As two similar strings must share a same key, they must be shuffled to the same reduce task. To reduce the transmission cost, in the key-value pairs, we use string ids to replace strings (an id is much smaller than its corresponding string.). The pseudo-code of our MASSJOIN algorithm is shown in Algorithm 1. It first generates key-value pairs for strings in $\mathcal{R}$ (lines 2-4) and then for strings in $\mathcal{S}$ (lines 5-9).

In the reduce phase, each reduce node takes a key-value pair $\langle sig, list(sid/rid) \rangle$ as input, where sig is a signature and $list(sid/rid)$ is the list of strings that contain the signature. It first splits the list into two groups: $list(sid)$ and $list(rid)$ (line 11). Notice that for any pair $\langle sid, rid \rangle$ from the two lists, it is a candidate pair. To reduce the transmission cost, for each $sid \in list(sid)$, we generate a key-value pair $\langle sid, list(rid) \rangle$ (lines 12-13). The map and reduce functions are as follows.

map: $\langle sid/rid, string \rangle \rightarrow \langle signature, sid/rid \rangle$
reduce: $\langle signature, list(sid/rid) \rangle \rightarrow \langle sid, list(rid) \rangle$

Comparing with existing works [15], [13] that used a single token as a key, our method utilizes signatures with multiple tokes as keys and thus has higher pruning power. Moreover, the number of pairs that share multiple tokens is smaller than that of a single token, thus our method can address the skewed token distribution problem as discussed in Section II-B.

Example 3: Consider the two string collections in Table I. Suppose we use JAC function and the threshold is $\delta = 0.8$. Figure 1 shows the running example. The map functions in the filter stage generate 2 key-value pairs for each r_1, r_2 and r_3 in $\mathcal{R}$ and 8, 4, and 8 key-value pairs for s_1, s_2 and s_3 in $\mathcal{S}$ as shown in the figure. The shuffle phase groups the intermediate $\langle key, value \rangle$ pairs and sends them to the reduce tasks. In the reduce phase, we get 18 key-value pairs. However only two of them, $\langle$["conference information", $1, 5], \{r_1, s_1, s_3\}\rangle$ and $\langle$["information management", $1, 5], \{r_2, s_2\}\rangle$ can generate outputs, which are $\langle s_1, \{r_1\} \rangle$, $\langle s_2, \{r_2\} \rangle$, and $\langle s_3, \{r_1\} \rangle$.

TABLE III. SIGNATURES GENERATED FOR $r \in \mathcal{R}$

	JAC	COS	DICE	ED
ℓ	$\lvert r \rvert$			
l_i^ℓ	$\lfloor \frac{\ell}{\triangle} \rfloor$ for $i \le \triangle - (\ell - \lfloor \frac{\ell}{\triangle} \rfloor * \triangle)$ $\lceil \frac{\ell}{\triangle} \rceil$ for $i > \triangle - (\ell - \lfloor \frac{\ell}{\triangle} \rfloor * \triangle)$			
p_i^ℓ	$p_1^\ell = 1$ and $p_i^\ell = \sum_{j=1}^{j<i} l_j^\ell + 1$			
sig	$\langle r[p_i^\ell, l_i^\ell], i, \ell \rangle$ for $1 \le i \le \triangle$			
$\lvert sig \rvert$	$\triangle$			

TABLE IV. SIGNATURES GENERATED FOR $s \in \mathcal{S}$
(FOR ED, $U_{r-s} = \ell - |s| + (\triangle - i)$ AND $U_{s-r} = i - 1$).

	JAC	COS	DICE	ED
ℓ	$L_o \le \ell \le L_u$			
l_i^ℓ	$\lfloor \frac{\ell}{\triangle} \rfloor$ for $i \le \triangle - (\ell - \lfloor \frac{\ell}{\triangle} \rfloor * \triangle)$ $\lceil \frac{\ell}{\triangle} \rceil$ for $i > \triangle - (\ell - \lfloor \frac{\ell}{\triangle} \rfloor * \triangle)$			
$\perp_i^\ell$	$max(p_i^\ell - (i-1), p_i^\ell - U_{r-s})$			
$\top_i^\ell$	$min(p_i^\ell + \lvert s \rvert - \ell + (\triangle - i), p_i^\ell + U_{s-r})$			
sig	$\langle s[x_i^\ell, l_i^\ell], i, \ell \rangle$ for $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$, $1 \le i \le \triangle$, $L_o \le \ell \le L_u$			
$\lvert sig \rvert$	$f_\delta \lvert s \rvert^3$			τ^3

Algorithm 1: MASSJOIN Algorithm

```
   // the filter stage
1  Map(⟨rid, r⟩/⟨sid, s⟩)
2      for r ∈ R do
3          for 1 ≤ i ≤ △ do
4              emit(⟨⟨r[p_i^ℓ, l_i^ℓ], i, ℓ = |r|⟩, rid⟩);
5      for s ∈ S do
6          for L_o ≤ ℓ ≤ L_u do
7              for 1 ≤ i ≤ △ do
8                  for ⊥_i^ℓ ≤ x_i^ℓ ≤ ⊤_i^ℓ do
9                      emit(⟨⟨s[x_i^ℓ, l_i^ℓ], i, ℓ⟩, sid⟩);
10 Reduce (⟨sig, list(id)⟩)
11     split list(id) into two groups list(sid) and list(rid) ;
12     foreach sid ∈ list(sid) do
13         output(⟨sid, list(rid)⟩);
   // first phase of verification stage
14 Map(⟨sid, list(rid)⟩/⟨sid, s⟩)
15     emit(⟨sid, list(rid)⟩); emit(⟨sid, s⟩) ;
16 Reduce (⟨sid, list(list(rid)/s)⟩)
17     Identify s and list(list(rid)) ;
18     Compute distinct list(rid) from list(list(rid)) ;
19     output ⟨s, list(rid)⟩ ;
   // second phase of verification stage
20 Map(⟨s, list(rid)⟩/⟨rid, r⟩)
21     emit(⟨rid, r⟩) ;
22     for each rid in list(rid) do
23         emit(⟨rid, s⟩) ;
24 Reduce (⟨rid, list(s/r)⟩)
25     Identify r and list(s) from list(s/r);
26     foreach s ∈ list(s) do
27         if SIM(r, s) ≥ δ then output(⟨⟨r, s⟩, SIM(r, s)⟩);
```

B. Verification Stage

In the verification stage, we verify the candidate pairs generated from the filter stage. As two strings may share multiple signatures, there may be many duplicate candidate pairs, and we want to remove the duplicate ones. In addition, we need to replace the id in candidate pairs with its real string to verify the candidate pairs. To achieve these two goals, we propose a two-phase method.

In the first phase, the map function takes dataset $\mathcal{S}$ and $\langle sid, list(rid)\rangle$ as input and emits two types of key-value pairs (line 15). The first one is $\langle sid, s\rangle$ from the dataset $\mathcal{S}$ which is used to replace sid with string s. The second one is $\langle sid, rid\rangle$ which is generated from input $\langle sid, list(rid)\rangle$ gotten from the filter stage. The reduce function gathers the list $list(rid)$ and the string s for the key sid. It first identifies the string s, and then removes the duplicates from $list(rid)$ to generate a list of distinct rids, and finally emits key-value pair $\langle s, list(rid)\rangle$ (lines 17-19). The map and reduce functions are as follows.

map: $\langle sid, list(rid)\rangle \rightarrow \langle sid, list(rid)\rangle; \langle sid, s\rangle \rightarrow \langle sid, s\rangle$
reduce: $\langle sid, list(list(rid)/s)\rangle \rightarrow \langle s, list(rid)\rangle$

In the second phase of verification, we need to replace the rid with string r and verify the candidate pairs. The map function takes dataset $\mathcal{R}$ and $\langle s, list(rid)\rangle$ as input and emits two types of key-value pairs: $\langle rid, r\rangle$ and $\langle rid, s\rangle$. The first one is generated from the dataset $\mathcal{R}$ (line 21). The second one is generated from $\langle s, list(rid)\rangle$ that is the output of the first phase. For each key s, it generates a key-value pair $\langle rid, s\rangle$ for each rid in $list(rid)$ (line 23). In the reduce phase, we have a string r and a list of string s such that $\langle r, s\rangle$ is a candidate pair. We first identify the string r and $list(s)$, then verify the candidate pairs, and finally output the final results (lines 25-27). The map and reduce functions are as follows.

map: $\langle s, list(rid)\rangle \rightarrow \langle rid, s\rangle$, $\langle rid, r\rangle \rightarrow \langle rid, r\rangle$;
reduce:$\langle rid, list(r/s)\rangle \rightarrow \langle (r, s), \text{SIM}(r, s)\rangle$

Example 4: Recall the running example in the last subsection and here we discuss the verification stage. In the filter step, we get three candidate pairs. In the first phase, we remove duplicates from the three pairs, replace sid with its string and output the key-value pairs, e.g., $\langle$"service information management policy", $\{r_2\}\rangle$. In the second phase, we replace rid with its string and verify the candidate pairs. Finally, we output $\langle r_2,\ s_2\rangle$ as a result. The details are shown in Figure 1.

C. Complexity

In this section, we analyze the complexity of our approach in each stage, including the space/time complexity and IO cost of the map and reduce phase, and the transmission complexity of the shuffle phase. For each string with length ℓ in $\mathcal{R}$, we need to generate $\mathcal{O}(\overline{U}) = \mathcal{O}(g_\delta \ell)$ signatures, where g_δ is the factor of ℓ in the formula of $\overline{U}$ as illustrated in Table II. For each string with length ℓ in $\mathcal{S}$, we generate $\mathcal{O}((L_u - \ell)^3 + (\ell - L_o)^3) = \mathcal{O}(f_\delta \ell^3)$ signatures where f_δ is the factor of ℓ^3 in $(L_u - \ell)^3 + (\ell - L_o)^3$ as illustrated in Table II. Let m_ℓ (n_ℓ) denote the number of the strings with length ℓ in $\mathcal{R}$ ($\mathcal{S}$). Based on the discussion in Section III, the number of generated signatures for strings in $\mathcal{R}$ ($\mathcal{S}$) is $\mathcal{O}(\sum_{\ell=\ell^{\mathcal{R}}_{min}}^{\ell^{\mathcal{R}}_{max}} g_\delta \ell m_\ell)$ ($\mathcal{O}(\sum_{\ell=\ell^{\mathcal{S}}_{min}}^{\ell^{\mathcal{S}}_{max}} f_\delta \ell^3 n_\ell)$), where $\ell^{\mathcal{R}}_{min}$ ($\ell^{\mathcal{S}}_{min}$) and $\ell^{\mathcal{S}}_{max}$ ($\ell^{\mathcal{S}}_{max}$) are the minimum and maximum string lengths in $\mathcal{R}$ ($\mathcal{S}$) respectively. For ease of presentation, let $N = \sum_{\ell=\ell^{\mathcal{R}}_{min}}^{\ell^{\mathcal{R}}_{max}} g_\delta \ell m_\ell$ and $M = \sum_{\ell=\ell^{\mathcal{S}}_{min}}^{\ell^{\mathcal{S}}_{max}} f_\delta \ell^3 n_\ell$.

Filter Stage: The map function loads the dataset from DFS and the IO cost is $\mathcal{O}(\mathcal{R} + \mathcal{S})$. The space/time complexity of generating one signature is $\mathcal{O}(1)$, thus the space/time complexity of the first map phase is $\mathcal{O}(N + M)$. As we emit one intermediate $\langle key, value\rangle$ pair for each signature and the space of each signature is $\mathcal{O}(1)$, the transmission complexity is also $\mathcal{O}(N + M)$. The reduce step needs to scan the lists of rids and sids in the value fields of the key-value pairs, thus the time/space complexity of the reduce phase is $\mathcal{O}(N \times M)$. The reduce step writes the candidate pairs to disk, thus the IO cost is also $\mathcal{O}(N \times M)$.

Verification Stage: The map function of the filter phase needs to read the dataset $\mathcal{S}$ and the candidate pairs, thus the IO cost is $\mathcal{O}(N \times M + \mathcal{S})$. Emitting a $\langle key, value\rangle$ takes $\mathcal{O}(1)$ time, and the space/time complexity is $\mathcal{O}(N \times M + |\mathcal{S}|)$. The shuffle transmission complexity is also $\mathcal{O}(N \times M + \mathcal{S})$. The reduce

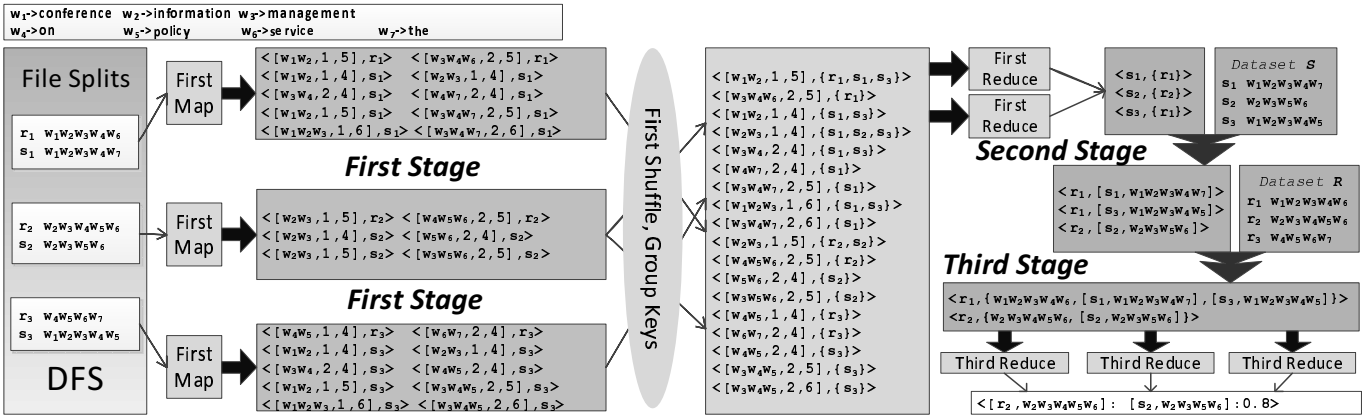


Fig. 1. Running example

function scans the input value list once and the time complexity is $\mathcal{O}\big(N \times M + |\mathcal{S}|\big)$. As we at most output each original string in $\mathcal{S}$ once, the IO cost is $\mathcal{O}(N \times M + \mathcal{S})$.

Suppose there are c distinct candidate pairs generated from the filter reduce of the verification stage, and the average size of the strings in $\mathcal{S}$ is $\mathcal{S}_{avg}$. As each candidate pair contains a string in $\mathcal{S}$, the IO complexity for this map phase is $\mathcal{O}(c\mathcal{S}_{avg} + \mathcal{R})$. The time complexity of the third map is $\mathcal{O}(c + |\mathcal{R}|)$. As each candidate pair contains a string of $\mathcal{S}$, the shuffle transmission complexity is $\mathcal{O}(c\mathcal{S}_{avg} + \mathcal{R})$. Suppose the average verification time for each candidate pair is v. The time complexity of the third reduce is $\mathcal{O}(cv)$. The IO cost in this stage is $\mathcal{O}\big(c(\mathcal{R}_{avg} + \mathcal{S}_{avg})\big)$. We show all the complexity analysis in Table V.

D. Supporting Self-Join

The MASSJOIN algorithm can inherently support the self-join scenario where two datasets are the same, i.e., $\mathcal{R} = \mathcal{S}$. In this case, for each string r, we first take its segments as *segment signatures* and then select its substrings for strings with length between L_o and $|r|$ as *substring signatures* (taking it as s). We can use a special mark to differentiate segment signatures and substring signatures. In this case, we can support self joins.

V. MERGING KEY-VALUE PAIRS

Based on the complexity analysis in Section IV-C, the basic framework generates large numbers of signature-based key-value pairs for string $s \in S$ in the filter stage. In this section, we discuss how to reduce the number of key-value pairs without sacrificing the pruning power. For example, we can decrease the number from $\mathcal{O}(|s|^3)$ to $\mathcal{O}(|s|)$ for the Jaccard function. To achieve this goal, we first introduce how to merge key-value pairs in Section V-B, and then devise an efficient merging algorithm in Section V-B, and finally discuss how to incorporate the method into our framework in Section V-C.

A. Merging Key-Value Pairs

For simplicity, the key-value pairs in this section refer to those generated from the map phrase in the filter stage.

Basic Idea. The basic framework will generate large numbers of key-value pairs for $s \in \mathcal{S}$: $\big\langle\langle s[x_i^\ell, l_i^\ell], i, \ell\rangle, s\big\rangle$ for $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$, $1 \le i \le \triangle$ and $L_o \le \ell \le L_u$. We aim to merge them to generate a small number of key-value pairs. Obviously if two signatures match, e.g., $\langle r[p_i^\ell, l_i^\ell], i, \ell\rangle = \langle s[x_i^\ell, l_i^\ell], i, \ell\rangle$, the segment $r[p_i^\ell, l_i^\ell]$ and the substring $s[x_i^\ell, l_i^\ell]$ must match, i.e., $r[p_i^\ell, l_i^\ell] = s[x_i^\ell, l_i^\ell]$. One simple method is to take $\langle s[x_i^\ell, l_i^\ell], s\rangle$ as key-value pairs for s and $\langle r[p_i^\ell, l_i^\ell], r\rangle$ as key-value pairs for r. Obviously, if r and s are similar, they must share a same key and the method will not miss a similar pair.

However, this method may reduce the pruning power. This is because a substring and a segment may match with different start positions, or with different lengths. Both cases will generate more false positives. To achieve the same pruning power as the basic framework, we need to check whether the start position x_i^ℓ of the substring $s[x_i^\ell, l_i^\ell]$ is within $[\perp_i^\ell, \top_i^\ell]$ as well as whether the length of $|r|$ is within $[L_o, L_u]$. These bounds are different for various i and ℓ, and it is expensive to add them into the key-value pairs. Fortunately, these bounds can be efficiently computed just based on the values of i and ℓ in $\mathcal{O}(1)$ time.

For each key-value pair $\langle s[x_i^\ell, l_i^\ell], s\rangle$ of s, we add x_i^ℓ and $|s|$ into its value field and the new key-value pair is $\big\langle\langle s[x_i^\ell, l_i^\ell]\rangle, \langle |s|, x_i^\ell, s\rangle\big\rangle$. For each key-value pair $\langle r[x_i^\ell, l_i^\ell], r\rangle$ of r, we add i and $\ell = |r|$ in its value field and the key-value pair is $\big\langle\langle r[p_i^\ell, l_i^\ell]\rangle, \langle i, \ell, r\rangle\big\rangle$. In the map phase, we generate these new key-value pairs. In the reduce phase, for two matching keys $r[p_i^\ell, l_i^\ell] = s[x_i^\ell, l_i^\ell]$ with values $\langle i, \ell, r\rangle$ and $\langle |s|, x_i^\ell, s\rangle$, we first calculate $\perp_i^\ell$, $\top_i^\ell$, L_o and L_u based on ℓ, i and $|s|$ in the value fields and δ from the configuration file. Then we check if $L_o \le \ell \le L_u$ and $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$. If yes, we output this pair as a candidate pair.

It is easy to prove that the method using $\big\langle\langle r[p_i^\ell, l_i^\ell]\rangle, \langle i, \ell, r\rangle\big\rangle$ and $\big\langle\langle s[x_i^\ell, l_i^\ell]\rangle, \langle |s|, x_i^\ell, s\rangle\big\rangle$ as keys generates the same candidate pairs as that using $\big\langle\langle r[p_i^\ell, l_i^\ell], i, \ell\rangle, r\big\rangle$ and $\big\langle\langle s[x_i^\ell, l_i^\ell], i, \ell\rangle, s\big\rangle$ as keys.

It is worth noting that this method can significantly reduce the number of key-value pairs. Given any string $s \in \mathcal{S}$, for set-based similarity functions, we can reduce the number of key-value pairs from $\mathcal{O}(f_\delta |s|^3)$ to $\mathcal{O}(|s|)$ as stated in Lemma 6.

TABLE V. COMPLEXITY ANALYSIS OF MASSJOIN ($\mathcal{R}/\mathcal{S}$ ARE THE DATASET SIZES AND $|\mathcal{R}|/|\mathcal{S}|$ ARE THE NUMBERS OF STRINGS IN $\mathcal{R}$/S).

Stage		Filter stage	First phase of the verification stage	Second phase of the verification stage
map	Time/Space	$\mathcal{O}(N+M)$	$\mathcal{O}(N\times M+\lvert\mathcal{S}\rvert)$	$\mathcal{O}(c+\lvert\mathcal{R}\rvert)$
	IO	$\mathcal{O}(\mathcal{R}+\mathcal{S})$	$\mathcal{O}(N\times M+\mathcal{S})$	$\mathcal{O}(c\mathcal{S}_{avg}+\mathcal{R})$
shuffle	Trans	$\mathcal{O}(N+M)$	$\mathcal{O}(N\times M+\mathcal{S})$	$\mathcal{O}(c\mathcal{S}_{avg}+\mathcal{R})$
reduce	Time/Space	$\mathcal{O}(N\times M)$	$\mathcal{O}(N\times M+\lvert\mathcal{S}\rvert)$	$\mathcal{O}(cv)$
	IO	$\mathcal{O}(N\times M)$	$\mathcal{O}(N\times M+\mathcal{S})$	$\mathcal{O}\Big(c(\mathcal{R}_{avg}+\mathcal{S}_{avg})\Big)$

Lemma 6: Given a string s and a set-based similarity function threshold δ, the number of key-value pairs in the merge-based method is $\mathcal{O}(|s|)$.

Proof Sketch: Here we use JAC as an example. For any string s and any JAC similarity threshold δ, $\frac{\ell}{\triangle}$ is monotonically increasing with the increasing of ℓ, thus the minimum and maximum value of l_i^ℓ is $\lfloor\frac{L_o}{\triangle}\rfloor$ and $\lceil\frac{L_u}{\triangle}\rceil$ respectively. $\lceil\frac{L_u}{\triangle}\rceil - \lfloor\frac{L_o}{\triangle}\rfloor + 1 \le \lfloor\frac{L_u}{\triangle} - \frac{L_o}{\triangle}\rfloor + 3 \le 4$. Thus there are at most 4 different values of l_i^ℓ. Since x_i^ℓ must be in $[1,|s|]$, there are at most $\mathcal{O}(|s|)$ keys. Moreover, each key $s[x_i^\ell, p_i^\ell]$ can solely determine the $value$ $\langle|s|, x_i^\ell, sid\rangle$. Thus the number of key-value pairs is $\mathcal{O}(|s|)$. Similarly, we can prove this lemma for COS and DICE functions. ■

We can also prove that for ED, the number of key-value pairs in this merge-based method is $\mathcal{O}\big(min(|s|,\tau^2)\big)$. This is formalized in Lemma 7.

Lemma 7: Given a string s and a ED threshold τ, the number of key-value pairs in the merge-based method is $\mathcal{O}\big(\min(|s|,\tau^2)\big)$.

Proof Sketch: Based on the similar idea in Lemma 6, we can prove that the number of key-value pairs generated by s is bounded by $\mathcal{O}(|s|)$. Next we prove the number of key-value pairs is also bounded by $\mathcal{O}(\tau^2)$. For any string s and threshold τ, $\frac{\ell}{\triangle}$ is monotonically increasing, thus the range of l_i^ℓ is $\big[\lfloor\frac{L_o}{\triangle}\rfloor, \lceil\frac{L_u}{\triangle}\rceil\big]$. As $\lceil\frac{L_u}{\triangle}\rceil - \lfloor\frac{L_o}{\triangle}\rfloor + 1 \le \lfloor\frac{2\tau+1}{\tau+1}\rfloor + 3 \le 4$, the size of the range of l_i^ℓ is at most 4. We also find that the size of the range of x_i^ℓ is at most $2\tau+4$ for all $L_o \le \ell \le L_u$ and a fixed $i \in [1,\triangle]$. Thus the total number of possible x_i^ℓ is $\mathcal{O}(\tau^2)$. Thus the number of key-value pairs is also bounded by $\mathcal{O}(\tau^2)$. ■

B. Merge Algorithm

For sting r, it is easy to generate its key-value pairs $\big\langle\langle r[p_i^\ell, l_i^\ell]\rangle, \langle i, |r|, r\rangle\big\rangle$ for $i \in [1,\triangle]$. For string s, a straightforward method enumerates all signatures for s as discussed in Section III-C and then merges them with the same $\langle s[x_i^\ell, l_i^\ell]\rangle$. However, the time complexity of this method is $\mathcal{O}(f_\delta|s|^3)$. Here we study how to efficiently merge the original key-value pairs to obtain the new key-value pairs for string s.

As proved in Lemma 6, there are only four different values for l_i^ℓ. As x_i^ℓ is a position in string s, $x_i^\ell \in [1,|s|]$. Then we can devise an efficient algorithm to directly generate the new key-value pairs as follows. For $x_i^\ell \in [1,|s|]$ and for each l_i^ℓ, we generate a key-value pair $\big\langle\langle s[x_i^\ell, l_i^\ell]\rangle, \langle|s|, x_i^\ell, s\rangle\big\rangle$. Obviously the time complexity is $\mathcal{O}(|s|)$. [3]

[3]Although this method may generate more key-value pairs than the straightforward method, the complexity is still $\mathcal{O}(|s|)$. In the reduce step, we can remove such pairs based on the checking method in Section V-B.

Complexity Improvement. The merge-based method can reduce the number of key-value pairs for s from $\mathcal{O}(|s|^3)$ to $\mathcal{O}(|s|)$. The total number of key-value pairs for all strings in $\mathcal{S}$ is from $\sum_{\ell=\ell_{min}^{\mathcal{S}}}^{\ell_{max}^{\mathcal{S}}} f_\delta \ell^3 n_\ell$ to $\sum_{\ell=\ell_{min}^{\mathcal{S}}}^{\ell_{max}^{\mathcal{S}}} \ell n_\ell$. The time/space complexity to generate the key-value pairs for string s is also from $\mathcal{O}(|s|^3)$ to $\mathcal{O}(|s|)$.

C. Changes on MASSJOIN Algorithm

This section discusses the changes we need to make for incorporating the merge-based method into our MASSJOIN framework. The pseudo-code shown in Algorithm 2 is the same as Algorithm 1 except for the filter stage. To generate the new key-value pairs, we replace Line 4 and Lines 6-9 in Algorithm 1 with Line 2 and Lines 3-5 in Algorithm 2 respectively. For each string $s \in \mathcal{S}$, we only need to scan the string once to generate all key-value pairs (Lines 3-5). In the reduce phrase, we still split the input value lists into two lists and output those pairs satisfying $L_o \le \ell \le L_u$ and $\perp_i^\ell \le x_i^\ell \le \top_i^\ell$ (Lines 7-12).

Example 5: Consider the strings r_3 and s_3 in Table I. Using the merge-based algorithm, we generate two key-value pairs for r_3, i.e., $\big\langle\langle r_3[1,2]\rangle, \langle 1,4,r_3\rangle\big\rangle$ and $\big\langle\langle r_3[3,2]\rangle, \langle 2,4,r_3\rangle\big\rangle$, and six key-value pairs for s_3, i.e., $\big\langle\langle s_3[1,2]\rangle, \langle 5,1,s_3\rangle\big\rangle$, $\big\langle\langle s_3[2,2]\rangle, \langle 5,2,s_3\rangle\big\rangle$, $\big\langle\langle s_3[3,2]\rangle, \langle 5,3,s_3\rangle\big\rangle$, $\big\langle\langle s_3[4,2]\rangle, \langle 5,4,s_3\rangle\big\rangle$, $\big\langle\langle s_3[1,3]\rangle, \langle 5,1,s_3\rangle\big\rangle$, and $\big\langle\langle s_3[3,3]\rangle, \langle 5,3,s_3\rangle\big\rangle$. Note that for the basic method, we generate eight key-value pairs for s_3, thus the merge-based algorithm reduces the transmission cost of two key-value pairs. When using the key-value pairs to generate the candidate pairs, although $r_3[1,2] = s_3[4,2]$, we do not output $\langle r_3, s_3\rangle$ since $4 \notin [\perp_1^4 = 1, \top_1^4 = 2]$.

VI. USING LIGHT-WEIGHT FILTER UNIT TO REDUCE CANDIDATE PAIRS

As the transmission and computation cost in the verification stage heavily relies on the number of candidate pairs, in this section, we aim to decrease the number of candidate pairs. One simple method is to attach the original strings to the value field of each $\langle key, value\rangle$ pair in the map phase of filter stage. In the reduce phase, for each candidate pair, we calculate their real similarity and remove those pairs whose similarity is smaller than the threshold δ. Although this method can reduce the transmission cost of candidate pairs, it increases the transmitting cost of original strings dramatically. To alleviate this problem, we incorporate an "light-weight filter unit" to replace the original strings. On one hand, filter units can be utilized to prune many dissimilar pairs. On the other hand, filter units should be light-weight in order not to significantly increase the transmission cost. To this end, we propose an effective filter unit in Section VI-A, and then discuss how

Algorithm 2: Merge-based Algorithm

```
   // the filtering stage
1  Map(⟨rid, r⟩/⟨sid, s⟩)
       // Replace Line 4 in Algo 1 with
2      emit(⟨⟨[p_i^ℓ, l_i^ℓ ], ⟨i,|r|,rid⟩⟩⟩) ;
       // Replace Lines 6∼9 in Algo 1 with
3      for 1 ≤ x_i^ℓ ≤ |s| − ℓ do
4          for ⌊L_o/△⌋ ≤ l_i^ℓ ≤ ⌈L_u/△⌉ do
5              emit(⟨⟨s[x_i^ℓ, l_i^ℓ]⟩, ⟨|s|, x_i^ℓ, sid⟩⟩);
6  Reduce (⟨sig, list(⟨i, ℓ, rid⟩/⟨|s|, x_i^ℓ, sid⟩)⟩)
7      split list(⟨i, ℓ, rid⟩/⟨|s|, x_i^ℓ, sid⟩) into two groups
       list(⟨i, ℓ, rid⟩) and list(⟨|s|, x_i^ℓ, sid⟩);
8      foreach ⟨|s|, x_i^ℓ, sid⟩ ∈ list(⟨|s|, x_i^ℓ, sid⟩) do
9          foreach ⟨i, ℓ, rid⟩ ∈ list(⟨i, ℓ, rid⟩) do
10             if L_o ≤ ℓ ≤ L_u & ⊥_i^ℓ ≤ x ≤ ⊤_i^ℓ then
11                 list(rid) ← rid;
12         output(⟨sid, list(rid)⟩);
```

Algorithm 3: Light-weight Filtering

```
   // the token count stage
1  Map(⟨id, string⟩)
2      For each token in the string, emit(⟨token, 1⟩);
3  Reduce (⟨token, list(1)⟩)
4      Output token frequency tf;
   // the filtering stage
5  MapSetup
6      Partition tokens into n groups ;
7  Map(⟨rid, r⟩/⟨sid, s⟩)
       // Add into Algo 1
8      Compute filter unit g^r/g^s for strings r/s;
       // Replace Line 4 in Algo 1 with
9      emit(⟨⟨r[p_i^ℓ, l_i^ℓ], i, ℓ = |r|⟩, ⟨rid, g^r⟩⟩);
       // Replace Line 9 in Algo 1 with
10     emit(⟨⟨s[x_i^ℓ, l_i^ℓ], i, ℓ⟩⟩, ⟨sid, g^s⟩⟩);
11 Reduce (⟨sig, list(⟨id, g⟩)⟩)
12     list(⟨id, g⟩) → list(⟨sid, g^s⟩) and list(⟨rid, g^r⟩);
13     foreach ⟨sid, g^s⟩ ∈ list(⟨sid, g^s⟩) do
14         foreach ⟨rid, g^r⟩ ∈ list(⟨rid, g^r⟩) do
15             if Σ_{1≤i≤n} |g_i^r − g_i^s| ≤ U then
16                 list(rid) → rid;
17         output(⟨sid, list(rid)⟩);
```

to integrate this technique into our MASSJOIN framework in Section VI-B.

A. Light-weight Filter Unit

Basic Idea. We first consider the set-based similarity functions. Given a string, we can use an integer to replace each token in the string and take the set of integers as the filter unit of the string. Obviously two strings are similar only if their corresponding integer sets are similar. However when the number of tokens is large, this method still involves large transmission cost. To reduce the size, we can group the integers into n buckets and keep a list of n integers as a filter unit.

Formally, we first partition all tokens in the two string sets into n groups, $\mathcal{G}_1, \mathcal{G}_2, \cdots, \mathcal{G}_n$. Then for each string $r(s)$, its filter unit is $\langle g_1^r, g_2^r, \cdots, g_n^r \rangle (\langle g_1^s, g_2^s, \cdots, g_n^s \rangle)$, where $g_i^r(g_i^s)$ is the number of tokens in $r(s)$ that are in the i-th group, i.e., $g_i^r = |\mathcal{G}_i \cap r| (g_i^s = |\mathcal{G}_i \cap s|)$. Two strings r and s are similar only if $\sum_{1 \le i \le n} |g_i^r - g_i^s| \le U$ as stated in Lemma 8, where U is an upper bound of $|r - s|+|s - r|$ (Section III-A).

Lemma 8: Given two strings r and s with filter unit $\langle g_1^r, g_2^r, \cdots, g_n^r \rangle$ and $\langle g_1^s, g_2^s, \cdots, g_n^s \rangle$, if $\sum_{1 \le i \le n} |g_i^r - g_i^s| > U$, r and s cannot be similar.

Proof Sketch: We divide r and s into n disjoint groups using a universal grouping method. The total number of different tokens between r and s is the sum of the number of different tokens in the n groups. For the i-th group, the number of different tokens in it is $|\mathcal{G}_i \cap r - \mathcal{G}_i \cap s| + |\mathcal{G}_i \cap s - \mathcal{G}_i \cap r| = |\mathcal{G}_i \cap r| + |\mathcal{G}_i \cap s| - 2|(\mathcal{G}_i \cap r) \cap (\mathcal{G}_i \cap s)| \le g_i^r + g_i^s - 2\max(g_i^r, g_i^s) = |g_i^r - g_i^s|$. Thus the total number of different tokens between r and s cannot be smaller than $\sum_{1 \le i \le n} |g_i^r - g_i^s|$. Meanwhile, the number of different tokens in two similar strings cannot exceed U(see Section III-A). Thus if $\sum_{1 \le i \le n} |g_i^r - g_i^s| > U$, r and s are not similar. ■

To utilize the filter unit, we add the filter unit into the value part in the key-value pair of the map function in the filter stage. As discussed in Section III-A, the upper bound U only depends on $|r|$, $|s|$ and δ. Obviously $|r| = \sum_{1 \le i \le n} g_i^r$ and $|s| = \sum_{1 \le i \le n} g_i^s$, thus we can compute U in the reduce step. Thus we can utilize the filter condition to prune dissimilar pairs. Notice that the filter unit based method is orthogonal to the merge-based method and they can be used together.

This method can be applied to the character-based similarity functions by taking each character as a token and $U = 2\tau$.

Finding the Optimal Grouping Strategy. We find that different grouping methods may have different pruning power. Next we study how to evaluate a grouping strategy and how to select the optimal grouping so as to generate high-quality filter unit.

Intuitively, for two strings r and s, the larger $\sum_{1 \le i \le n} |g_i^r - g_i^s|$, the two strings have higher probability to be pruned. Thus we want to maximize the value. Similarly, for all strings in $\mathcal{R}$ and $\mathcal{S}$, we want to find an optimal grouping strategy to maximize

$$\sum_{r \in \mathcal{R}} \sum_{s \in \mathcal{S}} \sum_{1 \le i \le n} |g_i^r - g_i^s|. \quad (1)$$

We can prove that the optimal grouping problem is NP-hard by a reduction from the 3-SAT problem.

Theorem 1: The optimal grouping problem is NP-hard.

Proof Sketch: We can prove the problem by a reduction from the 3-SAT problem. ■

We propose a heuristic approach to solve this problem. The approach is based on two observations. First, it is not good to put two tokens with large token frequencies into the same group, where the token frequency is the number of strings containing the token. This is because if we put them into the

same group, many string pairs will falsely consider them as the same token and the value $|g_i^r - g_i^s|$ for such string pairs will not increase; on the contrary, if they are assigned into two different groups, the value will be significantly increased. Second, the sum of token frequencies is a constant. To make the overall value as large as possible, we want to make the sum of token frequency in each group nearly equal. Based on these two observations, we can devise a greedy algorithm as follows. We first sort all the tokens by their frequencies in decreasing order. Then we access each token in order and add it into the group with the minimum sum of token frequencies. We repeat this step until all the tokens have been accessed.

B. Changes on the MASSJOIN Algorithm

To incorporate filter units into our method, we need make some changes on the MASSJOIN algorithm. The method can be utilized to both the basic method and the merge-based method. Here we take the basic method as an example. The pseudo-code is shown in Algorithm 3. We need to add another MapReduce phase to count token frequencies (Lines 1-4). We also modify the filter stage (Lines 5-17) as follows. We add a setup phase to read the token frequency and an approximation algorithm to divide all the tokens to n groups (Line 6). In the map phase, we load the token frequency, identify the tokens for each string, partition them into different groups based on token frequencies, and generate filter unit (Line 8). Then we emit the filter unit along with the $\langle key, value\rangle$ pairs (Lines 9-10). In the reduce step, we only output those pairs passing the grouping filter (Lines 12-17).

Example 6: Consider the two datasets in Table I. We use JAC and the threshold is $\delta = 0.8$. For simplicity, we use the ids of strings as shown in Figure 1. In the token count stage we get the token frequencies, $\langle w_2, 5\rangle$, $\langle w_3, 5\rangle$, $\langle w_4, 5\rangle$, $\langle w_5, 4\rangle$, $\langle w_6, 4\rangle$, $\langle w_1, 3\rangle$ and $\langle w_7, 2\rangle$. Suppose the group number is 4. We can divide the tokens into 4 groups $\mathcal{G}_1 = \{w_2, w_7\}$, $\mathcal{G}_2 = \{w_3, w_1\}$, $\mathcal{G}_3 = \{w_4\}$, and $\mathcal{G}_4 = \{w_5, w_6\}$. The filter unit for r_1 is $\langle 1, 2, 1, 1\rangle$ and that for s_1 is $\langle 2, 2, 1, 0\rangle$. We filter pair $\langle s_1, r_1\rangle$ in the reduce phase as $\sum_{i=1}^{4} |g_i^{r_1} - g_i^{s_1}| = 2 > U = 0$.

VII. EXPERIMENT

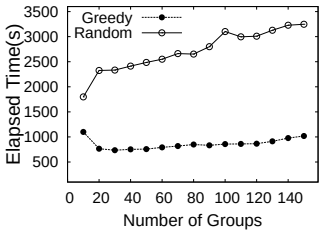
We have implemented our MASSJOIN method and conducted experiments on four real datasets: Enron email[4], PubMed paper abstract, PubMed paper title[5] and NCBI DNA sequence[6]. The details of the datasets are shown in Table VI.

TABLE VI. DATASETS

Datasets	Size(MB)	Cardinality	Avg Size/Len	$\lvert\Sigma\rvert$
Enron Email (Set)	1425	516,717	383.7	36
PubMed Abstract (Set)	3159	2,347,362	195.6	36
PubMed Title (String)	1494	10,394,374	144.4	36
DNA Sequence (String)	2148	18,299,728	117.2	5

For Enron email and PubMed paper abstract datasets, we used set-based similarity functions, where strings are tokenized by non-alphanumeric characters. For PubMed paper title and DNA datasets, we used the character-based distance functions. In the following experiments, we split each dataset into two datasets with equal size to conduct $\mathcal{R}$-$\mathcal{S}$ join. Due to space constraints, we focus on JAC and ED in the experiments and use them as default functions. We will show the results on COS and DICE in Section VII-D.

We compared with state-of-the-art method `PrefixFilter` [15]. We got their source code from their home page (asterix.ics.uci.edu/fuzzyjoin). All algorithms were implemented on Hadoop and run on a 10-node Dell cluster. Each node had two Intel(R) Xeon(R) E5420 2.5GHZ processors with 8 cores, 16GB RAM, and 1TB disk. Each node is installed 64-bit Ubuntu Server 10.04, Java 1.6, and Hadoop 1.0.4. We set the block size of the distributed file system to 16MB and allocate 2GB virtual memory to each task.



(a) PubMed Abstract

Fig. 2. Evaluating light-weight filter units.

A. Evaluating Our Proposed Techniques

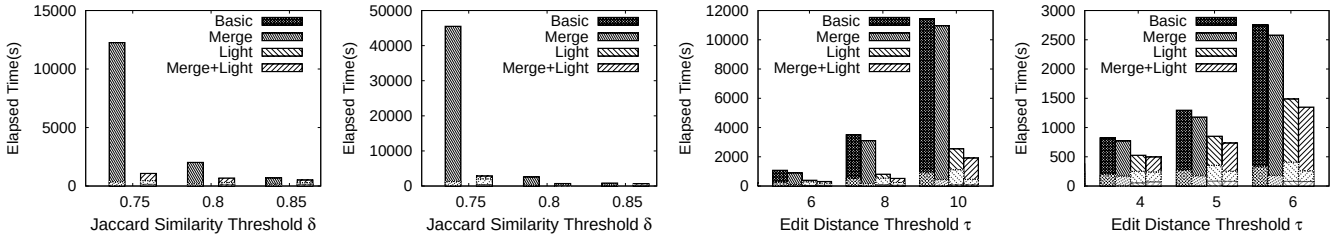
We first evaluated our filter unit based method by varying different group numbers from 10 to 150. We implemented two grouping methods: `Random` and `Greedy`. `Random` computed the hash code of each token and randomly assigned it into a group. `Greedy` used our greedy algorithm. Figure 2 shows the results. We can see that with the increase of groups, the performance of `Greedy` improved first and then depraved. This is because a small group number will reduce the transmission cost but with lower pruning power, and a large group number will improve the pruning power but with large transmission cost. In addition, our algorithm outperformed the `Random` grouping method since we considered the token distribution. In the remainder experiments, we used the `Greedy` algorithm and set the group number to 30.

We then evaluated our merge-based and filter unit techniques. We implemented four algorithms: our basic algorithm (`Basic`), merge-based algorithm (`Merge`), light-weight filter algorithm (`Light`), and MASSJOIN with both merge-based and light-weight filter techniques (`Merge+Light`). Figure 3 shows the elapsed time for each algorithm, including the running time of different MapReduce phases. Notice that on the Enron and PubMed abstract datasets, `Basic` and `Light` did not finish within 20 hours, because they had to generate large numbers of key-value pairs ($\mathcal{O}(\ell^3)$). `Light` and `Merge+Light` addressed this problem by merging the key-value pairs. On the PubMed title and DNA datasets, `Basic` and `Light` worked well because for ED the number of key-value pairs is not large ($\mathcal{O}(\tau^3)$). Notice that on all datasets, `Merge+Light` achieved the highest performance, because it merged the key-value pairs to reduce transmission cost and used filter units to reduce the number of candidate pairs. In the remainder experiments, we used the `Merge+Light` as the default algorithm for MASSJOIN.

[4]https://www.cs.cmu.edu/~enron/

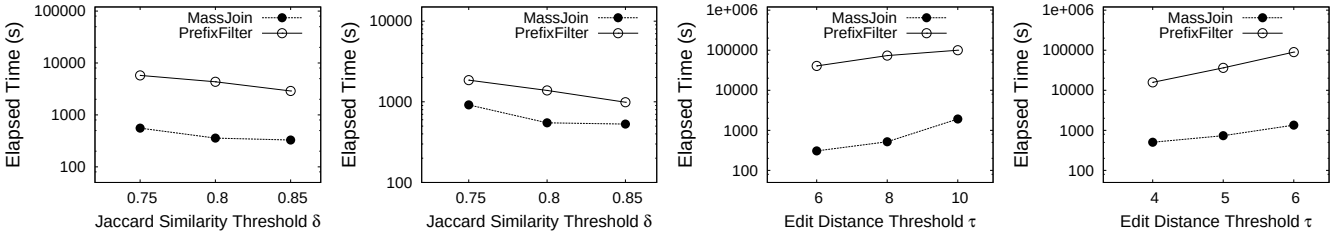
[5]http://www.ncbi.nlm.nih.gov/pubmed

[6]http://www.ncbi.nlm.nih.gov/guide/dna-rna/



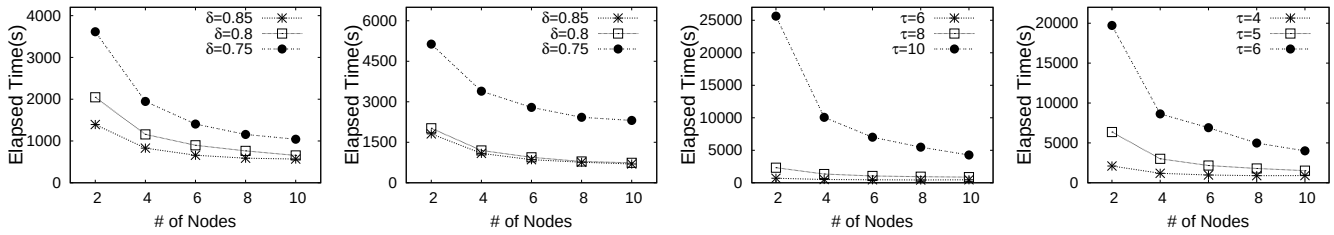
(a) Enron Email (b) PubMed Abstract (c) PubMed Title (d) DNA Sequence

Fig. 3. Evaluating our proposed techniques.



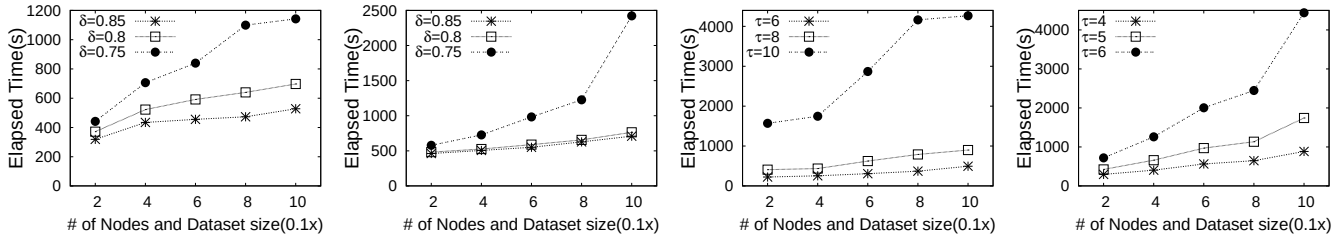
(a) Enron Email (b) PubMed Abstract (c) PubMed Title (d) DNA Sequence

Fig. 4. Comparison with state-of-the-art methods(`VSMARTJoin` and `FuzzyJoin` are out of memory).



(a) Enron Email (b) PubMed Abstract (c) PubMed Title (d) DNA Sequence

Fig. 5. Speedup by varying number of nodes.



(a) Enron Email (b) PubMed Abstract (c) PubMed Title (d) DNA Sequence

Fig. 6. Scaleup by increasing the dataset sizes and the number of nodes.

B. Comparison with State-of-the-art Method

We compared our algorithm with state-of-the-art method, `PrefixFilter` [15], `VSMARTJoin` [13] and `FuzzyJoin` [1]. For `PrefixFilter`, we used its source code on set-based similarity functions and extended the code to support character-based functions using the technique in ED-Join [21] to generate prefixes and the technique in PassJoin [12] to verify the results. We also implemented `VSMARTJoin` and `FuzzyJoin`. However they were always out of memory because they generated large numbers of key-value pairs which cannot load into the workers. Thus we did not include them in our experiment. Since `PrefixFilter` took rather long time on our large datasets, we used the 0.6x datasets, where is gotten by randomly sampling 60% of the original dataset. Figure 4 shows the results. Note that on the PubMed title dataset, `PrefixFilter` did not finish within 30 hours. We can see that our method significantly outperformed `PrefixFilter`, even by 1 to 2 orders of magnitude. For example, on PubMed paper title dataset with Edit Distance threshold $\tau = 6$, `PrefixFilter` took 50,000 seconds while MASSJOIN only took 500 seconds. This main reason is that their signatures are less selectivity than our algorithm and they generated large numbers of key-value pairs. In the verification stage, they involved much transmission and computation time. In addition, we used filter

units to reduce candidate pairs.

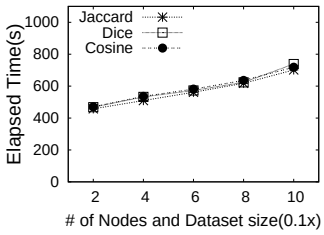
C. Speedup

We evaluated the speedup of our algorithm by varying the number of nodes from 2 to 10. The experimental results are shown in Figure 5. We can see that with the increase of nodes in the cluster, the performance of our algorithm significantly improved. For example, on the Enron email dataset with similarity threshold $\delta = 0.75$, the running time on the cluster with 2,4,6,8,10 nodes are 3600 seconds, 1900 seconds, 1400 seconds, 1200 seconds, and 1000 seconds respectively. This is attributed to our effective signatures which can significantly prune dissimilar pairs and avoid enumerating all pairs.

D. Scaleup

We evaluated the scaleup of our algorithm by increasing both dataset sizes and numbers of nodes in the cluster. Figure 6 shows the results. It is worth noting that as the dataset increased, the number of results will increase by quadratic, especially for small thresholds for set-based similarity functions and large thresholds for character-based similarity functions. Thus the running time increased slightly. For example, on the PubMed title dataset, when the Edit Distance threshold is $\tau = 8$, the running time on 2-node cluster and with 0.2x dataset is about 300 seconds; on 10-node cluster and with 1x dataset the time is about 900 seconds.

We also evaluated the scaleup on the other two set-based similarity functions COS and DICE. Figure 7 shows the results on the PubMed abstract dataset. MASSJOIN got similar results on COS and DICE as JAC because their difference is the verification method which took little time.



(a) PubMed Abstract

Fig. 7. Evaluating set-based similarity functions.

VIII. CONCLUSION

We proposed a MapReduce-based framework for scalable string similarity joins, which supports both set-based similarity functions and character-based similarity functions. We extended existing partition-based signature scheme to support set-based similarity functions. We utilized the signatures to generate key-value pairs on MapReduce. We proposed a merge-based method to significantly reduce the number of key-value pairs without sacrificing the pruning power. To improve the performance, we incorporated light-weight filtering units into key-value pairs to reduce the number of candidate pairs while not significantly increasing the transmission cost. Experimental results on real-world datasets show that our method outperforms state-of-the-art approaches.

Acknowledgement. This work was partly supported by the National Natural Science Foundation of China under Grant No. 61272090 and 61373024, National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302206, Beijing Higher Education Young Elite Teacher Project under grant No. YETP0105, a project of Tsinghua University under Grant No. 20111081073, Tsinghua-Tencent Joint Laboratory for Internet Innovation Technology, and the "NExT Research Center" funded by MDA, Singapore, under Grant No. WBS:R-252-300-001-490.

REFERENCES

[1] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy joins using mapreduce. In *ICDE*, pages 498–509, 2012.

[2] A. Arasu, V. Ganti, and R. Kaushik. Efficient exact set-similarity joins. In *VLDB*, pages 918–929, 2006.

[3] R. J. Bayardo, Y. Ma, and R. Srikant. Scaling up all pairs similarity search. In *WWW*, pages 131–140, 2007.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 5–16, 2006.

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

[6] D. Deng, Y. Jiang, G. Li, J. Li, and C. Yu. Scalable column concept determination for web tables using large knowledge bases. *PVLDB*, 6(13):1606–1617, 2013.

[7] J. Feng, J. Wang, and G. Li. Trie-join: a trie-based method for efficient string similarity joins. *VLDB J.*, 21(4):437–461, 2012.

[8] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33(2), 2008.

[9] Y. Kim and K. Shim. Parallel top-k similarity join algorithms using mapreduce. In *ICDE*, pages 510–521, 2012.

[10] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu. Distributed data management using mapreduce. *ACM Comput. Surv.*, 2014.

[11] G. Li, D. Deng, and J. Feng. A partition-based method for string similarity joins with edit-distance constraints. *ACM Trans. Database Syst.*, 38(2):9, 2013.

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

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

[14] J. Qin, W. Wang, Y. Lu, C. Xiao, and X. Lin. Efficient exact edit similarity query processing with the asymmetric signature scheme. In *SIGMOD Conference*, pages 1033–1044, 2011.

[15] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *SIGMOD*, pages 495–506, 2010.

[16] J. Wang, G. Li, and J. Feng. Trie-join: Efficient trie-based string similarity joins with edit-distance constraints. *PVLDB*, 3(1):1219–1230, 2010.

[17] J. Wang, G. Li, and J. Feng. Fast-join: An efficient method for fuzzy token matching based string similarity join. In *ICDE*, pages 458–469, 2011.

[18] J. Wang, G. Li, and J. Feng. Can we beat the prefix filtering?: an adaptive framework for similarity join and search. In *SIGMOD Conference*, pages 85–96, 2012.

[19] J. Wang, G. Li, and J. Feng. Extending string similarity join to tolerant fuzzy token matching. *ACM Trans. Database Syst.*, 2014.

[20] W. Wang, J. Qin, C. Xiao, X. Lin, and H. T. Shen. Vchunkjoin: An efficient algorithm for edit similarity joins. *IEEE Trans. Knowl. Data Eng.*, 25(8):1916–1929, 2013.

[21] C. Xiao, W. Wang, and X. Lin. Ed-join: an efficient algorithm for similarity joins with edit distance constraints. *PVLDB*, 1(1):933–944, 2008.

[22] C. Xiao, W. Wang, X. Lin, and H. Shang. Top-k set similarity joins. In *ICDE*, pages 916–927, 2009.

[23] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient similarity joins for near duplicate detection. In *WWW*, pages 131–140, 2008.