# An Efficient and Scalable MapReduce Algorithm for Sequence Similarity Join on Big Data

Sébastien RIVAULT (sebastien.rivault@univ-orleans.fr), Mostafa BAMHA (mostafa.bamha@univ-orleans.fr), Sébastien LIMET (sebastien.limet@univ-orleans.fr), Sophie ROBERT (sophie.robert@univ-orleans.fr)

Université d'Orléans, INSA Centre Val de Loire, LIFO ER 4022, France

**Corresponding Author:**
Sébastien RIVAULT
Université d'Orléans, INSA Centre Val de Loire, LIFO ER 4022, France
Email: sebastien.rivault@univ-orleans.fr

# An Efficient and Scalable MapReduce Algorithm for Sequence Similarity Join on Big Data

Sébastien RIVAULT[a,1], Mostafa BAMHA[a], Sébastien LIMET[a], Sophie ROBERT[a]

[a]*Université d'Orléans, INSA Centre Val de Loire, LIFO ER 4022, France*

**Abstract**

The edit similarity join operation consists in finding all the similar pairs of sequences according to an edit distance threshold. This problem has received a lot of attention in the literature. However, the most recent studies have observed that all existing algorithms are unsuitable for long sequences and high distance thresholds on very large datasets. In this paper, we introduce a new scalable algorithm called *MRSF-join* to perform edit similarity joins on huge sets of protein sequences using large distance thresholds. The *MRSF-join* algorithm drastically reduces processing time, communication, and disks I/O costs to only relevant data, while minimizing the number of computed distance. All our claims are supported by a theoretical cost model and a series of experiments showing the effectiveness of our approach in very large datasets processing.

*Keywords:* Similarity Join, MapReduce programming model, Data Skew, Hadoop framework, Locality Sensitive Hashing (LSH).

## 1. Introduction

The similarity join operation consists in finding all the pairs of sequences which are similar according to a given distance $\mathcal{D}$ and a given threshold. Formally, the similarity join for two given collections $R$ and $S$ of sequences and a

*Corresponding author.

*Email addresses:* `sebastien.rivault@univ-orleans.fr` (Sébastien RIVAULT), `mostafa.bamha@univ-orleans.fr` (Mostafa BAMHA), `sebastien.limet@univ-orleans.fr` (Sébastien LIMET), `sophie.robert@univ-orleans.fr` (Sophie ROBERT)

given threshold $\lambda$ is defined as follows:

$$R \bowtie_\lambda S = \{(u, v) \in R \times S \mid \mathcal{D}(u, v) \leq \lambda\}$$

where $\mathcal{D}(u, v)$ denotes a distance between two sequences $u$ and $v$, and $\lambda$ is the user-defined threshold parameter.

Similarity joins are primitive operators in database systems [1], and have a wide variety of applications, including data cleaning and integration, entity resolution, similar sequence detection in bioinformatics and collaborative filtering.

In this paper, we focus on one of the most popular distances in the literature, the edit distance, which determines the minimum number of edit operations (insertions, deletions, and substitutions) to transform one string to another. Although the algorithm can handle different variants of the distance, the experiments were performed using the normalized edit distance, as it is more meaningful for comparing strings of different lengths. In addition, this paper concentrates on long sequences using large threshold values. Although the sequence alphabet is fixed at 20, as for proteins, we expect the designed techniques to handle larger alphabets. According to [2, 3], this is a major challenge for most existing algorithms, as they are not scalable to very large datasets. The motivation for these criteria comes from the huge amount of data generated by different applications. For example, the throughput of massive parallel sequencing has risen over the past decade, resulting in very large datasets of metagenomes, genes and proteins. These datasets could greatly improve large-scale functional annotations and structure prediction.

To process such large datasets, the similarity join must drastically reduce the number of unnecessary comparisons to avoid the naive approach requiring a pairwise comparison and having a disastrous effect on performance. In the literature, the $q$-gram filtering has received much attention to approximate string matching under the edit distance [4, 5, 6, 2, 7, 8, 9]. This method approximates the edit distance using Jaccard distance [10, 7, 11][1]. In the literature, set similarity joins using Jaccard distance can be handled efficiently even for very large datasets by removing the constraint of the result completeness. We support this claim with a recent survey [12] that reported on the scalability of existing exact solutions for set similarity joins. This survey concludes that none of the evaluated algorithms can scale to process large datasets. The approximate approach is typically based on Locality Sensitive Hashing (LSH) which is a randomized

[1]As we will see later, the Jaccard distance is not used directly.

method for generating candidate pairs of similar sequences.

To compute set similarity joins, we introduced the *MRSS-join* algorithm [13]. It is built on top of the MapReduce [14] framework, allowing us to process very large datasets. It leverages LSH to reduce the number of comparisons while providing guarantees on the completeness of the result and ensuring that the load of the processing nodes is balanced. This algorithm is based on the theoretical guarantees on the load provided by [15, 16]. Recently, [17] extended the analysis and improved the algorithm using sketching and deduplication techniques. However, none of these algorithms can efficiently perform set similarity joins for high-value Jaccard distance thresholds, as it is the case for $q$-gram filtering, because they require too many LSH iterations, resulting in excessive communication costs. In this paper, we present a new algorithm called *MRSF-join* (MapReduce Sequence Filtering) to address this issue.

The remaining of this paper is organized as follows: Section 2 presents requirements to understand of the *MRSF-join* algorithm. Section 3 describes the *MRSF-join* algorithm in detail, including a cost model for each computation step. The cost model and the experimental results presented in Section 4 confirm the efficiency of our approach. We then conclude in Section 5.

## 2. Preliminaries

This section is organized as follows: Section 2.1 explains the MapReduce programming model; Section 2.3 introduces the $q$-gram filtering; Section 2.4 introduces Locality Sensitive Hashing technique; Section 2.5 reviews distributed histograms and randomized communication templates; Section 2.6 presents the *MRSS-join* algorithm based on LSH, distributed histograms and random communication templates as well as the main improvements of the *MRSF-join* algorithm compared to our previous algorithms.

### *2.1. The MapReduce programming model*

MapReduce is a simple yet powerful programming model for implementing scalable distributed applications without having extensive prior knowledge of issues related to data redistribution, task allocation or fault tolerance in large scale distributed systems.

Google's MapReduce programming model, described by [14] and illustrated in Figure 1, is based on two functions: **map** and **reduce**, provided by the user to the framework. These two functions should have the following signatures:

$$\textbf{map}: (K_{\text{in}}, V_{\text{in}}) \rightarrow \text{List}(K_{\text{m}}, V_{\text{m}}),$$
$$\textbf{reduce}: (K_{\text{m}}, \text{List}(V_{\text{m}})) \rightarrow \text{List}(K_{\text{out}}, V_{\text{out}}).$$

The **map** function has two input arguments, a key $K_{\text{in}}$ and its corresponding value $V_{\text{in}}$. The **map** output is a list of intermediate Key/Value pairs $(K_{\text{m}}, V_{\text{m}})$. This list is partitioned by the MapReduce framework according to the values $K_{\text{m}}$. All the pairs with the same value $K_{\text{m}}$ belong to the same partition and will be transmitted to a single *Reducer*.

The **reduce** function has two input arguments: an intermediate key $K_{\text{m}}$ and its corresponding list of intermediate values $\text{List}(V_{\text{m}})$. It applies the user-defined logic merge to the $\text{List}(V_{\text{m}})$ and outputs a list of Key/Value pairs $\text{List}(K_{\text{out}}, V_{\text{out}})$.

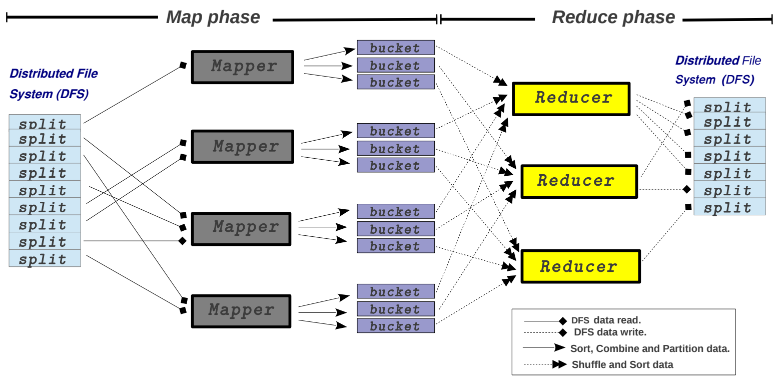


Figure 1: MapReduce framework.

In this paper, we used Apache Hadoop, which is an open-source framework that implements the MapReduce programming model. The Hadoop framework includes a distributed file system called HDFS[2] and designed to store very large files with streaming data access patterns.

For efficiency reasons, the Hadoop MapReduce framework also provides a **combine** function to reduce the amount of data transmitted from *Mappers* to *Reducers*. The **combine** function acts as a local *Reducer* and is applied before storing or transmitting intermediate results to the *Reducers* during the **map** phase. The signature of the **combine** function is as follows:

$$\textbf{combine}: (K_{\text{m}}, \text{List}(V_{\text{m}})) \rightarrow \text{List}(K'_{\text{m}}, V'_{\text{m}}).$$

[2]HDFS: Hadoop Distributed File System.

To address a wide range of application requirements in terms of computation and data redistribution, the Hadoop MapReduce framework also provides two additional functions: **init** and **close** called before and after each **map**, **reduce** or **combine** task. In addition, the **partition** function is also provided to allow each Key to be transmitted to a specific reducer during the shuffle step of the **reduce** phase.

### *2.2. Edit distance*

Let $\Sigma$ be a finite alphabet of size $\|\Sigma\|$, and let $\Sigma^+$ be the set of all non-empty strings of characters of $\Sigma$.

**Definition 1.** The edit distance $\mathcal{D}_{\mathsf{Lev}}(u, v)$ between two strings $u, v$ of $\Sigma^+$ is defined by the minimum number of edit operations (insertions, deletions, and substitutions) of single characters to transform one string into another.

In the context of similarity joins, when the range of string lengths is large, it is usual to use the normalized edit distance.

**Definition 2.** The normalized edit distance between two strings is defined as:

$$\mathcal{D}_{\mathsf{NLev}}(u, v) = \mathcal{D}_{\mathsf{Lev}}(u, v)/\max(\|u\|, \|v\|)$$

Let $\Lambda$ be the threshold of the normalized distance for the similarity join. Two strings $u, v \in \Sigma^+$ are considered similar if $\mathcal{D}_{\mathsf{NLev}}(u, v) \leq \Lambda$.

### *2.3. Sequence filtering using q-grams*

To filter out dissimilar sequences, we approximate the edit distance using the Jaccard distance, which consists in considering a string as a multiset of substrings of length $q$ called $q$-grams. Let $\Sigma^q$ be the set of all strings of length $q$ over the alphabet $\Sigma$. A $q$-gram is any string $w \in \Sigma^q$ for $q \in \mathbb{N}^*$.

**Definition 3.** Let $u = u_1 \cdots u_n$ be a string from $\Sigma^+$, the multiset of $q$-grams is defined as $G^q(u) = \{u_i \cdots u_{i+q-1} \mid \forall i : 1 \leq i \leq n - q + 1\}$.

For example, given the word *protein* and $q = 4$, the corresponding multiset of $q$-grams is $\{$*prot,rote,otei,tein*$\}$. Note that $G^q(u)$ contains $(\|u\| - q + 1)$ $q$-grams.

Let $u$ and $v$ be two strings of $\Sigma^+$, since a single edit operation can destroy at most $q$ $q$-grams by the definition 3, it holds that the size of the intersection can be expressed according to the edit distance threshold:

**Theorem 1.** Let $\Sigma$ a finite alphabet, $u, v \in \Sigma^+$, $q \in \mathbb{N}^*$, $\Lambda \in \mathbb{N}$, the size of the intersection of the q-gram multisets of sequences $u$ and $v$ is bounded by:

$$\|G^q(u) \cap G^q(v)\| \geq (\max(\|G^q(u)\|, \|G^q(v)\|)) - q \times \Lambda$$

For normalized edit distance, the cardinality of the intersection of the multisets of $q$-grams of two sequences is determined as follows when $q \times \Lambda < 1$:

**Theorem 2.** Let $\Sigma$ a finite alphabet, $u, v \in \Sigma^+$, $q \in \mathbb{N}^*$, $\Lambda \in [0, 1]$, the size of the intersection of the q-gram multisets of sequences $u$ and $v$ is bounded by:

$$\|G^q(u) \cap G^q(v)\| \geq (\max(\|G^q(u)\|, \|G^q(v)\|) * (1 - q \times \Lambda)$$

In addition, it should be noted that the number of destroyed $q$-grams depends on the positions of the edit operations. For example, when two edit operations are performed consecutively on a string $u$ from $\Sigma^+$, i.e., at positions $i$ and $i + 1$ for any $i$ such that $1 \leq i \leq \|u\|$, at most $(q + 1)$ $q$-grams are destroyed.

**Definition 4.** Let the Jaccard distance be defined as :

$$\mathcal{D}_{\text{Jaccard}}(G^q(u), G^q(v)) = 1 - \frac{\|G^q(u) \cap G^q(v)\|}{\|G^q(u) \cup G^q(v)\|}$$

Note that the size of the union of multisets of $q$-grams can be expressed according to the size of the intersection.

In addition, the operation of insertions and deletions are already included in the edit distance threshold, because the size difference is included in the edit distance to transform one string into another, so we can use two sequences of equal length to determine the size of the intersection without any loss of generality.

**Theorem 3.** Any sequence similar in terms of the edit distance satisfies the following inequality:

$$\begin{aligned}
\mathcal{D}_{\text{Jaccard}}(G^q(u), G^q(v)) &\leq 1 - \frac{\|G^q(u)\| - q \times \Lambda}{\|G^q(u) \cup G^q(v)\|} \\
&\leq 1 - \frac{\|G^q(u)\| - q \times \Lambda}{\|G^q(u)\| + \|G^q(v)\| - \|G^q(u) \cap G^q(v)\|} \\
&\leq 1 - \frac{\|G^q(u)\| - q \times \Lambda}{2 \times \|G^q(u)\| - \|G^q(u) \cap G^q(v)\|}
\end{aligned}$$

**Corollaire 3.1.** Using normalized edit distance, the following inequality is satisfied:

$$\mathcal{D}_{\mathsf{Jaccard}}(G^q(u), G^q(v)) \leq 1 - \frac{\|G^q(u)\| \times (1 - q \times \Lambda)}{2 * \|G^q(u)\| - \|G^q(u) \cap G^q(v)\|}$$

For example, by setting the threshold for the normalized edit distance to $\Lambda = 0.15$ and $q = 4$, it leads to compute a set similarity join where the Jaccard distance is lower than $\lambda = 0.75$. This is because using an equal length for the sequences implies a constant ratio for the Jaccard distance. It should be noted that this Jaccard distance threshold can be efficiently processed using set similarity join algorithms.

To reduce the number of comparisons, several $q$-grams are concatenated to form a join attribute value; the number of concatenations is denoted $\mathcal{K}$ in the following. We make the strong assumption that the edit operations between two strings are uniformly distributed, allowing to partition the multiset of $q$-grams into $\mathcal{K}$ parts. In addition, it is assumed that the number of insertions/deletions is equal in each part. Although this assumption seems to be very optimistic, the number of $q$-grams destroyed is lower on average, since the Jaccard distance threshold corresponds to the minimum size of the intersection 3,3.1, which ensures a certain degree of completeness.

**Definition 5.** Let $u \in \Sigma^+$, $p, q > 0$ and $\mathcal{K} \geq 1$ such that $\|G^q(u)\| = \|u\| - q + 1 = \mathcal{K} \times p$. For $j \in 1, \dots, \mathcal{K}$, the $j^{th}$ slice of $u$ is denoted $S_j^{\mathcal{K}}$ and defined as $S_j^{\mathcal{K}}(u) = u_{(j-1)\times p+1} \cdots u_{j \times p + q - 1}$. The corresponding $q$-grams multiset of the $j^{th}$ slice is then $G_j^q(u) = G^q(S_j^{\mathcal{K}}(u))$ for the sake of simplicity.

In practice, when the size is not a multiple of $\mathcal{K}$, the $\|G^q(u)\| \,\%\, \mathcal{K}$ first parts include an additional $q$-gram.

**Definition 6.** Let $\mathcal{G}^*$ denote the set of all multisets of $q$-grams, and let us assume that we have a function $\mathrm{MH} : \mathcal{G}^* \to \Sigma^q$ that selects a $q$-gram from a multiset of $q$-grams. A raw join attribute value is defined as $x = \mathrm{MH}(G_1^q(u)), \dots, \mathrm{MH}(G_{\mathcal{K}}^q(u))$.

The procedure for selecting a $q$-gram per slice is described later using LSH functions. As an example, the Figure 2 shows how an example of raw join attribute value is generated using $q$-gram filtering and by setting $q = 4$ and $\mathcal{K} = 3$. It should be noted that the slicing method allows several $q$-grams to be concatenated while preserving the order of these $q$-grams in the original sequence.

An important aspect of the problem is the choice of the parameter $q > 0$ in order to reduce the number of computed distances while minimizing the communication costs. The longer the $q$-grams, the more efficient the filtering is, as each raw join attribute value will be drawn from a domain of at least $(\|\Sigma\|^q)^{\mathcal{K}}$, considering the case where all the $q$-grams are distinct in the strings. However, the longer the $q$-grams, the smaller the size of the intersection is, so it can be more expensive to retrieve similar sets.

The size of the alphabet is also an important aspect of the problem. Indeed, the larger the alphabet, the more effective the filtering, since each $q$-gram selected on a sequence slice will be drawn from a domain of at least $\|\Sigma\|^q$.

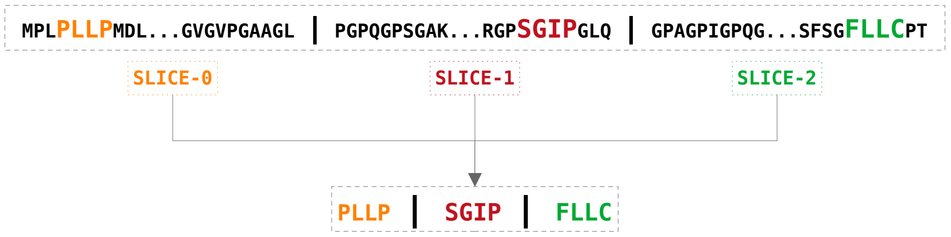


Figure 2: An example of raw join attribute value composed of three concatenated 4-grams from three independent slices of the original sequence.

## 2.4. *Locality Sensitive Hashing (LSH)*

Indyk and Motwani introduced a randomized hashing framework [18, 19] that ensures that nearby data points are more likely to collide than the distant ones. We first focus on the use of LSH for the Jaccard distance, and then we review the sequence sketches constructed using LSH.

Let $a$ and $b$ be two sets formed from a common universe $\mathcal{U}$. MinHash [20] is a family of LSH functions that estimates the Jaccard distance. It is defined from a random permutation $\pi$ of the universe $\mathcal{U}$. For any element $e$ of $\mathcal{U}$, let $\pi(e)$ be the position of $e$ in the permutation of $\mathcal{U}$. A hashing function from the family $\mathcal{H}$ is thus defined as $h(a) = \min_{e \in a} \pi(e)$. The hashing functions thus have the following probability, $\mathsf{P}_{h \in \mathcal{H}}[h(a) = h(b)] = 1 - \mathcal{D}_{\text{Jaccard}}(a, b)$, since the probabilities of having the same value under a function depend on the number of common elements. To handle the multiplicity of $q$-grams in the multisets, the hash function is extended to return a different position in the permutation for each occurrence of an element. This means counting the number of occurrences of each $q$-gram encountered during the hashing process.

Using the principle of concatenation of $q$-grams explained earlier, the function MH corresponds to a single independent MinHash function. Formally, let $\mathcal{H}^{\mathcal{K}}$ be the LSH family corresponding to a hashing function obtained by concatenating $\mathcal{K} \geq 1$ hashing functions uniformly and independently selected from $\mathcal{H}$, where each hashing function is applied to a different slice. Accordingly, the following equality holds:

$$\mathsf{P}_{g^{\mathcal{K}} \in \mathcal{H}^{\mathcal{K}}}[g^{\mathcal{K}}(u) = g^{\mathcal{K}}(v)] = \prod_{i=0}^{\mathcal{K}-1} \mathsf{P}[h_i(u) = h_i(v)], h_i \in \mathcal{H} = p_1^{\mathcal{K}}$$

This is due to the fact that slicing the sequence under the strong hypothesis guarantees that the probability is equal for each slice of the sequence.

To improve the completeness of the algorithm, it is common to use many independent iterations. The number of iterations is determined by the number of concatenations and the Jaccard distance threshold, thus $\mathcal{Q} = \lceil 1/(1-\lambda)^{\mathcal{K}} \rceil = 64$ with $\mathcal{K} = 3$ and $\lambda = 0.75$. To distinguish the raw join attribute values of each iteration, we extend the Definition 6 of the raw join attribute values.

**Definition 7.** For $i \in 1, \dots, \mathcal{Q}$, a concatenated LSH function $(h_1, \dots, h_{\mathcal{K}})$ is uniformly and independently selected from $\mathcal{H}^{\mathcal{K}}$, and given an additional universal hash function $f$, the $i^{th}$ join attribute value of a sequence $u$ is defined as:

$$x_i = \Big(i, f(h_1(G_1^q(u)), \dots, h_{\mathcal{K}}(G_{\mathcal{K}}^q(u)))\Big).$$

For very large scale datasets processing, LSH is known, in the literature, to be efficient when the number of iterations is low. We discuss later in the algorithm how to mitigate this issue to maintain efficiency even for very large datasets.

One Permutation Hashing [21, 22] is a variant of MinHash that efficiently computes several independent hashing functions using a single permutation. The idea is to partition the permutation into $m$ bins: $B_1, \dots, B_m$ and using $h_i(a) = \min_{e \in a \cap B_i} \pi(e)$. If $a \cap B_i$ is empty, $h_i(u)$ is the set to the first right value $h_j(u)$ (i.e., in a Round-Robin manner) such that $a \cap B_j$ is not empty. We refer the reader to [22] for more detailed information. Regarding the experiments, the number of bins is set to $\mathcal{Q} \times \mathcal{K}$. In the rest of the paper, MinHash refers to One Permutation Hashing.

For the sake of clarity, we have introduced the similarity join between two collections R and S, but in the remainder, we will consider self-joins of a given dataset $\Gamma$ with the objective of computing $\Gamma \bowtie_\lambda \Gamma$. From LSH's point of view, self-joins can be handled as R-S joins. Furthermore, as an LSH approach is a probabilistic method, MinHash may report sets whose Jaccard distance is in the interval $]\lambda, 1[$, we denote by $\overline{\Gamma} \bowtie_\sigma \overline{\Gamma}$ the evaluated similarity join, i.e., the pairs of sequences whose distance must be computed.

### *2.4.1. Sketching*

In our case, sequence sketches can be considered as fingerprints of sets of $q$-grams. When the size of the fingerprints is only a few bytes, they can be quickly generated and compared. These sketches are generated using LSH, which allows an approximation of the considered distance to use during the comparison. For sequence processing, it allows to quickly filter out most of the sequences whose Jaccard distance on their respective $q$-grams is greater than $\lambda$. The idea behind sketching is to trade a degree of completeness for improved pruning power. We extend the sketching definition from [23] for the slicing method.

**Definition 8.** Let $\mathcal{H}$ be a family of LSH functions and let $b = \mathcal{K} \times l$ be the length of the sketches for an integer $l \geq 1$. For $i \in 1, \dots, b$, an LSH function $h_i : \mathcal{G}^* \to \Sigma^q$ is randomly and independently selected from $\mathcal{H}$. Given an additional universal hash function $f : \Sigma^q \to \{0, 1\}$, the sketch of a sequence $u \in \Gamma$ is defined as $s(u) \in \Gamma \to \{0, 1\}^b$ such that each bit $i$ corresponds to $s(u)_i = f(h_i(G_k^q(u)))$ for an integer $1 \leq k \leq \mathcal{K}$. It means that the hash functions are applied on the different slices of the original string. For $i \in 1, \dots, b$, the $i^{th}$ hash function is applied on the $k^{th}$ slice of the original string so that $k = \lceil i/l \rceil$.

The probability that a bit $i$ is identical for two given sequence sketches depends on whether there is a collision under the LSH function $h_i$. For One-Bit Hashing [24], this probability is $(2 - \lambda)/2$. To decide whether two sequences are probably similar from the sketching point of view, the sum of the collisions is computed, i.e., $t = (\sum_{i=1}^{b} 1 \ (s(u)_i = s(v)_i))/b$, and a pair of sequences is pruned if $t < (2 - \lambda)/2$.

Increasing the sketch length $b$ generally leads to an improved accuracy in distinguishing between near and far points. However, it should be noted that increasing $b$ also incurs additional communication costs. There is therefore a trade-off between accuracy and efficiency that needs to be considered when choosing the length of the sketch. For the experiments, the sketches are computed using a single One Permutation Hashing function.

*2.5. Distributed histograms computation*

In large skewed datasets, the computation of the similarity join may be inefficiently distributed, i.e., few compute nodes are used for a large part of the distance computations. To avoid these situations, before the similarity join step, we compute the distributed histograms [25], allowing to reduce the computation and communication costs to only relevant data while guaranteeing perfect balancing properties among all processing nodes.

The histogram of a join is defined as the mapping between a join attribute value and its corresponding frequency. The histogram is used to generate the communication templates, allowing to transmit only relevant data fairly during the join phase. More formally, for a dataset $\Gamma$ where $\chi(\Gamma)$ denotes the set of its join attribute values, the histogram $\text{Hist}(\Gamma)$ is the list of all the pairs $(x, \mathbf{f}_x)$ where $x \in \chi(\Gamma)$ and $\mathbf{f}_x$ is its corresponding frequency in $\Gamma$.

In order to reduce communication costs to relevant data, only join attribute values that might appear in the join result are present in the histogram. Join attribute values that produce a result imply that their corresponding frequency is greater than one in the self-join case. Thus, the histogram of the similarity join $\overline{\Gamma} \bowtie_\sigma \overline{\Gamma}$ that contains only relevant data is defined as follows:

**Definition 9.** $\text{Hist}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma}) = \{(x, \mathbf{f}_x) \mid x \in \chi(\Gamma) \text{ and } \mathbf{f}_x > 1\}$

For very large datasets, we expect that the corresponding histogram will not fit in memory. Therefore, we introduced a redistribution scheme for multiple join attribute values presented in [26, 13]. Data redistribution principle is based on the occurrence of the join attribute values in the different parts of the input data. As in [26], we used a *split*, i.e., the part of the data processed by a **map** task, as a reference for the data redistribution. Basically, this method ensures that the size of the additional information required to process a **map** task or the histogram, is bounded by the number of records processed in that task. During the histogram computation, the identifiers of the *splits* are also stored for each join attribute value. During the redistribution step, each join attribute value and its corresponding entry in the histogram are transmitted for each split identifier. By defining the number of **reduce** tasks as the number of *splits*, each **reduce** task output corresponds to the distributed histogram required by a given *split*. This principle has been extended in [13] to ensure that a maximum number of histogram entries are stored in memory at the same instant. This ensures that the use of distributed histograms always fits in memory, even in the case of large *splits* or a large number of LSH iterations. The method consists in constructing the groups of consecutive records in a *split* such that each group contains at most $\mathbf{t}_{\max}$ records. This parameter is chosen so that each *Mapper* can store in memory a distributed histogram of size at most $\mathcal{Q} \times \mathbf{t}_{\max}$. Handling distributed histograms using groups do not require any new algorithm. Essentially, instead of storing only the split identifiers, the set of the pairs (*splitId*, *groupId*) is stored. Such a pair is called a *chunk* identifier, and a *chunk* is the corresponding portion of data. For the sake of simplicity, we assume that a *chunk* corresponds to a *split* in the following.

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, the communication templates use a frequency threshold parameter called $\mathbf{f}_{max}$. This parameter defines the number of records that a *Reducer* can handle during the similarity join step. Owing to this parameter, the records having a common join attribute value will be divided into several blocks such that the size of each block never exceeds a fixed user-defined value, so each block can be handled without memory overflow. This makes the *MRSF-join* algorithm scalable and insensitive to the effects of load imbalance, even for highly skewed data.

For a given join attribute value $x$, the communication templates will redistribute the input data blocks according to one of the two following cases:

a. $\mathbf{f}_x < \mathbf{f}_{max}$ : the records corresponding to low frequencies of join attribute values, these records have no effect on the load imbalance among processing nodes, so they are redistributed without any special processing, using a hashing approach.

b. $\mathbf{f}_{max} \leqslant \mathbf{f}_x$ : The join attribute values corresponding to high frequencies (i.e., values having generally a large effect on load imbalance). Data corresponding to these values are redistributed using a partition/replicate approach, as illustrated in the Figure 3. In this example, the data corresponding to the join attribute value $x$ is divided into several blocks.

In the Figure 3, the generated communication templates are arranged in rows and columns. Each cell corresponds to a block, that is, a part of the data corresponding to the join attribute value $x$. Each column corresponds to data transmitted to a **reduce** task. These tasks are identified starting from $t_0$, which is a random integer that can be derived from the join attribute value.

To ensure that the blocks are sorted in the correct order, appropriate MapReduce *key/value* pairs are used. The keys are composed of the join attribute value,

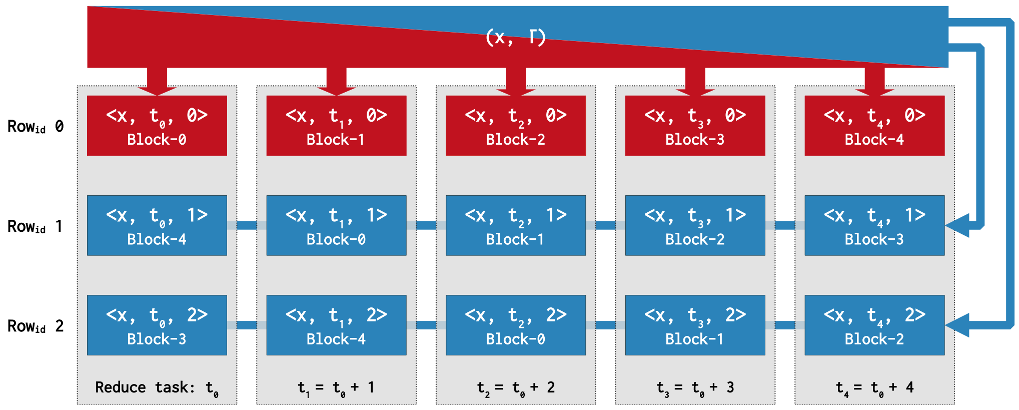


Figure 3: The communication templates for a highly frequent join attribute value in a self join case. Distributed blocks (red) are stored to compute the similarity join with replicated blocks (blue).

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

1. Store in memory the distributed blocks (i.e., the data blocks corresponding to the row $Row_{id} : 0$),

2. Compute the join within the stored blocks (blocks of $Row_{id} : 0$),

3. Compute the join between the stored blocks and the replicated blocks (i.e., the join among the blocks corresponding to row $Row_{id} : 0$ and those of rows $Row_{id} : i \geq 1$).

### *2.6. MRSS-join algorithm*

*MRSS-join* [13] is a set similarity join algorithm built on top of the MapReduce framework that uses LSH, distributed histograms and randomized communication templates to ensure balanced load and computation among the processing nodes. It proceeds in two steps with time and space guarantees as illustrated in the Figure 4 where:

❶ The histogram of the join is computed and redistributed to reduce computation to only relevant data while guaranteeing balanced communication patterns,

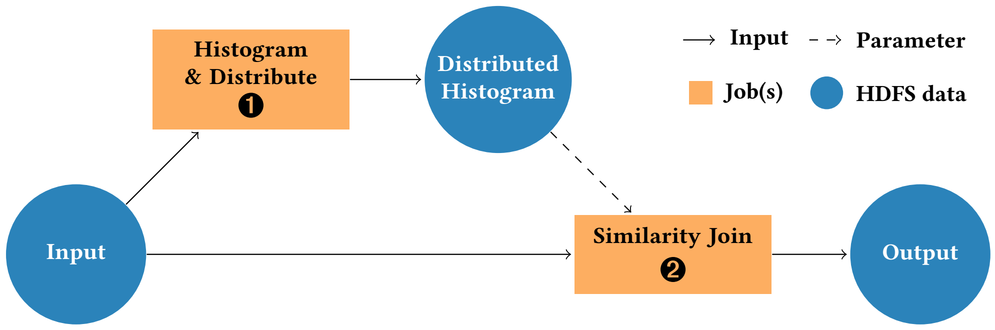


Figure 4: *MRSS-join* algorithm: MapReduce Similarity Join computation steps.

(i.e., the join attribute values are computed using MinHash for each record, along with their corresponding frequencies),

❷ Using distributed histograms, efficient and scalable communication templates are generated to balance the load of distance computations between all the pairs identified as similar,
(i.e., the join attribute values are recomputed and, using their corresponding frequencies, the original records are fairly distributed to compute the similarity join results).

## 3. *MRSF-join* algorithm computation steps

The *MRSF-join* algorithm proceeds in three steps; each step includes one or two MapReduce jobs. The Figure 5 shows the interactions between the different steps of the algorithm:

① The histogram of the join is computed and redistributed to guarantee balanced communication patterns regardless of data distribution,
(i.e., the join attribute values are computed using MinHash on the $q$-grams of each sliced sequence, along with their corresponding frequencies),

② Using distributed histograms, efficient and scalable communication templates are generated to filter and distribute the subproblems identifiers,
(i.e., the join attribute values are recomputed and, using their frequencies, the sequence sketches are fairly distributed to compute the pairs of sequence identifiers that are probably similar),

③ Using identified and distributed similar pairs, the edit distance between these identified pairs is computed using the original sequence to produce the similarity join results.

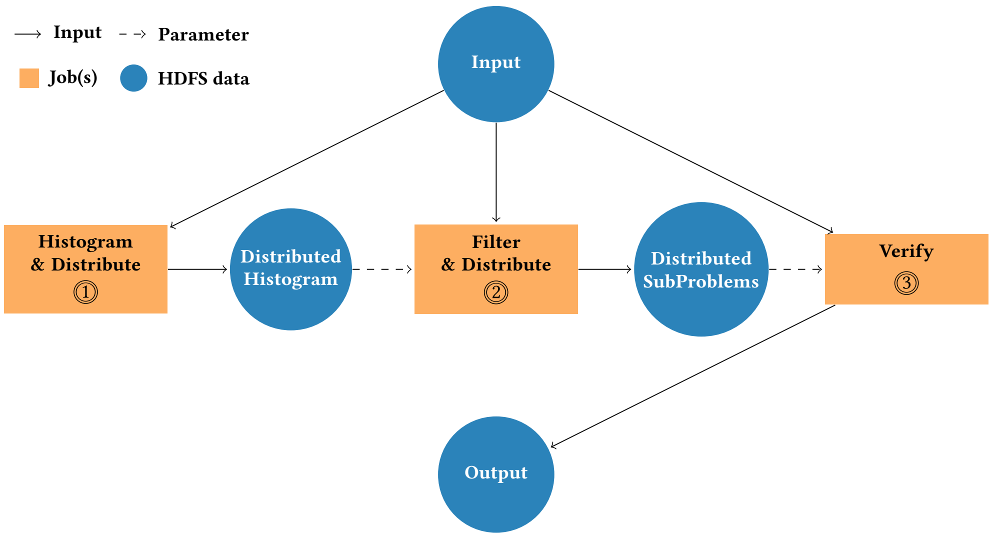


Figure 5: *MRSF-join* algorithm: MapReduce Sequence Filtering computation steps.

The main difference with the *MRSS-join* algorithm lies in the fact that an additional filtering step is inserted before the distances are computed, allowing many LSH iterations to be processed while minimizing the cost of the data distribution by only sending sketches of sequences. During the filtering step ②, a subproblem identifier, i.e., a pair containing the sequence identifiers, is generated for each pair of sequences whose distance must be computed. All these subproblems are then distributed to be verified using the original sequences. In summary, the step ❷ of *MRSS-join* is divided into two steps, the first one is used to filter, to generate and to redistribute the subproblem identifiers ②, and the second one to verify the distance of all the subproblems ③.

To compute the similarity join, $\Gamma \bowtie_\lambda \Gamma$, of a dataset $\Gamma$ for the threshold distance $\lambda$, we assume that the input dataset is divided into *splits* of data. These *splits* are stored in the Hadoop Distributed File System (HDFS) and replicated on several nodes for reliability reasons. Throughout this paper, we use the following notations for a dataset $\Gamma$:

| | |
|---|---|
| $\|\Gamma\|$ | The number of records in the dataset $\Gamma$, |
| $|\Gamma|$ | The number of blocks of data of $\Gamma$, |
| $\Gamma_i^{\text{map}}$ | The *Fragment* (set of *splits*) of $\Gamma$ affected to *Mapper* $i$, |
| $\Gamma_i^{\text{red}}$ | The *Fragment* of $\Gamma$ affected to *Reducer* $i$, |
| $\mathcal{S}(\Gamma)$ | The set of identifiers of *splits* of $\Gamma$, |
| $\|\mathcal{S}(\Gamma)\|$ | The number of identifiers of *splits* of $\Gamma$. |

| | |
|---|---|
| | HDFS split size is fixed at 256MB, |
| $\Gamma_j^{\text{split}} \mid j \in \mathcal{S}(\Gamma)$ | The *split* corresponding to the identifier $j$, |
| $\mathcal{Q}$ | The number of LSH iterations, |
| $\mathcal{K}$ | The number of concatenations of sequence $q$-grams, |
| $\mathcal{M}$ | The number of *Mappers*, |
| $\mathcal{R}$ | The number of *Reducers*, |
| $\mathrm{c}_{\mathbf{r/w}}$ | Read/write cost of a page of data from the Distributed File System (DFS), |
| $\mathrm{c}_{\mathbf{c}}$ | Communication cost per page of data, |
| $\chi(\Gamma)$ | Join attribute values of the dataset $\Gamma$, |
| $\mathrm{Hist}(\Gamma_i^{\text{map}})$ | The histogram of a fragment $\Gamma_i^{\text{map}}$; i.e., each join attribute value of $\chi(\Gamma_i^{\text{map}})$ is paired with its fragment frequency, |
| $\mathrm{Hist}_i^{\text{map}}(\Gamma)$ | The part of $\mathrm{Hist}(\Gamma)$ affected to *Mapper* $i$, |
| $\mathrm{Hist}_i^{\text{red}}(\Gamma)$ | The part of $\mathrm{Hist}(\Gamma)$ affected to *Reducer* $i$, |
| $\mathrm{Hist}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})$ | The histogram of the join, see Definition 9, |
| $\mathrm{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})$ | The part of $\mathrm{Hist}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})$ corresponding to a split $\Gamma_j^{\text{split}}$, |
| $\overline{\Gamma}$ | Part of $\Gamma$ corresponding to relevant join attribute values, |
| $\mathrm{Est}(\overline{\Gamma}_i^{\text{map}})$ | Estimator of a fragment $\overline{\Gamma}_i^{\text{map}}$, i.e., each record identifier that occurs via a join attribute value in $\mathrm{Hist}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})$ is paired with information for filtering, |
| $\mathrm{ID}(\overline{\overline{\Gamma}} \bowtie_\sigma \overline{\overline{\Gamma}})$ | Subproblem identifiers of the join $\overline{\overline{\Gamma}} \bowtie_\sigma \overline{\overline{\Gamma}}$, i.e., the pairs of sequence identifiers requiring that the distance must be computed, |
| $\overline{\overline{\Gamma}} \bowtie_\sigma \overline{\overline{\Gamma}}$ | Similarity join evaluated by the algorithm, i.e., all original data pairs that are present in the subproblem identifiers of the join, |
| $\overline{\overline{\Gamma}}$ | The part of $\Gamma$ present in the subproblem identifiers of the join, |
| $\overline{\overline{\Gamma}} \bowtie_\lambda \overline{\overline{\Gamma}}$ | The similarity join result produced by the algorithm. |

We assume that, the MinHash functions are randomly and uniformly selected and stored in the HDFS before the start of the **map** phase of the steps ① and ②. The MinHash function is implemented using Zobrist hashing [27]. It has been shown theoretically and practically that the Zobrist hashing has strong MinHash properties with fast performance in practice [28, 29]. In the remainder of this section, we describe the *MRSF-join* algorithm in detail while giving a cost analysis for each computation step. Throughout the paper, the $\mathcal{O}(...)$ notation only hides small constant factors which depend on program's implementation but neither on data nor on processing machine parameters.

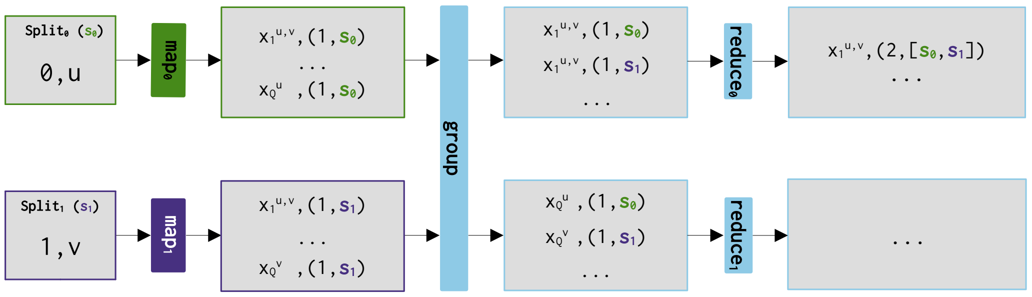
### *3.1. MRSF-join: Histogram computation step*

The histogram computation step is described in the Algorithm 1.A and a working example is illustrated in the Example 1.A. For the sake of clarity, the examples show only two sequences $u$ and $v$ belonging respectively to two *splits* $s_0$ and $s_1$. To compute the frequency of each join attribute value, the **map** phase emits a *key/value* pair containing the frequency and the *split* identifiers for each record, and for each join attribute value $x$. In the examples, we only track the join attribute value, $x_1^{u,v}$, that produces a join result. This value is produced by hashing the corresponding MinHash values of the sliced sequence. In addition, if the set of *split* identifiers does not fit in the memory, it is possible to send the frequency and the *split* identifiers apart to compute the global frequency before the union of the identifiers. For more details on the implementation of this memory extension, we refer the reader to the paper [13].

**Algorithm 1.A:** Histogram computation step (①.a)

```
Map: <id, u> → List(<$x_i$, (1, $\mathrm{split_{id}}$)>)
    init:
        Read from HDFS the $\mathcal{Q}$ LSH functions.
    $\mathrm{split_{id}}$ ← getSplitId();
    Compute the $\mathcal{Q}$ join attribute values, $x_i$ of the record $u$ for $i = 1 ... \mathcal{Q}$
    For each join attribute value $x_i$:
        Emit a pair <$x_i$, (1, $\mathrm{split_{id}}$)>
Reduce: <$x_i$, List(1, $\mathrm{split_{id}}$)> → <$x_i$, ($\mathbf{f}_{x_i}$, Set($\mathrm{split_{id}}$))>
    Compute the global frequency $\mathbf{f}_{x_i}$ of the join attribute value.
    Compute the set of $\mathrm{split_{id}}$.
    If ($\mathbf{f}_{x_i} > 1$):
        Emit the pair <$x_i$, ($\mathbf{f}_{x_i}$, Set($\mathrm{split_{id}}$))>.
```

*Analysis of the histogram computation step:* The cost model uses the $\mathcal{O}(\dots)$ notation, which hides only small constant factors: they depend only on the program's implementation, but neither on input datasets nor on processing machine parameters. It is assumed that there are $\mathcal{M}$ nodes in the cluster performing the



Example 1.A: An example of the execution of the histogram computation step: The two input sequences $u$ and $v$ identified respectively by 0 and 1, belonging respectively to two splits $s_0$ and $s_1$, produce $\mathcal{Q}$ join attribute values. Each join attribute value corresponds to the value of an LSH iteration, that is, the application of a concatenated LSH hash function on a record. Records with the same value for an iteration are denoted by superscripts. The value $x_1^{u,v}$ means that the record $u$ and $v$ have the same value for the first join attribute.

**map** tasks, so the cost of processing all the **map** tasks depends on the slowest machine.

On node $i$, the *Mapper* reads its assigned fragment at a cost $\mathrm{c}_{\mathbf{r/w}} * |\Gamma_i^{\text{map}}|$. By using a single MinHash function with $\mathcal{Q} * \mathcal{K}$ partitions, the cost to compute the $\mathcal{Q}$ join attribute values depends on the length of the sequences. Let $\|\tilde{U}\|$ be the average length of the input sequences. The term $\|\tilde{U}\| * \|\Gamma_i^{\text{map}}\|$ represents the cost to compute the join attribute values for each record in the fragment. The cost of sorting data, on the *Mapper* $i$, is represented by the term $\mathcal{Q} * \|\Gamma_i^{\text{map}}\| * \log(\mathcal{Q} * \|\Gamma_i^{\text{map}}\|)$. Data are then sent to the *Reducers* at the cost of $\mathrm{c}_{\mathbf{c}} * |\text{Hist}(\Gamma_i^{\text{map}})|$. Therefore, this **map** phase requires at most:

$$\text{Time}(1.\text{A.Mapper}) = \mathcal{O}\Big(\max_{i=0}^{\mathcal{M}} \mathrm{c}_{\mathbf{r/w}} * |\Gamma_i^{\text{map}}| + \|\tilde{U}\| * \|\Gamma_i^{\text{map}}\| + \mathcal{Q} * \|\Gamma_i^{\text{map}}\| * \log(\mathcal{Q} * \|\Gamma_i^{\text{map}}\|) + \mathrm{c}_{\mathbf{c}} * |\text{Hist}(\Gamma_i^{\text{map}})|\Big).$$

Respectively, for the **reduce** phase, it is assumed that there are $\mathcal{R}$ nodes in the cluster performing the **reduce** tasks. Each node receives its fragment of the histogram of the join and computes the sum of the frequencies and the union of each *split* identifier. The cost of this step is represented by the term $\|\text{Hist}_i^{\text{red}}(\Gamma)\|$. The cost to write *Reducer*'s $i$ output is represented by the term $\mathrm{c}_{\mathbf{r/w}} * |\text{Hist}_i^{\text{red}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|$. Therefore,

$$\text{Time}(1.\text{A.Reducer}) = \mathcal{O}\Big(\max_{i=0}^{\mathcal{R}} \|\text{Hist}_i^{\text{red}}(\Gamma)\| + \mathrm{c}_{\mathbf{r/w}} * |\text{Hist}_i^{\text{red}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|\Big).$$

This histogram computation step will have the following cost:

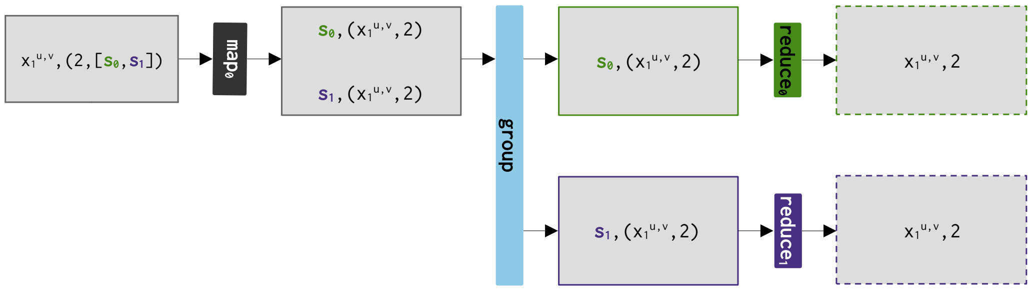
$$\mathsf{Time}(1.A.) = \mathsf{Time}(1.A.Mapper) + \mathsf{Time}(1.A.Reducer).$$

### *3.2. MRSF-join: Histogram redistribution step*

The histogram of the join is then redistributed using the Algorithm 1.B. A working example is illustrated in the Example 1.B. For a join attribute value, the **map** phase sends a *key/value* pair for each *split* in which it appears. Since the algorithm partitions the pairs according to the *split* identifier, each **reduce** task output corresponds to the distributed histogram required by a *split*.

**Algorithm 1.B:** Histogram redistribution step (①.b)

```
Map: <$x_i$, ($\mathbf{f}_{x_i}$, Set(split$_{\mathrm{id}}$))> $\rightarrow$ List(<split$_{\mathrm{id}}$, ($x_i$, $\mathbf{f}_{x_i}$)>)
    For each id $\in$ Set(split$_{\mathrm{id}}$) :
        Emit a pair <id, ($x_i$, $\mathbf{f}_{x_i}$)>.
Reduce: <split$_{\mathrm{id}}$, List($x_i$, $\mathbf{f}_{x_i}$)> $\rightarrow$ Set(<$x_i$, $\mathbf{f}_{x_i}$>)
    For each value in the received list :
        Emit a pair <$x_i$, $\mathbf{f}_{x_i}$>.
```



Example 1.B: An example of the execution of the histogram redistribution step. The value $x_1^{u,v}$ was present in the splits $s_0$ and $s_1$, so the corresponding histogram tuple is redistributed to the tasks 0 and 1.

*Analysis of the histogram redistribution step:* Each *Mapper* $i$ reads its fragment of the histogram of the join at the cost $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{Hist}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|$. For each entry in the histogram, at most $\|\mathcal{S}(\Gamma)\|$ pairs are sent if the corresponding join attribute value appears in all the *splits*. The cost of this step is the term $\|\mathcal{S}(\Gamma)\| * \|\mathrm{Hist}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|$. The cost to sort data on the *Mapper* $i$ is represented by the term $\|\mathcal{S}(\Gamma)\| * \|\mathrm{Hist}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| * \log(\|\mathcal{S}(\Gamma)\| * \|\mathrm{Hist}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|)$. The term $\mathsf{c}_{\mathbf{c}} * |\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \|\mathcal{S}(\Gamma)\|$ represents the cost to transmit data to *Reducers*. Therefore, this **map** phase requires at most:

$$\begin{aligned}\text{Time}(1.\text{B.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| + \|\mathcal{S}(\Gamma)\| * \|\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| \\ + \|\mathcal{S}(\Gamma)\| * \|\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| * \log(\|\mathcal{S}(\Gamma)\| * \|\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|) \\ + \mathsf{c}_{\mathbf{c}} * |\text{Hist}_i^{\text{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \|\mathcal{S}(\Gamma)\|\Big).\end{aligned}$$

Each **reduce** task processed by the node $i$ corresponds to the distributed histogram of a *split*. Since the number of **reduce** tasks is set to be equal to the number of *split*, each of these tasks has an identifier $j$ such that $j \in \mathcal{S}(\Gamma)$. Consequently, each *Reducer* $i$ processes a distinct subset of **reduce** tasks $\mathcal{S}_i^{\text{red}}(\Gamma) \subseteq \mathcal{S}(\Gamma)$.

$$\text{Time}(1.\text{B.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \sum_{j \,\in\, \mathcal{S}_i^{\text{red}}(\Gamma)} \|\text{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| + \mathsf{c}_{\mathbf{r/w}} * |\text{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|\Big).$$

The term $\|\text{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|$ represents the processing cost of a **reduce** task $j$, on a *Reducer* $i$. The term $\mathsf{c}_{\mathbf{r/w}} * |\text{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|$ represents the cost to store the output. Thus, this histogram redistribution step will have the following cost:

$$\text{Time}(1.\text{B.}) = \text{Time}(1.\text{B.Mapper}) + \text{Time}(1.\text{B.Reducer}).$$

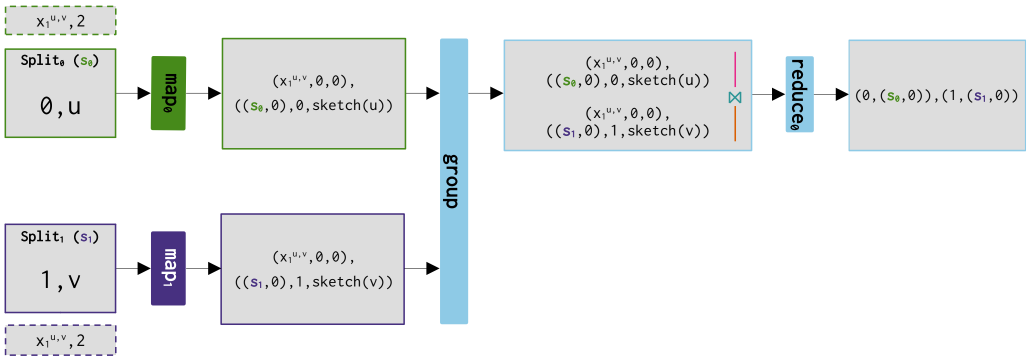
### *3.3. MRSF-join: Sequence filtering step*

The sequence filtering step is described in the Algorithm 2.A and illustrated in the Example 2.A. Using the distributed histograms, the **map** phase sends for each record and for each remaining join attribute value a *key/value* pair. The *key* is composed of the join attribute value, supplemented by some information to handle highly frequent join attribute values, as explained in Section 2.5. The *value* contains the record localization, the record length and a sketch of the record. Basically, the record localization is the *split* identifier and an offset corresponding to the position of the record in the *split*. The sketch of a record is constructed using LSH as explained in Section 2.4. The **reduce** phase computes the join and emits a pair, called a subproblem identifier, that contains the record localizations for each pair of sequences that has not been pruned by the sketch and length filter, and whose edit distance on the original sequences must be computed. It is worth noting that a subproblem identifier may appear multiple times in the output. To remove these duplicates, it would be necessary to send the join attribute values for each iteration. However, the large number of iterations makes it prohibitively expensive. In addition, this step minimizes the amount of data transmitted over the network by drastically reducing the number of subproblems without sending the original sequences. In the end, this allows a large number of iterations to be processed.

**Algorithm 2.A:** Sequence Filtering step (②.a)

```
Map: <id, u> $\rightarrow$ List(<(x, $\mathrm{reducer_{id}}$, $\mathrm{row_{id}}$), (id, localization, length, sketch)>)
    Init:
        Read from HDFS the $\mathcal{Q}$ LSH functions.
    Read and store the corresponding distributed histogram of the current $\mathrm{split_{id}}$.
    Compute the $\mathcal{Q}$ join attribute values $x_i$ of the record $u$ for $i = 1 \dots \mathcal{Q}$.
    Retain only the join attribute values that appear in the distributed histogram.
    If there are any remaining join attribute value:
        localization $\leftarrow$ getLocalization();
        length $\leftarrow$ $\|u\|$;
        Compute the LSH sequence sketch as described in Section 2.4.1.
        For each remaining join attribute value $x_i$:
            Emit pairs using the communication templates described in Section 2.5.
Partition: <($x_i$, $\mathrm{reducer_{id}}$, $\mathrm{row_{id}}$), (id, localization, length, sketch)> $\rightarrow$ $\mathrm{reducer_{id}}$ :
  Integer
    Redirect each pair according to the value $x_i$ or the destination task $\mathrm{reducer_{id}}$,
      randomly generated during the map phase, following the different scenarios of
      the communication schemes.
Reduce: <($x_i$, $\mathrm{reducer_{id}}$, $\mathrm{row_{id}}$), List(id, localization, length, sketch)>
        $\rightarrow$ List(<($\mathrm{id_u}$, $\mathrm{localization_u}$), ($\mathrm{id_v}$, $\mathrm{localization_v}$)>)
    Compute the join using communication templates.
    For each pair (u, v) corresponding to ($\mathrm{id_u}$, $\mathrm{id_v}$) such that $\mathrm{id_u} \neq \mathrm{id_v}$:
        Exit if their length ratio is higher than the edit distance threshold.
        Exit if the sketch distance is too low as described in Section 2.4.
        Emit a pair <($\mathrm{id_u}$, $\mathrm{localization_u}$), ($\mathrm{id_v}$, $\mathrm{localization_v}$)>.
```

*Analysis of the sequence filtering step:* The term $\mathrm{c}_{\mathbf{r/w}} * |\Gamma_i^{\mathrm{map}}|$ represents the cost to read the fragment on *Mapper* $i$. For each *split* in its fragment, identified by $j \in \mathcal{S}(\Gamma_i^{\mathrm{map}})$, the *Mapper* $i$ reads the corresponding distributed histogram at a cost given by the term $\mathrm{c}_{\mathbf{r/w}} * |\mathrm{Hist}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|$. The term $\|\tilde{U}\| * \|\Gamma_i^{\mathrm{map}}\|$ represents the cost to compute the join attribute values and the sketches for each record in the fragment. The cost of sorting the data on the *Mapper* $i$ is represented by the term $\mathcal{Q} * \|\overline{\Gamma}_i^{\mathrm{map}}\| * \log(\mathcal{Q} * \|\overline{\Gamma}_i^{\mathrm{map}}\|)$. The term $\mathrm{c}_{\mathbf{c}} * |\mathrm{Est}(\overline{\Gamma}_i^{\mathrm{map}})|$ corresponds to the cost of transmitting the data to the *Reducers*. Therefore, the cost of this


x₁^{u,v},2
Split₀ (s₀)
0,u
map₀
(x₁^{u,v},0,0), ((s₀,0),0,sketch(u))
group
(x₁^{u,v},0,0), ((s₀,0),0,sketch(u))
⋈
(x₁^{u,v},0,0), ((s₁,0),1,sketch(v))
reduce₀
(0,(s₀,0)),(1,(s₁,0))
Split₁ (s₁)
1,v
map₁
(x₁^{u,v},0,0), ((s₁,0),1,sketch(v))
x₁^{u,v},2

Example 2.A: An example of the execution of the filtering step: The records $u$ and $v$, identified respectively by 0 and 1, and belonging to the *splits* $s_0$ and $s_1$, emit at most $\mathcal{Q}$ key/value pairs corresponding to their $\mathcal{Q}$ join attribute values. The communication schemes for highly frequent join attribute values are not used in this example, since the value $x_1^{u,v}$ appears only twice in the input. The values of "reducerId" and "rowId" of the transmitted keys are therefore set to 0. In this example, we represent the localization by the tuples ($s_0$,0) and ($s_1$,0) for the records $u$ and $v$. This means that the record $u$ is in the *split* $s_0$ at the position 0. During the **reduce** phase, the records sketches are compared. In this example, we assume that the sketches of $u$ and $v$ are similar and that the edit distance must be computed during the verification step.

step is as follows:

$$\mathsf{Time}(2.\mathsf{A.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\Gamma_i^{\text{map}}| + \sum_{j \,\in\, \mathcal{S}(\Gamma_i^{\text{map}})} (\mathsf{c}_{\mathbf{r/w}} * |\text{Hist}_j^{\text{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|) + \|\tilde{U}\| * \|\Gamma_i^{\text{map}}\| + \mathcal{Q} * \|\overline{\Gamma}_i^{\text{map}}\| * \log(\mathcal{Q} * \|\overline{\Gamma}_i^{\text{map}}\|) + \mathcal{Q} * \mathsf{c}_{\mathbf{c}} * |\text{Est}(\overline{\Gamma}_i^{\text{map}})| \Big).$$

The term $\|\text{Est}(\overline{\Gamma}_i^{\text{red}}) \bowtie \text{Est}(\overline{\Gamma}_i^{\text{red}})\|$ represents the cost to compute the join and filtering the dissimilar sequences using the lengths and the sequence sketches, whereas the term $\mathsf{c}_{\mathbf{r/w}} * |\text{ID}(\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}})|$ corresponds to the cost to write the output of the *Reducer* $i$. Therefore,

$$\mathsf{Time}(2.\mathsf{A.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \|\text{Est}(\overline{\Gamma}_i^{\text{red}}) \bowtie \text{Est}(\overline{\Gamma}_i^{\text{red}})\| + \mathsf{c}_{\mathbf{r/w}} * |\text{ID}(\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}})| \Big).$$

Thus, this filtering step will have the following cost:

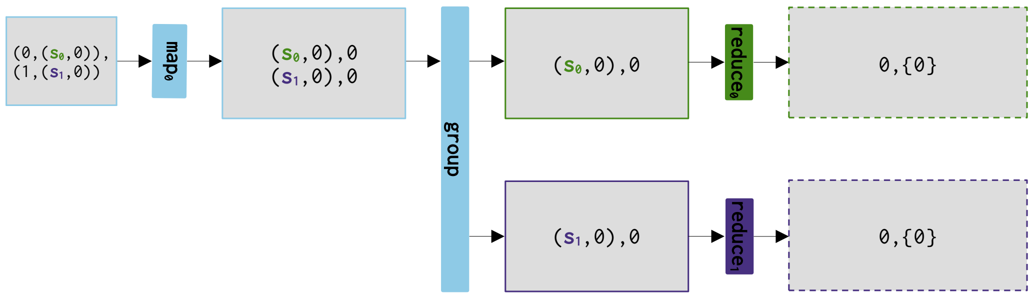
$$\mathsf{Time}(2.\mathsf{A.}) = \mathsf{Time}(2.\mathsf{A.Mapper}) + \mathsf{Time}(2.\mathsf{A.Reducer}).$$

### *3.4. MRSF-join: Subproblems identifiers redistribution step*

This step allows the identified similar pairs to be redistributed to their corresponding *splits* while removing the duplicates. It is described in the Algorithm 2.B and illustrated in the Example 2.B. In the case of very large datasets, the set of the received values may not fit in memory. The algorithm can be modified so that the emitted *value* is given in the *key*. In this way, the deduplication is performed during the shuffle phase and the set of the received values can be yielded in several batches that fit into memory.

**Algorithm 2.B:** Subproblems identifiers: redistribution step (②.b)

```
Map: <(id_u, localization_u), (id_v, localization_v)> → List(<localization, id_u>)
    Emit a pair <localization_u, id_u>.
    Emit a pair <localization_v, id_u>.
Partition: <localization, id_u> → Integer
    Return the split_id from the localization.
Reduce: <localization, List(id_u)> → <localization, Set(id_u)>
    Compute the set of received values to eliminate the duplicates.
    Emit a pair <localization, Set(id_u)>.
```



Example 2.B: An example of the execution of the redistribution step. The record $u$ is in the *split* $s_0$ at the position 0. It now remains to transmit the record $u$ on the key 0. For record $v$, which is in the split $s_1$ at the position 0, it will also have to be transmitted on key 0 in order to compute the distance between the two records during the verification step.

*Analysis of the redistribution step:* The term $c_{\mathbf{r/w}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \mathcal{Q}$ corresponds to the cost of reading the identified similar pairs on node $i$. The term $\mathcal{Q}$ is due to the fact that the identified similar pairs may be duplicated. The term $\mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| * \log(\mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|)$ represents the cost of sorting the data. The term $\mathsf{c}_{\mathbf{c}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \mathcal{Q}$ corresponds to the cost of transmitting data to the *Reducers*.

$$\mathsf{Time}(2.\mathsf{B.Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \mathcal{Q} + \mathsf{c}_{\mathbf{c}} * |\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})| * \mathcal{Q} \\ + \mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| * \log(\mathcal{Q} * \|\mathrm{ID}_i^{\mathrm{map}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|)\Big).$$

Similarly to the histogram redistribution step, the number of **reduce** tasks is set to be equal to the number of *splits*. The term $\mathcal{Q} * \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|$ represents the cost of eliminating duplicates for a **reduce** task $j$. The term $\mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|$ is the cost to store the reducer's output.

$$\mathsf{Time}(2.\mathsf{B.Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \sum_{j \,\in\, \mathcal{S}_i^{\mathrm{red}}(\Gamma)} \mathcal{Q} * \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\| + \mathsf{c}_{\mathbf{r/w}} * |\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})|\Big).$$

Therefore, this redistribution of identified similar pairs will have the following cost:

$$\mathsf{Time}(2.\mathsf{B.}) = \mathsf{Time}(2.\mathsf{B.Mapper}) + \mathsf{Time}(2.\mathsf{B.Reducer}).$$

### *3.5. MRSF-join: Verification step*

The final step is described in the Algorithm 3 and illustrated in the Example 3. The **map** phase sends a *key/value* pair for each identified similar pair containing the record. The **reduce** phase computes the edit distance for each identified similar pair using the original sequences to produce the similarity join results.

*Analysis of the verification step:* The cost of the **map** phase is as follows:

$$\mathsf{Time}(3.\mathsf{Mapper}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathsf{c}_{\mathbf{r/w}} * |\Gamma_i^{\mathrm{map}}| + \|\Gamma_i^{\mathrm{map}}\| + \sum_{j \,\in\, \mathcal{S}(\Gamma_i^{\mathrm{map}})} \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|\Big).$$

The term $\mathsf{c}_{\mathbf{r/w}} * |\Gamma_i^{\mathrm{map}}|$ represents the cost of reading the fragment on *Mapper* $i$. For each *split*, the corresponding identified similar pairs are read at a cost of $\mathsf{c}_{\mathbf{r/w}} * \|\mathrm{ID}_j^{\mathrm{split}}(\overline{\Gamma} \bowtie_\sigma \overline{\Gamma})\|$.

The cost of the **reduce** phase is as follows:

$$\mathsf{Time}(3.\mathsf{Reducer}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{R}} \mathsf{c}_{\mathbf{c}} * |\overline{\overline{\Gamma}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\Gamma}_i^{\mathrm{red}}| + \|\overline{\overline{\Gamma}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\Gamma}_i^{\mathrm{red}}\| * \log(\|\overline{\overline{\Gamma}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\Gamma}_i^{\mathrm{red}}\|) \\ + \mathsf{t}_{\mathbf{D}} * \|\overline{\overline{\Gamma}}_i^{\mathrm{red}} \bowtie_\sigma \overline{\Gamma}_i^{\mathrm{red}}\| + \mathsf{c}_{\mathbf{r/w}} * |\overline{\overline{\Gamma}}_i^{\mathrm{red}} \bowtie_\lambda \overline{\Gamma}_i^{\mathrm{red}}|\Big).$$

**Algorithm 3:** Verification step (③)

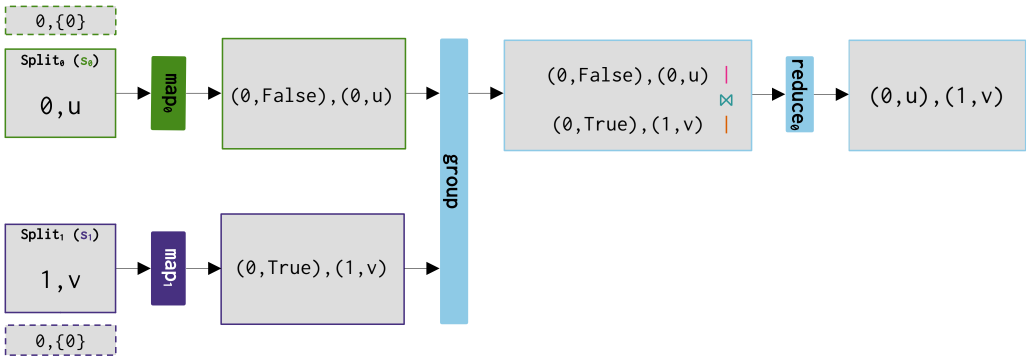
```
Map: <id, u> → List(<($s$, 0|1), (id, u)>)
    Read the identified similar pairs corresponding to the current record if there are
      any.
    For each identified similar pairs $s$:
        If $s$ is equal to id:
            Emit a pair <($s$, 0), (id, u)>.
        Else:
            Emit a pair <($s$, 1), (id, u)>.
Partition: <($s$, 0|1), (id, u)> → Integer
    Redirect each pair using the $s$ value.
Reduce: <($s$, 0), (id, u)>
    Store the record (id, u).
Reduce: <($s$, 1), List(<id, v>)> → List(<($id_u$, u), ($id_v$, v)>)
    For each received value:
        Compute the edit distance with the stored record.
        If the distance is lower than the threshold:
            Emit a pair <($id_u$, u), ($id_v$, v)>.
```

The term $c_{\mathbf{c}} * |\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}|$ corresponds to the cost of transmitting data to the *Reducer* $i$. The term $\|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\| * \log(\|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\|)$ represents the cost of sorting the data. Let $t_{\mathbf{D}}$ be the cost of computing the edit distance. The cost to compute the similarity join on the *Reducer* $i$ is represented by the term $t_{\mathbf{D}} * \|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\|$. The cost of storing the similarity join output is represented by the term $c_{\mathbf{r/w}} * |\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\lambda \overline{\overline{\Gamma}}_i^{\text{red}}|$.

Thus, the overall cost of this verification step is:

$$\mathsf{Time}(3.) = \mathsf{Time}(3.\mathsf{Mapper}) + \mathsf{Time}(3.\mathsf{Reducer}).$$

*Analysis of the MRSF-join algorithm:* The cost of the *MRSF-join* algorithm is therefore the sum of all previous steps, which can be rewritten as the following cost, since the histogram of the join and its redistribution is by definition bounded by the histogram of each *split*. Furthermore, the redistribution of the identified similar pairs is also by definition bounded by the evaluated similarity



Example 3: The execution of the example verification step. Using the redistributed identifiers from the previous step, the records $u$ and $v$ are transmitted on the same key 0, and the edit distance is computed and compared to the user-defined threshold. In this example, we assume that the computed distance is below the threshold, so a key/value pair containing the two records is produced.

join, since it only sends the localized identifiers. Therefore,

$$\begin{aligned}\mathsf{Time}(\textit{MRSF-join}) = \mathcal{O}\Big( \max_{i=0}^{\mathcal{M}} \mathrm{c}_{\mathbf{r/w}} * |\Gamma_i^{\text{map}}| + \|\tilde{U}\| * \|\Gamma_i^{\text{map}}\| + \mathcal{Q} * \|\Gamma_i^{\text{map}}\| * \log(\mathcal{Q} * \|\Gamma_i^{\text{map}}\|) \\ + \mathrm{c}_{\mathbf{c}} * |\text{Hist}(\Gamma_i^{\text{map}})| + \mathrm{c}_{\mathbf{c}} * |\text{Est}(\overline{\Gamma}_i^{\text{map}})| \\ + \max_{i=0}^{\mathcal{R}} \|\text{Hist}_i^{\text{red}}(\Gamma)\| + \|\text{Est}(\overline{\Gamma}_i^{\text{red}}) \bowtie \text{Est}(\overline{\Gamma}_i^{\text{red}})\| + \mathrm{c}_{\mathbf{c}} * |\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}| \\ + \|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\| * \log(\|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\|) + \mathrm{t}_{\mathbf{D}} * \|\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\sigma \overline{\overline{\Gamma}}_i^{\text{red}}\| + \mathrm{c}_{\mathbf{r/w}} * |\overline{\overline{\Gamma}}_i^{\text{red}} \bowtie_\lambda \overline{\overline{\Gamma}}_i^{\text{red}}| \Big).\end{aligned}$$

For large similarity join results, *MRSF-join* is dominated by the similarity join computations, which corresponds to the *MRSF-join* verification step. However, for small similarity join results, *MRSF-join* processing is either proportional to the input dataset size or the evaluated similarity join.

## 4. *MRSF-join* Experiments

In this section, we discuss the efficiency and the strength of our theoretical analysis by experimenting the *MRSF-join* algorithm on real world and synthetic datasets. We measured the efficiency of the *MRSF-join* algorithm in terms of execution time and data communication costs, as well as the filtering quality in terms of the *recall* and the *precision* parameters. The *recall* parameter corresponds to the fraction of the number of pairs of similar sequences correctly produced by the algorithm over the exact number of similar sequence, whereas the *precision* corresponds to the fraction of the number of pairs of similar sequences correctly produced over the number of comparisons. We evaluate these two measures during the distance computation, i.e., the similarity join output does not contain any record pair with a distance greater than the given threshold. The experiments were performed using Hadoop 3.3.4 framework on a heterogeneous cluster of 6 machines. There are four machines with the following specifications: Intel(R) Xeon(R) CPU E5-2650 @2.60 GHz, 64 GB of memory and two HDD with a capacity of 1 TB each. In addition, two other machines are equipped with an Intel(R) Xeon(R) Gold 6248R CPU @3.00GH, 256 GB of memory and two HDD disk of 8TB. The nodes are connected by a 1Gbit/s *Ethernet* network. The map output compression is enabled using Snappy codec except for the filtering and the verification steps that use the Gzip codec. The output compression is achieved using the Gzip codec only. The heap memory configuration for the **map** and **reduce** tasks was fixed to 2GB and 6GB respectively.

### *4.1. Synthetic dataset experiments*

In order to use different sized datasets for the experiments, we employed a synthetic data generator that allows us to specify the number of similarity join outputs. To this end, the generator takes as parameters the number of clusters and their sizes, as well as an edit distance threshold. For each cluster, the algorithm generates a random sequence which is used as a template. The cluster sequences is generated using this template and randomly altering it (using insertion, deletion, substitution operations) according to the given threshold. This leads to clusters with a normalized edit distance between 10 % and 20 %. The dataset is supplemented with noise, i.e., a number of random sequences that do not produce any similarity join output.

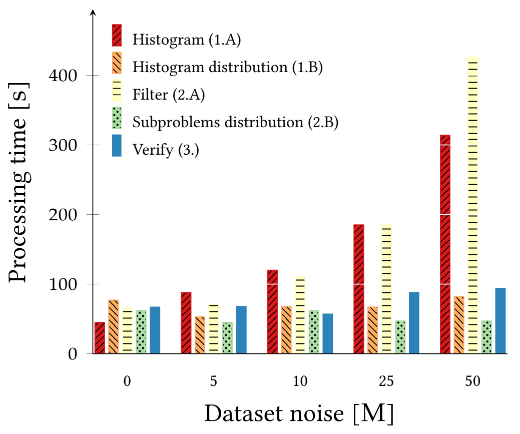
To analyze the filtering quality of the algorithm according to the average length of the sequences, we prepared several datasets by varying the sequence length from 25 to 10,000. For each dataset, we generated 1,000 clusters of size 10 supplemented by 1,000 random sequences. The results are presented in Table 1. We observe that the *MRSF-join* algorithm is very efficient and makes it possible to achieve at least 90 % for the *recall* parameter for the sequences longer than 50. Whereas for short sequences, its performance is relatively low to generate all similar pairs: this bias can be explained, for small sequences, by the use of a single MinHash function. In addition, although the use of three concatenations provides effective filtering, as reflected in the *precision* values, it does not allow small sequences to be properly handled. Moreover, we recall that *MRSF-join* only sends the sketch of the sequence during the filtering phase, which implies, for very small sequences, to send as much data as the original sequence. Therefore, we suggest handling this case differently by reducing the number of concatenations and the LSH iterations.

| Sequence lengths | 25 | 50 | 75 | 100 | 250 | 500 | 750 | 1000 | 2500 | 5000 | 7500 | 10000 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Number of input | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 | 11000 |
| Number of output | 19910 | 22475 | 22003 | 19126 | 20479 | 20890 | 20933 | 19979 | 20164 | 19680 | 20575 | 20853 |
| Number of comparisons | 22924 | 30566 | 29146 | 29919 | 32275 | 33067 | 33397 | 32931 | 33235 | 33385 | 33640 | 32964 |
| *recall* | 0.74 | 0.91 | 0.93 | 0.94 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 | 0.99 |
| Filtering *precision* | 0.87 | 0.74 | 0.75 | 0.67 | 0.63 | 0.63 | 0.63 | 0.61 | 0.61 | 0.59 | 0.61 | 0.63 |

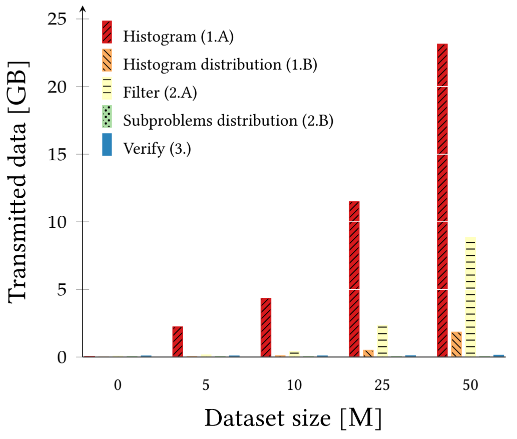
Table 1: The filtering quality of the *MRSF-join* algorithm in terms of *recall* and *precision* parameters and the number of computed distances by varying the average length of sequences.

To illustrate the efficiency of filtering for large datasets, we prepared several datasets by varying only the amount of noise. Basically, all the generated datasets contain exactly the same clusters of sequences. There are 1,000 sequence clusters of size 100. For the sake of illustration, the "0" dataset in the Table 2 is not supplemented with noise, whereas the "50" dataset is supplemented with 50 million of random sequences. All the generated sequences have an average length of 1,000, and each million of sequence represents approximately 1GB on disk. To allow a fair comparison of the different datasets by varying only the noise, the random seeds are identical for each dataset.

The Figure 6 presents the performance of the *MRSF-join* algorithm in terms of processing time and data redistribution cost. The processing time of *MRSF-join* remains very low, even for the "50" dataset. This is because, although the input data is read three times, only relevant data (i.e., that can produce a result) is transmitted during the communication phases. In particular, this illustrates that the transmitted data to compute the histogram of the join increases linearly with the number of input sequences. Regarding the computation time for the filtering step, it depends essentially on the number of join attribute values that have been discarded during the histogram computation step. Since the size of the similarity join output is very small, most of the processing time is spent on the histogram and filtering computations. Additionally, we observe that the transmitted data for the verification step remains constant, this is due to the fact that LSH and the filtering step drastically reduce the number of comparisons and these comparisons are usually limited to the most relevant sequence pairs, that is, the sequences from the same cluster.



(a) Processing times of the *MRSF-join* algorithm.



(b) Transmitted data of the *MRSF-join* algorithm.

Figure 6: The filtering quality of the *MRSF-join* algorithm when varying the size of the supplementary noise.

The Table 2 presents the filtering quality of the algorithm, providing a detailed analysis. The “sketch filtering” is the number of pruned sequence pairs during the filtering step. It is worth noting that this number includes some duplicates, since identified similar pair deduplication is performed during the redistribution step. Furthermore, we observe that the *MRSF-join* algorithm produces almost all similarity join results achieving 99 % for the *recall* parameter while drastically reducing the number of distance computations as reflected by the good values of the *precision* parameter which are on averages at 48 % : this means that only half of the pairs compared are not relevant and do not appear in the sequence similarity join result (we recall that, the naive similarity join algorithm requires a pairwise comparison of all the sequences of the input dataset and therefore up to $\sim 1.2 * 10^{15}$ comparisons are required for the dataset “50” using the naive algorithm).

| *Dataset noise [M]* | 0 | 5 | 10 | 25 | 50 |
|---|---|---|---|---|---|
| *Number of input* | 100,000 | 5,100,000 | 10,100,000 | 25,100,000 | 50,100,000 |
| *Sketch filtering* | 976,236 | 2,284,439 | 6,112,586 | 32,697,631 | 127,353,919 |
| *Number of output* | 923,916 | 923,916 | 923,916 | 923,916 | 923,916 |
| *Number of comparisons* | 1,915,057 | 1,915,567 | 1,916,983 | 1,926,660 | 1,960,962 |
| *Recall* | 0.995 | 0.995 | 0.995 | 0.995 | 0.995 |
| *Filtering precision* | 0.48 | 0.48 | 0.48 | 0.48 | 0.47 |

Table 2: The filtering quality of the *MRSF-join* algorithm in terms of sketch filtering, *recall*, *precision* and the number of computed distances while varying the amount of noise.

### *4.2. A comparison with the MRSS-join algorithm*

To demonstrate the efficiency of the *MRSF-join* algorithm compared to our previous approaches, we compare the number of computed distances, the processing time and communication costs between the *MRSS-join* algorithm and the *MRSF-join* algorithm using the dataset "50". The Table 3 presents the detailed summary. First, the *MRSF-join* algorithm produces slightly fewer similarity join results, this is due to the additional sketch filtering process that has been performed. However, it allows reducing the number of distance computations by a factor of 60 while minimizing the data redistribution cost. Secondly, the processing time of *MRSF-join* is reduced by a factor of 3 owing to the filtering step allowing to reduce the communication cost to only relevant sequences: the transmitted data is reduced by a factor of 20 by considering the sum of the transmitted data for the *MRSF-join* algorithm steps. In a nutshell, although the *MRSS-join* algorithm cannot fail to compute the similarity join, it remains less efficient for dealing with high Jaccard distance thresholds.

### *4.3. Metaclust protein dataset experiments*

To exhibit the *MRSF-join*'s ability to perform sequence similarity joins on very large datasets, we have experimented it on a subset of the Metaclust dataset [9]. The original dataset is composed of 1.59 billion protein sequence fragments predicted by Prodigal [30] in 2200 metagenomic and metatranscriptomic datasets [31, 32, 33]. For the following experiments, duplicate sequences in the input dataset are removed, since the exact join is a different problem. In addition, since the main objective was to compute the similarity join of long sequences, we discarded sequences shorter than 100. The resulting dataset is composed of

| *Algorithm* | | *MRSS-join* | *MRSF-join* |
|---|---|---|---|
| *Number of input* | | 50,100,000 | 50,100,000 |
| *Number of output* | | 927,113 | 923,916 |
| *Number of comparisons* | | 129,030,002 | 1,964,159 |
| *Recall* | | 0.998 | 0.995 |
| *Filtering precision* | | 0.007 | 0.472 |
| *Processing time* | [s] | 3,447 | 1,049 |
| *Transmitted data* | [GB] | 242 | 10 + 0.2 + 2 ($\sim 12.2$ GB) |

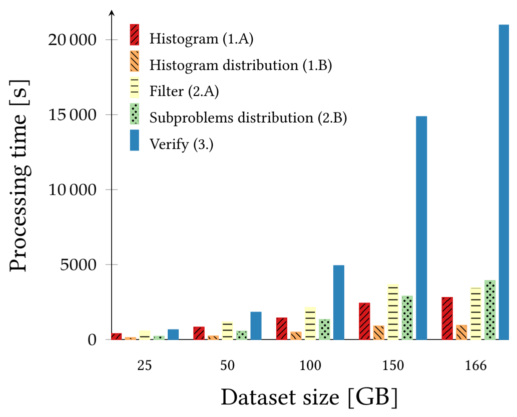
Table 3: The comparison of the *MRSS-join* and *MRSF-join* algorithms in terms of the number of computed distances, processing time and data communication cost: As the histogram computation step (❶, ①) is the same, only the join step for *MRSS-join* (❷) and the filtering step, its distribution (②) and verification step (③) for *MRSF-join* are compared for the transmitted data.

680 million sequences with an average length of 220, corresponding to 166GB of identified raw sequences.

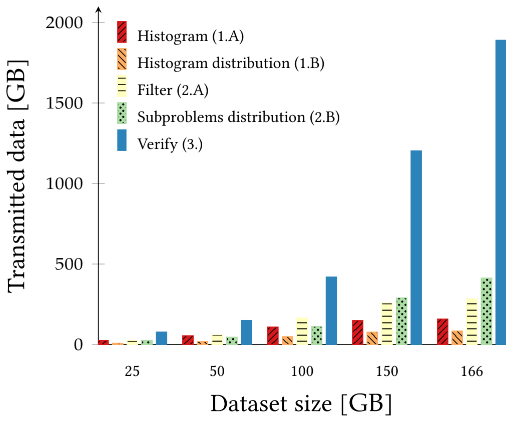
The Figure 7 highlights the performance of the *MRSF-join* algorithm in terms of processing time and communication cost when varying the size of the input dataset from 25GB to 166GB. It shows that most of the processing time is spent on the verification step especially for the largest datasets, in contrast to the previous synthetic datasets. This is due to the fact that the similarity join output is considerably more important, as revealed in Table 4. In addition, the *precision* values are high, especially for the largest datasets. In other words, although a significant amount of data is transmitted during the verification step, a large part of it corresponds to similarity join outputs, so we cannot expect a significant reduction of the transmitted data if the original sequences are sent during the verification step.

## 5. Conclusion

In this paper, we have introduced the *MRSF-join* algorithm to perform edit similarity joins on very large sequence datasets. Through the strategic integration of Locality-Sensitive Hashing (LSH), distributed histograms, and randomized communication templates, we drastically reduce the number of sequence comparisons and limit communication costs to only relevant data.
The *MRSF-join* algorithm improves the performance of our previous *MRSS-join* algorithm by further reducing the number of comparisons by using an efficient



(a) Processing times of *MRSF-join* algorithm.



(b) Transmitted data of *MRSF-join* algorithm.

Figure 7: The performance of the *MRSF-join* algorithm when varying the size of the input dataset.

| *Dataset size* | [GB] | 25 | 50 | 75 | 100 | 125 | 150 | 166 |
|---|---|---|---|---|---|---|---|---|
| *Number of input* | $[\times 10^6]$ | 111 | 231 | 360 | 468 | 580 | 643 | 680 |
| *Number of output* | $[\times 10^6]$ | 227 | 347 | 558 | 789 | 1065 | 4365 | 7408 |
| *Number of comparisons* | $[\times 10^6]$ | 1112 | 1952 | 3306 | 4936 | 6770 | 13715 | 20633 |
| *Recall** | | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* | 0.99* |
| *Filtering precision* | | 0.2 | 0.18 | 0.17 | 0.16 | 0.16 | 0.32 | 0.36 |

Table 4: The filtering quality of the *MRSF-join* algorithm when varying the size of the input: the *recall* has been measured on samples of the dataset.

filtering step, especially when using LSH with a high number of iterations. The theoretical cost model and the experiments show that the overhead involved using this multistep algorithm remains very small compared to the gain in performance obtained by reducing communication and data processing to almost all relevant. We showed that, the *MRSF-join* algorithm avoids memory overflows even in the case of very large dataset. This makes the algorithm scalable and insensitive to the data distribution skew. It also solves the limitations of existing approaches to handle large datasets with large number of LSH iterations.

Future work will be devoted to the comparison of the *MRSF-join* algorithm with approaches based on Locality-Sensitive Filtering (LSF) [34]. Furthermore, we would like to reuse this family of LSH functions to handle similarity search at a very large scale, and adapt these techniques to specific applications in biology.

## References

S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: 22nd International Conference on Data Engineering (ICDE'06), 2006, pp. 5–5. doi:10.1109/ICDE.2006.9.

S. Wandelt, D. Deng, S. Gerdjikov, S. Mishra, P. Mitankin, M. Patil, E. Siragusa, A. Tiskin, W. Wang, J. Wang, U. Leser, State-of-the-art in string similarity search and join, SIGMOD Rec. 43 (1) (2014) 64–76. doi:10.1145/2627692.2627706.

H. Zhang, Q. Zhang, Embedjoin: Efficient edit similarity joins via embeddings, in: Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 585–594. doi:10.1145/3097983.3098003.

C. E. Shannon, A mathematical theory of communication, The Bell System Technical Journal 27 (3) (1948) 379–423. doi:10.1002/j.1538-7305.1948.tb01338.x.

E. Ukkonen, Approximate string-matching with q-grams and maximal matches, Theoretical Computer Science 92 (1) (1992) 191–211. doi:10.1016/0304-3975(92)90143-4.

A. Arasu, V. Ganti, R. Kaushik, Efficient exact set-similarity joins, in: Proceedings of the 32nd International Conference on Very Large Data Bases, VLDB '06, VLDB Endowment, 2006, p. 918–929.

Y. Jiang, G. Li, J. Feng, W.-S. Li, String similarity joins: An experimental evaluation, Proc. VLDB Endow. 7 (8) (2014) 625–636. doi:10.14778/2732296.2732299.

M. Yu, G. Li, D. Deng, J. Feng, String similarity search and join: a survey, Frontiers of Computer Science 10 (3) (2016) 399–417. doi:10.1007/s11704-015-5900-5.

M. Steinegger, J. Söding, Clustering huge protein sequence sets in linear time, Nature Communications 9 (1) (2018) 2542. doi:10.1038/s41467-018-04964-5.

L. Gravano, P. Ipeirotis, H. Jagadish, N. Koudas, S. Muthukrishnan, D. Srivastava, Approximate string joins in a database (almost) for free, in: P. Apers, P. Atzeni, R. Snodgrass, S. Ceri, K. Ramamohanarao, S. Paraboschi (Eds.), VLDB 2001 - Proceedings of 27th International Conference on Very Large Data Bases, VLDB '01, Morgan Kaufmann, 2001, pp. 491–500.

G. Marçais, D. DeBlasio, P. Pandey, C. Kingsford, Locality-sensitive hashing for the edit distance, Bioinformatics 35 (14) (2019) i127–i135. doi:10.1093/bioinformatics/btz354.

F. Fier, N. Augsten, P. Bouros, U. Leser, J.-C. Freytag, Set similarity joins on mapreduce: An experimental survey, Proceedings of the VLDB Endowment 11 (10) (2018) 1110–1122. doi:10/gnqjwg.

S. Rivault, M. Bamha, S. Limet, S. Robert, Towards a scalable set similarity join using mapreduce and lsh, in: Computational Science – ICCS 2022, Springer International Publishing, Cham, 2022, pp. 569–583.

J. Dean, S. Ghemawat, MapReduce: Simplified data processing on large clusters, Communications of the ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.

X. Hu, Y. Tao, K. Yi, Output-optimal Parallel Algorithms for Similarity Joins, in: Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2017, pp. 79–90. doi:10.1145/3034786.3056110.

X. Hu, K. Yi, Y. Tao, Output-Optimal Massively Parallel Algorithms for Similarity Joins, ACM Transactions on Database Systems 44 (2) (2019) 1–36. doi:10/gnr4r4.

M. Aumüller, M. Ceccarello, Implementing distributed similarity joins using locality sensitive hashing, OpenProceedings.org, 2022, p. 13.

P. Indyk, R. Motwani, Approximate nearest neighbors: Towards removing the curse of dimensionality, in: Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing, 1998, pp. 604–613. doi:10.1145/276698.276876.

A. Gionis, P. Indyk, R. Motwani, Similarity search in high dimensions via hashing, in: Proceedings of the 25th International Conference on Very Large

Data Bases, VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999, p. 518–529.

A. Z. Broder, S. C. Glassman, M. S. Manasse, G. Zweig, Syntactic clustering of the Web, Computer Networks and ISDN Systems 29 (8) (1997) 1157–1166. doi:10/br259g.

P. Li, A. B. Owen, C.-H. Zhang, One permutation hashing, in: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2, NIPS'12, Curran Associates Inc., Red Hook, NY, USA, 2012, p. 3113–3121.

A. Shrivastava, P. Li, Densifying One Permutation Hashing via Rotation for Fast Near Neighbor Search, in: Proceedings of the 31st International Conference on Machine Learning, 2014, pp. 557–565.

T. Christiani, Fast locality-sensitive hashing frameworks for approximate near neighbor search, in: Similarity Search and Applications: 12th International Conference, SISAP 2019, Newark, NJ, USA, October 2–4, 2019, Proceedings, Springer-Verlag, Berlin, Heidelberg, 2019, p. 3–17. doi:10.1007/978-3-030-32047-8_1.

P. Li, C. König, B-bit minwise hashing, in: Proceedings of the 19th International Conference on World Wide Web, WWW '10, Association for Computing Machinery, New York, NY, USA, 2010, p. 671–680. doi:10.1145/1772690.1772759.

M. A. H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using mapreduce, Procedia Computer Science 29 (2014) 145–158, 2014 International Conference on Computational Science. doi:10.1016/j.procs.2014.05.014.

S. Rivault, M. Bamha, S. Limet, S. Robert, A Scalable Similarity Join Algorithm Based on MapReduce and LSH, International Journal of Parallel Programming 50 (3–4) (2022) 360–380. doi:10.1007/s10766-022-00733-6.

A. L. Zobrist, A new hashing method with application for game playing, ICGA Journal 13 (1990) 69–73.

M. Thorup, Fast and powerful hashing using tabulation, Communications of the ACM 60 (7) (2017) 94–101. doi:10.1145/3068772.

S. Dahlgaard, M. B. T. Knudsen, M. Thorup, Practical hash functions for similarity estimation and dimensionality reduction, in: Proceedings of the 31st International Conference on Neural Information Processing Systems, 2017, pp. 6618–6628.

D. Hyatt, G.-L. Chen, P. F. LoCascio, M. L. Land, F. W. Larimer, L. J. Hauser, Prodigal: prokaryotic gene recognition and translation initiation site identification, BMC Bioinformatics 11 (1) (2010) 119. doi:10.1186/1471-2105-11-119.

V. M. Markowitz, I.-M. A. Chen, K. Chu, E. Szeto, K. Palaniappan, M. Pillay, A. Ratner, J. Huang, I. Pagani, S. Tringe, M. Huntemann, K. Billis, N. Varghese, K. Tennessen, K. Mavromatis, A. Pati, N. N. Ivanova, N. C. Kyrpides, IMG/M 4 version of the integrated metagenome comparative analysis system, Nucleic Acids Research 42 (D1) (2013) D568–D573. doi:10.1093/nar/gkt919.

Y. Kodama, M. Shumway, o. b. o. t. I. N. S. D. C. Leinonen, Rasko, The sequence read archive: explosive growth of sequencing data, Nucleic Acids Research 40 (D1) (2011) D54–D56. doi:10.1093/nar/gkr854.

S. Sunagawa, L. P. Coelho, S. Chaffron, J. R. Kultima, K. Labadie, G. Salazar, B. Djahanschiri, G. Zeller, D. R. Mende, A. Alberti, F. M. Cornejo-Castillo, P. I. Costea, C. Cruaud, F. d'Ovidio, S. Engelen, I. Ferrera, J. M. Gasol, L. Guidi, F. Hildebrand, F. Kokoszka, C. Lepoivre, G. Lima-Mendez, J. Poulain, B. T. Poulos, M. Royo-Llonch, H. Sarmento, S. Vieira-Silva, C. Dimier, M. Picheral, S. Searson, S. Kandels-Lewis, T. O. Coordinators, C. Bowler, C. de Vargas, G. Gorsky, N. Grimsley, P. Hingamp, D. Iudicone, O. Jaillon, F. Not, H. Ogata, S. Pesant, S. Speich, L. Stemmann, M. B. Sullivan, J. Weissenbach, P. Wincker, E. Karsenti, J. Raes, S. G. Acinas, P. Bork, Ocean plankton. structure and function of the global ocean microbiome, Science 348 (6237) (2015) 1261359.

C. Rashtchian, A. Sharma, D. Woodruff, Lsf-join: Locality sensitive filtering for distributed all-pairs set similarity under skew, in: Proceedings of The Web Conference 2020, WWW '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 2998–3004. doi:10.1145/3366423.3380069.