Towards a Scalable Set Similarity Join using MapReduce and LSH

Sébastien RIVAULT, Mostafa BAMHA, Sébastien LIMET, and Sophie ROBERT

Université Orléans, INSA Centre Val de Loire, LIFO EA 4022, France
{Sebastien.Rivault,Mostafa.Bamha,Sebastien.Limet,Sophie.Robert}@univ-orleans.fr

Abstract. Set similarity joins consists in computing all pairs of similar sets from two collections of sets. In this paper, we introduce an algorithm called *MRSS-join*, an extended version of our previous MRS-Join algorithm for the treatment of similarity in the trajectories. *MRSS-join* algorithm is based on the MapReduce computation model and a randomized redistribution approach guaranteeing perfect load balancing properties during all similarity join calculation steps while significantly reducing communication costs and the number of sets comparisons with regard to the best known algorithms based on prefix filtering. All our claims are supported by theoretical guarantees and a series of experiments that show the effectiveness of our approach in handling large datasets collections on large-scale systems.

Keywords: Similarity join operations · Local Sensitive Hashing (LSH) · MapReduce model · Data Skew · Hadoop framework.

1 Introduction

The Set Similarity Join (SSJ) consists in finding all the pairs of sets having a distance smaller than a given threshold. SSJ has a large amount of applications including data cleaning [4], entity resolution [7], similar text detection [19, 22], and collaborative filtering [2]. The pruning power of the SSJ is also used to reduce the number of candidate pairs for edit-based string similarity joins [1].

Formally, the R-S join for two collections R and S of sets from the universe $\mathcal{U}$ is $R \bowtie_\lambda S = \{(u, v) \in R \times S \mid Dist(u, v) \leq \lambda\}$ where Dist(u, v) is a distance between u and v, and λ is the threshold parameter. Throughout this paper, a set $u \in R \cup S$ is called a record and the elements of u are called tokens.

We restrict the scope in this paper to one of the most popular distances in the literature, namely the similarity function Jaccard. It is defined as follows: $Jaccard(u, v) = \|u \cap v\|/\|u \cup v\|$ where $\|\cdot\|$ is the cardinality of a set. Note that by design $0 \leq Jaccard(u, v) \leq 1$, and $Jaccard(u, v) = 1$ if and only if u and v are equals. To satisfy the metric space properties, we define the corresponding Jaccard distance as $Dist_J(u, v) = 1 - Jaccard(u, v)$.

Naively, the SSJ computations can be performed by comparing all the data pairs which requires a Cartesian product computation. This may have a disastrous effect on performance and limits their scalability to process large datasets.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

In a BigData context, the amount of data easily exceeds the storage capacity and the processing capability of a single machine. Accordingly, a cluster of machines and scalable distributed algorithms are required.

In the literature, we distinguish two classes of SSJ algorithms according to their result completeness. We refer to algorithms that produce the full similarity join result as exact and others as approximate. Exact SSJ has received much attention and an experimental survey has been conducted on the most recent algorithms [8]. It concludes that none of the evaluated algorithms scale for large datasets processing.

Approximate SSJ is usually based on Locality Sensitive Hashing (LSH) that is a randomized method for generating candidate pairs. In the massively parallel computation model, [11, 12] present an algorithm relying on LSH that achieves guarantees on the result completeness and balanced load of the processing nodes. However, it assumes that the dataset is not skewed. This issue is solved by the *MRS-join* algorithm [17] that guarantees perfect balancing properties among the processing nodes while reducing communication costs. *MRS-join* has been described to perform similarity joins on trajectories using MapReduce [6]. The MapReduce paradigm has received a lot of attention for being a scalable parallel shared-nothing data-processing platform. In order to generalize our framework, we present *MRSS-join* that performs set similarity joins using the Jaccard distance. Furthermore, additional filtering steps and new communication templates are introduced in the self join case.

The *MRSS-join* is compared to VernicaJoin (VJ) [21] which is the state-of-the-art algorithm in terms of runtime and robustness in an exact computation according to [8]. VJ is a multistep algorithm based on prefix and length filtering. By sorting records according to the global frequency of tokens, the w-prefix of a record corresponds to its w first tokens. By determining the prefix sizes depending on the Jaccard similarity threshold and the record length, a candidate pair of records can be pruned if their prefixes have no common token. In the experiments, we compare the performance and the quality of the LSH filtering. The quality is measured in terms of *recall* and *precision*. The *recall* is the fraction of the number of pairs of similar records correctly produced over the exact number of similar records, whereas *precision* corresponds to the fraction of the number of pairs of similar records correctly produced over the number of candidates.

The remaining of this paper is organized as follows: Section 2 presents requirements for the understanding of the *MRSS-join* algorithm. Section 3 describes the *MRSS-join* algorithm. Experimental results presented in Section 4 confirms the efficiency of our approach. We then conclude in Section 5.

2 Preliminaries

This section is organized as follows: Section 2.1 introduces LSH and its associated algorithm to perform set similarity joins using LSH; Section 2.2 explains distributed histograms and randomized communication templates; Section 2.3 presents the *MRS-join* algorithms based on LSH, distributed histograms and

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

randomized communication templates to guarantee a perfect balancing of the load and computation among the processing nodes while reducing communication and computation to only relevant data.

2.1 Locality Sensitive Hashing (LSH)

Indyk and Motwani introduced a randomized hashing framework [9, 13] that solves efficiently the (λ, c)-near neighbor problem even in high dimensional spaces. It is based on a hashing scheme that ensures that close data points are more likely to collide than distant ones. More formally, it is characterized by the following definition.

Let u, v be two records from a common universe $\mathcal{U}$, $Dist$ a distance and λ the threshold distance parameter. Given an approximation factor $c > 1$ and two probabilities p_1 and p_2 such that $0 \leq p_2 < p_1 \leq 1$, $\mathcal{H}$ is a family of LSH functions, if it satisfies the following conditions for any hash function $h \in \mathcal{H}$ chosen uniformly:

1. If $Dist(u, v) \leq \lambda$ then $\mathsf{P}[h(u) = h(v)] \geq p_1$
2. If $Dist(u, v) \geq c * \lambda$ then $\mathsf{P}[h(u) = h(v)] \leq p_2$

Subsequently, we focus on the set similarity join using Jaccard similarity function, however the approach may be generalized using any LSH family that has constant probabilities. MinHash [3] is a family of LSH function that estimates the Jaccard distance. It is defined from a random permutation π of the universe $\mathcal{U}$. For any element e of $\mathcal{U}$, let note $\pi(e)$ be the position of e in the permutation of $\mathcal{U}$. The hashing function h is then defined by $h(u) = \min_{e \in u} \pi(e)$. It is easy to prove that $\mathsf{P}[h(u) = h(v)] = Jaccard(u, v)$.

It is common to concatenate several independent hash functions to improve *precision* and use many independent repetitions to improve *recall*. In a formal way, let $\mathcal{H}_K$ be the LSH family in which a hash function is obtained by concatenating $K \geq 1$ hash functions uniformly and independently selected from $\mathcal{H}$. Accordingly, it holds that $\mathsf{P}_{g^K \in \mathcal{H}_K}[g^K(u) = g^K(v)] = \mathsf{P}_{h \in \mathcal{H}}[h(u) = h(v)]^K = p_1^K$.

K-partition [15, 18] is a variant of MinHash which is efficient to compute several independent hash functions using a single permutation. The idea is to partition the permutation into K bins $B_1, \ldots, B_K$ and $h_i(u) = \min_{e \in u \cap B_i} \pi(e)$. If $u \cap B_i$ is empty, $h_i(u)$ is the set to the first on right (circular) $h_j(u)$ where $u \cap B_j$ is not empty. We refer the reader to [18] for more detailed information. In the rest of the paper MinHash refers to K-partition one-permutation MinHash.

For now, we have left K unspecified, we primarily review Hu et al.'s algorithm [11, 12] that provides the following load bounds in the massively parallel computation model.

Theorem 1 ([11, 12]). *There is a randomized similarity join algorithm that runs in $O(1)$ rounds on P processors that reports each join result with at least a constant probability and the following expected load:*

$$\tilde{O}\left(\sqrt{\frac{\|R \bowtie_\lambda S\|}{P^{1/(1+\rho)}}} + \sqrt{\frac{\|R \bowtie_\sigma S\|}{P}} + \frac{N}{P^{1/(1+\rho)}}\right).$$

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

where $\sigma = c * \lambda$, $\rho = \frac{\log(p_1)}{\log(p_2)} < 1$ *and* N *is the number of inputs.*

The corresponding algorithm is given by the three following steps by setting $K = \left\lceil \log(p_1, 1/P^{\frac{\rho}{1+\rho}}) \right\rceil$:

1. Randomly and independently select Q hash functions $g_1^K, g_2^K, ..., g_Q^K$ from $\mathcal{H}_K$,
2. For each record u, emit a key/value pair $<(i, g_i^K(u)), u>$ for all $i \in 1, ..., Q$,
3. Perform a join by treating $(i, g_i^K(u))$ as the join attribute value, i.e, two records u, v join if $g_i^K(u) = g_i^K(v)$ for all i. For a pair of records (u, v), output them if $Dist_J(u, v) \leq \lambda$.

The number of repetitions is given by $Q = \lceil p_1^{-K} \rceil$ in [11] that gives an optimal output sensitive algorithm. Although result of the similarity join is reported with at least a constant probability, users may want to generate the full similarity join result. By setting $Q = \lceil 3 * p_1^{-K} * \ln(N) \rceil$, the probability to report all join results is $1 - 1/N$ [12]. Thereafter, we prefer to let users specify the desired result expectation by setting $Q = \lceil E * p_1^{-K} \rceil$ with $1 \leq E \leq 3 * \ln(N)$. This algorithm forms the basic building blocks of the similarity join using LSH.

For sake of clarity, we introduced the similarity join using LSH between two collections R and S but in the following, we will consider self joins of a dataset Γ which aim is to compute $\Gamma \bowtie_\lambda \Gamma$. Self joins can be handled as R-S joins from the LSH side.

2.2 Distributed histograms and communication templates

We first review the notion of distributed histograms introduced in [10] to reduce communication costs while guaranteeing perfect balancing properties among all processing nodes. Then, we explain the histogram distribution used in [17]. We implement the memory extensions following the ideas of [17] to guarantee that a distributed histogram always fits in processing nodes' memory. At last, we introduce communication templates for the self join case in the same spirit as the ones introduced for the general case [10, 17] to which we refer the reader for further information.

The histogram of a join is defined as the association between a join attribute value and its frequency. It is used to generate communication templates, allowing to transmit only relevant data fairly during the join phase. More formally, for a dataset Γ where $L(\Gamma)$ denotes the set of its LSH join attribute values, the histogram Hist(Γ) is the list of the pairs $<x, \mathbf{f_x}>$ where $x \in L(\Gamma)$ and $\mathbf{f_x}$ is its frequency.

In order to reduce communication costs to relevant data, only join attribute values which might appear in the join result are present in the histogram. Join attribute values that produce a result imply that their frequencies are greater than or equal to two. Thus, the histogram for the similarity join $\Gamma \bowtie_\lambda \Gamma$ which contains only relevant data is defined as follows.

Definition 1. $Hist(\Gamma \bowtie_\lambda \Gamma) = \{<x, \mathbf{f_x}>, \forall x \in L(\Gamma) \mid \mathbf{f_x} > 1\}$

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

For large datasets, we expect that the corresponding histogram does not fit in memory. Therefore, [17] introduced a way to distribute it for multiple join attribute values. The distribution principle is based on the appearance of join attribute values in different *splits* where a *split* is the portion of data that a **map** function processes. When constructing the histogram, for all join attribute values, the *split* identifiers are also stored. The distribution job requires as many **reduce** tasks as the total number of *splits*. Each join attribute value and its corresponding entry in the histogram are transmitted for each split identifier. At the end, each **reduce** task output corresponds to the distributed histogram required by a *split*.

In order to guarantee that a distributed histogram always fits in memory even in the case of large *splits*, an additional parameter noted $\mathbf{t_{max}}$ is used. This parameter is chosen in such a way that each *Mapper* can buffer a distributed histogram of size at most $Q * \mathbf{t_{max}}$. The method consists in constructing groups of consecutive records in a *split* so that a group is composed of at most $\mathbf{t_{max}}$ records. Handling distributed histogram on groups requires no new algorithm. Essentially, instead of storing only the splits identifiers, the set of pairs (*splitId*, *groupId*) is stored. Later, a pair (*splitId*, *groupId*) is called a *chunk* identifier. A *chunk* is the corresponding portion of data. At the end, each **reduce** task output corresponds to the sorted list of *chunk*'s histogram required by a *split*.

Distributed histograms are then used to reduce communication costs while guaranteeing perfect balancing properties among all processing nodes. It also avoids the effects of data skew in large datasets processing. To this end, communication templates use a parameter denoted by $\mathbf{f_{max}}$. This parameter defines the number of records that a *Reducer* will have to store and process during the similarity join step. Owing to this parameter, the records having a common join attribute value will be divided into several buckets (blocks), so that each bucket can be stored in memory. This makes the *MRS-join* algorithm scalable and insensitive to the effects of data distribution skew.

For a given join attribute value x, communication templates will distribute all buckets according to two cases:

a. $\mathbf{f_x} < \mathbf{f_{max}}$: the records corresponding to the join attribute value are transmitted to a single *Reducer*, without special processing, using a hashing approach.
b. $\mathbf{f_{max}} \leqslant \mathbf{f_x}$: The join attribute value is highly frequent and as illustrated Figure 1. In order to balance the computations, the join attribute value x is divided into several blocks (5 on our example).

In Figure 1, the generated communication templates are arranged in rows and columns. Each cell corresponds to a bucket. Each column corresponds to data transmitted to a **reduce** task. These tasks are identified starting from i_0 that is a random integer which can be stored in the histogram. For the sake of the clarity, it will not be mentioned in the following.

To ensure that the buckets are sorted in the correct order appropriate MapReduce <Key, Value> pairs are used. The keys are composed of the join attribute

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

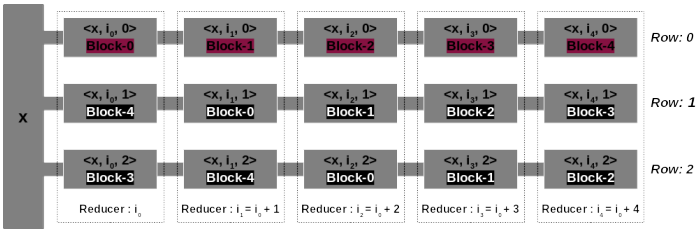

Fig. 1. Communication templates for a highly frequent join attribute value in a self join case. Distributed buckets (red) are stored to compute the similarity join with replicated buckets (black).

value, the column and the row identifier. Pairs are then redirected by the MapReduce **partition** function by means of the **reduce** task identifier. For a *Reducer* task, the join is computed using the following algorithm for a highly frequent join attribute value:

- Store in memory the distributed buckets (i.e., the row identifier is at zero),
- Compute the join within the stored buckets,
- Compute the join with replicated buckets.

Later, we introduce an additional filtering step to this algorithm in the case of a highly frequent join attribute value to reduce the number of comparisons.

2.3 MRS-join

MRS-join [17] is an algorithm built on top of the MapReduce framework that uses LSH, distributed histograms and randomized communication templates to guarantee balanced load and computation among the processing nodes. It is a multi-steps algorithm with time and space guarantees for all the join computation steps.

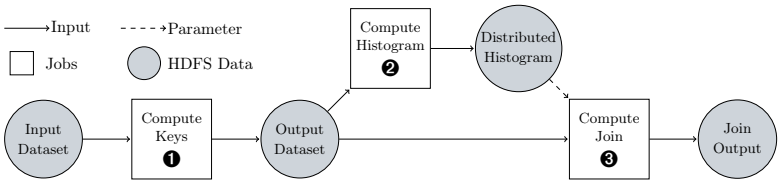

Fig. 2. MapReduce similarity join computation steps.

MRS-join proceeds in 3 steps, each step including one or two MapReduce jobs. The Figure 2 represents the interactions between the different steps of the algorithm:

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

❶ Compute the LSH join attribute values,
❷ The histogram of the join is computed and distributed to guarantee balanced communication patterns regardless the data distribution,
❸ By using distributed histograms, efficient and scalable communication templates are generated and the distance between the pairs identified as similar, is computed to produce the similarity join output.

The step ❶ computes the Q LSH join attribute values, in our implementation it is performed before steps ❷ and ❸. The step ❷ is composed of two MapReduce *jobs*, the first one is used to compute the histogram of the join and the second to distribute it. The main difference is that the histogram is constructed for a self-join and distributed by *chunks* instead of *splits* as explained previously. Using distributed histograms, the step ❸ computes the similarity join. To reduce the number of comparisons, we introduce additional filters during this step in the *MRSS-join* algorithm.

For space reason, we do not include a theoretical analysis for each computation step in the following. However, the load of each *Reducer* during the last step of *MRSS-join* is bounded by Theorem 1. In addition, the *MRS-join* algorithm has an asymptotic optimal complexity when the maximum cost to read all distributed histograms corresponding to a *Mapper* is less than the maximum cost to transmit all data and to perform the similarity join in a processing node. We recall that the sizes of distributed histograms are very small compared to input datasets sizes, and we refer the reader to [17] for more details regarding the cost model. Using *chunks* instead of *splits* to distribute the histogram only adds to the cost model a constant factor that depends on $\mathbf{t_{max}}$ and the number of records in a *split*. By maximizing $\mathbf{t_{max}}$ while ensuring that the corresponding distributed histograms fit in memory, we expect that using *chunks* instead of *splits* has a limited impact on the size of distributed histograms. Of course, this is only an intuitive argument that would deserve a detailed cost model.

3 *MRSS-join*: A scalable set similarity join algorithm using LSH and MapReduce

We assume that, before the start of the **map** phase, the Q MinHash functions are randomly and uniformly selected and stored in the HDFS. The MinHash function is implemented using Zobrist hashing [23]. Zobrist hashing has been shown theoretically and practically to have strong MinHash properties while being fast in practice [5, 20].

The histogram computation *job* is described in the Algorithm A. To compute frequency of each join attribute value, the **map** phase emits, for each record, and for each join attribute value x, two key/value pairs, allowing to handle the frequency and the set of *chunkId* apart. To ensure that the *groups* within distributed histograms are sorted in the correct order, a *chunkId* is implemented using a 64 bits integer. The first 32 bits correspond to the *splitId* and the last ones to the *groupId*.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

Algorithm A: Histogram computation step

Map: $<id,\ u> \rightarrow <(x,\ 0/1),\ (1/,\ /chunkId)>$
- **init:**
  - Read from HDFS the Q MinHash functions.
- $chunkId \leftarrow$ getCurrentChunkId();
- Compute the Q LSH join attribute values of the record u.
- For each join attribute value x emit the pairs:
  - $<(x,\ 0),\ (1,\ \emptyset)>$
  - $<(x,\ 1),\ (\varepsilon,\ chunkId)>$

Combine: $<(x,\ 0),\ (1,\ \emptyset)^*> \rightarrow <x,\ (lf_x,\ \emptyset)>$
- Compute local frequency using the size of input values.
- Emit a pair $<(x,\ 0),\ (lf_x,\ \emptyset)>$.

Combine: $<(x,\ 1),\ (\varepsilon,\ chunkId)^*> \rightarrow <x,\ (\varepsilon,\ chunkId^*)>$
- Compute the set of *chunkId*.
- Emit a pair $<(x,\ 1),\ (\varepsilon,\ chunkId^*)>$.

Reduce: $<(x,\ 0),\ (lf_x,\ \emptyset)^*>$
- Compute the global frequency of the join attribute value.

Reduce: $<(x,\ 1),\ (\varepsilon,\ chunkId^*)^*> \rightarrow <x,\ (\mathbf{f_x},\ chunkId^*)>$
- Compute the set of *chunkId*; whenever the set size exceeds a given limit, the following pair is emitted and the set cleared.
- Emit the pair $<x,\ (\mathbf{f_x},\ chunkId^*)>$ if $\mathbf{f_x} > 1$.

Algorithm B: Histogram distribution step

Map: $<x,\ (\mathbf{f_x},\ chunkId^*)> \rightarrow <chunkId,\ (x,\ \mathbf{f_x})>$
- Emit a pair $<id,\ (\mathrm{x},\ \mathbf{f_x})>$ for all $id \in chunkId^*$.

Partition: $<chunkId,\ (x,\ \mathbf{f_x})> \rightarrow Integer$
- Return the *splitId* from the *chunkId*.

Reduce: $<chunkId,\ (x,\ \mathbf{f_x})^*> \rightarrow <chunkId,\ (x,\ \mathbf{f_x})^*>$
- Compute the set of values received to eliminate duplicates.
- Emit a pair $<\mathrm{x},\ \mathbf{f_x}>$ for all remaining values.

The *Combiner* computes the local frequencies and the set of *chunkIds* of the current *split*. *Reducers* sum up the received local frequencies and filter out join attribute values that produce no results. The set of *chunkIds* where the join attribute value appears is stored to be able to distribute the histogram. Since the set of *chunkIds* is computed after the frequency, it does not have to fit in memory. Duplicate entries may be produced, which will be eliminated during the distribution step according to Algorithm B. This algorithm relies on the **partition** function to distribute the histogram based on the stored *split* identifiers. Owing to the parameter $\mathbf{t_{max}}$, duplicates can be eliminated in memory during the **reduce** phase. The similarity join *job*, described in the Algorithm C, uses distributed histograms to generate efficient communication templates. The similarity join is computed by filtering out false positive pairs. It should be noted that the computation of the Jaccard distance is performed only once for a pair of records. We introduce additional filtering steps to reduce the number of

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

Algorithm C: Similarity join computation step

Map: *<id, u> → <(x, reducerId, rowId), (u, LSHKeys)>*

init:

Read from HDFS the Q MinHash functions.

Read and store the corresponding distributed histogram of the current *chunkId*.

Compute the Q LSH join attribute values of the record u.

Retain only the join attribute values that appear in the distributed histogram, which we will denote by $LSHKeys$.

Emit pairs according to communication templates described previously 2.2.

Partition: *<(x, reducerId, rowId), (u, LSHKeys)> → Integer*

Redirect each pair according to communication templates.

Reduce: *<(x, reducerId, rowId), (u, LSHKeys)> → (id_1, id_2)*

Compute the similarity join using the communication templates.

For each pair of different record:

- Compute the intersection of the sets $LSHKeys$.
- If the current join attribute value is the minimal among the intersection, compute the Jaccard distance and output them if the distance is lower than λ.

comparisons. We distinguish two cases depending on whether the similarity join computations of a join attribute value are performed by one or several **reduce** tasks.

In the case where the similarity join computations are performed by a single *Reducer* task, the length filter can be applied [1]. This filter allows only the Jaccard distance between pairs of records of similar size to be computed. For this purpose, the length of the records is appended to the transmitted keys during the join step, allowing the records to be processed in a sorted manner.

In the case of a highly frequent join attribute value, the filtering step relies on LSH. For a **reduce** task and a join attribute value, instead of storing the distributed block, each record is partitioned into F_Q buckets using MinHash. The number of concatenate MinHash function F_K is based on F_Q and the desired expectation F_E, i.e., formally, $F_K = \log_{p_1^{-1}}(F_Q/F_E)$.

When $K < F_K$, we expect the number of comparisons to be drastically reduced, resulting in a time saving. However, this time saving is only guaranteed if the cost of filtering and comparing remaining records is less than the cost of comparing all records. The filtering cost is increasing with F_Q, we set $F_Q = 32$ and $F_E = 4$ in our experiments.

4 Experiments

In this section, we discuss the efficiency and the strength of our theoretical analysis by experimenting the *MRSS-join* algorithm on real world and synthetic datasets. We measured the *recall* and *precision* as well as the efficiency

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

Dataset	N	Record length		Universe	Size (B)
	$\cdot 10^5$	max	avg	$\cdot 10^3$	
AOL	100	245	3	3900	396MB
ENRO	2.5	3162	135	1100	254MB
LIVE	31	300	36	7500	873MB
NETF	4.8	18000	210	18	576MB
ORKUT	2	40000	120	8700	2.5GB
WDC	41	17000	15	184644	5.8GB
UNIFORM	1	25	10	0.21	4.5MB
ZIPF (1.0)	4.4	84	50	100	33MB
ALL	182	40000	21	205962	10.3GB

Table 1. Characteristics of the experimental datasets.

of the *MRSS-join* compared to the state-of-the-art algorithm. The experiments were performed using Hadoop 3.2.1 framework on a cluster of 11 machines. Each machine has the following characteristics: Intel(R) Xeon(R) CPU E5-2650 @2.60GHz, 16Gb of memory, 300Gb of HDD disk and 6Gb as a value for Heap memory of Map/Reduce tasks. The nodes are connected by a 1Gb/s network. The map output compression is enabled as well as the output compression. For the following experiments, the duplicates in the input dataset are removed since duplicate removal is a different problem from similarity joins. In addition, this makes our results comparable to the existing surveys [8, 16].

4.1 Performance and results quality

To analyze the performance and the quality of the *MRSS-join* algorithm, we used 6 real-world and 2 synthetic datasets. The datasets mainly come from the survey [16] and the distributed survey [8] on the exact set similarity join. The entire English relational subset of the WDC Web Table Corpus 2015 [14] has been added to test the scalability of the algorithms. Textual datasets are preprocessed to translate original strings to integers using the tools from [16]. Since we focus on the set similarity join, this pre-processing step is not measured in our experiments. Table 1 presents characteristics of the selected datasets. The characteristics of the datasets are given by the number of records, the maximum, and average size of a record and the size of the universe. Records in AOL, LIVE and WDC datasets are short with a large universe which favors the *VJ* algorithm. The token frequencies follow a Zipfian token distribution for most of real-world dataset, that means a large part of the universe is infrequent. An exception is the NETF dataset which has few infrequent tokens. The two synthetic datasets are UNIFORM and ZIPF which are generated to follow a uniform and a Zipfian token distribution using Zipf factor 1.0 and the generators from [16]. The dataset ALL is simply the union of the datasets presented above and is used to test scalability. We used the state-of-the-art algorithm *VJ* to compare the performance of *MRSS-join*. We performed a self join on all previous datasets by varying the Jaccard similarity threshold in $\{0.6, 0.7, 0.8, 0.9, 0.95\}$. Table 2 shows the average join processing time in seconds over three independent runs. For each run and practical reasons, we set a timeout of 2 hours, which is

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

more than three times the runtime of *MRSS-join* for all dataset. The letter "T" denotes that this timeout has been reached for all runs. For the additional parameters, the desired expectation of generating any output pair is set to $E = 2$, and we set $\rho = 0.5$. For space reasons, we omit the experiments varying these parameters. However, we have selected parameters that give the best trade-off between time and quality in our setup.

The processing time for *MRSS-join* is always lower than the corresponding processing time of *VJ* except for AOL. However, in this setting the runtimes are of the same magnitude. In the remaining cases, *MRSS-join* achieves a speedup that can exceed an order of magnitude. Especially, for datasets such as ALL, NETF, ORKUT and WDC, where *VJ* fails to compute the similarity join within the time allocated for one or more similarity thresholds.

More in-depth analysis is given Table 3 that compares the transmitted data during the communication phase of the joining step between *MRSS-join* and *VJ*. It shows that *VJ* is inefficient in terms of transmitted data for dataset containing long records on average as ENRO, NETF and ORKUT. This is due to the fact that *VJ* uses prefixes to compute the similarity join which makes it very sensitive to long records and low similarity join thresholds. This is not the case in *MRSS-join* because it is based on LSH framework which is independent of dimensionality. In addition, we recall that only relevant data is transmitted in *MRSS-join* which drastically reduces the transmitted data during the communication phase of the similarity join step.

In Table 3, the size of transmitted data for ALL and WDC datasets are not reported for *VJ* since it failed before the similarity join step. Indeed, to compute prefixes according to the lowest frequency, *VJ* constructs the histogram of the universe. This histogram is then buffered in memory. For datasets with a very large universe, such as ALL and WDC, memory overloads may occur which limits its efficiency and scalability. This cannot happen in *MRSS-join* because the histogram of LSH join attribute values is distributed and read by *chunks* that fit in memory.

Finally, *VJ* groups intermediate data by prefix token during the joining step. The size of a group depends on the token frequency thus for large datasets, a group may hit the memory limit of a Reducer which limits its scalability. Even in the case of small datasets with few infrequent tokens as NETF, this limits significantly its efficiency because the join similarity computations are not well distributed among all the processing nodes. This cannot happen in *MRSS-join* because the join computations for a highly frequent join attribute value, are partitioned into buckets and transmitted to distinct **reduce** tasks in a randomized manner. This makes *MRSS-join* scalable and insensitive to the data distribution. To achieve these performances, *MRSS-join* is based on LSH that produces almost all the results of the similarity join. Table 4 shows the quality of the LSH filtering by *MRSS-join*. ZIPF is omitted because the similarity join produces no result on the queried thresholds since a large part of the tokens are unique. Each cell reports the *recall* and the *precision* values. The full similarity join is computed using *VJ* except for ALL and WDC, where we used *MRSS-join*

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

Threshold	0.6		0.7		0.8		0.9		0.95	
Dataset	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS
AOL	**260**	383	**198**	249	**172**	220	**168**	195	**165**	191
ENRO	481	**142**	341	**137**	295	**136**	231	**131**	196	**137**
LIVE	480	**202**	349	**177**	306	**157**	227	**149**	221	**147**
NETF	T	**217**	2809	**172**	909	**148**	460	**139**	299	**143**
ORKUT	5176	**172**	3654	**167**	2717	**154**	1183	**156**	695	**153**
WDC	T	**1798**	T	**831**	T	**620**	T	**458**	T	**401**
UNIFORM	175	**135**	153	**129**	149	**134**	145	**130**	147	**134**
ZIPF	151	**132**	152	**131**	147	**125**	152	**127**	156	**127**
ALL	T	**2086**	T	**1043**	T	**766**	T	**606**	T	**549**

Table 2. Time in seconds of *MRSS-join* algorithm compared to *VJ* algorithm.

Threshold	0.6		0.7		0.8		0.9		0.95	
Dataset	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS
AOL	**455MB**	2GB	**409MB**	1GB	**333MB**	811MB	**255MB**	327MB	209MB	**133MB**
ENRO	16GB	**741MB**	12GB	**403MB**	8GB	**247MB**	4GB	**146MB**	2GB	**78MB**
LIVE	13GB	**1GB**	10GB	**762MB**	7GB	**222MB**	3GB	**96MB**	2GB	**45MB**
NETF	79GB	**3GB**	59GB	**1GB**	40GB	**742MB**	20GB	**9MB**	10GB	**1MB**
ORKUT	234GB	**1GB**	176GB	**780MB**	118GB	**37MB**	59GB	**17MB**	30GB	**12MB**
WDC	T	**14GB**	T	**7GB**	T	**5GB**	T	**2GB**	T	**1GB**
UNIFORM	**13MB**	87MB	**10MB**	37MB	**8MB**	28MB	5MB	**1MB**	4MB	**110KB**
ZIPF	306MB	**51MB**	234MB	**29MB**	161MB	**2MB**	87MB	**20KB**	52MB	**20KB**
ALL	T	**25GB**	T	**13GB**	T	**8GB**	T	**4GB**	T	**2GB**

Table 3. Transmitted data of *MRSS-join* algorithm compared to *VJ* algorithm.

with the expectation set to $E = 3 * \ln(N)$. We observe that, *MRSS-join* achieves at least 90% *recall* for all datasets. One can notice, the low values of the *precision* in the experiments. These values are not bad due to the fact that, there is no hashing or sorting techniques allowing to find similar pairs and even for these low precision values, *MRSS-join* reduces drastically the number of set comparisons compared to *VJ* as shown in the Table 5.

	Threshold				
Dataset	0.6	0.7	0.8	0.9	0.95
AOL	0.93 | 0.010	0.96 | 0.003	0.96 | 0.003	0.96 | 0.000	0.98 | 0.000
ENRO	0.99 | 0.086	0.99 | 0.124	0.99 | 0.402	0.98 | 0.202	0.97 | 0.355
LIVE	0.93 | 0.027	0.97 | 0.010	0.97 | 0.014	0.98 | 0.033	0.98 | 0.077
NETF	0.96 | 0.001	0.97 | 0.001	0.97 | 0.001	0.96 | 0.015	1.00 | 0.024
ORKUT	0.97 | 0.005	0.98 | 0.003	0.97 | 0.015	0.97 | 0.019	0.97 | 0.017
WDC	0.98 | 0.488	0.98 | 0.032	0.98 | 0.024	0.97 | 0.052	0.98 | 0.063
UNIFORM	0.92 | 0.001	0.96 | 0.001	0.98 | 0.001	1.0 | 0.0	1.0 | 0.0
ALL	0.97 | 0.460	0.98 | 0.030	0.98 | 0.026	0.98 | 0.045	0.98 | 0.049

Table 4. Results quality of the *MRSS-join* algorithm.

5 Conclusion

In this article, we have introduced *MRSS-join* an efficient and scalable MapReduce set similarity join algorithm, using LSH and randomized communication

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

Threshold	0.6		0.7		0.8		0.9		0.95	
Dataset	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS	VJ	MRSS
AOL	1.3E+10	**1.2E+09**	5.5E+09	**5.1E+08**	1.6E+09	**1.4E+08**	2.4E+08	**3.0E+07**	1.2E+08	**7.8E+06**
ENRO	2.3E+09	**2.4E+07**	6.2E+08	**9.6E+06**	1.1E+08	**1.8E+06**	1.2E+07	**5.4E+05**	1.8E+06	**1.4E+05**
LIVE	7.6E+09	**1.5E+08**	2.1E+09	**7.6E+07**	4.1E+08	**1.1E+07**	5.3E+07	**7.7E+05**	8.9E+06	**1.3E+05**
NETF	6.4E+10	**1.1E+08**	2.2E+10	**3.9E+07**	5.0E+09	**3.0E+06**	4.6E+08	**4.9E+03**	4.9E+07	**2.6E+02**
ORKUT	4.6E+09	**4.9E+06**	1.1E+09	**2.0E+06**	1.7E+08	**1.4E+05**	1.2E+07	**2.6E+04**	1.5E+06	**6.4E+03**
WDC	T	**8.2E+09**	T	**5.1E+09**	T	**1.4E+09**	T	**1.1E+08**	T	**2.0E+07**
UNIFORM	2.4E+09	**3.6E+07**	1.3E+09	**1.9E+07**	5.4E+08	**5.4E+05**	1.1E+08	**2.5E+03**	3.2E+07	**2.4E+02**
ALL	T	**8.2E+09**	T	**5.5E+09**	T	**1.4E+09**	T	**1.3E+08**	T	**2.7E+07**

Table 5. Computed distances of the *MRSS-join* algorithm compared to the *VJ* algorithm.

templates approach allowing to reduce drastically the number of sets comparisons and communication costs while guaranteeing perfect balancing properties during all the steps of large datasets similarity join computation.

MRSS-join theoretical guarantees and experiments, using real world and synthetic benchmarks datasets, show that the overhead related to the use both MinHash and our communication templates remains very small compared to the gain in performance by reducing communication and data processing to almost all relevant data (this avoids sets pairwise comparisons). We showed that, *MRSS-join* avoids memory overflows by controlling the size of generated buckets. This makes the algorithm scalable and insensitive to the data distribution. It also solves the limitations of existing approaches to handle large datasets whenever data associated to a MapReduce key cannot fit in the available reducer's local memory.

Future work will be devoted to extend *MRSS-join* algorithm to a more general purpose framework for most similarity joins operations by using LSH techniques. We also plan to compute sequences similarity processing in large datasets using similar techniques based on our randomized MapReduce data redistribution to balance load among processing nodes while guaranteeing the scalability of the proposed solutions in large scale systems.

References

1. Arasu, A., Ganti, V., Kaushik, R.: Efficient exact set-similarity joins. In: Proceedings of the 32nd International Conference on Very Large Data Bases. pp. 918–929 (2006)
2. Bayardo, R.J., Ma, Y., Srikant, R.: Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web. pp. 131–140 (2007)
3. Broder, A.Z., Glassman, S.C., Manasse, M.S., Zweig, G.: Syntactic clustering of the Web. Computer Networks and ISDN Systems **29**(8), 1157–1166 (1997)
4. Chaudhuri, S., Ganti, V., Kaushik, R.: A primitive operator for similarity joins in data cleaning. In: 22nd International Conference on Data Engineering (2006)
5. Dahlgaard, S., Knudsen, M.B.T., Thorup, M.: Practical hash functions for similarity estimation and dimensionality reduction. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 6618–6628 (2017)

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41

6. Dean, J., Ghemawat, S.: MapReduce: Simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
7. Dey, D., Sarkar, S., De, P.: A distance-based approach to entity reconciliation in heterogeneous databases. IEEE Transactions on Knowledge and Data Engineering **14**(3), 567–582 (2002)
8. Fier, F., Augsten, N., Bouros, P., Leser, U., Freytag, J.C.: Set similarity joins on mapreduce: An experimental survey. Proceedings of the VLDB Endowment **11**(10), 1110–1122 (2018)
9. Gionis, A., Indyk, P., Motwani, R.: Similarity Search in High Dimensions via Hashing. In: Proceedings of the 25th International Conference on Very Large Data Bases. pp. 518–529 (1999)
10. Hassan, M.A.H., Bamha, M., Loulergue, F.: Handling data-skew effects in join operations using mapreduce. Procedia Computer Science **29**, 145–158 (2014)
11. Hu, X., Tao, Y., Yi, K.: Output-optimal Parallel Algorithms for Similarity Joins. In: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. pp. 79–90. ACM (2017)
12. Hu, X., Yi, K., Tao, Y.: Output-Optimal Massively Parallel Algorithms for Similarity Joins. ACM Transactions on Database Systems **44**(2), 1–36 (2019)
13. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing. pp. 604–613 (1998)
14. Lehmberg, O., Ritze, D., Meusel, R., Bizer, C.: A Large Public Corpus of Web Tables containing Time and Context Metadata. In: Proceedings of the 25th International Conference Companion on World Wide Web. pp. 75–76 (2016)
15. Li, P., Owen, A., Zhang, C.h.: One permutation hashing. In: Pereira, F., Burges, C.J.C., Bottou, L., Weinberger, K.Q. (eds.) Advances in Neural Information Processing Systems. vol. 25. Curran Associates, Inc. (2012)
16. Mann, W., Augsten, N., Bouros, P.: An empirical evaluation of set similarity join techniques. Proceedings of the VLDB Endowment **9**(9), 636–647 (2016)
17. Rivault, S., Bamha, M., Limet, S., Robert, S.: A Scalable MapReduce Similarity Join Algorithm Using LSH. Research report, LIFO, Université d'Orléans. To appear to IJPP int. Journal (2021)
18. Shrivastava, A., Li, P.: Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search. In: Proceedings of the 31st International Conference on Machine Learning. pp. 557–565 (2014)
19. Theobald, M., Siddharth, J., Paepcke, A.: Spotsigs: Robust and efficient near duplicate detection in large web collections. In: Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval (2008)
20. Thorup, M.: Fast and powerful hashing using tabulation. Communications of the ACM **60**(7), 94–101 (2017)
21. Vernica, R., Carey, M.J., Li, C.: Efficient parallel set-similarity joins using MapReduce. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data (2010)
22. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., Leser, U.: State-of-the-art in string similarity search and join. SIGMOD Record **43**(1), 64–76 (2014)
23. Zobrist, A.L.: A new hashing method with application for game playing. ICGA Journal **13**, 69–73 (1990)

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_41