Fast and Scalable Distributed Set Similarity Joins for Big Data Analytics

Chuitian Rong[†♯], Chunbin Lin[‡], Yasin N. Silva[♯], Jianguo Wang[‡], Wei Lu[♭§], Xiaoyong Du[♭§]

† Tianjin Polytechnic University ♯ Arizona State University ‡ University of California, San Diego
§ Key Laboratory of Data Engineering and Knowledge Engineering, Ministry of Education, China
♭ School of Information, Renmin University of China
† chuitian@tjpu.edu.cn ♯ ysilva@asu.edu ‡ {chunbinlin, csjgwang}@cs.ucsd.edu § {lu-wei,duyong}@ruc.edu.cn

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, in this paper, we propose 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. To further improve the performance and scalability, FS-Join integrates horizontal partitioning. Experimental results on three 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.

I. 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 [18]. Similarity joins are widely used in a variety of applications including data integration [6], data cleaning [7], duplicate detection [22], record linkage [20] and entity resolution [8].

Most of the existing similarity join algorithms are in-memory approaches [5], [17], [3], [1], [2], [22], [12]. 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 [18], [13], [4]. 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., s_2 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 token[1]. 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 [18], [4]. Even though such signature-based methods improve the performance, they still have several limitations.

- **Generation of many duplicates.** If n tokens are selected as signatures of a string, then n duplicates are generated in order to create the (key, value) pairs, where each key is a token in the signature set [18], as shown in Figure 1. Although some previous techniques tried to merge key-value pairs, e.g., [4], duplicates still cannot be avoided.
- **Skewness problem.** Using signature tokens as keys may lead to a skewness problem, as the number of key-value pairs sharing the same key is not controllable or predictable, which heavily depends on the data distribution. Each reduce node has to passively receive strings containing the corresponding signatures.
- **Expensive verification processing.** Existing approaches usually use the original strings to calculate the accurate similarity score for each candidate string pair in the verification phase. The main operation is to compute the intersection size for each candidate, which is $O(m+n)$, where m and n are the lengths

---

[1]Sometimes, other combined information is also used to help pruning dissimilar string pairs, e.g., length information.
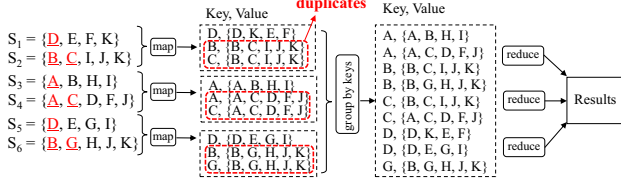
Fig. 1. Example of existing similarity joins using MapReduce. Each string is tokenized as a set, and the tokens within a string are ordered based on the the lexicographical order. The underscored tokens are selected as signatures. Each signature token is regarded as a key, and the corresponding string is regarded as its value.
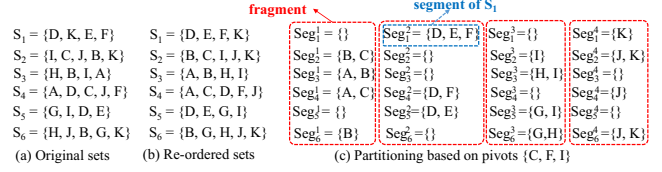

Fig. 2. Vertical partitioning based on selected pivots. (a) the original data, (b) the sorted data, and (c) a pivot-based partitioning.

of the two strings[2]. When the number of candidates is large, which is usually true[3], then the verification cost will dominate the whole execution time.

To tackle all these problems, we propose an innovative distributed string similarity join algorithm, *FS-Join*, which uses a filter-and-verification framework. In the filter phase, FS-Join partitions each original string into several disjoint *segments* according to carefully selected *pivots* and a global ordering (See Section IV). For example, Figure 2(c) shows the segments of each string for pivots $\{C, F, I\}$. Seg_1^1 and Seg_1^2 are the first two segments of string s_1. FS-Join assigns equivalent amount of strings to each map task with the partition ids as keys and the segments as values, e.g., $(1, Seg_1^1)$ and $(2, Seg_2^1)$ are two (key, value) pairs. By using (id, segment) pairs in the map phase instead of (token, string) pairs of existing algorithms, FS-Join avoids generating duplicates (*solves the first problem*). This is the case because each segment is unique. Then, FS-Join transmits all the segments with the same partition id into the same reducer. The segments with the same partition id constitute a *fragment*, e.g., each red dotted rectangle in Figure 2 denotes a fragment. From the perspective of data partitioning, the string dataset is partitioned vertically, which is orthogonal to the horizontal partitioning in existing methods. The number of tokens in fragments are the same (or similar), which is guaranteed by the selection of pivots. Therefore, each reduce node has the same size data, which guarantees proper load balancing on the reduce phase (*solves the second problem*). Then, each reduce task generates a list of candidates along with the number of common tokens. To decrease the number of candidates, we use a prefix-based inverted list index and a length filter. In addition, we also introduce three new segment-aware filters and novel hybrid partitioning optimization technique to further reduce the number of candidates (See Section V-A). In the verification phase, FS-Join uses a MapReduce job to simply aggregate the candidates to get the total number of common tokens (i.e., intersection size) without calling $O(m + n)$ algorithms to compute it on-the-fly (*solves the third problem*) (See Section V-B).

As MapReduce is a shared-nothing framework, it may seem that generating duplicates cannot be avoided when performing similarity joins. Duplicates are in fact generated by all previous techniques [18], [4]. However, this paper gives a surprising solution by successfully avoiding duplicates and achieving higher performance and better scalability than existing techniques.

The contributions of our work can be summarized as follows.

1) We propose a vertical-partitioning based algorithm, called FS-Join, to support parallel set similarity joins without generating duplicates. In addition, it guarantees load balancing in both map and reduce phases.

2) We introduce three new segment-based filtering methods, which significantly reduce the number of candidates.

3) We propose an optimization method by integrating horizontal data partitioning with vertical data partitioning to achieve higher scalability.

4) We implemented our method and experimental results on three real-world datasets show that our method outperforms state-of-the-art approaches by one order of magnitude.

The rest of the paper is structured as follows. In Section II we formulate our problem and review the related work. We introduce the architecture of FS-Join in Section III. Section IV presents the details of the vertical partitioning method used in FS-Join. Section V illustrates the details of FS-Join including its computation framework, its design choices and its optimizations. We theoretically and experimentally evaluate FS-Join in Section V-C and Section VI, respectively. Finally, we give the conclusion in Section VII.

II. PRELIMINARY

In string similarity joins, a string is usually treated as a set of tokens, where each token is a word [18]. For example, the set of tokens of string "ICDE conference" is {ICDE, conference}. For simplicity, we do not differentiate a string and its set of tokens unless there is ambiguity.

A. Problem Definition

*Definition 1 (*String Similarity Join*):* Given two string collections $\mathcal{S}$ and $\mathcal{T}$, a similarity function SIM and a similarity threshold θ, the string similarity join problem finds all the string pairs $(s, t) \in \mathcal{S} \times \mathcal{T}$ such that $\mathsf{SIM}(s, t) \geq \theta$.

Set-based Similarity Measures. There are three widely used set-based similarity functions, namely Jaccard [14], Dice [16] and Cosine [21], whose computation problem can be reduced to the set overlap problem [2]. These three similarity measures are defined in Table II along with examples. Unless otherwise specified, we use Jaccard as the default function, i.e., $sim(s, t) = sim_{Jaccard}(s, t)$.

B. MapReduce

MapReduce was proposed to facilitate the processing of large-scale datasets on shared-nothing clusters, particularly

---

[2]If two strings are not sorted according to a global ordering, then the time complexity is $O(mn)$.

[3]Assume two string collections having one billion strings and only 0.01% of the string pairs are candidates, the number of candidates is still $\tilde{O}(10^{14})$.

TABLE I.   COMPARISON OF MAPREDUCE-BASED STRING SIMILARITY JOIN ALGORITHMS.

Algorithm	# runs of reading original data	Filtering Phase						Verification Phase
		avoid duplicates	apply filters	load balancing		predictable size in Reduce	candidates generation	avoid reading original data
				Map	Reduce			
RIDPairsPPJoin[18]	≥ 3	✗	✓	✓	✓	✗	common prefix	✗
V-Smart-Join[13]	≥ 3	✓	✗	✓	✗	✗	enumerate pairs in inverted lists	✓
MassJoin[4]	≥ 3	✗	✓	✓	✗	✗	enumerate pairs in inverted lists	✗
FS-Join	2	✓	✓	✓	✓	✓	common prefix + segmentation	✓

TABLE II.   SIMILARITY FUNCTIONS. T_s AND T_t ARE THE SETS OF TOKENS OF STRING s AND t, RESPECTIVELY.

Similarity Function	Definition	Example
$sim_{jaccard}(s,t)$	$\frac{|T_s \cap T_t|}{|T_s \cup T_t|}$	$sim_{jaccard}("ab","bc") = \frac{1}{3}$
$sim_{dice}(s,t)$	$\frac{2 \times |T_s \cap T_t|}{|T_s| + |T_t|}$	$sim_{dice}("ab","bc") = \frac{1}{2}$
$sim_{cosine}(s,t)$	$\frac{|T_s \cap T_t|}{|T_s| \times |T_t|}$	$sim_{cosine}("ab","bc") = \frac{1}{4}$

commodity machines. Due to its high scalability and built-in fault tolerance, it has become the de facto platform for big data processing. A MapReduce program consists of two primitives, Map and Reduce. In the MapReduce framework, all the records are represented as key/value pairs. The input dataset is usually stored in a distributed file system (DFS) across several nodes that execute the job. Once the MapReduce job is submitted, the records are provided to mapper nodes in chunks. Each record will be parsed into key/value pair $<key_1, value_1>$, and the mapper applies the map function on each record to emit a list of new key/value pairs $list(<key_2, value_2>)$. The emitted key/value pairs with the same key will be sent to the same reduce node during the shuffle phase and will be grouped by their key into the form of value list $<key_2, list(value_2)>$. The reducer will receive the list of values $<key_2, list(value_2)>$ and apply the reduce function one them. Finally, the reducer will produce a list of key/value pairs $list(<key_3, value_3>)$ and write them out to the distributed file system. The above process can be formalized as below.

Map: $<key_1, value_1> \rightarrow list(<key_2, value_2>)$;
Reduce: $<key_2, list(value_2)> \rightarrow list(<key_3, value_3>)$;

C. Related Work

In-memory string similarity joins. There are many studies on in-memory similarity join algorithms [5], [17], [3], [1], [2], [22], 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 [9], [19], [15], [11]. 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. [22] 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. [9], [22] proposed more optimization techniques for the inverted list. [19] extended the fixed-length based prefix filtering and proposed a variable-length prefix filtering. For the same string, [19] extracts more tokens than fixed-length based prefix filtering methods and constructs multiple inverted indices incrementally. [15] explored various global orderings and proposed a multiple prefix filtering based algorithm. The multiple prefix filters are applied in an incrementally manner. [11] 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.

MapReduce-based string similarity joins. To support string similarity joins over big data, many recent contributions focus on implementing algorithms on Map-Reduce [18], [13], [4]. More precisely, [18] 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[4]. 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 [13].

[13] 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 [13], *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:

---
[4]The tokens in a string are ordered according to a pre-defined global order.
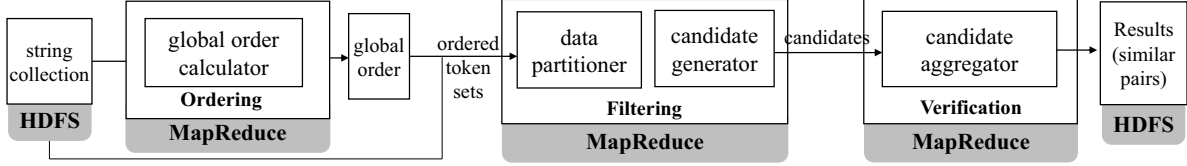
Fig. 3.   Architecture of FS-Join.

(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.

[4] proposed a similarity join method, called *MassJoin*, for long strings based on their centralized algorithm in [10]. In [4], 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* [18], *V-Smart-Join* [13] and *MassJoin* [4].

III.   FS-JOIN ARCHITECTURE

Figure 3 shows the architecture of FS-Join, which contains three phases: *Ordering*, *Filtering* and *Verification*.

Ordering. The first step of FS-Join is to obtain a global order $\mathcal{O}$. FS-Join adopts the ordering method proposed in [18], 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 [18].

Filtering. 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* (See Section V). More precisely, the *data partitioner* first receives the global order $\mathcal{O}$ returned from the previous ordering step and selects a number of pivots from $\mathcal{O}$. 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 (See Section IV).

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 (See Section V). 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.

Verification. 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 Section V-B.

IV.   VERTICAL PARTITIONING

In this paper, we propose 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. In order to present our main ideas clearly, we give several definitions below.

Definition 2 (Token Domain): All distinct tokens from a given collection constitute a token set. We define this token set as the Token Domain, denoted as U.

Definition 3 (Global Ordering): For the tokens domain U, we define a total order on its tokens by a sorting method. For example, we sort all the tokens according to their term frequencies. We call the defined total order a *Global Ordering*, denoted as $\mathcal{O}$.

Definition 4 (Pivots): A pivot is a token. A set of selected tokens from U form a set of pivots, denoted as $\mathcal{P}$.

Definition 5 (Segment): When the tokens of a string are sorted by a global ordering $\mathcal{O}$, the pivots $\mathcal{P}$ split the string (s_i) into several disjoint segments $Seg_i^1, ..., S_i^{|p|+1}$. There is no overlap between any two segments of a string, i.e., $\forall a \neq b \in [1, |\mathcal{P}| + 1]$, $Seg_i^a \cap Seg_i^b = \emptyset$.

Definition 6 (Fragment): For all strings in the given collection S, the segments produced by the same pivot constitute a fragment $\mathcal{F}$. The i-th fragment is defined as $\mathcal{F}_i = \{Seg_1^i, Seg_2^i, ..., Seg_{|S|}^i\}$.

Example 1: Taking the data set in Figure 2(a) as an example. First, we define a global ordering, e.g., the dictionary order $\mathcal{O} = \{A \rightarrow B \cdots \rightarrow Z\}$. Each string is sorted by

the same global ordering, as shown in Figure 2(b). Then, we select a set of pivots $\mathcal{P}=\{C, F, I\}$. The pivots split each sorted tokens set into four segments (See Figure 2(c)). For example, string s_1 is split into four segments $Seg_1^1=\{\}$, $Seg_1^2=\{D, E, F\}$, $Seg_1^3=\{\}$ and $Seg_1^4=\{K\}$. The segments belonging to the same partition (indicated by the second subscripts) form a fragment. For example, the first fragment $\mathcal{F}_1$ contains the first segments of all the strings, i.e., Seg_1^1, Seg_2^1, Seg_3^1, Seg_4^1, Seg_5^1 and Seg_6^1.

Based on the above analysis, it should be clear that 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, we 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.

Pivots Selection Methods. There are many kinds of possible methods to select pivots from the token domain. In this paper, we study the following three approaches.

Random Selection (Random). We assign each token a equivalent probability to be selected as a pivot. Then we randomly choose $|\mathcal{P}|$ tokens from the token domain $\mathcal{O}$ as pivots. The distance between each adjacent pivots pair in $\mathcal{P}$ may vary significantly. Thus, this method cannot ensure an even workload partitioning.

Even Interval (Even-Interval). In order to get a more even workload partitioning, a better way to select the pivots is to split the global ordering $\mathcal{O}$ into even segments. This method can ensure that the token domain is partitioned evenly as the interval between each pair of adjacent pivots is equal. Still, as the tokens' term frequency is different, this method cannot achieve workload balancing either. The pivots set $\mathcal{P}$ can be formalized as below ($N_p = |\mathcal{P}|$).
$$\mathcal{P} = \{\mathcal{O}_i | i = k \times |\mathcal{O}|/N_p, 0 < k \leq N_p\} \quad (1)$$

Even Token Frequency (Even-TF). Based on the above analysis, we 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 $\mathcal{O}$ evenly. The pivots set $\mathcal{P}$ can be formalized as below ($N_p = |\mathcal{P}|$, $TF_{\mathcal{O}_n}$ is the term frequency of the n-th token in $\mathcal{O}$).

$$\mathcal{P} = \{\mathcal{O}_{m_i} | \sum_{n=m_{i-1}}^{m_i} TF_{\mathcal{O}_n} = \sum_{t=0}^{|\mathcal{O}|} TF_{\mathcal{O}_t}/N_p, 0 \leq i \leq N_p\} \quad (2)$$

In this paper, FS-Join applies the *Even-TF* to choose pivots. Experiments in Section VI-D demonstrate that using *Even-TF* is more efficient than the other two approaches. *Even-TF* has a load balancing guarantee, while the others do not.

The Number of Pivots. 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 $N_c - 1$ ($N_c \geq D/M$).

Algorithm 1: FS-Join Algorithm

1 //Ordering phase
2 **SetUp**(context)
3 $\mathcal{O} \leftarrow$ Load Global Ordering;
4 $\mathcal{P} \leftarrow PiovtsSelection(\mathcal{O}, \mathcal{N})$;

5 //Filtering phase
6 **Map**(rid, string)
7 $segments \leftarrow VerticalPartition(string, \mathcal{O}, \mathcal{P})$;
8 **foreach** *segment $\in$ segments* **do**
9 $\quad$ context.write(*partitionID*, $<segment, segInfo>$);

10 **Reduce**(*partitionID*, $list(<segment, segInfo>)$)
11 Candidates $\leftarrow PerformJoin(<segment, segInfo>)$;
12 **foreach** *candidate $\in$ Candidates* **do**
13 $\quad$ context.write($RidPair, Count$);

14 //Verification phase
15 **Map**(*RidPair*, *Count*)
16 context.write($RidPair, Count$);

17 **Reduce**(*RidPair*, *list(Count)*)
18 $sim \leftarrow Verification(list(Count))$;
19 **if** $sim \geq \theta$ **then**
20 $\quad$ context.write(*RidPair*, *sim*);

V. 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 shared-nothing clusters, data duplication is a common technique utilized to improve parallel processing [18], [4]. 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 [18], [4] 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.

A. Generating candidates.

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 4. The algorithm is given in Algorithm 1. 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 $\mathcal{N}_p$ pivots $\mathcal{P}$ using the global ordering $\mathcal{O}$ (Line 3 - 4). The pivots split the sorted token set $\mathcal{O}$ into $\mathcal{N}_p + 1$ segments $Seg_k (0 \leq k \leq \mathcal{N}_p + 1)$. 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, [18], [4] choose the tokens in signatures as the keys, which results in large number of duplicates, as shown in Figure 1. However, FS-Join uses partition ids as keys, which has no duplicates as shown in Figure 4.

Join Algorithms. Once the partition task is completed, the segments with the same key will be shuffled to the same node. From the perspective of data partitioning, the strings collection are partitioned vertically and the segments of the same fragment will be processed by one node. The main task of performing joins is to get the number of common tokens between each segment pairs. Based on the work in relational databases, there are many techniques to implement join operations.

Loop Join. The naive method to implement the join operation is the loop join. Since the tokens in each segment are sorted according to $\mathcal{O}$, the time cost to identify segment intersection is determined by their length.

Index Join. Inverted lists can be utilized to accelerate the process of identifying the number of common tokens between each segment pair. All the tokens of each segment of the same fragment will be mapped to corresponding inverted lists. For each segment, we probe the related inverted list to get the segments' identifiers. Then, the results can be computed by an aggregation.

Prefix Based Index Join. Prefix filtering [18] is a popular technique applied in set similarity join due to its efficient pruning power. Given that the strings are sorted by the tokens' ascending order of term frequency, the prefix of each string is composed of the tokens with smaller term frequency. This minimizes the size of constructed index. According to the property of prefix filtering, similar strings should share at least one common prefix token. In practice, many segments in the same fragment do not share any prefix tokens, especially for the fragments containing the tokens with high term frequency. In order to reduce the unnecessary cost, we use the prefix filtering for segment intersections in each fragment.

The above three methods are widely used in main memory based algorithms. FS-Join chooses the prefix-join, which achieves higher performance than the others (This is verified in Section VI-E).

Filtering Methods. The join algorithms should be utilized in conjunction with other filtering techniques to achieve high efficiency. In this work, we applied four filtering methods: *string length filtering (StrL-Filter)*, *segment length filtering (SegL-Filter)*, *segment intersection filtering (SegI-Filter)* and *segment difference filtering (SegD-Filter)*. SegL-Filter, SegI-Filter and SegD-Filter are novel contributions.

In order to apply these filtering methods, each segment should be accompanied with other information, including the number of tokens in the string ($|s|$), the number of tokens ahead of the segment($|s^h|$), the tokens in the current segment Seg_s^m and the number of tokens behind the segment($|s^e|$).

String Length Filtering (StrL-Filter). The motivation of the length filtering is that two similar strings have similar lengths.

Lemma 1 (StrL-Filter): Given two strings s and t ($|s| < |t|$), a threshold value θ, if $|s| < \theta \cdot |t|$, then $\mathsf{sim}(s,t) < \theta$.

Proof: As $|s \cap t| \leq |s|$ and $|s \cup t| \geq |t|$ always hold. So $\mathsf{sim}(s,t) = \frac{|s \cap t|}{|s \cup t|} \leq \frac{|s|}{|t|}$. If $|s| < \theta \cdot |t|$ holds, then $\mathsf{sim}(s,t) < \theta$. ∎

Segment Length Filtering (SegL-Filter). Recall that given a pivot set, a string is partitioned into several segments. Let Seg_s^i be the i-th ($i > 0$) segment of string s, s^h be the token set of the substring of s from the beginning to the $(i-1)$-th segment, s^e be the token set of the substring of s from the $(i+1)$-th segment to the end. For example, consider the string token set $s = \{B, C, I, J, K\}$, $Seg_s^2 = \{I\}$, then $s^h = \{B, C\}$ and $s^e = \{J, K\}$.

Lemma 2 (SegL-Filter): Given two strings s and t ($|s| < |t|$), a threshold value θ, and let n be the size of pivots, if $\forall i \in [1, n+1]$, $min(|Seg_s^i|, |Seg_t^i|) < \frac{\theta}{1+\theta} \times (|s| + |t|) - min(|s^h|, |t^h|) - min(|s^t|, r^t|)$ holds, then $\mathsf{sim}(s,t) < \theta$.

Proof: Since $min(|s^h|, |t^h|) \geq |s^h \cap t^h|$, $min(|s^e|, t^e|) \geq |s^e \cap t^e|$ and $min(|Seg_s^i|, |Seg_t^i|) \geq |Seg_s^i \cap Seg_t^i|$ always hold, we have

$$min(|Seg_s^i|, |Seg_t^i|) < \frac{\theta(|s| + |t|)}{1 + \theta} - min(|s^h|, |t^h|) - min(|s^e|, t^e|)$$

$$\Rightarrow min(|Seg_s^i|, |Seg_t^i|) < \frac{\theta}{1+\theta} \times (|s| + |t|) - |s^h \cap t^h| - |s^e \cap t^e|$$

$$\Rightarrow |Seg_s^i \cap Seg_t^i| < \frac{\theta}{1+\theta} \times (|s| + |t|) - |s^h \cap t^h| - |s^e \cap t^e|$$

Since all the strings in the collection are sorted by the same global ordering $\mathcal{O}$ and split by the same pivots $\mathcal{P}$, their common tokens come from segments that belong to the same fragment. Therefore, $\forall i$, $|s \cap t| = |s^h \cap t^h| + |Seg_s^i \cap Seg_t^i| + |s^e \cap t^e|$. The above inequation can be transformed to

$$|Seg_s^i \cap Seg_t^i| + |s^h \cap t^h| + |s^e \cap t^e| < \frac{\theta}{1+\theta} \times (|s| + |t|)$$

$$\Rightarrow |s \cap t| < \frac{\theta}{1+\theta} \times (|s| + |t|) \Rightarrow \frac{|s \cap t|}{|s| + |t| - |s \cap t|} < \theta$$

Since $Sim(s,t) = \frac{|s \cap t|}{|s \cup t|} = \frac{|s \cap t|}{|s| + |t| - |s \cap t|}$, we have $Sim(s,t) < \theta$. ∎

Example 2: Consider two strings s="A, B, D, E, G" and t="B, D, E, F, K", θ=0.8. Let $\{D, G\}$ be the selected pivots. Then $Seg_s^1 = \{A, B, D\}$, $Seg_s^2 = \{E, G\}$, $Seg_s^3 = \{\}$ and $Seg_t^1 = \{B, D\}$, $Seg_t^2 = \{E, F\}$, $Seg_t^3 = \{K\}$. For i=1, $min(|Seg_s^1|, |Seg_t^1|) = min(3, 2) = 2$, $\frac{\theta}{1+\theta} \times (|s| + |t|) - min(|s^h|, |t^h|) - min(|s^t|, r^t|) = \frac{0.8}{1+0.8} \times 10 - min(0, 0) - min(2, 3) > min(|Seg_s^1|, |Seg_t^1|)$. Similarly, for i=2 and i=3, the condition holds, so it is safe to claim that the similarity of s and t is less than 0.8, which means (s,t) can be pruned without verification.

Segment Intersection Filtering (SegI-Filter). In addition, we observe that if two strings have few common tokens (below a certain threshold value), then the two strings are not similar. Based on this observation, we propose segment intersection filtering (called SegI-Filter).
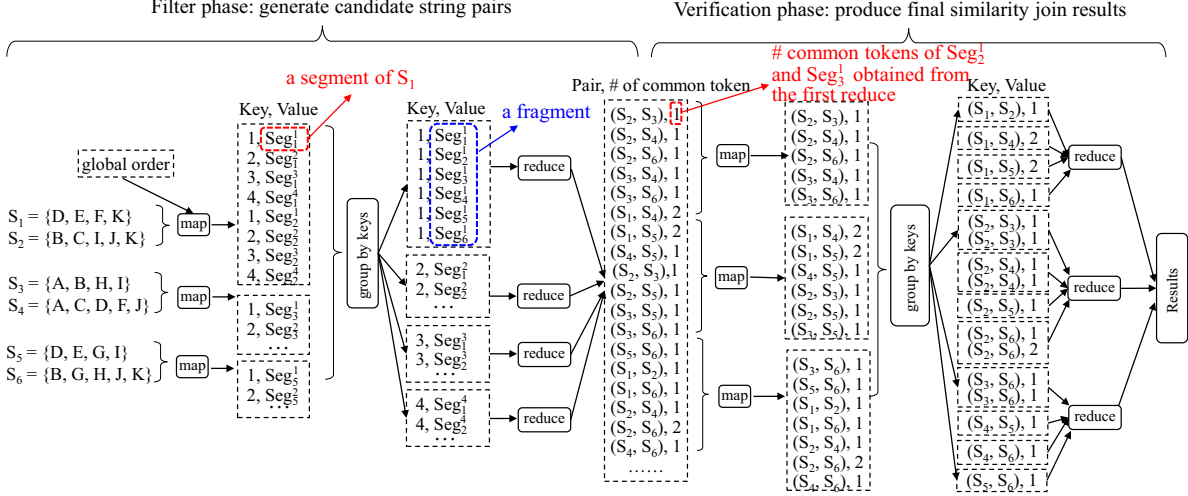
Fig. 4. Computation framework of FS-Join.

Lemma 3 (SegI-Filter): Given two strings s and t ($|s| < |t|$), a threshold value θ, and let n be the size of pivots, if $\forall i \in [1, n+1]$, $|Seg_s^i \cap Seg_t^i| < \frac{\theta}{1+\theta} \times (|s|+|t|) - |s^h \cap t^h| - |s^e \cap t^e|$ holds, then $\mathsf{sim}(s,t) < \theta$.

The proof of Lemma 3 is similar to that of Lemma 2, so we omit it here due to space sontraints.

Segment Difference Filtering (SegD-Filter). We further propose another segment-based filtering method *Segment Difference Filtering*, which is based on the difference of two segments.

Lemma 4 (SegD-Filter): Given two strings s and t, and a threshold value θ, if $\forall i$, $|Seg_s^i - Seg_t^i| + |Seg_t^i - Seg_s^i| > \frac{(1-\theta)(|s|+|t|)}{1+\theta} - (abs(|s^h| - |t^h|) + abs(|s^e| - |t^e|))$ holds, then $\mathsf{sim}(s,t) < \theta$.

Proof: $|s^h - t^h| + |t^h - s^h| \geq abs(|s^h| - |t^h|)$ and $|s^e - t^e| + |t^e - s^e| \geq abs(|s^e| - |t^e|)$ always hold, so if $\forall i$, $|Seg_s^i - Seg_t^i| + |Seg_t^i - Seg_s^i| > \frac{(1-\theta)(|s|+|t|)}{1+\theta} - (abs(|s^h| - |t^h|) + abs(|s^e| - |t^e|))$ holds, we have

$$|Seg_s^i - Seg_t^i| + |Seg_t^i - Seg_s^i| > \frac{(1-\theta)(|s|+|t|)}{1+\theta}$$
$$- ((|s^h - t^h| + |t^h - s^h|) + (|s^e - t^e| + |t^e - s^e|))$$

Since the strings are split into several segments, the difference between two strings can be expressed by the differences between their corresponding segments. That is

$$|s-t| + |t-s| = |s^h - t^h| + |t^h - s^h| + |Seg_s^i - Seg_t^i| + |Seg_t^i - Seg_s^i|$$
$$+ |s^e - t^e| + |t^e - s^e|$$

Thus, we can get the following expression:

$$|s-t| + |t-s| > \frac{(1-\theta)(|s|+|t|)}{1+\theta}$$

As $|s-t| = |s| - |s \cap t|$ and $|t-s| = |t| - |s \cap t|$, we have: $|s \cap t| < \frac{\theta(|s|+|t|)}{1+\theta} \Rightarrow \frac{|s \cap t|}{|s \cup t|} < \theta$. ∎

Optimization: Horizontal Partitioning. FS-Join adopts vertical partitioning to achieve high scalability by avoiding duplicates. In the filtering phase, each fragment is processed by one reduce node, which requires the reduce nodes to have enough memory space for the whole fragment. To enhance the scalability and performance of FS-Join without requiring large memory space in the reduce nodes, we employ *horizontal*

partitioning to divide strings based on their length. *Horizontal partitioning* is based on the observation that similar strings have similar lengths. More precisely, given two strings s and t, if $\frac{|s \cap t|}{|s \cup t|} \geq \theta$ holds, then their lengths should satisfy $\lceil |t| \times \theta \rceil \leq |s| \leq \lfloor |t| / \theta \rfloor$.

FS-Join uses *horizontal partitioning* to divide each fragment into several sections by using a set of *Horizontal pivots*. In this subsection, we first describe how horizontal pivots are selected. Then, we present the details of how they are used.
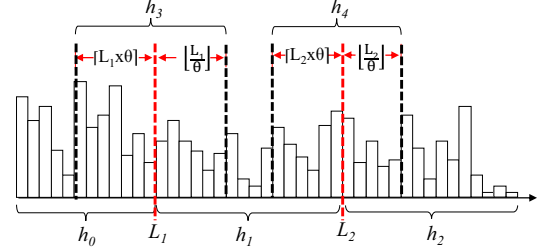

Fig. 5. Horizontal Partitioning.

Horizontal pivots selection

FS-Join first creates a statistic length histogram $\mathcal{H}$, which describes the length distribution of the string collection. Then FS-Join chooses a set of horizontal pivots $\mathcal{P}_{\mathcal{H}} = \{L_1, ..., L_t\}$ based on $\mathcal{H}$ in order to get the same total length in each partition. The length histogram $\mathcal{H}$ can be created in the *Ordering* phase and the horizontal pivots $\mathcal{P}_{\mathcal{H}}$ can be selected in the *SetUp* procedure of the *Filtering* phase. Let k be the number of bins in $\mathcal{H}$, l_i the length corresponds to the i-th bin, c_i the number of strings with length l_i, and t the number of pivots. The pivot set $\mathcal{P}_{\mathcal{H}} = \{L_1, ..., L_t\}$ can be computed as follows($N_h = |\mathcal{P}_{\mathcal{H}}|$):

$$\mathcal{P}_{\mathcal{H}} = \{L_p | \sum_{j=p-1}^{p} (l_j \times c_j) = (\sum_{i=0}^{k} l_i \times c_i) / N_h\} \quad (3)$$

FS-Join uses the horizontal pivots $\mathcal{P}_{\mathcal{H}} = \{L_1, ..., L_t\}$ to partition strings into $2|\mathcal{P}_{\mathcal{H}}| + 1$ (i.e., $2t+1$) partitions $h_0, ..., h_{2t}$ performing the following two steps: (1) For the first $t+1$ partitions $h_0, ..., h_t$, FS-Join assigns all the strings with length in the range $[L_i, L_{i+1})$ to the partition h_i ($i > 0$). Strings with

length less than L_1 are assigned to partition h_0, and those with lengths greater than or equal to L_t are distributed to partition h_t. (2) For the last t partitions $h_{t+1}, ..., h_{2t}$, FS-Join assigns all the strings with length in the range $[\lceil L_i \times \theta \rceil, \lfloor L_i/\theta \rfloor]$ to the partition h_{t+i}. Observe that strings in h_{t+i} come from h_{t-1} and h_t, and that strings in h_{t+i}, h_{t-1} and h_t will be shuffled to different nodes according to their horizontal partition ids. In order to avoid producing duplicate results, we perform joins in h_{t+i} only among strings s and r that satisfy $|s| < L_i$ and $|r| \geq L_i$.

Example 3: Figure 5 shows a sample length histogram. Assume that the selected pivot set is $\mathcal{P}_{\mathcal{H}} = \{L_1, L_2\}$. This set is used to partition all the strings into $2 \times 2 + 1 = 5$ partitions. The strings with length less than L_1 are mapped to horizontal partition h_0, while those with lengths between L_1 and L_2 are mapped to horizontal partition h_1, and those with lengths greater than or equals to L_2 are assigned to horizontal partition h_2. Additionally, the strings with length between $\lceil L_1 \times \theta \rceil$ and $\lfloor L_1/\theta \rfloor$ (or $\lceil L_2 \times \theta \rceil$ and $\lfloor L_2/\theta \rfloor$) are mapped to h_3 (or h_4).

Combining vertical and horizontal partitioning
FS-Join uses vertical partitioning to achieve high performance and good scalability. Horizontal partitioning is integrated to further improve the scalability and performance. FS-Join first maps each string to horizontal partitions according to its length and the horizontal pivots. Then, each string is partitioned into segments and mapped to the corresponding fragment by using vertical partitioning. Finally, each segment is associated with a pair (h, v), where h is the horizontal partition number and v is the vertical partition number. Segments are shuffled and aggregated based on the partition id pairs in the filtering phase.

B. Candidate Verification

FS-Join first generates candidates by pruning many dissimilar pairs using the filter methods mentioned in the previous section. In this subsection, we describe how the candidates are verified. FS-Join uses one MapReduce job to combine partial results and get the id pairs of similar strings as the final results. Taking the output of the second MapReduce job as input, the mapper outputs the Key/Value pairs using the same format as the input. The number of common tokens between different segments of the same string pair are shuffled to the same reducer. FS-Join aggregates the numbers of common tokens for each string pair. Finally, reducers compute the similarity score and output the id pairs of records whose similarity score is not less than the given threshold. Let c be the number of final common tokens for string pair (s, t). It is easy to know whether (s, t) is a similar pair or not.

- If $\frac{c}{|s| + |t| - c} \geq \theta$, then $sim_{jaccard}(s, t) \geq \theta$.
- If $\frac{2c}{|s| + |t|} \geq \theta$, then $sim_{dice}(s, t) \geq \theta$.
- If $\frac{c}{|s| \times |t|} \geq \theta$, then $sim_{cosine}(s, t) \geq \theta$.

Observe that, FS-Join computes the accurate similarity scores without using the original data.

C. Cost Analysis

In this part, we analyze the cost of FS-Join[5] performing self-join over a collection S using MapReduce. We assume that

---

[5]Here we analyze the cost of the filtering and verification phases excluding the ordering phase.

DataSet	Size (GB)	# of Records	Length(Number of tokens)		
			Min	Max	Average
Email	0.994	517,401	51	148,624	320.69
PubMed	4.390	7,400,308	1	1,142	80.59
Wiki	1.650	4,305,022	1	42,639	55.95

all the worker nodes are equipped with the same computational capabilities. The costs includes the cost of the mapper, shuffle, reducer and results output. Let C_m, C_s, C_r and C_o be the cost of processing one unit of data in the mapper, shuffle, reducer and output respectively. Let M be the number of strings in the dataset, N be the number of partitions which is equal to the numbers of workers in the cluster.

Lemma 5 (Cost of FS-Join): The cost of FS-Join is $\sum_{i=1}^{M}(|s_i|C_m) + \sum_{i=1}^{M}(|s_i|C_s) + (\frac{M \times P}{N})^2 \times (\frac{N}{M} \sum_{i=1}^{M} |s_i| \times C_r + N \times \alpha(C_m + C_s + C_r + C_o) + \alpha \times \beta \times C_o)$, where α is the proportion of the string pairs identified as candidates pairs, and β is the proportion of the candidates pairs that are included in the result.

The detailed proof of Lemma 5 is provided in Appendix A.

VI. EXPERIMENTS

In this section, we evaluate FS-Join's capbilities by running string similarity joins on three real-world datasets. We first compare our algorithms with the state-of-art algorithms under different thresholds on three datasets (Section VI-B). 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.

- We evaluate the scalability of our algorithms with the variations of data scale and the number of worker nodes in the cluster (Section VI-C).
- We evaluate the effect of different pivots selection methods (Section VI-D).
- We evaluate the effect of different join approaches (Section VI-E).
- We evaluate the effect of different optimizations including the filter methods and the horizontal partitioning (Section VI-F).

A. Experiments Setup

Cloud Platform. All the algorithms were implemented and evaluated using Hadoop (0.20.2), the most popular open-source MapReduce framework. The experiments were performed using a Hadoop cluster running on the Amazon Elastic Compute Cloud (EC2). Unless otherwise stated, we used a cluster of 11 nodes (1 master + 10 worker nodes) with the following specifications: 15 GB of memory, 4 virtual cores with one EC2 Compute Unit [6], 400 GB of local instance storage, 64-bit platform. On each worker node, the maximum map and reduce tasks is 3. So, we set the number of reduce tasks to be three times the number of nodes.

DataSets. We evaluated all the systems with three public datasets, which cover a wide range of data distributions and are

---

[6]https://aws.amazon.com/ec2/

widely used in previous studies. We generate different scales of the datasets by random sampling.

- **Enron Email (Email)**[7] is an email messages dataset collected and prepared by the CALO project. It contains the messages of 150 users.
- **PubMed Abstract (PubMed)**[8] is an abstract dataset of biomedical literature from MEDLINE.
- **Wiki Abstract (Wiki)**[9] is an abstract dataset extracted from Wikipedia dumps generated in February / March 2015.

The statistical information of the three datasets are given in Table III.

Alternative Techniques. To provide an end-to-end comparison, we compared FS-Join with the state-of-art methods *RIDPairsPPJoin* [18], *V-Smart-Join* [13] and *MassJoin* [4]. Each of them proposed several optimized algorithms. In our experiments, we used the algorithms that perform the best.

- *RIDPairsPPJoin* [18]. As the authors published the source code, we directly used their code in our experiments.
- *V-Smart-Join* [13]. Since there is no public source code, so we implemented its *Online-Aggregation* algorithm from scratch.
- *MassJoin* [4]. We implemented two of its algorithms, *Merge* and *Merge+Light*, from scratch as there is no publicly available code.

B. Comparisons with State-of-the-art Methods

Figure 6 and Figure 7 present the running time of string similarity join queries (self-join) for each approach on big and small datasets, respectively. Since *MassJoin* and *V-Smart-Join* cannot run successfully on the large datasets, we also conduct the experiments on small datasets generated by random sampling. From the experimental results, we can observe that FS-Join performs and scales significantly better than the other techniques. We further observed that:

- With smaller threshold values, FS-Join achieves better performance. For instance, FS-Join outperforms *RID-PairsPPJoin* by 5 times and 10 times, respectively, when the threshold is set to 0.9 and 0.75 in Figure 6(a). Observe that a lower threshold means larger signatures for each string. It also generates a huge amount of duplicates in *RIDPairsPPJoin*, which results in expensive shuffle cost and unnecessary computation.
- *MassJoin* is unable to answer string similarity joins on the large datasets, since it generates huge amounts of duplicates. More precisely, *MassJoin* has four MapReduce jobs to complete the join operation. The first job is to get the token frequency. The other three jobs constitute the three phases of the join operation. When running its *Merge* algorithm on the whole *Wiki* dataset with threshold 0.8, the first phase job takes more than 13 minutes to complete, which is much more than

the whole execution time of *RIDPairsPPJoin* and FS-Join. The output size of the first phase job of *Merge* is 105GB while the size of the input *Wiki* dataset is 1.65GB . The shuffle cost and HDFS writing cost is very high. The same problem is found when running *MassJoin* on the other two datasets. In the last job of *MassJoin*, it will enumerate each item in the inverted lists and output the same string multiple times with the items. Although *Merge+Light* applied a light filtering by token grouping method, it cannot run successfully on large datasets.

- *V-Smart-Join* cannot run completely on the large datasets. One of its drawbacks is that it must emit each token of the string with related information in the Map. Then, the tokens' information will be shuffled to reduce nodes in the form of value lists. In the Reduce, *V-Smart-Join* enumerates each pair of items in each value list. Another drawback is that no filtering methods are applied. The output size of the first *Similarity* phase is very large. Even on the small *PubMed* dataset, its algorithm *Online-Aggregation* cannot run completely. When running on small *Email* and *Wiki* datasets, the performance of *Online-Aggregation* is the worst among the tested algorithms (See Figure 7). Since *V-Smart-Join* uses the threshold only in the last reduce task to get the final results, its time is not sensitive to the variation of the threshold.
- For small datasets, both *RIDPairsPPJoin* and FS-Join outperform *Merge* and *Merge+Light* on all the datasets (See Figure 7). In particular, in the Email dataset, when the threshold is low, the performance of *Merge* is extremely slow (more than 100x slower than the others). As *Merge+Light* adopted a light filtering by token grouping technique, the shuffle cost decreases significantly. So, *Merge+Light* performs better than *Merge*. Since the datasets are small, the time cost difference between *RIDPairsPPJoin* and FS-Join is also very small.

C. Scalability

In order to study the scalability of FS-Join, we conducted experiments to test its execution time with the variation of (1) data scale, and (2) the number of computing nodes. We plotted the experimental results in Figure 8 and Figure 9, respectively.

Figure 8 shows the execution time of FS-Join on three datasets under different data sizes. In this experiment, we used four different scales for each dataset, 4X, 6X, 8X and 10X. Specifically, 6X means that we extracted 60% of strings randomly. Since the similarity join is a pairwise-comparison problem, the number of candidate pairs increases quadratically on the size of dataset. From Figure 8, we can observe that when the data size increases by 2X, the time cost will increase less than 33% in most cases(under the same threshold).

Figure 9 shows the scalability of FS-Join with the variation of the number of computing nodes. We conducted experiments on 5 nodes, 10 nodes and 15 nodes, respectively. In this experiment, we set the number of reduce tasks to be three times the number of nodes. From the three sub-figures in Figure 9, we can observe that the time cost decreases about 35%-48% when the computing nodes increases from 5 to 10 on all the
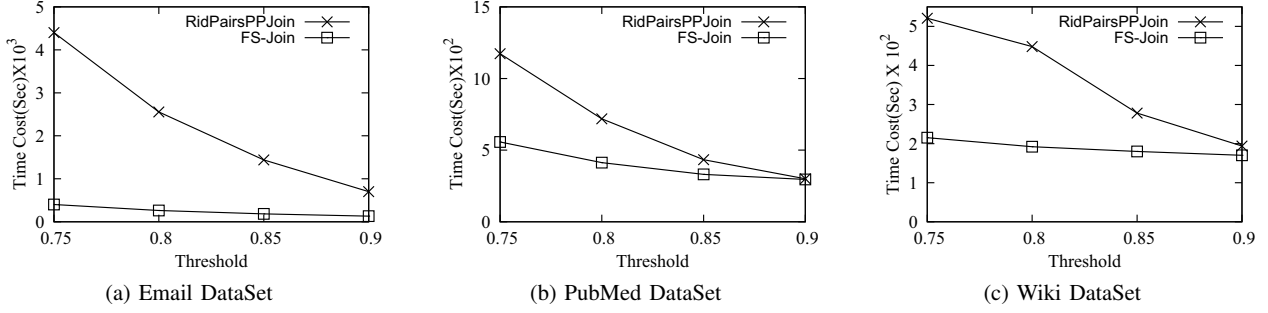
(a) Email DataSet   (b) PubMed DataSet   (c) Wiki DataSet

Fig. 6.   Comparison With Existing Methods (Large Datasets)

(a) Email DataSet(2%)   (b) PubMed DataSet(1%)   (c) Wiki DataSet(1%)
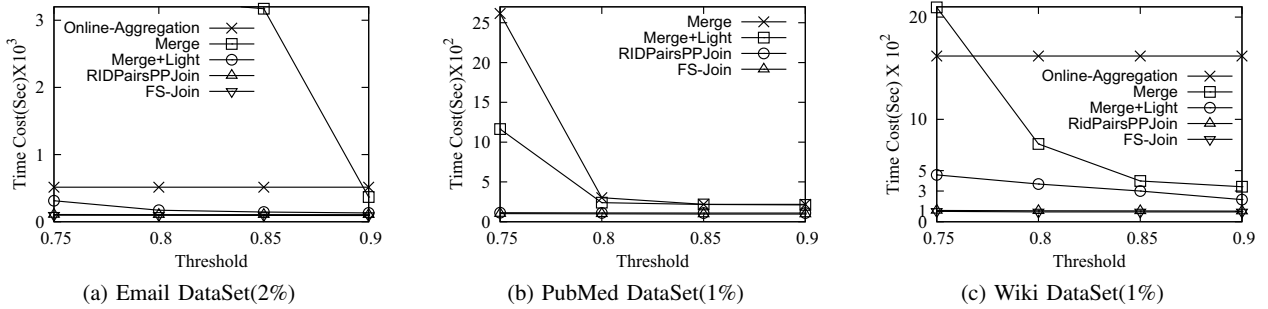
Fig. 7.   Comparison With Existing Methods (Small Datasets)

datasets. As the number of nodes increasing, the number of reduce tasks also increases. At the same time, the shuffle cost among computing nodes also increases. Thus, the time costs decrease about 10%-20% when the nodes number increases from 10 to 15. Overall, FS-Join has good scalability with the number of computing nodes in the cluster.

D. Effect Of Pivot Selection

In Section IV, we studied three pivot selection methods, i.e., *Random*, *Even-Interval* and *Even-TF*. We conducted experiments to evaluate the effects of these three different approaches. As shown in Figure 11, *Even-TF* performs the best on the three datasets followed by *Even-Interval* and *Random*. This is because both *Random* and *Even-Interval* incur on workload imbalance among worker nodes, while *Even-TF* achieves good workload balancing. More specifically, *Even-Interval* ensures an equal number of distinct tokens in each fragment. As we adopt the ascending order of token frequency as the global ordering, the last fragment contains the tokens with high term frequency. So, the reduce tasks that process the sections from the last fragment take much more time than other reduce tasks, which results in workload imbalance. Nevertheless, *Even-TF* carefully selects the pivots to ensure each fragment contains an equal number of tokens, which guarantees proper workload balancing among worker nodes.

E. Effect Of Join Methods

Recall that there are three types of join methods studied in Section V. They are *Loop Join*, *Index Join* and *Prefix Join*. In this subsection, we conducted experiments to evaluate the effects of these three different join methods. As shown in Figure 12, *Prefix Join* is clearly the winer among those three. In particular, for datasets with long strings (e.g., Email dataset), *Prefix Join* is about two times faster than *Loop Join* and *Index Join*.

F. Effect Of Optimizations

Effects of different filters. FS-Join uses four filtering strategies in the join procedure as mentioned in Section V-A. They are *StrL-Filter (String length filtering)*, *SegL-Filter (Segment length filtering)*, *SegI-Filter (Segment intersection filtering)*, *SegD-Filter (Segment difference filtering)* and *Prefix-Filter*. Table IV shows the filtering power of each of them and also the combination of all of them (i.e., All). For efficiency considerations, we combined *StrL-Filter* with the other four filters and conducted experiments on three datasets: Email(10%), Wiki(1%) and PubMed(1%). The numbers in the table are the output number of records of the filter job.

TABLE IV.   FILTER EFFECTIVENESS

Filter	Email(10%)	Wiki(1%)	PubMed(1%)
StrL	271,385,025	1,473,167,384	1,403,760,351
StrL + SegL	233,063,886	1,449,842,593	1,399,927,097
StrL + SegI	1,164,102	2,287,718	31,498
StrL + SegD	1,143,783	1,236,775	8,342
StrL + Prefix	1,011,428	1,147,016	792,185
All	493,644	515,664	6,840

From Table IV, we can observe that in general *SegD-Filter* has the most efficient and stable pruning power among all the filters on the three datasets, followed by *SegI-Filter*. *SegD-Filter* can prune out more than 90% false positives based on the results of *StrL-Filter*. *Prefix-Filter* performs the best on Email and Wiki. However, it is not as stable as *SegD-Filter* and *SegI-Filter* due to the token distributions of the datasets. When the filters are applied together, FS-Join prunes most of the false positives. See from the Figure 10, the time cost of verification phase is much smaller than that of the join phase.
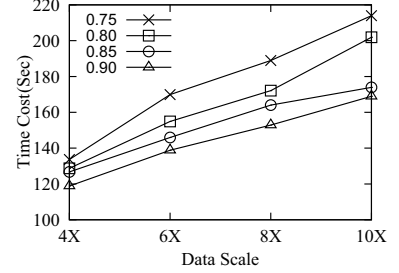
Effects of horizontal partition. To further speedup the performance, we proposed horizontal partitioning in Section V-A. To evaluate the effect of horizontal partitioning, we compared FS-Join with FS-Join-V (the algorithm without horizontal
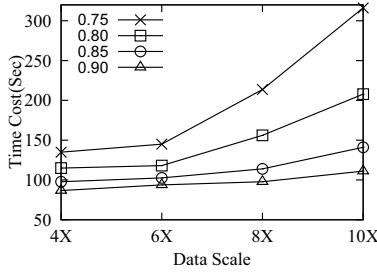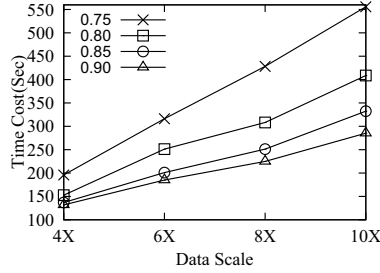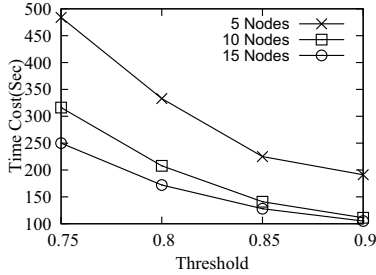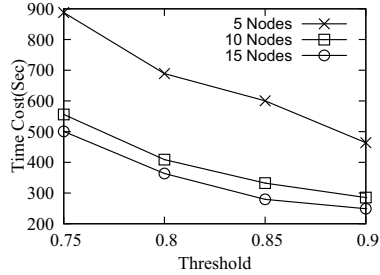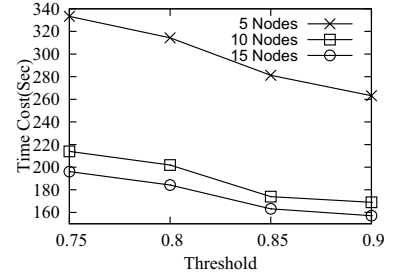
(a) Email DataSet   (b) PubMed DataSet   (c) Wiki DataSet

Fig. 8. Scalability Tests Increasing Dataset Size (X=0.1)

(a) Email DataSet   (b) PubMed DataSet   (c) Wiki DataSet

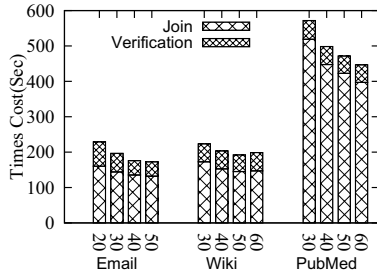Fig. 9. Scalability Tests Increasing The Number Of Cluster Nodes

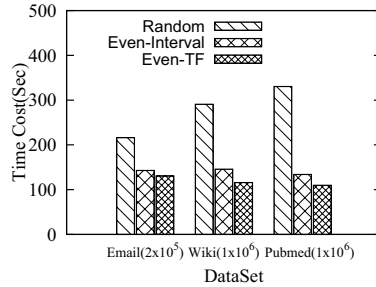

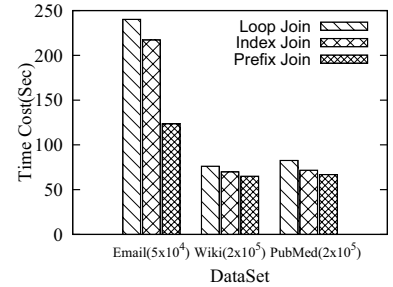Fig. 10. Time Cost of Filtering and Verification.   Fig. 11. Pivots Selection.   Fig. 12. Join Type.

partitioning) on three datasets. In this experiment, the number of vertical partitions is set to 30 for all the datasets; the number of horizontal partitions is set to 10 for the *Email* dataset, 50 for the *Wiki* dataset and 70 for the *PubMed* dataset. The experimental results are shown in Figure 13. FS-Join outperforms FS-Join-V on the three datasets under different thresholds. In FS-Join-V, each dataset is partitioned vertically into 30 disjoint fragments. Each fragment is processed on one node and each node should process 3 fragments. The segments of different strings are produced by different map tasks and grouped into 30 groups. Before shuffling to the reduce nodes, the segments belonging to the same fragment should be sorted and outputted to HDFS. As each fragment is large, the spilling procedure is invoked multiple times to write the map buffer to HDFS during the sort and shuffle phase. Consequently, each reduce node will incur on high time latency to load its fragment from HDFS and to start the reduce task. In FS-Join, each dataset is partitioned into sections using vertical and horizontal partitioning. The segments from different strings are grouped into 300 groups for the *Email* dataset, 1500 groups for the *Wiki* dataset and 2100 groups for the *PubMed* dataset, not including the very small partitions at the intersection of adjacent horizontal paritions. All these groups will be shuffled

to 10 reduce nodes to guarantee workload balancing among different nodes. Unlike FS-Join-V, each group is much smaller in FS-Join. During the shuffle phase, most groups can be sorted without multiple spilling procedures. Each reduce task can load its groups from the output of related map tasks with low time latency. Furthermore, each reduce task can process its groups fast in its local memory space. Therefore, FS-Join significantly outperforms FS-Join-V. We evaluated the effects of the number of horizontal partitions. The experimental results is shown in Figure 10, where the numbers above the dataset names represent the number of horizontal partitions. From Figure 10, we observe that on the three datasets, the overall execution time of FS-Join decreases when the number of horizontal partitions increases. We also find that the time cost of filtering phase is much more than that of the verification phase. This is the result of the powerful filtering methods that have pruned out most of the false positives in the filtering phase.

VII. CONCLUSIONS

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 Par-
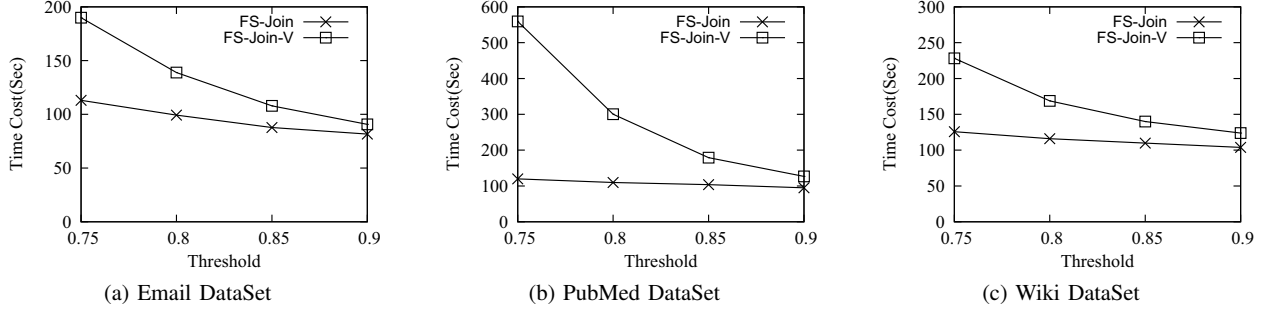
(a) Email DataSet (b) PubMed DataSet (c) Wiki DataSet

Fig. 13. Effects Of Horizontal Partitioning.

titioning). Moreover, we proposed highly effective optimization techniques and introduced new filtering approaches. We thoroughly analyze the performance of FS-Join theoretically and experimentally. The results show that FS-Join significantly outperforms the state-of-the-art techniques and that it has excellent scalability and performance properties. In the future, we plan to extend our methods to approximate approaches and other Big Data platforms, like Spark.

REFERENCES

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

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

[3] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, pages 61–72. IEEE, 2006.

[4] D. Deng, G. Li, S. Hao, J. Wang, and J. Feng. Massjoin: A mapreduce-based method for scalable string similarity joins. In *ICDE*, pages 340–351. IEEE, 2014.

[5] L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, pages 491–500. ACM, 2001.

[6] M. Hernández and S. Stolfo. The merge/purge problem for large databases. In *SIGMOD*, pages 127–138. ACM, 1995.

[7] M. Hernández and S. Stolfo. Real-world data is dirty: Data cleansing and the merge/purge problem. *Data mining and knowledge discovery*, 2(1):9–37, 1998.

[8] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *ICDE*, pages 618–629. IEEE, 2012.

[9] C. Li, J. Lu, and Y. Lu. Efficient merging and filtering algorithms for approximate string searches. In *ICDE*, pages 257–266. IEEE, 2008.

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

[11] G. Li, J. He, D. Deng, and J. Li. Efficient similarity join and search on multi-attribute data. In *SIGMOD*, pages 1137–1151. ACM, 2015.

[12] J. Lu, C. Lin, W. Wang, C. Li, and H. Wang. String similarity measures and joins with synonyms. In *SIGMOD*, pages 373–384. ACM, 2013.

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

[14] A. Monge and C. Elkan. The field matching problem: Algorithms and applications. In *SIGKDD*, pages 267–270, 1996.

[15] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, and A. K. Tung. Efficient and scalable processing of string similarity join. *TKDE*, 25(10):2217–2230, 2013.

[16] G. Salton and M. McGill. *Introduction to modern information retrieval*. McGraw-Hill, Inc., 1986.

[17] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *SIGMOD*, pages 743–754. ACM, 2004.

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

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

[20] W. Winkler. The state of record linkage and current research problems. In *Statistical Research Division*, 1999.

[21] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

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

APPENDIX A
PROOF OF LEMMA 5

Recall that FS-Join uses one MapReduce job to generate candidates and one MapReduce job to verify the candidates. In the first MapReduce job, the mapper splits strings into segments and outputs them with segment information. The cost of mapper is $\sum_{i=1}^{M} |s_i| \times C_m$. Since there is no data duplicates are produced, the shuffle cost is $\sum_{i=1}^{M} |s_i| \times C_s$. Each reducer performs joins on its fragment and the loop join based implementation will perform joins on each pairs of segments. Let p be the probability of the token's occurrence. The probability that one segment belongs to a fragment is $\frac{1}{N} \times p$. The expected number of segments in one fragment is $\frac{1}{N} \times M \times P$. So, the cost of the loop join based implementation for one reducer is $(\frac{1}{N} \times M \times P)^2 \times \frac{1}{M} \sum_{i=1}^{M} |s_i| \times C_r$. The total cost of all reducers is $N \times (\frac{1}{N} \times M \times P)^2 \times \frac{1}{M} \sum_{i=1}^{M} |s_i| \times C_r$. The output cost of one reducer is $(\frac{1}{N} \times M \times P)^2 \times \alpha \times C_o$. The cost of the first job is $\sum_{i=1}^{M} |s_i| \times C_m + \sum_{i=1}^{M} |s_i| \times C_s + N \times (\frac{1}{N} \times M \times P)^2 \times \frac{1}{M} \sum_{i=1}^{M} |s_i| \times C_r + N \times (\frac{1}{N} \times M \times P)^2 \times \alpha \times C_o$. In the second MapReduce job, the mapper reads the input and outputs the candidates pairs, the cost of the mapper is $N \times (\frac{1}{N} \times M \times P)^2 \times \alpha \times C_m$. The shuffle cost is $N \times (\frac{1}{N} \times M \times P)^2 \times \alpha \times C_s$. The cost of the reducer is $N \times (\frac{1}{N} \times M \times P)^2 \times \alpha \times C_r$. The output cost is $(\frac{1}{N} \times M \times P)^2 \times \alpha \times \beta \times C_o$. The total cost of the second job is $N \times (\frac{1}{N} \times M \times P)^2 \times \alpha(C_m + C_s + C_r) + (\frac{1}{N} \times M \times P)^2 \times \alpha \times \beta \times C_o$.

Therefore, the total cost of FS-Join is $\sum_{i=1}^{M} |s_i| \times C_m + \sum_{i=1}^{M} |s_i| \times C_s + (\frac{1}{N} \times M \times P)^2 \times (\frac{N}{M} \sum_{i=1}^{M} |s_i| \times C_r + N \times \alpha(C_m + C_s + C_r + C_o) + \alpha \times \beta \times C_o)$.