Document Similarity Self-Join with MapReduce

Ranieri Baraglia
ISTI-CNR
Pisa, Italy
ranieri.baraglia@isti.cnr.it

Gianmarco De Francisci Morales
IMT Institute for Advanced Studies
Lucca, Italy
gianmarco.dfmorales@imtlucca.it

Claudio Lucchese
ISTI-CNR
Pisa, Italy
claudio.lucchese@isti.cnr.it

***Abstract*—Given a collection of objects, the Similarity Self-Join problem requires to discover all those pairs of objects whose similarity is above a user defined threshold. In this paper we focus on document collections, which are characterized by a sparseness that allows effective pruning strategies.**

Our contribution is a new parallel algorithm within the MapReduce framework. This work borrows from the state of the art in serial algorithms for similarity join and MapReduce-based techniques for set-similarity join. The proposed algorithm shows that it is possible to leverage a distributed file system to support communication patterns that do not naturally fit the MapReduce framework. Scalability is achieved by introducing a partitioning strategy able to overcome memory bottlenecks.

Experimental evidence on real world data shows that our algorithm outperforms the state of the art by a factor 4.5.

***Keywords*-Similarity Self-Join, MapReduce, Web Information Retrieval**

I. Introduction

In this paper, we address the *similarity self-join* problem: given a collection of objects, we aim to discover every pair of objects with high similarity, according to some similarity function. The task of discovering similar objects within a given collection is common to many real world applications and machine learning problems.

Item-based and user-based *recommendation algorithms* require computing, respectively, pair-wise similarity among users or items [1]. *Near duplicate detection* is commonly performed as a preprocessing step before building a document index [2]. It may be used to detect redundant documents, which can therefore be removed, or it may be a hint for spam websites exploiting content repurposing strategies. Finding similar objects is also a basic step for *data cleaning* tasks. When dealing with real world data, we wish to normalize different but similar representations of the same entity, e.g. different names or addresses, caused by mistakes or formatting conventions.

We specifically focus on document collections. Documents are a particular kind of data that exhibits a significant sparseness: only a small subset of the whole lexicon occurs in any given document. This sparsity allows exploiting pruning strategies that reduce the potentially quadratic number of candidate pairs to evaluate. In addition, the size of the collection at hand poses new interesting challenges. This is particularly relevant for Web-related collections, where the number of documents involved is measured in billions. For this setting, distributed solutions become mandatory.

The main contributions of this work are:

- we present SSJ-2R, an efficient distributed algorithm based on the MapReduce framework for computing the similarity self-join on a collection of documents;
- we provide detailed analytical and empirical analysis of the proposed and related algorithms across the various phases of the MapReduce framework;
- we show that SSJ-2R performs nicely on a sample of the WT10g TREC Web Corpus.

II. Problem formulation and preliminaries

We address the similarity self-join problem applied to a collection $\mathcal{D}$ of documents. Let $\mathcal{L}$ be the lexicon of the collection. Each document d is represented as a $|\mathcal{L}|$-dimensional vector, where $d[x]$ denotes the number of occurrences of the x-th term in the document d. We adopt cosine distance to measure the similarity between two documents. Without loss of generality, we assume that documents are normalized, and therefore the cosine similarity is equivalent to the dot product between document vectors.

Definition 1 (Sim-SJ Problem): Given a collection $\mathcal{D} = \{d_1, \dots, d_N\}$ of normalized document-vectors, and a minimum similarity threshold $\overline{\sigma}$, the *Similarity Self-Join (Sim-SJ)* problem requires to discover all those document pairs $d_i, d_j \in \mathcal{D}$ with high similarity:

$$\sigma(d_i, d_j) = \sum_{0 \le t < |\mathcal{L}|} d_i[t] \cdot d_j[t] \ge \overline{\sigma}$$

Borrowing from [3, 4], we classify state-of-the-art Sim-SJ algorithms by their solution to the following sub-problems:

1) **Signature scheme:** a compact representation of each document;
2) **Candidate generation:** identifying potentially similar document pairs given their signature;
3) **Verification:** computing document similarity;
4) **Indexing:** data structures for speeding up candidate generation and verification.

Broder et al. [5] adopt a simple signature scheme, that we name *Full-Filtering*. The signature of a document is given by the terms occurring in that document. These signatures are stored in an inverted index, which is used to find pairs of documents sharing at least one term.

Prefix-Filtering [6] is an extension of the above idea. Let $\hat{d}$ be an artificial document such that $\hat{d}[t] = \max_{d \in \mathcal{D}} d[t]$.

Given a document d, let $b(d)$ be the largest integer such that $\sum_{0 \leq t < b(d)} d[t] \cdot \hat{d}[t] < \overline{\sigma}$. We define the signature of a document d as the set of terms occurring in the document greater or equal to $b(d)$, $S(d) = \{b(d) \leq t < |\mathcal{L}| \mid d[t] \neq 0\}$. It is easy to show that if the signatures of two documents d_i, d_j have empty intersection, than the two documents have similarity below the threshold, since:

$$\begin{aligned} S(d_i) \cap S(d_j) &= \emptyset \quad \Rightarrow \\ \sigma(d_i, d_j) &= \sum_{0 \leq t < b(d_i)} d_i[t] \cdot d_j[t] \\ &\leq \sum_{0 \leq t < b(d_i)} d_i[t] \cdot \hat{d}[t] \quad < \overline{\sigma} \end{aligned}$$

Only part of each document (i.e. its signature) is indexed and used to find candidate document pairs. Eventually, the full document must be retrieved to compute the actual similarity.

Bayardo et al. [7] adopt an incremental indexing and matching approach. Each document is matched against the current index to find potential candidates. Similarity scores are computed by retrieving the candidate documents from the collection. Then, a signature is extracted from the current document and added to the index. This approach was shown to outperform alternative techniques such as LSH [8], PartEnum [4] and ProbeCount-Sort [9], which are therefore not considered here. Unfortunately, an incremental approach with a global shared index is not feasible on a large parallel system.

Finally, some additional pruning techniques are presented by Xiao et al. [10]. However, these techniques are based on positional information, are tailored for set-similarity joins, and cannot be directly applied to document similarity.

III. Related work

MapReduce [11] is a distributed computing paradigm inspired by concepts of functional languages. More specifically, MapReduce is based on two higher order functions: Map and Reduce. The Map function applies a User Defined Function (UDF) to each key-value pair in the input, which is treated as a list of independent records. The result is a second list of intermediate key-value pairs. This list is sorted and grouped by key, and used as input to the Reduce function. The Reduce function applies a second UDF to every intermediate key with all its associated values to produce the final result. The signatures of the functions that compose the phases of a MapReduce computation are as follows:

$$\begin{aligned} \textsf{Map}: \quad & \langle k_1, v_1 \rangle \quad \rightarrow \quad [\langle k_2, v_2 \rangle] \\ \textsf{Reduce}: \quad & \langle k_2, [v_2] \rangle \quad \rightarrow \quad [\langle k_3, v_3 \rangle] \end{aligned}$$

MapReduce assumes a distributed file system from which the Map instances retrieve their input data. The framework takes care of moving, grouping and sorting the intermediate data produced by the various mappers to the corresponding reducers. This phase is called *shuffle*, and strongly affects the efficiency of any MapReduce based implementation. In our work we used Hadoop (http://hadoop.apache.org), an open-source implementation of MapReduce.

A. MapReduce Full-Filtering (Elsayed et al.)

A MapReduce implementation of the *Full-Filtering* approach was presented by Elsayed et al. [12]. The authors propose an algorithm for computing the similarity of every document pair, which we extended with a simple post-filtering. The proposed algorithm runs two consecutive MapReduce jobs, the first to build an inverted index and the second to compute similarities, as follows:

Indexing: given a document d_i, for each term, the mapper emits the term as the key, and a tuple consisting of document ID and weight as the value, i.e. $\langle i, d_i[t] \rangle$. The shuffle phase of MapReduce, groups these tuples by term, and delivers these inverted lists to the reducers, that write them to disk.

$$\begin{aligned} \textsf{Map}: \quad & \langle i, d_i \rangle \; \rightarrow \; [\langle t, \langle i, d_i[t] \rangle \rangle \mid d_i[t] > 0] \\ \textsf{Reduce}: \quad & \langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle \rightarrow \\ & \qquad [\langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle] \end{aligned}$$

Similarity: given the inverted list of term t, the mapper produces the contribution $w = d_i[t] \cdot d_j[t]$ to every pair of documents where this occurs. This value is associated with a key consisting in the pair of document IDs $\langle i, j \rangle$. For any document pair, the shuffle phase will provide a reducer with the contributions list W from the various terms, which simply need to be summed up.

$$\begin{aligned} \textsf{Map}: \quad & \langle t, [\langle i, d_i[t] \rangle, \langle j, d_j[t] \rangle, \ldots] \rangle \rightarrow \\ & \qquad [\langle \langle i, j \rangle, w = d_i[t] \cdot d_j[t] \rangle \mid i < j] \\ \textsf{Reduce}: \quad & \langle \langle i, j \rangle, W = [w_0, \ldots, w_{|W|}] \rangle \rightarrow \\ & \qquad [\langle \langle i, j \rangle, \sigma(d_i, d_j) = \textstyle\sum_{w \in W} w \rangle] \end{aligned}$$

Full-Filtering is exploited implicitly: the similarity of two documents not sharing any term is never evaluated, because none of their terms may occur in the same inverted list.

B. MapReduce Prefix-Filtering (Vernica et al.)

Vernica et al. [13] present a MapReduce algorithm based on Prefix-Filtering that uses only one MapReduce job. For each term in the signature $S(d_i)$ of a document d_i, as defined by the Prefix-Filtering technique, the Map function outputs the term itself as the key and the whole document as the value. The shuffle phase delivers to each reducer a small sub-collection of documents having one term in common in their signatures. This can be thought as the creation of an inverted index of the signatures, where each posting is the document itself, rather than a simple ID. Finally, the reducer finds the similar pairs in the received set of candidates, e.g. by exploiting techniques like [10]. The Map and Reduce functions are as follows:

$$\begin{aligned} \textsf{Map}: \quad & \langle i, d_i \rangle \; \rightarrow \; [\langle t, d_i \rangle \mid t \in S(d_i)] \\ \textsf{Reduce}: \quad & \langle t, D = [d_0, \ldots, d_{|D|}] \rangle \rightarrow \\ & \quad [\langle i, j \rangle, \sigma(d_i, d_j) \mid \\ & \qquad d_i, d_j \in D \wedge \sigma(d_i, d_j) \geq \overline{\sigma}] \end{aligned}$$

IV. Algorithms

In the following, we propose two algorithms based on *Prefix-Filtering* that overcome the weak points of the algorithms presented above. The first algorithm, named SSJ-2, performs a Prefix-Filtering in two MapReduce jobs. The second algorithm, named SSJ-2R, uses a *remainder file* to broadcast data likely to be used by every reducer. It also exploits a partitioning of the search space to avoid memory constraints.

A. Double-Pass MapReduce Prefix-Filtering (SSJ-2)

SSJ-2 is an extension of Elsayed et al. [12]. It shortens the inverted lists by exploiting *Prefix-Filtering*. As shown in Fig. 1, the effect of *Prefix-Filtering* is to reduce the portion of document indexed. The terms occurring in d_i up to term $b(d_i)$, or b_i for short, are not indexed. By sorting terms in decreasing order of frequency the most frequent terms are discarded, thus reducing the length of the longest inverted lists.

To improve load balancing we employ a simple bucketing technique. During the indexing phase, we randomly hash the inverted lists to different buckets (files). This spreads the longest lists uniformly among the buckets. Each bucket is then consumed by a different mapper in the similarity phase.

On the flip-side of the coin, the reducers do not have enough information to compute the similarity between d_i and d_j. They receive only the contribution from terms $t \geq b_j$, assuming $b_i \leq b_j$. For this reason, the reducers need two additional remote I/O operations to retrieve the two documents from the underlying distributed file system.

SSJ-2 improves over both Elsayed et al. and Vernica et al. in terms of the amount of intermediate data produced. Thanks to Prefix-Filtering, inverted lists are shorter than Elsayed et al., and therefore the number of candidate pairs generated decreases quadratically. In Vernica et al. each document is *always* sent to a number of reduce instances that depends on the size of its signature, even if no other signature shares a term with it. In SSJ-2 a couple of documents is sent to the reducers only if they share at least one term in their signatures. Also, Vernica et al. computes the similarity of a two docments in as many Reduce instances as the number of terms in the intersection of their signatures. Instead, SSJ-2 computes the similarity between any two given documents just once. Therefore, SSJ-2 has no computational overhead, and significantly reduces communication costs.

B. Double-Pass MapReduce Prefix-Filtering with Remainder File (SSJ-2R)

For any given couple of documents $\langle d_i, d_j\rangle$, the reducers of SSJ-2 receive only partial information. Therefore, they need to retrieve the full documents to compute their similarity. Thus, each reducer performs two remote random accesses to the input collection. Since a node runs multiple reducers over time, this can lead to broadcasting the full collection to every node (multiple times). We thus propose an improved algorithm named SSJ-2R, that avoids remote random accesses and makes better use of the non-indexed portion of the input.

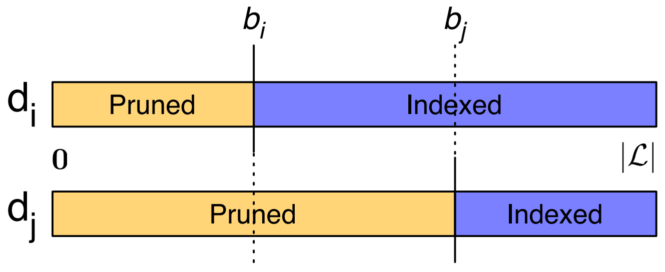

Figure 1. Pruned Document Pair: the left part (orange/light) has been pruned, the right part (blue/dark) has been indexed.

We can make a few interesting observations. First, some terms are very common, and therefore used to compute the similarity of most document pairs. For those terms, it would be more effective to broadcast their contribution once rather than pulling them every time. Indeed, such piece of information is the one pruned via Prefix-Filtering during the indexing phase. We thus propose to store the portion of each document pruned during indexing in a *remainder file* $\mathcal{D}_R$. This can be later retrieved by every node from the distributed file system.

Second, the *remainder file* $\mathcal{D}_R$ does not contain all the information needed. Consider Fig. 1: each reducer receives a contribution $w = d_i[t] \cdot d_j[t]$ for every $t \geq b_j$. But, subtly, no information is available for those terms of document d_i such that $b_i \leq t \leq b_j$. On the one hand, these terms are not in the remainder file because they have been indexed. On the other hand, the corresponding inverted lists contain the weight of the terms in d_i but not of those occurring in d_j, and therefore, the contribution $w = d_i[t] \cdot d_j[t]$ cannot be produced.

We thus propose to deliver the document with the missing information to the reducer, by shuffling it together with candidate pairs. Given two documents d_i and d_j as shown in Fig. 1, if $b_i \leq b_j$ we call i the *Least Pruned Document* (LPD_{ij}) and j the *Most Pruned Document* (MPD_{ij}). Our goal is to group at the reducer every document d_i together with the contributions w for every document pair $\langle d_i, d_j\rangle$ where i is the LPD_{ij}. We achieve this by properly defining the keys produced by the mappers, and their comparison operator.

First, we slightly modify the similarity Map function so that, for every document pair in a given inverted list, it produces as keys a couple of document IDs where the first is always the LPD, and as values the MPD and the usual contribution w:

$$\begin{aligned}\textsf{Map}: \quad & [\langle t, [\langle i, d_i[t]\rangle, \langle j, d_j[t]\rangle, \ldots]\rangle] \rightarrow \\ & [\langle\langle LPD_{ij}, MPD_{ij}\rangle, \\ & \quad \langle MPD_{ij}, w = d_i[t] \cdot d_j[t]\rangle\rangle]\end{aligned}$$

Second, we take advantage of the possibility of running independent Map functions whose output is then shuffled together. We define an additional Map function that takes the input collection and outputs its documents:

$$\textsf{Map}: \quad \langle i, d_i\rangle \;\rightarrow\; [\langle\langle i, \star\rangle, d_i\rangle]$$

where $\star$ is the minimal document ID, i.e. $\star < i \;\forall i \mid d_i \in \mathcal{D}$.

The effect of sorting increasingly the keys produced by the two Map functions is to get a pair $\langle i, \star \rangle$ with the corresponding document d_i, followed by every document pair for which i is the Least Pruned Document.

Finally, we need to override the so-called *group operator*. This is used by the MapReduce framework to tell whether two keys are equivalent, in which case the corresponding values must be delivered to the same reduce function. We based our group operator only on the LPD. Two keys $\langle i, j \rangle$ and $\langle i', j' \rangle$ are equivalent *iff* $i = i'$.

The input of the Reduce function is thus a key $\langle i, \star \rangle$, followed by a sequence of values:

$$[d_i, \langle j, w_0 \rangle, \langle j, w_1 \rangle, \ldots, \langle k, w_0 \rangle, \langle k, w_1 \rangle, \ldots]$$

The first value is the LPD document d_i, i.e. the weights of each term it contains. Thanks to key sorting, this is followed by a stripe of contributions associated to the same document j being the MPD_{ij}. We sum up these contributions to get a partial result. To this, we add the dot product between d_i and the pruned portion of d_j. The former is an input to the Reduce function, the latter can be retrieved from the *remainder file*. The function consumes the following stripes in the same way.

Differently from SSJ-2, no remote access to the input collection is performed. For each document pair, one is delivered fully through the MapReduce framework, and the other is retrieved from the remainder file $\mathcal{D}_R$. The advantage in terms of communication costs is given by the remainder file. Indeed, $\mathcal{D}_R$ is much smaller than the input collection. We show experimentally that its size is about 1/10th of the input. Therefore, $\mathcal{D}_R$ can be efficiently broadcast to every node through the distributed file system. Each reducer can load it and store it in main memory. Therefore, the similarity computation can be performed without any further disk access.

C. Partitioning

We mentioned that the remainder file $\mathcal{D}_R$ is much smaller than the original collection, so that it can be loaded in main memory by the reducer and used to retrieve the unindexed portion of the documents. However, the size of $\mathcal{D}_R$ will grow when increasing the collection size, therefore hindering the scalability of the algorithm.

To overcome this issue, we introduce a partitioning of the remainder file. Given a user defined parameter K, the remainder file $\mathcal{D}_R$ is split into K nearly equi-sized chunks, such that the unindexed portion of document d_i falls into the $\lfloor i/K \rfloor$-th chunk. We overrode Hadoop's partitioning function which maps a key emitted by the mapper to a reducer instance. Mapping is done on the basis of the MPD. Every key $\langle i, j \rangle$ is mapped to a reducer in the $\lfloor j/K \rfloor$-th group.

The special key $\langle i, \star \rangle$ and its associated value d_i are replicated at most K times, so that the full content of the LPD can be delivered at every reducer. Each reducer in a group thus needs to recover and load in memory only one chunk of $\mathcal{D}_R$, whose size can be set arbitrarily small by varying K. This is achieved at the cost of replicating the documents delivered through the MapReduce framework.

Table I
SYMBOLS AND QUANTITIES

Symbol	Description
$\overline{d}$ $\widehat{d}$	Avg. and Max. document length
$\overline{s}$ $\widehat{s}$	Avg. and Max. signature length
$\overline{p}$ $\widehat{p}$	Avg. and Max. inverted list length with Prefix-Filtering
$\overline{l}$	Avg. inverted list length without pruning
r	Cost of a remote access to a document
R	Cost of retrieving the remainder file $\mathcal{D}_R$

Table II
COMPLEXITY ANALYSIS

Algorithm	Map	Reduce
Elsayed et al.	$\widehat{l}^2$	$\widehat{d}$
Vernica et al.	$\widehat{s} \cdot \widehat{d}$	$\widehat{p}^2 \cdot \overline{d}$
SSJ-2	$\widehat{p}^2$	$\widehat{d} + r$
SSJ-2R	$\widehat{p}^2$	$\widehat{d}$

V. COMPLEXITY ANALYSIS

There exist a few proposals for modeling MapReduce algorithms [14, 15]. Indeed, estimating the cost of a MapReduce algorithm is very difficult, because of the inherent parallelism, the cost of the shuffling phase, the overlaps among computations and communications managed implicitly by the framework, and so on.

We prefer to model separately the two main steps of a MapReduce job, i.e. the Map and Reduce functions, to provide a more detailed comparison. In particular, we take into consideration the cost associated to the function instance with the largest input. In this way, we roughly estimate the maximum possible degree of parallelism.

We refer to Table I for the symbols used in this section. The quantity $\overline{l}$ can also be seen as the average frequency of the terms in the lexicon $\mathcal{L}$. Similarly, $\overline{p}$ can also be seen as the number of signatures containing any given term. Clearly, it holds that $\overline{p} \leq \overline{l}$.

The running time of Elsayed et al. is dominated by the similarity phase. Its Map function generates every possible pair for each inverted list, with a cost of $O(\widehat{l}^2)$. Thus, it generates a large number of candidate pairs with a similarity below the threshold. Furthermore, the presence of long inverted lists introduces *stragglers*. Since reducers may start only after every mapper has completed, mappers processing the longest lists keep resources idle waiting. The Reduce function just sums up the contributions of the various terms, with a cost $O(\widehat{d})$.

Vernica et al. runs only one MapReduce job. The Map replicates each document s times, where s is its signature length, with a cost of $O(\widehat{s} \cdot \widehat{d})$. The Reduce evaluates the similarity of every pair of documents in its input, with cost $O(\widehat{p}^2 \cdot \overline{d})$. Due to Prefix-Filtering, the maximum reducer input size is $\widehat{p}$, which is smaller than $\widehat{l}$. Document pairs sharing multiple terms in their signatures are evaluated multiple times at different reducers. These duplicates create a computational overhead, and require post-processing to be removed.

By comparing the two algorithms, we notice the superiority of Vernica et al. in the first phase. The Map function generates

Table III
SAMPLES FROM THE TREC WT10G COLLECTION

	D17K	D30K	D63K
# documents	17,024	30,683	63,126
# terms	183,467	297,227	580,915
# all pairs	289,816,576	941,446,489	3,984,891,876
# similar pairs	94,220	138,816	189,969

a number of replicas of each document, rather then producing a quadratic number of partial contributions. Conversely, the Reduce cost for Vernica et al. is quadratic in the size of the pruned lists $\widehat{p}$.

Our proposed algorithms SSJ-2 and SSJ-2R have a structure similar to Elsayed et al.. The Map cost $O(\widehat{p}^2)$ depends on the pruned inverted lists. For SSJ-2R, we are disregarding the cost of creating replicas of the input in the independent Map function, since it is much cheaper than processing the largest inverted list. The Reduce cost is different. In addition to the contributions coming from the pruned inverted index, SSJ-2 needs to access documents remotely with a total cost of $O(\widehat{d}+r)$. While SSJ-2R has a cost $O(\widehat{d})$ since the unindexed portion of the document is already loaded in memory at the local node, delivered through the distributed file system.

Thanks to Prefix-Filtering, the Map cost of our proposed algorithm smaller than Elsayed et al., yet larger than Vernica et al.. The Reduce function of SSJ-2 needs to remotely recover the documents being processed to retrieve their unindexed portions. This dramatically increases its cost beyond Elsayed et al., but can hardly be compared with Vernica et al.. SSJ-2R has about the same cost as Elsayed et al., assuming that the remainder file is already loaded in memory.

VI. EXPERIMENTAL RESULTS

We ran the experiments on a 5-node cluster. Each node was equipped with two Xeon E5520 CPUs at 2.27GHz, for a total of 40 cores, a 2TiB disk, 8GiB of RAM, and Gigabit Ethernet. One of the nodes was used to run Hadoop's master daemons, while the rest were configured as slaves.

For each algorithm, we wrote an appropriate *combiner* to reduce the shuffle size (a combiner is a reduce-like function that runs inside the mapper to aggregate partial results).

We used different subsets of the TREC WT10G Web corpus. The dataset has 1,692,096 english language documents. The size of the entire uncompressed collection is around 10GiB. In Table III we describe samples of the collections we used, ranging from ~17K to ~63K documents. We performed a preprocessing step to prepare the data. We parsed the dataset, removed stop-words, performed stemming and normalization of the input, and we extracted the lexicon. We also sorted the features inside each document in decreasing order of term frequency, as required by the pruning strategy.

A. Running Time

We compared the four algorithms described in the previous sections. The algorithms proposed in this paper, SSJ-2 and SSJ-2R, and the baselines, *Elsayed et al.* and *Vernica et al.*. Elsayed et al. was adapted by adding a filtering in the last

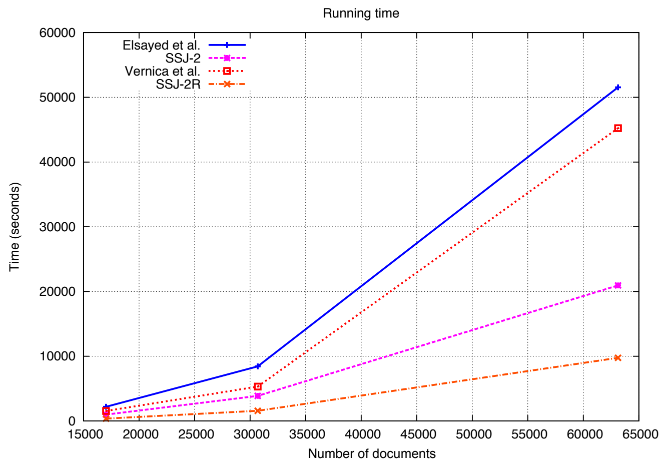

Figure 2. Running times of the algorithms.

phase. Vernica et al. would require an additional step to remove duplicate pairs. We did not implement this final step, so our analysis does not take into account its overhead.

For SSJ-2R, we used a partitioning factor $K = 4$. This was done for the sake of completeness, because the mappers were not even close to fill the available memory. We set the number of mappers to 56 and the number of reducers to 28, so that the mappers finish in two waves and all the reducers can start copying and sorting the partial results while the second wave of mappers is still running. For all the experiments, we set the similarity threshold $\overline{\sigma}$ to 0.9.

Figure 2 shows the running times of the four algorithms. All of them have a quadratic step, so doubling the size of the input roughly multiplies by 4 the running time. Unexpectedly, Vernica et al. does not improve significantly over Elsayed et al.. This shows that Prefix-Filtering is not fully exploited.

Both our proposed algorithms outperform the baselines. In particular, SSJ-2 is more than twice faster than Vernica et al., and SSJ-2R is ~4.5 times faster. Notice that SSJ-2R is twice faster than SSJ-2. The advantage given by the exploitation of the remainder file is therefore significant.

B. Map phase

It is clear from Table IV that Vernica et al. is the fastest. The mapper only replicates input documents. The time required by the other algorithms grows quadratically as expected. Our proposed algorithms are twice faster than Elsayed et al. for two main reasons.

First, Prefix-Filtering reduces the maximum inverted list length. Even a small improvement here brings a quadratic gain. Since Prefix-Filtering removes the most frequent terms, the longest inverted lists are shortened or even removed. For D17K, the length of the longest inverted list is reduced from 6600 to 1729 (not shown due to space constraints). The time to process such inverted list decreases dramatically, and also the amount of intermediate data generated is reduced. Interestingly, Prefix-Filtering does not affect significantly the index size. Only 10% of the index is pruned as reported in

Table IV
STATISTICS ON VARYING INPUT SIZE.

dataset	D17K				D30K				D63K			
algorithm	Elsayed	SSJ-2	Verinca	SSJ-2R	Elsayed	SSJ-2	Verinca	SSJ-2R	Elsayed	SSJ-2	Verinca	SSJ-2R
# evaluated pairs (M)	109	**65**	401	**65**	346	**224**	1,586	**224**	1,519	**1,035**	6,871	**1,035**
index size (MB)	46	41		41	92	82		82	189	170		170
remainder size (MB)				4.7				8.2				15.6
shuffle size (GB)	3.3	**2.1**	3.7	2.6	11.3	**8.1**	8.3	10.5	49.2	35.8	**20.7**	49.7
running time (s)	2,187	971	1,543	**364**	8,430	3,862	5,313	**1,571**	51,540	20,944	28,534	**9,745**
avg. map time (s)	276	148	**42**	122	1,230	635	**112**	560	7,013	3,908	**338**	3,704
std. map time (%)	127.50	37.92	68.10	**26.50**	132.08	32.06	68.10	**23.41**	136.39	24.32	59.61	**20.26**
avg. reduce time (s)	**49**	575	892	**49**	**82**	2,183	3,847	155	1,136	11,328	16,849	**846**
std. reduce time (%)	16.68	**3.58**	22.20	15.50	15.91	**5.81**	13.94	15.50	11.84	**2.76**	9.36	12.11

Table IV, meaning that its effects are limited to shortening the longest inverted lists.

Second, we employ a bucketing technique to improve load balancing. Since mappers process the inverted lists in chunks, the longest inverted lists might end up in the same chunk and be processed by a single mapper. In SSJ-2 and SSJ-2R inverted lists are randomly hashed into different buckets, so that each mapper is likely to process lists of diverse length. As a result, the running time of the slowest mapper is almost halved. The reduction in standard deviation of map completion time w.r.t. Elsayed et al. is evident from Table IV.

C. Reduce phase

Vernica et al. is the most expensive, as expected. While the other algorithms take advantage of the MapReduce infrastructure to perform partial similarity computations, Vernica et al. just delivers a collection of potentially similar documents. The similarity is entirely computed at the reducer, with a cost quadratic in the number of documents received. In addition, the same document pair is evaluated several times at different reducers, generating a large overhead.

SSJ-2 is the second most expensive. This is due to the remote retrieval of documents from the distributed file system.

Both SSJ-2R and Elsayed et al. are embarrassingly faster than the competitors. Very little computation is done by the reducer: the contributions from the various terms are simply summed up. The use of the remainder file significantly speeds up SSJ-2R compared to SSJ-2. The remainder file is quickly retrieved from the distributed file system and no random remote access is performed.

The performance of the reducer depends heavily on the number of candidate pairs evaluated. As shown in Table IV, both SSJ-2 and SSJ-2R evaluate about 33% less candidate than Elsayed et al.. Notice that Vernica et al. evaluates a number of pairs which is about 3 times larger than $N(N-1)/2$, i.e. the size of the entire search space.

VII. CONCLUSIONS

We presented SSJ-2R, a MapReduce based algorithm for the Sim-SJ problem that is 4.5x faster than the state of the art. Future work will test the algorithm on larger collections, specifically examining the impact on the shuffle phase.

ACKNOWLEDGEMENTS

This work was supported by the EU-FP7-215483 (S-Cube), the EU-FP7-250527 (Assets) and the POR-FESR 2007-2013 (VISITO Tuscany) projects.

REFERENCES

[1] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems," *IEEE TKDE*, 2005.

[2] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *Proc. WWW*, 2007.

[3] C. Xiao, W. Wang, X. Lin, and H. Shang, "Top-K Set Similarity Joins," in *Proc. ICDE*, 2009.

[4] A. Arasu, V. Ganti, and R. Kaushik, "Efficient exact set-similarity joins," in *Proc. VLDB*, 2006.

[5] A. Broder, S. Glassman, M. Manasse, and G. Zweig, "Syntactic clustering of the web," *Computer Networks and ISDN Systems*, vol. 29, no. 8-13, 1997.

[6] S. Chaudhuri, V. Ganti, and R. Kaushik, "A Primitive Operator for Similarity Joins in Data Cleaning," in *Proc. ICDE*, 2006.

[7] R. J. Bayardo, Y. Ma, and R. Srikant, "Scaling Up All Pairs Similarity Search," in *Proc. WWW*, 2007.

[8] A. Gionis, P. Indyk, and R. Motwani, "Similarity Search in High Dimensions via Hashing," in *Proc. VLDB*, 1999.

[9] S. Sarawagi and A. Kirpal, "Efficient set joins on similarity predicates," in *Proc. SIGMOD*, 2004.

[10] C. Xiao, W. Wang, X. Lin, and J. X. Yu, "Efficient similarity joins for near duplicate detection," in *Proc. WWW*, 2008.

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified Data processing on Large Clusters," in *Proc. OSDI*, 2004.

[12] T. Elsayed, J. Lin, and D. W. Oard, "Pairwise document similarity in large collections with MapReduce," in *Proc. HLT*, 2008.

[13] R. Vernica, M. J. Carey, and C. Li, "Efficient parallel set-similarity joins using MapReduce," in *Proc. SIGMOD*, 2010.

[14] H. Karloff, S. Suri, and S. Vassilvitskii, "A Model of Computation for MapReduce," in *Proc. SODA*, 2010.

[15] F. Afrati and J. Ullman, "A New Computation Model for Cluster Computing," Tech.Rep.