A Scalable MapReduce Similarity Join Algorithm Using LSH

Sébastien Rivault, Mostafa Bamha, Sébastien Limet, Sophie Robert

A Scalable MapReduce Similarity Join Algorithm Using LSH

Sébastien RIVAULT · Mostafa BAMHA · Sébastien LIMET · Sophie ROBERT

Abstract. Similarity Joins are recognized to be among the most useful data processing and analysis operations. A similarity join is used to retrieve all data pairs whose distances are smaller than a predefined threshold λ.
In this paper, we introduce the *MRS-join* algorithm to perform similarity joins on large trajectories datasets. The MapReduce model and a randomized LSH (Local Sensitive Hashing) keys redistribution approach are used to balance load among processing nodes while reducing communications and computations to almost all relevant data by using distributed histograms.
A cost analysis of *MRS-join* algorithm shows that our approach is insensitive to data skew and guarantees perfect balancing properties, in large scale systems, during all stages of similarity join computations. These performances have been confirmed by a series of experiments using the Fréchet distance on large datasets of trajectories from real world and synthetic data benchmarks.
Keywords: Similarity-Join operations, Local Sensitive Hashing (LSH), MapReduce model, Data Skew, Hadoop framework.

1 Introduction

Parallel joins have been widely studied and efficiently implemented on distributed architectures [4–6, 17, 18]. However, all of the existing parallel join approaches cannot be used in the case of similarity-joins because no hashing or sorting technique makes it possible to find all potentially similar data pairs. So, all the data pairs must be compared which requires a Cartesian product computation. This can have a disastrous effect on performance and limit the scalability for large datasets processing [23]. Therefore, it is of utmost importance to efficiently implement scalable similarity join algorithms on large scale systems.
The aim of this paper is to focus on similarity joins on trajectories or time series. Trajectories or time series are seen as polygonal lines where each point belongs to $\mathbb{R}^d$ with d the dimension of the trajectory. A similarity join consists of finding all the pairs of trajectories having a similarity distance smaller than a given threshold. Given two collections of trajectories R and S, the goal is to compute

$$R \bowtie_\lambda S = \{(x, y) \in R \times S \quad | \quad \text{sim}(x, y) \leq \lambda\}$$

where $\text{sim}(x, y)$ is a distance between two trajectories x and y and λ is the threshold parameter.
To compute a similarity score between two trajectories, several distances can be used depending on the use cases. We focus on the study of similarity join using the Fréchet distance. In the literature, this distance was used for the recognition of handwriting [24], the prediction of protein structures [24] as well as the study of moving objects [22].
The Fréchet distance is often explained by the following metaphor: a man walking on a finite trajectory holds his dog on a leash, which is on another finite trajectory. Man and dog can vary their speeds but cannot turn back. The Fréchet continuous distance is the minimum length of the leash to connect man to his dog during the entire journey.
Although the Fréchet distance has been studied in detail in the literature, there are three aspects related to modern systems that have not yet been addressed for large scale similarity join computation. We therefore focus in this paper to develop an algorithm called *MRS-Join* (MapReduce Similarity Join) with provable guarantees which meets the following challenging aspects:

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

- **Scalability**: The algorithm must be scalable and efficient on very large datasets processing,
- **Completeness**: The algorithm must be able to generate, in a reasonable processing time, all the pairs of trajectories having a strong similarity,
- **Insensitive to data skew**: The algorithm must be insensitive to data skew while guaranteeing perfect balancing properties during all the stages of similarity join computations.

The motivation for these criteria comes from the huge amount of data collected by different applications. For example, a taxi or an Uber generates a sequence of locations throughout its taxi ride. This sequence of points in the space forms a trajectory. This leads to massive amounts of location data. For instance, the dataset ECML/PKDD (Porto) [1] contains 1.7 million trajectories from the ride of 442 taxis during a year. Uber is likely to generate trajectory dataset orders of magnitude larger than those generated from only 442 taxis. The amount easily exceeds the capacity of storage and the processing capability of a single machine. Accordingly, a cluster of machines and scalable and distributed algorithms are needed to process such large scale trajectory data.
There are multiple and useful query operations on trajectory dataset. For example, for a given trajectory, return all taxis that share similar routes. The trajectory similar search could help decision-making when building new roads, recommending faster taxi rides or more efficient customer care. It is also useful in sports analytic, weather forecast and it allows to build many advanced mining and learning tasks such as clustering and classification.
We guarantee, in *MRS-Join* algorithm, perfect balancing properties, among cluster processing nodes, during all MapReduce computation steps by using distributed histograms and randomized communication templates. To guarantee the scalability of the algorithm, it is not necessary to compute the distance between all the pairs of trajectories. Recently, a Locale sensitive hashing (LSH) family has been introduced for trajectories [15] for the Dynamic time Warping (DTW) and the Fréchet distance. LSH is a hashing technique where near points are more likely to have a common partition than distant points. In the literature, the standard use of LSH, to find strongly similar objects, is very efficient if the number of required iterations of LSH is low. We will, therefore, use this LSH family to reduce the search space and show results in terms of performance and quality.
The quality of the results is measured in terms of *recall* and *precision*. The *recall* is the fraction of the number of pairs of close trajectories correctly generated by the algorithm over the exact number of close trajectories, whereas *precision* corresponds to the fraction of the number of pairs of close trajectories correctly generated over the number of pairs of trajectories predicted as close.
In the literature, there is no distributed and scalable similarity join algorithm of trajectory using LSH. Many in-memory filtering-based similarity join algorithms provide exact algorithms [27,28] however, these results do not apply to our setting because we focus on very large datasets.
In the massively-parallel computation model, [19] presents an algorithm relying on LSH to approximate similarity joins to achieve output-optimal guarantees on the maximum load. However, this algorithm assumes that the dataset is not skewed. We primarily review that work and adapt it to our settings.
The remaining of this paper is organized as follows: Section 2 presents requirements for the understanding of the *MRS-join* algorithm. Section 3 describes the *MRS-join* algorithm with its complexity analysis. 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 describes the MapReduce programming model; Section 2.2 explains the Fréchet distance and its computation; Section 2.3 introduces LSH and its associated algorithm for trajectories processing; Section 2.4 presents an algorithm called "Standard" which is the best approach to perform similarity join in high dimension using LSH; Section 2.5 explains distributed histograms and randomized communication templates.

2.1 MapReduce programming model

MapReduce [13] is a simple yet powerful framework for implementing 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 presented in [13] is based on two functions: **map** and **reduce**, that the programmer is supposed to provide to the framework. These two functions should have the following signatures:

map: $(k_1, v_1) \longrightarrow list(k_2, v_2)$,
reduce: $(k_2, list(v_2)) \longrightarrow list(k_3, v_3)$.

The user must write the **map** function that has two input parameters, a key k_1 and an associated value v_1. Its output is a list of intermediate key/value pairs (k_2, v_2). This list is partitioned by the MapReduce framework depending on the values of k_2, where all pairs having the same value of k_2 belong to the same group.

The **reduce** function, that must also be written by the user, has two parameters as input: an intermediate key k_2 and a list of intermediate values $list(v_2)$ associated with k_2. It applies the user defined merge logic on $list(v_2)$ and outputs a list of key/values $list(k_3, v_3)$.

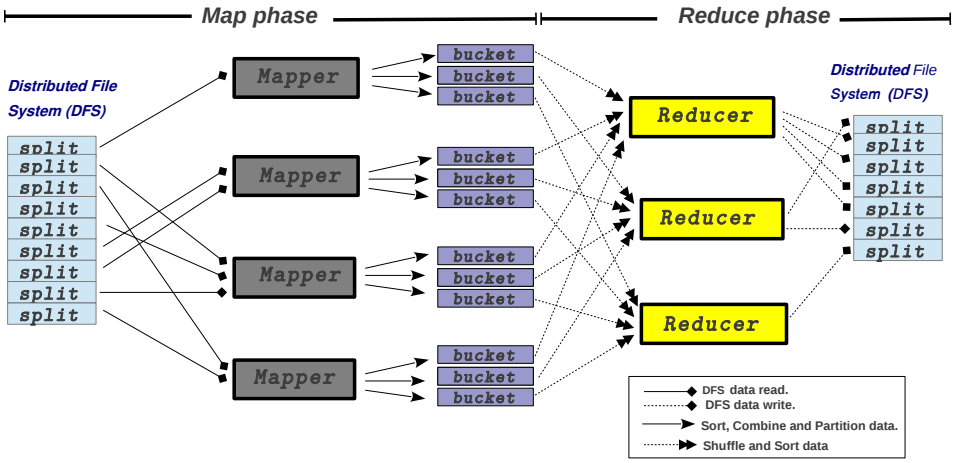


Fig. 1: MapReduce processing workflow.

In this paper, we used an open source version of MapReduce called Hadoop developed by "The Apache Software Foundation". The Hadoop framework includes a distributed file system called HDFS[1] designed to store very large files with streaming data access patterns.
For efficiency reasons, in Hadoop MapReduce framework, users may also specify a "Combine function", to reduce the amount of data transmitted from *Mappers* to *Reducers* during *shuffle* phase (see Fig 1). The "Combine function" is like a local reduce applied (at map worker) before storing or sending intermediate results to the *Reducers*. The signature of **combine** function is:

combine: $(k_2, list(v_2)) \longrightarrow (k_2, list(v_3))$.

To cover a large range of application needs in terms of computation and data redistribution, in Hadoop framework, the user can optionally implement two additional functions: **init**() and **close**() called before and after each **map** or **reduce** task. The user can also specify a "partition function" to send each key k_2 generated in **map** phase to a specific *Reducer* destination. The *Reducer* destination may be computed using only a part of the input key k_2. The signature of the **partition** function is:

partition: $k_2 \longrightarrow Integer$,

where the output of **partition** should be a positive number strictly smaller than the number of *Reducers* task. Hadoop's default **partition** function is based on "hashing" the whole input key k_2.

2.2 Fréchet distance

Let U, V be two trajectories defined by theirs vertices $U_0, ...U_n$ and $V_0, ..., V_m$. We denote by $U_{i,j}$ the subcurve $(U_i, U_{i+1}, ..., U_j)$ from U. The continuous Fréchet distance is defined by viewing a trajectory as a continuous function $U, V : [1, n] \rightarrow \mathbb{R}^d$ and interpolating between vertices with $U_{i+\tau} = (1-\tau) * U_i + \tau * U_{i+1}$ with $\tau \in [0, 1]$. Let Φ_n be the set of all continuous and non-decreasing functions ϕ from $[0, 1]$ to $[1, n]$. Then the continuous distance of Fréchet is defined as follows:

$$d_F(U, V) = \inf_{\substack{\phi_1 \in \Phi_{|U|} \\ \phi_2 \in \Phi_{|V|}}} \max_{t \in [0,1]} \|U_{\phi_1(t)} - V_{\phi_2(t)}\|$$

Alt and Godau were the first in 1995 [2] to introduce a polynomial algorithm computing Fréchet distance in a discrete manner. The discrete distance of Fréchet is an approximation of the continuous distance that considers

[1] HDFS: Hadoop Distributed File System.

only the leash length when the man and his dog are located on the vertices of their respective trajectory. In the literature, it is proven that the discrete distance of Fréchet between two trajectories cannot be decided in strictly subquadratic time in the number of vertices of the trajectories unless the Strong Exponential Time Hypothesis is false [7]. Accordingly, the most efficient algorithms are quadratic [9]. By taking some realistic assumptions on the shape of the input curves, it is possible to approximate the distance in near-linear time [14]. More recently, during the ACM SIGSPATIAL GIS Cup 2017, the challenge was to develop an exact and efficient algorithm to select close trajectories and test quickly if two trajectories have a distance less than a threshold distance. Winner of the ACM SIGSPATIAL GIS Cup 2017 presented an algorithm using several heuristics [26]. The following filters are applied in the order where λ denotes a threshold distance.

1. For two trajectories, if the distance between two points with the same index on each trajectory is always less than λ then the Fréchet distance is necessarily smaller than λ [10];
2. For two trajectories U and V, if there is a finite sequence $(U_{i_0}, V_{j_0}), \ldots, (U_{i_n}, V_{j_n})$ such that
   - $i_0 = j_0 = 0$ and i_n and j_n are the last indexes of U and V respectively,
   - $\forall k \in [0, n-1]$ either i_k and i_{k+1} are equal or $i_k = i_{k+1} + 1$ (idem for j_k and j_{k+1}),
   - the distance between U_k and V_k is less than λ

   then the overall distance is less than λ [3];
3. For all the points of a trajectory, if there does not exist at least one point in the other trajectory having a distance less than λ, then the global distance is necessarily greater than λ [3];
4. If none of the previous filters returns a result, the discrete Fréchet distance is computed using the recursive implementation of the Alt and Godau algorithm [2] described in [3].

In the literature, improvements have been made, and a procedure using the concept of simplification is shown in [12]. An overview of efficient algorithms to compute the Fréchet distance can be found in [8].

2.3 Local Sensitive Hashing (LSH)

Indyk and Motwani introduced an approximate search method [21] to simplify the search for the nearest neighbor in a high dimensional Euclidean space by using a hash function ensuring that close points are more likely collide than distant points. This strategy was subsequently extended to several distances and types of objects; it is characterized by the following definition.
Let two trajectories U and V, a distance measure d and the threshold distance λ. Given an approximation factor $c > 1$ and two probabilities p_1 and p_2 such that $0 \leq p_2 \leq p_1 \leq 1$, $\mathcal{H}$ is a family of LSH functions, if it satisfies the following conditions for any hash function chosen uniformly $g \in \mathcal{H}$:

1. If $d(U, V) \leq \lambda$ then $P(g(U) = g(V)) \geq p_1$
2. If $d(U, V) \geq c * \lambda$ then $P(g(U) = g(V)) \leq p_2$

Several LSH families have been introduced in the literature for the Fréchet distance. The oldest one was described by Indyk [20]. However, this strategy has exponential space complexity, which makes it impractical for our application.
Recently a LSH family, called $\mathcal{G}_\sigma$, was introduced by Driemel and Silvestri [12, 15]. It uses grids and transforms a trajectory into a sequence of grid points.
A grid G_σ^t of dimension d is defined by its origin t and its resolution σ. More formally, the canonical definition of a grid is as follows:

$$G_\sigma^t = \left\{(x_1, ..., x_d) \in \mathbb{R}^d \mid \forall\, i \in [1..d]\ \exists\, j \in \mathbb{N} : x_i = j * \sigma + t_i\right\}$$

The origin of the grid is chosen randomly in the half-open hypercube $[0, \sigma[^d$ whereas the grid resolution parameter σ is fixed to $\sigma = 4 * d * \lambda$ as in the experiments of LSH strategy in the paper [12]. The value $\sigma = 4 * d * \lambda$ is not proven to be optimal, but produces good results in practice as our experiments will show.
For a given grid G_σ^t, the hash function associated to G_σ^t takes a trajectory U and produce a sequence of grid nodes as follows:

1. For each point of the trajectory, find the nearest grid node,
2. Add the node to the result sequence if it is different from the last added node.

The LSH family $\mathcal{G}_\sigma$ is defined by $\mathcal{G}_\sigma = \left\{G_\sigma^t \mid \forall t \in [0, \sigma[^d\right\}$. In the context of LSH, an element of this family is called a hash function (in other words G_σ^t denotes both the grid and its associated hash function). A LSH strategy consists in choosing an element of $\mathcal{G}_\sigma$. To generate a good proportion of all similar pairs of trajectories, several hash functions are used. We denote by L_K the number of iterations of the LSH strategy. For each iteration $i, 1 \leq i \leq L_K$, a hash function g_i is uniformly and independently selected from $\mathcal{G}_\sigma$.

2.4 Standard algorithm to perform a similarity join using LSH

In order to compute a similarity join using LSH, an output optimal algorithm was described by [19]. By setting $L_K = \frac{1}{p_1}$, the algorithm is defined in three steps:

1. Randomly and independently select L_K hash functions $g_1, ..., g_{L_K} \in \mathcal{G}_\sigma$, and broadcast them to all servers.
2. For each trajectory t, emit a key/value pair $<(i,\ g_i(t)),\ t>$ for all $i \in 1, ..., L_K$
3. Perform an equi-join on all copies of the pairs by treating $(i,\ g_i(t))$ as the join value, i.e., two trajectories U, V join if $g_i(U) = g_i(V)$ for all i. For a pair of trajectories U, V, output them if $dist(U, V) \leq \lambda$.

As p_1 is not defined according to the grid resolution σ, we present in the experiments several values of the iteration's number L_K and their results in terms of *recall* and *precision* parameters. During the third step (*Reducer* phase), the distance of Fréchet between all trajectories having a common **key** (join value) will be computed.
Since the node sequences resulting from the LSH may require a lot of memory, a hashing function using the multiply-shift [25] is used to associate an integer to each sequence. i.e., if h a hashing function using the multiply-shift, a **key** $(i,\ g_i(t))$ will be hashed by $(i,\ h(g_i(t)))$. To simplify notation of **keys** from each LSH iteration, H_i^j will denote the tuple $(i,\ H^j)$ with H^j the sequence of hashed nodes of a given trajectory for $i \in 1, ..., L_K$.

2.5 Distributed histograms

To reduce communication costs while guaranteeing perfect balancing properties among all processing nodes, a distributed histogram of the relation is constructed in the same manner as in [18]. This histogram is the association between a **key** and its frequency in both datasets R and S. Distributed histograms are used to generate communication templates, allowing to distribute only relevant data fairly during the join phase. Distributed histograms avoid the effects of data skew, due to the fact that data will be partitioned into buckets fitting in memory. For a set R, let us denote $L(R)$ the set of **keys** of R and f_R^x the number of elements of R which have x as **key** (when it is clear from context x will be omitted).

Definition 1 For a dataset R, a histogram is defined as a function $\mathbf{Hist}_R$ that maps each element x of $L(R)$ to its frequency f_R^x.

In order to distribute only relevant data, only **keys** which might be present in join result are present in the histogram. A **key** producing a result implies that none of its frequencies in R and S are zero. The set of **keys** which produce a result are in $L(R) \cap\ L(S)$. Thus, we can define a histogram for the similarity join $R \bowtie_\lambda S$ which contains only relevant data.

Definition 2 Let $\text{Hist}(R \bowtie_\lambda S)$ be the function: $x \in L(R) \cap L(S) \longrightarrow (f_R^x, f_S^x)$.

Communication templates require a parameter which is denoted by f_{max}. This parameter defines the number of trajectories from a relation that a *Reducer* will have to store and process during the join. Owing to this parameter, the trajectories having a common **key** will be divided into several buckets (blocks), so that, each bucket can be loaded in memory. f_{max} is chosen in a manner that each bucket will fit in *Reducer*'s memory. This makes *MRS-join* algorithm insensitive to the effects of data skew. Partitioning data into buckets guarantees that, all join tasks are generated in a manner that the input data for each join task will fit in the memory of processing nodes and never exceed a user defined size, even for highly skewed datasets.
For a **key** x belonging to the histogram $\text{Hist}(R \bowtie_\lambda S)$ during the join, the communication templates will distribute all buckets according to one of the following three cases:

- If $f_R^x < f_{max}$ and $f_S^x < f_{max}$
  (i.e. the key corresponds to low frequencies in both datasets R and S and holds in memory on a single *Reducer*),
- If $f_R^x > f_{max}$ and $f_R^x > f_S^x$
  (i.e. the frequency in relation R is very high and is greater than in S),
- If $f_S^x > f_{max}$ and $f_S^x > f_R^x$
  (i.e. conversely, the frequency in dataset S is higher than in R).

In the first case, the trajectories corresponding to the **key** x are sent to a single *Reducer*, without special processing, using hashing whereas in the following two cases, a preprocessing is performed to balance the join computation on several *Reducers*. Thus, for a given **key** x, the relation corresponding to the lowest frequency is replicated on several nodes while the relation having the highest frequency is distributed over these same

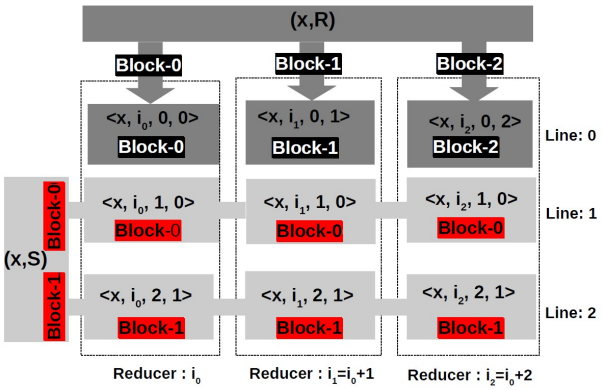


Fig. 2: Distribution of trajectories from R and replication of trajectories from S for a given **key** x.

nodes. An example of the second case where a **key** x is more frequent in the dataset R than S is shown in Figure 2.
In this example, each column corresponds to the data sent to a **reduce** task. All trajectories having the **key** x from the dataset R are divided into 3 buckets (blocks) and sent over 3 different *Reducer* tasks, whereas all trajectories corresponding to the **key** x from the dataset S are replicated on these tasks. To balance the load among processing nodes, the first **reduce** task identified by i_0 is computed using a random integer which can be derived from the **key** x.
Appropriate MapReduce (Key,Value) pairs are used to ensure that the buckets are sorted in the correct order. To this end, each emitted tuple is of the form <**key**, reducerId, line, bucketId>. The line value is used to ensure the order in the *Reducer* side. It is represented in Figure 2. The pairs are redirected by the **partition** function by means of the reducerId value. For a given *Reducer* task, the join is computed by using the following algorithm for each key corresponding to a high frequency:

- Store in memory the distributed buckets (i.e. the line is at zero);
- Compute the join with replicated buckets (i.e. line is greater than zero).

To use this communication template, the histogram must hold in memory. The histogram distribution algorithm from [18] was designed for a single join attribute value distribution. We present a new way to distribute it for multiple join values. Distribution is based on the appearance of the **keys** in the different splits. When building the histogram, the identifiers of splits where the **key** appears are saved. This set of identifiers holds in memory even for very large datasets since a HDFS split corresponds by default to 128Mb. Since entries are compressed, the splits are consistent during all steps. To work, the job distribution must have as many **reduce** tasks as the number of splits. Let <x, ((f_R^x, f_S^x), splitId*)> be an entry of the distribution job with a **key** $x \in L(R) \cap L(S)$, (f_R^x, f_S^x) its entry in the histogram and splitId* the set of identifiers of split where the **key** x was found. The algorithm is as follows for each entry of the histogram:

Mapper: Emit a pair <id, (x, (f_R^x, f_S^x))> for all $id \in splitId*$.
Reducer: Write to HDFS all received values

During the next *Mapper* task requiring the distributed histogram, only retrieving their split identifier is needed to fetch and store in memory the *Reducer* corresponding output. Due to the fact that the size of distributed histogram does not depend on the global size of the entries, it fits in memory.

3 MRS-Join: A scalable Similarity-join Algorithm using LSH

To compute similarity join, $R \bowtie_\lambda S$, of two datasets R and S for a threshold distance λ, we assume that input datasets are divided into splits (blocks) of data. These splits are stored in Hadoop Distributed File System (HDFS) and are also replicated on several nodes for reliability issues. Throughout this paper, for a dataset $\Gamma \in \{R, S\}$, we will use the following notations:

$\|\Gamma\|$	Number of entries or records in dataset Γ,
$\|\Gamma\|$	Number of pages (or blocks of data) of Γ,
Γ_i^{map}	*Fragment* (set of *split(s)*) of Γ affected to Mapper i,
$\{\Gamma\}$	Set of identifiers of *split(s)* of Γ,

$\|\{\Gamma\}\|$	Number of identifiers of *splits* of Γ. This number, is in general, very small since only the identifiers of *splits* are stored. We recall that HDFS default split size is 128Mb.
$\Gamma_j^{\text{split}}, j \in \{\Gamma\}$	*Split* identified by the integer j belonging to Γ
Γ_i^{red}	*Fragment* of Γ affected to Reducer i,
$L(\Gamma)$	Set of LSH **keys** of Γ,
$\Gamma(x), x \in L(\Gamma)$	Returns the **partition** (a subset) of Γ corresponding to the LSH key x,
$\text{Hist}(\Gamma_i^{\text{map}})$	Histogram of fragment Γ_i^{map}; i.e., each **key** of $L(\Gamma_i^{\text{map}})$ have an entry in the histogram,
$\text{Hist}(R \bowtie_\lambda S)$	Global histogram reduced to contain only the **keys** of $L(R) \cap L(S)$. Note that the size of this histogram is much smaller than $\text{Hist}(\Gamma)$,
$\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)$	*Fragment* of Hist $(R \bowtie_\lambda S)$ affected to Mapper i,
$\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)$	*Fragment* of Hist $(R \bowtie_\lambda S)$ affected to Reducer i,
$\text{Hist}_j(R \bowtie_\lambda S)$	Part of $\text{Hist}(R \bowtie_\lambda S)$ needed by *split* Γ_j^{split}; all the **keys** of $L(R) \cap L(S) \cap L(\Gamma_j^{\text{split}})$ have an entry in the histogram. The size of this histogram depends on the number of trajectories in a *split*, it can be stored in memory,
$\text{Hist}(\Gamma)(x)$	Returns the frequencies associated to the **key** $x \in L(\Gamma)$,
$\overline{\text{Hist}}(R \bowtie_\lambda S)$	Global histogram reduced of **partitions** contained in another **partition**. With $y \in L(R) \cap L(S)$, all **key** $x \in L(R) \cap L(S) \setminus y$ such that $\Gamma(x) \subset \Gamma(y)$ are deleted. In addition, if $\Gamma(x) = \Gamma(y)$, only one **key** among $x \cup y$ is saved. This histogram is smaller compared to $\text{Hist}(R \bowtie_\lambda S)$,
$\overline{\text{Hist}}_j(R \bowtie_\lambda S)$	Part of $\overline{\text{Hist}}(R \bowtie_\lambda S)$ required to compute the *split* Γ_j^{split},
$\overline{\Gamma}$	Subset of Γ corresponding to **keys** that are present in the reduced histogram $\overline{\text{Hist}}(R \bowtie_\lambda S)$,
L_K	Number of iterations of the LSH strategy (it also corresponds to the number of LSH **keys** per trajectory),
N_M	Number of Mappers,
N_R	Number of Reducers,
$c_{r/w}$	Read/write cost of a page of data from Distributed File System (DFS),
c_c	Communication cost per page of data.

We will describe the *MRS-join* algorithm while giving a cost analysis for each computation step. The O(...) notation only hides small constant factors: they only depend on the program's implementation, but neither on input datasets nor on processing machine parameters. *MRS-join* proceeds in 4 steps, including one or more MapReduce jobs. The Figure 3 represents the interactions between these different steps of the algorithm:

❶ Compute LSH **keys** of each trajectory,
❷ A histogram of the join is computed to guarantee balanced communication patterns regardless the data distribution [18],
❸ The histogram computed previously is reduced to optimize the communication costs to only relevant data. All **keys** which have their **partitions** included in another **partition** are deleted,
❹ By using reduced histogram, an efficient and scalable communication template scheme is generated and the distance between pairs of trajectories identified as similar is computed using the most efficient algorithm in the literature [3, 12] to generate similarity-join output.

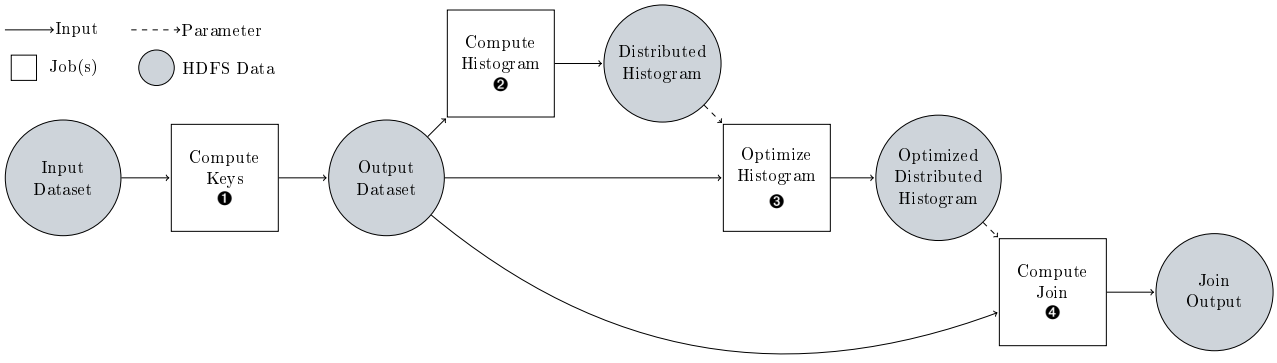


Fig. 3: MapReduce Similarity-join computation steps.

3.1 LSH keys computation

LSH keys of trajectories are computed using a MapReduce *job* where only a **map** phase is performed as shown in Figure 3.1.

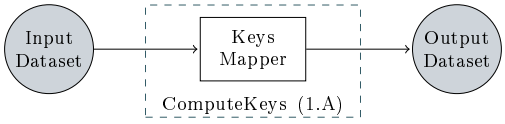


Fig. 3.1: Computation of the LSH **keys**.

In the input data, each trajectory is defined by a unique identifier, the tag of its data source (R or S) and its sequence of points. This step computes the L_K **keys** of each trajectory using the **map** function defined in the algorithm 1.A. Before the start of this **map** phase, the L_K hash functions are randomly and uniformly selected from $\mathcal{G}_\sigma$ and then stored in the HDFS.

Algorithm 1.A: ComputeKeys – MapReduce job ❶

Map: <*id, tag, trajectory*> → <*id, tag, trajectory, lsh_ keys*>
  init:
    Read from HDFS the L_K hash function.
  Compute lsh_keys: the L_K LSH **keys** of trajectory.
  Emit a key-value: <(id, tag, trajectory, lsh_keys)>.

By noting N_p, the average number of points in input trajectories, the cost of this step is at most:
$Time(1.A) = O(\max_{i=0}^{N_M} c_{r/w} * |\Gamma_i^{\text{map}}| + L_K * N_p * \|\Gamma_i^{\text{map}}\|)$.

The first term $c_{r/w} * |\Gamma_i^{\text{map}}|$ is the time to read trajectories from HDFS on *Mapper i*. The second term $L_K * N_p * \|\Gamma_i^{\text{map}}\|$ is the time needed to compute LSH **keys** of all the trajectories of the fragments affected to *Mapper i*. In order to reduce the number of *jobs*, it is possible to include this algorithm at the beginning of the two following steps.

3.2 Creation and distribution of the histogram

Once the **keys** have been computed for each trajectory, a histogram of the relation is computed. A histogram is the association between a key and its frequency both in R and S datasets. The aim of histograms is to identify the **keys** associated to high frequencies (these **keys** are generally those having a large effect on load imbalance among processing nodes). Identifying these **keys** will allow us to generate appropriate communication templates to avoid the effects of data skew while guaranteeing perfect balancing of load, among processing nodes, during all the steps of MapReduce similarity-join computation.
To distribute the trajectories equitably and efficiently using histograms, it is necessary to load the histogram in memory. The size of the global histogram depends on the size of the entry to be processed. Therefore, it is impossible to guarantee that it can hold in memory. However, when joining, a **map** function only processes its *split* data, it will not need the entire histogram. Therefore, a second *job* DistributeHistogram is needed to distribute the histogram entries according to their appearance in the different *splits*. These two *jobs* are represented by Figure 3.2 and will be described in the next two subsections.

Fig. 3.2: Distributed histogram computation steps.

3.2.A Creation of the histogram

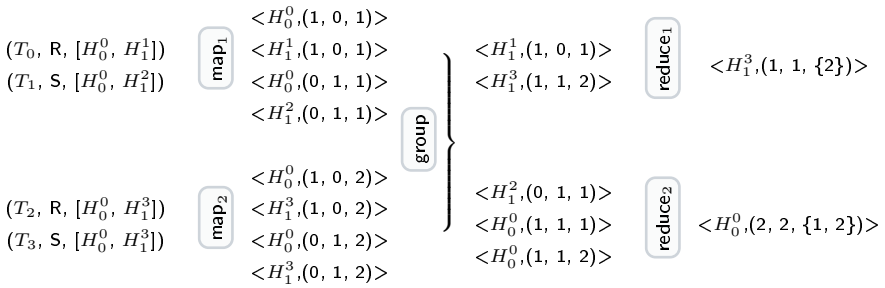
The MapReduce job of histogram creation is described in Algorithm 2.A and a working example is shown in Figure 2.A. In order to compute frequencies of each **key**, the **map** phase emits for each trajectory and for all $x \in$ lsh_keys a tuple of the form (0 | 1, 0 | 1, splitId). The first two values depend on the data source of the trajectory. For a trajectory belonging to the dataset R, the tuple will be of the form (1, 0, splitId).
The *Combiner* computes the local frequencies of the current split lf_R and lf_S for each **key** by adding each column separately. *Reducers* sum up the local frequencies received and filter out **keys** with zero frequency for a data source. The set of *splitId* where the **key** was found is saved in order to be able to distribute the global histogram during the *job* DistributeHistogram. For the sake of clarity, in the following examples the sequence of points forming a trajectory is never represented.

Algorithm 2.A: CreateHistogram – Step A of MapReduce job ❷

Map: <*id, tag, trajectory, lsh_keys*> → <*x, (0 | 1, 0 | 1, splitId)*>
  splitId ← getCurrentSplitId ();
  Emit a pair according to its data source for any $x \in$ lsh_keys:
  - <x, (1, 0, *splitId*)> when the trajectory belongs to R
  - <x, (0, 1, *splitId*)> when the trajectory belongs to S

Combine: <*x, (token$_R$, token$_S$, splitId)**> → <*x, (lf$_R$, lf$_S$, splitId)*>
  Compute the local frequencies by adding respectively received tokens.
  Emit a pair <x, (lf_R, lf_S, *splitId*)>.

Reduce: <*x, (lf$_R$, lf$_S$, splitId)**> → <*x, (f$_R$, f$_S$, splitId*)*>
  Compute the global frequencies in R and S by adding the received local frequencies respectively.
  Compute the union of splitIds.
  Emit the pair <x, (f_R, f_S, *splitId**)> if the **key** is present in R and S.



Example 2.A: Execution example of the *CreateHistogram job*.

Analysis of CreateHistogram job: The cost of the **map** phase is as follows. The term $c_{r/w} * |\Gamma_i^{\text{map}}|$ is the time, for the *Mapper i*, to read its associated Γ_i^{map} fragment from HDFS. The term $L_K * \|\Gamma_i^{\text{map}}\|$ is the number of pairs emitted by *Mapper i*. The last term $L_K * \|\Gamma_i^{\text{map}}\| * \log(L_K * \|\Gamma_i^{\text{map}}\|)$ is the time required to sort the emitted pairs by the **map** function.
$Time(2.A.Mapper) = O(\max_{i=0}^{N_M} c_{r/w} * |\Gamma_i^{\text{map}}| + L_K * \|\Gamma_i^{\text{map}}\| + L_K * \|\Gamma_i^{\text{map}}\| * \log(L_K * \|\Gamma_i^{\text{map}}\|))$.

Emitted pairs are then combined, partitioned and sent to the different *Reducers*. The term $L_K * \|\Gamma_i^{\text{map}}\|$ is the cost to compute the local frequencies. The term $c_c * |\text{Hist}_i^{\text{map}}(\Gamma)|$ is the required time for communication phase between *Mappers* and *Reducers*.
$Time(2.A.Combiner) = O(\max_{i=0}^{N_M} L_K * \|\Gamma_i^{\text{map}}\| + c_c * |\text{Hist}(\Gamma_i^{\text{map}})|)$.

The different partitions are finally retrieved from the *Mappers*, the term
$\|\text{Hist}_i^{\text{red}}(\Gamma)\|$ is the necessary time to compute the sum of the local frequencies and the union of *splitId* on the *Reducer i*. The term $c_{r/w} * |\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)|$ is the cost for *Reducer i* to write the histogram to the HDFS. Therefore,

$Time(2.A.Reducer) = O(\max_{i=0}^{N_R} \|\text{Hist}_i^{\text{red}}(\Gamma)\| + c_{r/w} * |\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)|)$.

This *job* will have the following cost:

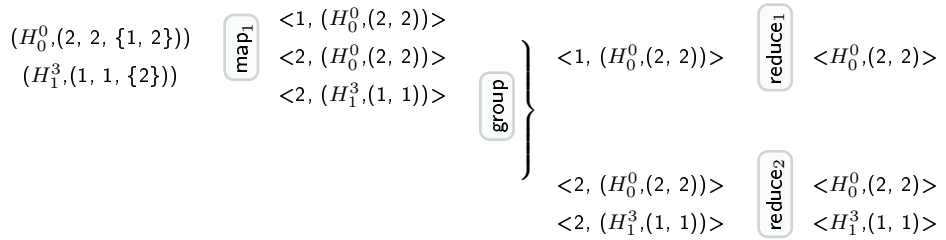
$$Time(2.A) = Time(2.A.Mapper) + Time(2.A.Combiner) + Time(2.A.Reducer).$$

3.2.B Histogram distribution

The global histogram is then distributed using DistributeHistogram *job* described in the 2.B algorithm. By setting the number of **reduce** tasks to be equal to the number,$\|\{\Gamma\}\|$, of splits of Γ, the output of a **reduce** task j corresponds to the histogram needed for the split Γ_j^{split}. For the following tasks requiring the histogram, it will therefore suffice to retrieve their *splitId* and read the output of this *job* associated with their *splitId*. An example of the output of the CreateHistogram *job* is shown in the Figure 2.B.

Algorithm 2.B: DistributeHistogram – Step B of MapReduce job ❷

Map: <*x*, (*f*$_R$, *f*$_S$, *splitId**)> → <*splitId*, (*x*, (*f*$_R$, *f*$_S$))>
| Emit a pair <*splitId*, (x, *f*$_R$, *f*$_S$)> for any *splitId* of the union computed previously.
Reduce: <*splitId*, (*x*, (*f*$_R$, *f*$_S$))*> → <*x*, (*f*$_R$, *f*$_S$)>
| Emit a pair <x, (*f*$_R$, *f*$_S$)> for all the received values.



Example 2.B: Execution example of *DistributeHistogram job*.

Analysis of DistributeHistogram job: Each *Mapper* reads its fragment of the histogram, thus the cost is $c_{r/w} * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)|$. For each entry in the histogram, there is a maximum of $\|\{\Gamma\}\|$ pairs sent. The cost of this step corresponds to the term $\|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|$. The term $\|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(\|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|$ corresponds to the time required to sort the intermediate results on the *Mapper i*. The intermediate results are then sent; at the cost of $c_c * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)| * \|\{\Gamma\}\|$. Therefore this mapper's step requires at most:
$Time(2.B.Mapper) = O(\max_{i=0}^{N_M} c_{r/w} * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)| +$
$\|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| + \|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(\|\{\Gamma\}\| * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|) + c_c * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)| * \|\{\Gamma\}\|)$

Each fragment processed by a **reduce** function corresponds to a necessary histogram of a *split*. There are several **reduce** tasks executed on the *Reducer i*. Each of these tasks has an identifier j such as $j \in 0..\|\{\Gamma\}\|$, since the number of **reduce** tasks is set to be equal to the number of *split*. By noting $\{F_i\}$, the set of **reduce** tasks executed on the *Reducer i*, the cost of this step is as follows.
$Time(2.B.Reducer) = O(\max_{i=0}^{N_R} \sum_j^{\{F_i\}} \|\text{Hist}_j(R \bowtie_\lambda S)\| + c_{r/w} * |\text{Hist}_j(R \bowtie_\lambda S)|)$.

The first term $\|\text{Hist}_j(R \bowtie_\lambda S)\|$ corresponds to the cost of executing a **reduce** task on *Reducer i*. The second term $c_{r/w} * |\text{Hist}_j(R \bowtie_\lambda S)|$ is required time to write output of a **reduce** task. Thus, this *job* will have the following cost:

$$Time(2.B) = Time(2.B.Mapper) + Time(2.B.Reducer).$$

3.3 Reduction of the global histogram

In order to reduce the intermediate results of the next Similarity join step ❹ and therefore decrease the number of trajectories comparisons, it is possible to reduce the overall histogram. Indeed, by reducing the histogram, the trajectories will be less replicated on the *Reducers*.
The aim of this step is therefore to delete all the **partitions** which are contained in another. All these **partitions** are not necessary for the computation since the similarity of these trajectories can be ensured by another **partition**.
To remove these **partitions**, the algorithm represented by the Figure 3.3 is composed of two MapReduce jobs followed by a *job* to distribute the histogram. This last *job* is equivalent to the one described in the previous subsection 3.2.B. By noting the set of trajectories having the **key** $x \in L(R) \cap L(S)$ by $\Gamma(x)$, the purpose of the *job* SuperPartition is to compute the set of **keys** $y \in L(R) \cap L(S) \setminus x$ such as $\Gamma(x) \subseteq \Gamma(y)$. We will denote by $I(x)$, the set of **keys** verifying this property.
The aim of the second *job* is to check for any $y \in I(x)$ whether the inclusion is strict. The three scenarios are as follows. If $\Gamma(x) \subset \Gamma(y)$ for at least one **key** y then the **key** x can be removed from the histogram since the join will be provided by the **key** y. The second case is the following, if for all $y \in I(x)$, $\Gamma(x) = \Gamma(y)$ then only one of the **key** of $x \cup I(x)$ is necessary and should be present in the reduced histogram. Finally, if $I(x) = \emptyset$ then no other **key** produces the join of this set of trajectories and the **key** x has to be present in the reduced histogram.

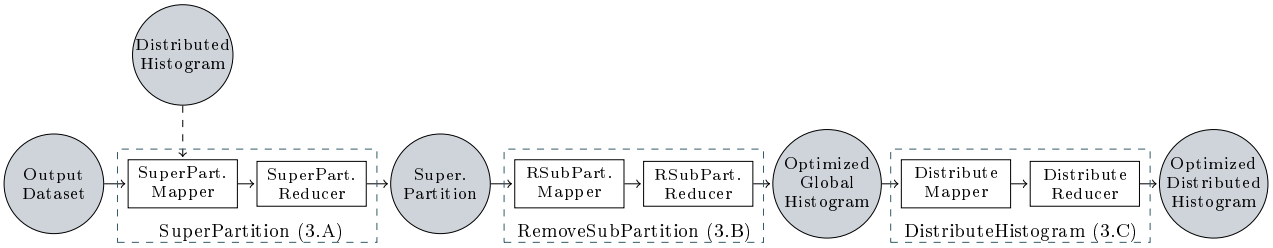


Fig. 3.3: Histogram reduction steps

3.3.A Preparation for histogram reduction

This *job* is described by the algorithm 3.A. Each **map** function takes a *split* as input. The identifier of *split* is retrieved and the distributed histogram of step ❷ associated is fetched and stored. The example of step ❷ is used for the example of this *job*. The distributed histogram is represented in the Table 3 and a working example in 3.A.

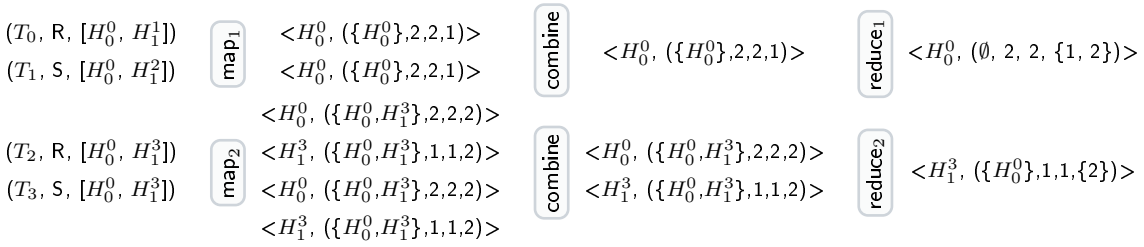
Algorithm 3.A: SuperPartition — Step A of MapReduce job ❸

Map: <*id, tag, trajectory, lsh_keys*> → <*x, (*$\overline{lsh\_keys}$*, Hist*(Γ)(x)*, splitId)*>
  init:
    splitId ← getCurrentSplitId ();
    Read and store the distributed histogram associated to the current *splitId*.
  Compute for each trajectory the set of **keys** present in the histogram, noted $\overline{lsh\_keys}$.
  Get the tuples associated to each **key** in the histogram.
  Emit a pair <x, ($\overline{lsh\_keys}$, Hist(Γ)(x), *splitId*)> for all $x \in \overline{lsh\_keys}$.
Combine: <*x, (*$\overline{lsh\_keys}$*, Hist*(Γ)(x)*, splitId)**> → <*x, (*$I_i^{map}(x)$*, Hist*(Γ)(x)*, splitId)*>
  Compute the intersection of the $\overline{lsh\_keys}$ denoted by $I_i^{\text{map}}(x)$.
  Emit a pair <x, ($I_i^{\text{map}}(x)$, Hist(Γ)(x), *splitId*)>
Reduce: <*x, (*$I_i^{map}(x)$*, Hist*(Γ)(x)*, splitId)**> → <*x, (I*(x)*, Hist*(Γ)(x)*, splitId*)*>
  Compute $I(x)$, the intersection of the received $I_i^{\text{map}}(x)$.
  Remove x from $I(x)$.
  Compute the union of *splitIds*.
  Emit a pair <x, ($I(x)$, Hist(Γ)(x), *splitId**)> for any received key x.

splitId	Distributed histogram
1	$<H_0^0, (2, 2)>$
2	$<H_0^0, (2, 2)>$
2	$<H_1^3, (1, 1)>$

Table 3: Distributed histogram using the output of step ❷.

Using the histogram, the $\mathbf{map}_1$ function will send only two pairs. These pairs are then combined using intersections on the sets of emitted **keys** in order to compute $I(H_0^0)$. The **key** H_0^0 is then sent to the *Reducer* to complete the intersection between all sets of **keys**. For the **key** H_0^0, the set is empty, so there is no other **key** whose **partition** contains the **partition** of H_0^0. For the **key** H_1^3, the **partition** is as follows $\{T_2, T_3\}$, this **partition** is contained by the **partition** of the **key** H_0^0. Therefore, $I(H_1^3)$ has for only one element H_0^0.



Example 3.A: ComputeSuperPartititon *job* example.

Analysis of job ComputeSuperPartititon: Each fragment is made up of a set of *split*. The cost of the **init** function is as follows. The term $c_{r/w} * |\text{Hist}_j(R \bowtie_\lambda S)|$ corresponds to the time, for a **map** function, to read the necessary histogram for its corresponding *split* of data.

$$Time(3.A.Mapper.\mathbf{init}) = O(\max_{i=0}^{N_M} \sum_{j}^{\{\Gamma_i^{\text{map}}\}} c_{r/w} * |\text{Hist}_j(R \bowtie_\lambda S)|).$$

The cost of the Mappers is as follows. The term $c_{r/w} * |\Gamma_i^{\text{map}}|$ is the time for the *Mapper* i to read its fragment from HDFS. At most, a pair is sent for each **key** of each trajectory, which corresponds to the term $L_K^2 * \|\Gamma_i^{\text{map}}\|$. These intermediate results are then sorted at a cost of $L_K^2 * \|\Gamma_i^{\text{map}}\| * \log(L_K^2 * \|\Gamma_i^{\text{map}}\|)$.

$$Time(3.A.Mapper) = Time(3.A.Mapper.\mathbf{init}) + O(\max_{i=0}^{N_M} c_{r/w} * |\Gamma_i^{\text{map}}| + L_K^2 * \|\Gamma_i^{\text{map}}\| + L_K^2 * \|\Gamma_i^{\text{map}}\| * \log(L_K^2 * \|\Gamma_i^{\text{map}}\|))).$$

Using the **combine** function, intermediate results can be reduced and the communication phase optimized. The cost is as follows.

$$Time(3.A.\text{Combiner}) = O(\max_{i=0}^{N_M} L_K^2 * \|\Gamma_i^{\text{map}}\| + \sum_{j}^{\{\Gamma_i^{\text{map}}\}} (c_c * |\text{Hist}_j(R \bowtie_\lambda S)|)).$$

The term $L_K^2 * \|\Gamma_i^{\text{map}}\|$ corresponds to the size of the input of the function **combine** on the *Mapper* i. The second term $\sum_j^{\{\Gamma_i^{\text{map}}\}} (c_c * |\text{Hist}_j(R \bowtie_\lambda S)|$ corresponds to the communication cost for the *Mapper* i.

$$Time(3.A.Reducer) = O(\max_{i=0}^{N_R} N_M * \|\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)\| + c_{r/w} * |\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)|).$$

The *job* ComputeSuperPartititon will have the following global cost:

$$Time(3.A) = Time(3.A.Mapper) + Time(3.A.Combiner) + Time(3.A.Reducer).$$

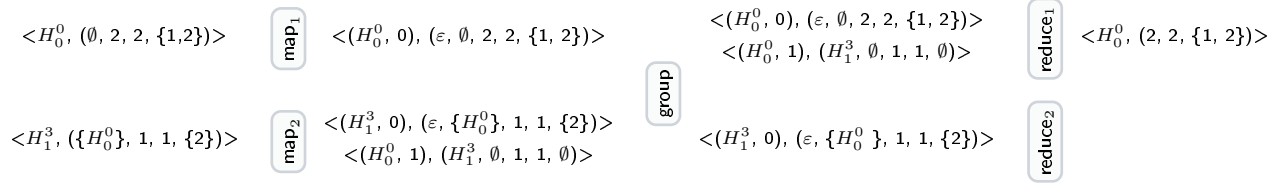
3.3.B Histogram reduction

For each input of its *split*, the RemoveSubMapper, presented by the 3.B algorithm, emits a primary pair <(x, 0), (ε, $I(x)$, Hist(Γ)(x), *splitId**)> for each input **key**. For some element y of its set $I(x)$ a secondary pair <(y, 1), (x, $\emptyset$, Hist(Γ)(x), $\emptyset$>) is emitted. When sorting, adding this integer to the sent key allows the primary pairs to arrive before the secondary pairs. The primary pair (<x, 0>) is saved by the *Reducer* and the emitters of the secondary pairs are deleted from $I(x)$. At the end of the execution of a **key**, if its set $I(x)$ is empty then it has to be present in the reduced histogram. Indeed, if $I(x)$ is empty then the **partition** x is not included in another **partition**. Moreover, due to the added relation order in the **map** function at the line 4, the case where one **partition** is equal to all the others is solved.
To optimize the size of the intermediate results, a *Combiner* is implemented using the following two observations. For a given **key** $x \in L(R) \cap L(S)$, all the elements y of $I(x)$ have frequencies greater than or equal to the frequencies of x since $\Gamma(x) \subseteq \Gamma(y)$. Moreover, $I(x)$ has at most a size equal to $L_K - 1$ by its construction. As the purpose of the *Reducer* is to remove emitters from $I(x)$, i.e **partitions** containing the **partition** of x, the *Combiner* can search and send only the secondary pairs with the maximum frequency if the number of pairs is less than $L_K - 1$. The following example 3.B continues from the previous example. The **key** H_0^0 will only send

Algorithm 3.B: RemoveSubPartition — Step B of MapReduce job ❸

Map: <*x, (I(x), Hist(Γ)(x), splitId*)*> → <*x | y, 0 | 1, (ε|x , I(x)|, Hist(Γ)(x), splitId*|)*>
  Emits a pair <(x, 0), (ε, Hist(Γ)(x), $I(x)$, *splitId**)>.
  foreach $y \in I(x)$ **do**
4   **if** $y < x$ **then**
      Emit a pair <(y, 1), (x, Hist(Γ)(x), $\emptyset$, $\emptyset$)>
    end
  end
Combine: <*(x, label), (ε | emitter, I(x) |, Hist(Γ)(x)|Hist(Γ)(emitter), splitId*|)*>
  → <*(x, label), (ε | emitter, I(x)|, Hist(Γ)(x)|Hist(Γ)(emitter), splitId*|)*>
  Emits the primary pairs (x, 0) without any processing.
  The secondary pairs (x, 1) are discarded if:
  - the frequency of the pair is not maximum,
  - the number of pair with the maximum frequency is greater than $L_K - 1$.
Reduce: <*x, label, (ε | emitter, I(x)|, Hist(Γ)(key)|Hist(Γ)(emitter), splitId*|)*>
  → <*x, (Hist(Γ)(x), splitId*)*>
  Save the primary pair <(x, 0), (I(x), Hist(Γ)(x), *splitId**)>.
  For each secondary pair received, the *emitter* is deleted from the saved set I(x).
  Emits a <x, (Hist(Γ)(x), *splitId**)> pair if the saved set I(x) is empty after this processing.

a primary pair since its set $I(H_0^0)$ is empty. The **key** H_1^3 emits two pairs. A primary containing its set $I(H_1^3)$, its associated histogram values, and the identifier of its *split*. A secondary $(H_0^0, 1)$ containing only the emitting **key** and its associated histogram values. The *Combiner* in this example is not shown since it does not filter any pair. The *Reducer* computing the **key** H_1^3 does not receive any other secondary pair. As its set $I(H_1^3)$ is not empty, the **key** H_1^3 is removed.



Example 3.B: RemoveSubPartition *job* example.

Analysis of job RemoveSubPartition: The cost of the *Mapper* phase is as follows. The term $c_{r/w} * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)|$ corresponds to the time for the *Mapper* i to read the histogram resulting from the previous step from the HDFS. The term $L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|$ refers to the maximum number of the emitted pairs by *Mapper* i.

The last term $L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|)$ represents the maximum time to sort the emitted pairs by the *Mapper i*.

$$Time(3.B.Mapper) = O(\max_{i=0}^{N_M} c_{r/w} * |\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)| + L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| + L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|)).$$

Owing to the Combine phase, only L_K secondary pairs are sent for each **key**. The cost of this phase corresponds to the term $L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|$.
$Time(3.B.\text{Combiner}) = O(\max_{i=0}^{N_M} L_K * \|\text{Hist}_i^{\text{map}}(R \bowtie_\lambda S)\|)$,
The communication cost is represented by the term $c_c * |\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)|$. Thanks to the Combine phase, the time required for the execution of the *Reducer i* can be increased by the term $N_M * L_K * \|\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)\|$. The last term $c_{r/w} * |\overline{\text{Hist}}_i^{\text{red}}(R \bowtie_\lambda S)|$ corresponds to the cost of writing the results.

$$Time(3.B.Reducer) = O(\max_{i=0}^{N_R} c_c * |\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)| + N_M * L_K * \|\text{Hist}_i^{\text{red}}(R \bowtie_\lambda S)\| + c_{r/w} * |\overline{\text{Hist}}_i^{\text{red}}(R \bowtie_\lambda S)|),$$

And the overall cost of this *job* is:
$Time(3.B) = Time(3.B.Mapper) + Time(3.B.Combiner) + Time(3.B.Reducer)$.

3.3.C Histogram distribution

This *job* is similar to the Algorithm 2.B described in the previous section 3.2.B. The goal is always to distribute the global histogram according to the appearance of the **keys** in the *splits*. Each *split* will be associated to a histogram containing only the necessary **keys**. This histogram holds in memory since its size depends only on the number of trajectories in the *split*. The associated algorithms are equivalent but the cost is different since the histogram has been reduced. The time required for the *Mapper* phase is as follows.

$$Time(3.C.Mapper) = O(\max_{i=0}^{N_M} c_{r/w} * |\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)| + \|\{\Gamma\}\| * \overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S) + \|\{\Gamma\}\| * \|\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(\|\{\Gamma\}\| * \|\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)\|) + c_c * |\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)| * \|\{\Gamma\}\|),$$

The term $c_{r/w} * |\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)|$ corresponds to the time to read the fragments of the reduced histogram on the *Mapper i* from the HDFS. The *Mapper i* emits a maximum of $\|\{\Gamma\}\| * \overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)$ couples. These pairs are sorted at a cost of $\|\{\Gamma\}\| * \|\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)\| * \log(\|\{\Gamma\}\| * \|\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)\|)$. The intermediate results are then communicated to the *Reducers* at a time corresponding to the term $c_c * |\overline{\text{Hist}}_i^{\text{map}}(R \bowtie_\lambda S)| * \|\{\Gamma\}\|$.
The **reduce** phase will have the following complexity. The term $\|\overline{\text{Hist}}_j(R \bowtie_\lambda S)\|$ corresponds to the execution time of a **reduce** function on the *Reducer i*. Each *Reducer j* emits the necessary histogram of the *split* Γ_j^{split}, this cost corresponds to the term $c_{r/w} * |\overline{\text{Hist}}_j(R \bowtie_\lambda S)|$.

$$Time(3.C.Reducer) = O(\max_{i=0}^{N_R} \sum_{j}^{\{F_i\}} \|\overline{\text{Hist}}_j(R \bowtie_\lambda S)\| + c_{r/w} * |\overline{\text{Hist}}_j(R \bowtie_\lambda S)|).$$

And, this *job* will have the following complexity: $Time(3.C) = Time(3.C.Mapper) + Time(3.C.Reducer)$.

3.4 Similarity-Join computation step

This *job* is composed of a **map** phase and a **reduce** phase as shown in the Figure 3.4. The goal of this algorithm is to compute the similarity-join by using distributed histogram allowing us to generate efficient communication templates to redistribute only relevant data while guaranteeing perfect balancing properties even for highly skewed input datasets. The similarity-join computation steps are described in the Algorithm 4.A. The phase of **map** corresponds to the communication templates presented in the subsection 2.5. The *Reducer* computes the join between buckets by filtering out false positive trajectories. Some pairs of trajectories are present in several **keys**. To avoid duplicate computation of the Fréchet distance, the minimal **key** is computed dynamically.

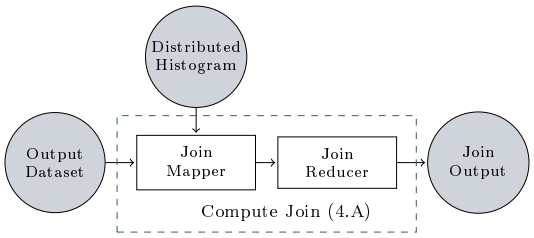


Fig. 3.4: Similarity join computation step.

Algorithm 4.A: Similarity join computation step – MapReduce job ❹

Map: *<id, tag, trajectory, lsh_keys>* → *<(x, reducerId, line, bucket), (trajectory, $\overline{lsh\_keys}$)>*

init:

$splitId \leftarrow$ getCurrentSplitId ();

Read and store the distributed histogram.

Compute for each trajectory the set of **keys** present in the histogram, noted $\overline{lsh\_keys}$.

Generate for each $\overline{lsh\_keys}$ the communication templates according to the frequencies saved in the histogram. Only the relevant data is sent using the communication templates defined previously (2.5).

Emit pairs according to communication templates (subsection 2.5).

Partition: *<(x, reducerId, line, bucket), (trajectory, $\overline{lsh\_keys}$)>* → *Integer*

Redirect each pair to the computed reducerId.

Reduce: *<(x, reducerId, line, bucket), (trajectory, $\overline{lsh\_keys}$)>* → *(id_R, id_S)>*

Compute the join of the trajectories using to the communication templates.

For each pair of trajectories coming from R and S:

- Check if this pair of trajectories is computed by another **key** using emitted sets $\overline{lsh\_keys}$.
- Compute the intersection of the sets $\overline{lsh\_keys}$.
- A **key** is dynamically computed to know which *Reducer* filters this pair of trajectories. 0.A
- If the current **key** is the minimal among the intersection, compute the Fréchet distance (subsection 2.2) and output the pair (id_R, id_S) if the distance is lower than λ.

Analysis of similarity join computation job: The cost of the *Mapper* phase is as follows. The term $c_{r/w} * |\Gamma_i^{\text{map}}|$ is the cost for the *Mapper* i to read its fragment from HDFS. For each split identified by $j \in \{\Gamma_i^{\text{map}}\}$, in its fragment, the *Mapper* i reads its associated distributed histograms at a cost corresponding to the term $c_{r/w} * |\overline{\text{Hist}}_j(R \bowtie_\lambda S)|$. The term $L_K * \|R_i^{\text{map}}\| + L_K * \|S_i^{\text{map}}\|$ corresponds to the cost to perform search operations in the distributed histogram for a given **key**. The term $\|\overline{R}_i^{\text{map}}\| * \log(\|\overline{R}_i^{\text{map}}\|) + \|\overline{S}_i^{\text{map}}\| * \log(\|\overline{S}_i^{\text{map}}\|)$ is time to sort relevant data on *Mapper* i. The term $L_K * c_c * (|\overline{R}_i^{\text{map}}| + |\overline{S}_i^{\text{map}}|)$ is time to send data to Reducers using our communication templates.

$Time(4.A.Mapper) = O(\max_{i=0}^{N_M} c_{r/w} * |\Gamma_i^{\text{map}}| + \sum_j^{\{\Gamma_i^{\text{map}}\}} (c_{r/w} * |\overline{\text{Hist}}_j(R \bowtie_\lambda S)|) + L_K * \|R_i^{\text{map}}\| + L_K * \|S_i^{\text{map}}\| + \|\overline{R}_i^{\text{map}}\| * \log(\|\overline{R}_i^{\text{map}}\|) + \|\overline{S}_i^{\text{map}}\| * \log(\|\overline{S}_i^{\text{map}}\|) + L_K * c_c * (|\overline{R}_i^{\text{map}}| + |\overline{S}_i^{\text{map}}|))$

We recall that, in this step, only relevant data is sent by *Mappers* because of the distributed histogram. Records associated with a highly frequent **key** are redistributed according to our efficient dynamic partition/replicate schema to balance load among reducers and to avoid the effect of data skew. Records associated with low frequency **key** are redistributed using default hashing functions.

The cost of the *Reducer* phase is as follows. Denoting by $t_{frechet}$ the complexity of computing the Fréchet distance. The term $t_{frechet} * \|\overline{R}_i^{\text{red}} \bowtie_\lambda \overline{S}_i^{\text{red}}\|$ corresponds to the cost to filter out false positive emitted by the LSH strategy on the *Reducer* i. The term $c_{r/w} * |\overline{R}_i^{\text{red}} \bowtie_\lambda \overline{S}_i^{\text{red}}|$ refers to the required time for the *Reducer* i to write results in HDFS.

$$Time(4.A.Reducer) = O(\max_{i=0}^{N_R} t_{frechet} * \|\overline{R}_i^{\text{red}} \bowtie_\lambda \overline{S}_i^{\text{red}}\| + c_{r/w} * |\overline{R}_i^{\text{red}} \bowtie_\lambda \overline{S}_i^{\text{red}}|).$$

Therefore the overall cost of this *job* is:

$$Time(4.A) = Time(4.A.Mapper) + Time(4.A.Reducer).$$

Analysis of MRS-join algorithm: The global cost of *MRS-join* is therefore the sum of all previous steps:

$$Time(MRS{-}join) = Time(1.A){+}Time(2.A){+}Time(2.B){+}Time(3.A){+}Time(3.B){+}Time(3.C){+}Time(4.A).$$

Using hashing techniques, the Similarity-join computation, $R \bowtie_\lambda S$, of two relations R and S requires at least the following lower bound :

$$\mathbf{bound_{inf}} = \Omega\Bigg(\max_{i=1}^{N_M} \big((c_{r/w} + c_{comm}) * (|R_i^{map}| + |S_i^{map}|) + ||R_i^{map}|| * \log||R_i^{map}|| + ||S_i^{map}|| * \log||S_i^{map}||\big) + \max_{i=1}^{N_R} \big(t_{Frechet} * \|R_i^{red} \bowtie_\lambda S_i^{red}\| + c_{r/w} * |R_i^{red} \bowtie_\lambda S_i^{red}|\big)\Bigg),$$

where $c_{r/w} * (|R_i^{map}| + |S_i^{map}|)$ is the cost of reading input relations from HDFS on node i. The term $||R_i^{map}|| * \log(||R_i^{map}||) + ||S_i^{map}|| * \log(||S_i^{map}||)$ represents the cost to sort input relations records on map phase. The term $c_{comm} * (|R_i^{map}| + |S_i^{map}|)$ represents the cost to communicate data from *Mappers* to *Reducers*, the term $t_{frechet} * \|R_i^{red} \bowtie_\lambda S_i^{red}\|$ is time to compute Fréchet distance on *Reducer* i and $c_{r/w} * |R_i^{red} \bowtie_\lambda S_i^{red}|$ represents the cost to store *Reducer*'s i join output on the HDFS.
MRS-join algorithm has asymptotic optimal complexity when: $L_K^2 * \|\Gamma_i^{\text{map}}\| * \log(L_K^2 * \|\Gamma_i^{\text{map}}\|)$

$$\leq \max\left(\max_{i=1}^{N_M} \Big(||R_i^{map}|| * \log(||R_i^{map}||), ||S_i^{map}|| * \log(||S_i^{map}||)\Big), \max_{i=1}^{N_R} t_{frechet} * \|R_i^{red} \bowtie_\lambda S_i^{red}\|\right), \quad (1)$$

this is due to the fact that, all other terms in $Time(MRS\text{-}join)$ are bounded by those of $\mathbf{bound_{inf}}$. We recall that the sizes of distributed histograms are very small compared to input datasets sizes and all the terms of $Time(MRS\text{-}join)$ remains very small when compared against naive algorithms requiring pairwise comparison of all trajectories.

Remark: In practice, data imbalance related to the use of hashing functions can be due to:

- *a bad choice of used hash function.* This imbalance can be avoided by using the hashing techniques presented in the literature which have the property of distributing uniformly with a very high probability [11],
- *an intrinsic data imbalance* which appears when some values of the join attribute appears more frequently than others. There is no way for a clever hash function to avoid load imbalance that results from these repeated values. However, this case cannot arise here since we applied hashing functions to histograms which contain only distinct values or to randomized keys.

4 Experiments

In this section, we discuss the efficiency and the strength of our theoretical analysis by experimenting *MRS-Join* algorithm on real world and synthetic datasets by measuring recall, precision and efficiency. This analysis was performed on a cluster of 11 machines. Each machine has the following characteristics: Intel(R) Xeon(R) CPU E5-2650 @2.60GHz, 16Gb of memory and 300Gb of HDD disk. Experiments are performed on the top of Hadoop 3.2.1 framework using 6Gb of Heap memory for Map/Reduce tasks.

4.1 The Taxis dataset

The first ECML / PKDD (Porto) [1] dataset used describes the mobility of 442 taxis during a year. The dataset contains 1.7 million trajectories. Each trajectory corresponds to a travel of a passenger with one of the taxis. The GPS locations of the taxi during a trip were recorded every 15 seconds in the WGS84 format. The length of the trajectory varies between 1km and 15km. To simulate a R-S join, a random label (R or S) has been assigned to each trajectory. The Table 4.1 shows that most of similar pairs were found by LSH. In the following Table, *True positive* denotes the number of close trajectories correctly generated and *False positive* denotes the number of distances computed by the algorithm on far trajectories. We notice that *False positive* number remains very small compared to naive algorithms which require pairwise comparisons of all trajectories of input datasets. Execution times for the Taxi dataset, for both Standard and MRS-join algorithms, are similar to those observed the synthetic datasets of the same size.

L_K	True positive	Exact	False positive $\cdot 10^8$	Recall	Precision
16	18086086	18827613	6,3	0.961	0.028
32	18557037	18827613	9,9	0.985	0.018
64	18734572	18827613	10,5	0.995	0.017

Table 4.1: Similarity-Join results using the Porto taxi database benchmark [1].

4.2 Similarity-Join processing using synthetic datasets

Synthetic datasets were generated to study the scalability of the algorithm. Between 1 and 10 million trajectories were generated. Each million represents approximately 1GB on the HDFS. The dataset generator was designed such that each trajectory represents a car on a very long highway. All cars start randomly at one point on the highway. Each car has its own speed. On average, each trajectory is made up of 20 points. In the Figure 4.1, the execution times of the *MRS-join* algorithm are compared to the Standard version (subsection 2.4). For the experiment, the number of iterations of the LSH strategy was set at 32.

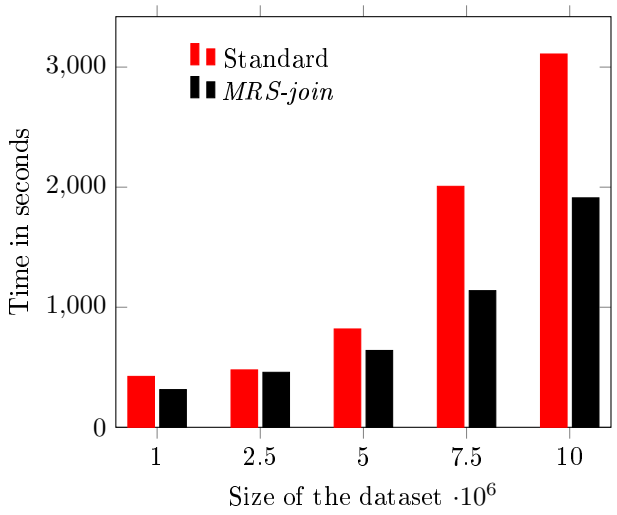


Fig. 4.1: Effects of the dataset size on Hadoop join processing time.

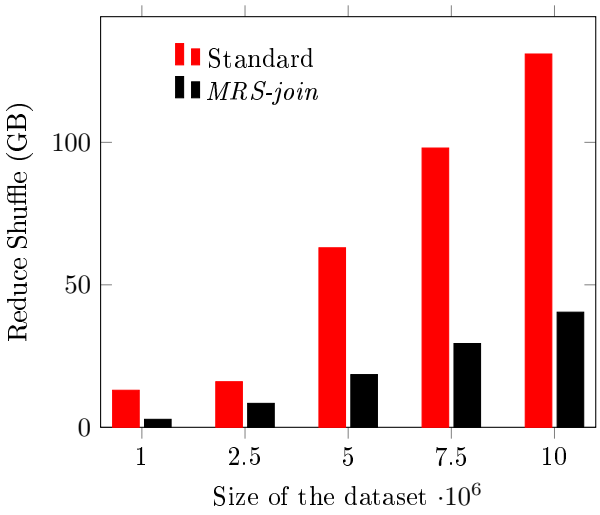


Fig. 4.2: Size of the dataset effect on the communication cost.

We recall that the *MRS-join* algorithm requires four reads of the inputs to compute the similarity join. The first one is used to compute the LSH hash keys according to the grid, the second to compute the histogram, the third to optimize the histogram and finally the last one to compute similarity join. Although these multiple steps seem to have a significant cost, the gain induced by sending only relevant data makes it possible to drastically reduce the data transmitted by the Mappers during the communication phase of the joining step. The Figure 4.2 presents the comparison of the data transmitted during the communication phase between the standard algorithm and the *MRS-join* algorithm. The Figure 4.1 presents the comparison of the processing time between the standard and the *MRS-join* algorithm.

To show when the LSH strategy needs more iterations, we compute the similarity join on trajectories of 20 and 50 points on average, the results are shown in the Tables 4.2 and 4.3.

L_K	True positive	Exact	False positive $\cdot 10^8$	Recall	Precision
8	361362	365824	3	0.9878	0.0011
16	365725	365824	5,7	0.999	0.0006
32	365808	365824	9,2	0.999	0.0004
64	365812	365824	12,7	0.999	0.0003

Table 4.2: Results using 10^6 trajectories composed of 20 points on average.

L_K	True positive	Exact	False positive $\cdot 10^6$	Recall	Precision
8	7095	9605	3,2	0.739	0.002
16	8611	9605	8	0.897	0.001
32	9235	9605	14,7	0.961	0.0006
64	9473	9605	24	0.986	0.0003

Table 4.3: Results using 10^6 trajectories composed of 50 points on average.

4.3 Data skew effects on Similarity-join processing using synthetic datasets

To show when the Standard version fails, we present results on a dataset with an imbalance in the data distribution. To create this dataset, we took the previous dataset and added trajectory copies. The number of added copies follows a Zipf distribution [16]. The Zipf factor varies from 0 (for a uniform distribution) to 1.6 (for a highly skewed distribution). The Figure 4.3 shows the data skew effect on the execution time for the standard and *MRS-join* algorithm. We see that for a Zipf parameter varying from 1.0 to 1.6, the Standard algorithm jobs fail due to lack of memory. Since all the trajectories having the same join **key** are forwarded to the same Reducer, Standard algorithm is very sensitive to data skew which limits its scalability. This cannot happen in *MRS-join* because the join computations for a highly frequent **key**, are partitioned into buckets and transmitted to distinct *Reducers* in a random manner. This makes *MRS-join* algorithm insensitive to the imbalance of data. Figure 4.4 shows that the number of distances computed by each **reduce** tasks following a highly skewed distribution, remains well distributed on the different nodes of the cluster.

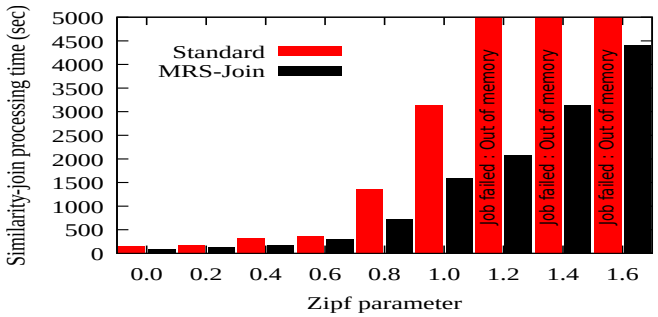


Fig. 4.3: Data skew effect on the Hadoop join processing time using 10^6 of trajectories.

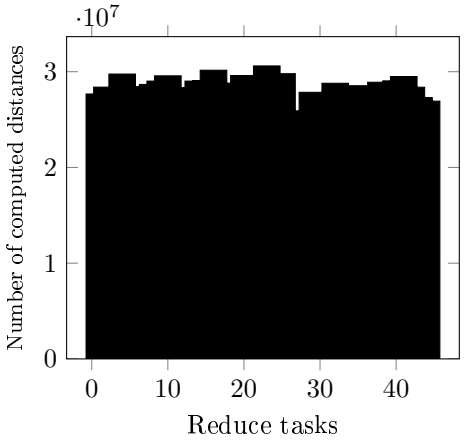


Fig. 4.4: Data skew effect on the distribution of the distance computation using *MRS-join* for Zipf factor 1.0.

5 Conclusion

In this article, we have introduced *MRS-join* an efficient and scalable MapReduce Similarity-join, for trajectories, using LSH and randomized communication templates to avoid the effects of data skew even for highly skewed large datasets. The *MRS-Join* cost analysis and experiments, using real world and synthetic benchmarks datasets, show that the overhead related to the use of distributed histograms remains very small compared to the gain in performance by reducing communication and data processing to only relevant data (this avoids pairwise comparison of all trajectories) while guaranteeing perfect balancing properties during all the stages of similarity join computation. We recall that in the MRS-join, all MapReduce generated buckets never exceed a user defined size. This makes the algorithm scalable and insensitive to data skew. 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 MapReduce sequences similarity processing in large datasets using prefix filtering and/or similar techniques based on randomized MapReduce data redistribution to balance load among processing nodes while guaranteeing the scalability of the proposed solutions in large scale systems.

References

1. Ecml/pkdd porto taxi data. `https://www.kaggle.com/c/pkdd-15-predict-taxi-service-trajectory-i`.
2. H. Alt and M. Godau. Computing the fréchet distance between two polygonal curves. *International Journal of Computational Geometry & Applications*, 05(1):75–91, 1995.
3. J. Baldus and K. Bringmann. A fast implementation of near neighbors queries for fréchet distance (GIS cup). In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, SIGSPATIAL '17, pages 1–4. Association for Computing Machinery, 2017.
4. M. Bamha. An optimal and skew-insensitive join and multi-join algorithm for distributed architectures. In *Proceedings of the International Conference on Database and Expert Systems Applications (DEXA'2005). 22-26 August, Copenhagen, Danemark*, volume 3588 of *LNCS*, pages 616–625. Springer, 2005.
5. M. Bamha and M. Exbrayat. Pipelining a skew-insensitive parallel join algorithm. *Parallel Processing Letters*, 13(3):317–328, 2003.
6. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 975–986, New York, NY, USA, 2010. ACM.
7. K. Bringmann. Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. *arXiv:1404.1448 [cs]*, 2014.
8. K. Bringmann, M. Künnemann, and A. Nusser. Walking the dog fast in practice: Algorithm engineering of the fréchet distance. *arXiv:1901.01504 [cs]*, 2019.
9. K. Buchin, M. Buchin, W. Meulemans, and W. Mulzer. Four soviets walk the dog-improved bounds for computing the fréchet distance. *Discrete and Computation Geometry*, 58(1):180–216, 2017.
10. K. Buchin, Y. Diez, T. van Diggelen, and W. Meulemans. Efficient trajectory queries under the fréchet distance (GIS cup). In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 1–4. ACM, 2017.
11. J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.
12. M. Ceccarello, A. Driemel, and F. Silvestri. FRESH: Fréchet similarity with hashing. *arXiv:1809.02350 [cs]*, 2019.
13. J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.
14. A. Driemel, S. Har-Peled, and C. Wenk. Approximating the fréchet distance for realistic curves in near linear time. *Discrete and Computation Geometry*, 48(1):94–127, 2012.
15. A. Driemel and F. Silvestri. Locality-sensitive hashing of curves. *arXiv:1703.04040 [cs]*, 2017.
16. P. S. Florence. Human Behaviour and the Principle of Least Effort. *The Economic Journal*, 60(240):808–810, 12 1950.
17. M. A. H. Hassan and M. Bamha. Towards scalability and data skew handling in groupby-joins using mapreduce model. *Procedia Computer Science*, 51:70–79, 2015. International Conference On Computational Science, ICCS 2015.
18. M. A. H. Hassan, M. Bamha, and F. Loulergue. Handling data-skew effects in join operations using mapreduce. *Procedia Computer Science*, 29:145–158, 2014. 2014 International Conference on Computational Science.
19. X. Hu, Y. Tao, and K. Yi. Output-optimal parallel algorithms for similarity joins. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 79–90. ACM, 2017.
20. P. Indyk. Approximate nearest neighbor algorithms for frechet distance via product metrics. In *Proceedings of the eighteenth annual symposium on Computational geometry - SCG '02*, pages 102–106. ACM Press, 2002.
21. P. Indyk. *Nearest Neighbors In High-Dimensional Spaces*. 2004.
22. M. Konzack, T. J. Mcketterick, T. Ophelders, M. Buchin, L. Giuggioli, J. Long, T. Nelson, M. A. Westenberg, and K. Buchin. Visual analytics of delays and interaction in movement data. *International Journal of Geographical Information Science*, 31(2):320–345, 2017.
23. A. Metwally and C. Faloutsos. V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors. *Proc. VLDB Endow.*, 5(8):704–715, Apr. 2012.
24. E. Sriraghavendra, K. K, and C. Bhattacharyya. Fréchet distance based approach for searching online handwritten documents. In *Ninth International Conference on Document Analysis and Recognition (ICDAR 2007)*, pages 461–465. IEEE Computer Society, 2007.
25. M. Thorup. High speed hashing for integers and strings. *CoRR*, abs/1504.06804, 2015.

26. M. Werner and D. Oliver. ACM SIGSPATIAL GIS cup 2017: range queries under fréchet distance. *SIGSPATIAL Special*, 10(1):24–27, 2018.
27. D. Xie, F. Li, and J. M. Phillips. Distributed trajectory similarity search. *Proc. VLDB Endowment*, 10(11):1478–1489, Aug. 2017.
28. H. Yuan and G. Li. Distributed In-memory Trajectory Similarity Search and Join on Road Network. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1262–1273. IEEE, 2019.

6 Annexe

Algorithm 0.A: getKey – Dynamic key generation of the reduce step in the Algorithm 4.A of Job ❹.

getKey : $(\overline{lsh\_keys}_R, \overline{lsh\_keys}_S) \rightarrow x \in L(R) \cap L(S)$
  E $\leftarrow \overline{lsh\_keys}_R \cap \overline{lsh\_keys}_S$
  sum $\leftarrow \sum_{e \in E} e$
  index $\leftarrow$ sum % $|E|$
  return E[index]