# MapReduce-based Similarity Join for Metric Spaces

Yasin N. Silva
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
ysilva@asu.edu

Jason M. Reed
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
jmreed3@asu.edu

Lisa M. Tsosie
Arizona State University
4701 W. Thunderbird Road
Glendale, AZ 85306, USA
lmtsosi1@asu.edu

## ABSTRACT

Cloud enabled systems have become a crucial component to efficiently process and analyze massive amounts of data. One of the key data processing and analysis operations is the Similarity Join, which retrieves all data pairs whose distances are smaller than a predefined threshold $\varepsilon$. Even though multiple algorithms and implementation techniques have been proposed for Similarity Joins, very little work has addressed the study of Similarity Joins for cloud systems. This paper focuses on the study, design and implementation techniques of cloud-based Similarity Joins. We present *MR-SimJoin*, a MapReduce based algorithm to efficiently solve the Similarity Join problem. This algorithm efficiently partitions and distributes the data until the subsets are small enough to be processed in a single node. MRSimJoin is general enough to be used with data that lies in any metric space, thus it can be used with multiple data types and distance functions. We present guidelines to implement the algorithm in Hadoop, an open-source cloud system. The experimental evaluation of MRSimJoin shows that it has very good execution time and scalability properties.

## Categories and Subject Descriptors

H.2.4 [**Database management**]: Systems—*query processing, parallel databases*

## General Terms

Algorithms, Design, Experimentation, Performance

## Keywords

Similarity Join, MapReduce, Hadoop, Metric Space

## 1. INTRODUCTION

Similarity Join is one of the most useful data processing and analysis operations. It retrieves all data pairs whose distances are smaller than a predefined threshold $\varepsilon$. Multiple application scenarios need to perform this operation over large amounts of data. Internet companies, for instance, collect massive amounts of data such as content produced by web crawlers or service logs, and can use similarity queries to gain valuable understanding of the use of their services, e.g., identify customers with similar buying patterns, generate recommendations, perform correlation analysis, etc. Cloud systems and MapReduce [5], its main framework for distributed processing, constitute an answer to the requirements of processing massive amounts of data in a highly scalable and distributed fashion. Cloud systems are composed of large clusters of commodity machines and are often dynamically scalable, i.e., nodes can be added or removed based on the workload. The MapReduce framework quickly processes massive datasets by splitting them into independent chunks that are processed in parallel. Multiple Similarity Join algorithms have been proposed. They range from approaches for only in-memory or external memory data to techniques that make use of database operators to answer Similarity Joins. Unfortunately, there has not been much work on the study of this operation on cloud systems. This paper focuses on the design and implementation of MapReduce-based Similarity Joins for metric spaces. Our contributions are:

- We present MRSimJoin, an efficient MapReduce-based Similarity Join algorithm. MRSimJoin extends the single-node QuickJoin algorithm [9] by adapting it to the MapReduce framework and integrating grouping, sorting and parallelization techniques.
- The proposed algorithm is general enough to be used with any dataset that lies in a metric space. The algorithm can be used with various distance functions and data types e.g., numerical data, vector data, text, etc.
- We present guidelines to implement the algorithm in Hadoop [1], a highly used open-source cloud system.
- We thoroughly evaluate the performance and scalability properties of the implemented operation with synthetic and real-world data. We show that MRSimJoin performs significantly better than an adaptation of the state-of-the-art MapReduce Theta-Join algorithm [11] (up to 15 times faster). MRSimJoin scales very well when important parameters like epsilon, data size, number of nodes, and number of dimensions increase.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 describes the MR-SimJoin algorithm. The guidelines to implement MRSimJoin in Hadoop are described in Section 4. Section 5 presents the performance evaluation and Section 6 concludes the paper.


*Cloud-I '12,* August 31 2012, Istanbul, Turkey

## 2. RELATED WORK

Most of the work on Similarity Join has considered the case of non-distributed solutions, e.g., EGO [3] and Quick-Join [9]. The Quickjoin algorithm [9], which has been shown to outperform EGO, recursively partitions the data until the subsets are small enough to be efficiently processed using a nested loop join. The algorithm makes recursive calls to process partitions and the *windows* around the partitions' boundaries. The MRSimJoin approach presented in this paper extends the single-node QuickJoin algorithm by adapting it to the distributed MapReduce framework and integrating grouping, sorting and parallelization techniques (physical partitioning and distribution of data in a computer cluster). Also of importance is the work on Similarity Join techniques in the context of database systems. Some work focused on answering Similarity Join queries using standard database operators [4, 7]. More recently, Similarity Joins have been studied as first-class database operators [12].

The MapReduce framework was introduced in [5]. The Map-Reduce-Merge variant [17] extends this framework with a *merge* phase after the reduce stage to facilitate the implementation of operations like join. Map-Join-Reduce [10] is another MapReduce variant that adds a *join* stage before the reduce stage. In this approach, mappers read from input relations, the output of mappers is distributed to joiners where the actual join task takes place, and the output of joiners is processed by the reducers. Most of the previous work on MapReduce-based Joins considers the case of equi-joins. The two main types of MapReduce-based joins are Map-side joins (e.g., Map-Merge and Broadcast Join) and Reduce-side joins (e.g., Repartition join). The Map-Merge approach [16] has two steps: in the first one, input relations are partitioned and sorted, and in the second one, mappers merge the intermediate results. Broadcast Join [2] considers the case where one of the relations is small enough to be sent to all mappers and maintained in memory. In the Repartition join approach [16], the mappers augment each record with a label that identifies the relations where it comes from. All the records that have the same join attribute value are sent to the same reducer. Reducers in turn produce the join pairs.

Recently, a MapReduce-based approach was proposed to implement Theta-joins [11]. This previous work proposed a randomized algorithm that requires some basic statistics (input cardinality). The approach proposes a model that partitions the input relations using a matrix that considers all the combinations of records that would be required to answer a cross product. The matrix cells are then assigned to reducers in a way that minimizes job completion time. A memory-aware variant is also proposed for the common scenario where partitions do not fit in memory. This previous work represents the state-of-the-art approach to answer arbitrary joins in MapReduce.

A related work that also addresses the problem of Similarity Joins in the context of cloud systems is the one presented in [15]. The work in [15], however, focuses on the study of a different and more specialized type of Similarity Join (Set-Similarity Join) which constrains its applicability to set-based data. The main differences between the work in [15] and the work in this paper are: (1) we consider the case of the most extensively used type of Similarity Join (distance range join), and (2) our approach can be used with data that lies in any metric space, i.e., our approach can be used with a wide variety of data types and distance functions.

A demonstration paper that shows the use of MRSimJoin in several real-world scenarios is presented in [13]. The contributions of this paper that go beyond what was presented in [13] include: (1) algorithmic details of the different MR-SimJoin functions (map, reduce, compare), (2) an extensive evaluation of MRSimJoin's performance, and (3) additional key information about MRSimJoin, e.g., how it is affected by the number of pivots and an expression to compute a good value for this parameter.

## 3. THE MRSimJoin ALGORITHM

The MapReduce framework works by dividing the processing task into two phases: *map* and *reduce*. The framework user is required to provide two functions (*map* and *reduce*). These functions have the following general form:

```
map: (k1,v1) → list(k2,v2)
reduce: (k2,list(v2)) → list(k3,v3)
```

Multiple *map* tasks process independent input chunks in parallel. Each *map* call is given a pair `(k1,v1)` and produces a list of `(k2,v2)` pairs. The output of the *map* calls is transferred to the *reduce* nodes (*shuffle* phase). All the intermediate records with the same intermediate key (k2) are sent to the same reducer node. At each *reduce* node, the received intermediate records are sorted and grouped. Each formed group is processed in a single *reduce* call. The processes of transferring the *map* outputs to the *reduce* nodes, sorting the records at each *reduce* node, and grouping these records are driven by the following functions, respectively:

```
partition: k2 → partitionNumber
sortCompare: (k2_1,k2_2) → {-1, 0, 1}
groupCompare: (k2_1,k2_2) → {-1, 0, 1}
```

The default *partition* function receives an intermediate key ($k2$) as input and generates a partition number based on a hash value for $k2$. When the default comparator functions are used, the intermediate records in a *reduce* node are sorted by the intermediate key and a group is formed for each different value.

The Similarity Join (SJ) operation between two datasets $R$ and $S$ is defined as $R \bowtie_{\theta_\varepsilon(r,s)} S = \{\langle r, s\rangle | \theta_\varepsilon(r,s), r \in R, s \in S\}$, where $\theta_\varepsilon(r,s)$ is the SJ predicate ($dist(r,s) \leq \varepsilon$). In general, the input data can be given in one or multiple distributed files. Each input data file contains a sequence of key-value records of the form $(id, (id, elem))$ where $id$ contains two components: the id of the dataset or relation this record belongs to ($id.relID$) and the id of the record in the relation ($id.uniqueKey$).

The MRSimJoin algorithm iteratively partitions the input data into smaller partitions until each partition is small enough to be efficiently processed by a single-node Similarity Join routine, i.e., the entire partition can be stored in memory in a single machine. The overall process is divided into a sequence of rounds. The initial round partitions the input data while any subsequent round partitions the data of a previously generated partition. Each round corresponds to a MapReduce job. The input and output of each job is read from or written to the distributed file system. The output of a round includes: (1) result pairs (links) for the small partitions that were processed in a single-node, and (2) intermediate data for the partitions that will require further

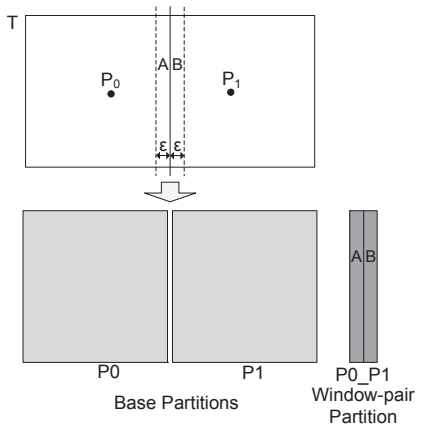


**Figure 1: Partitioning a base partition.**

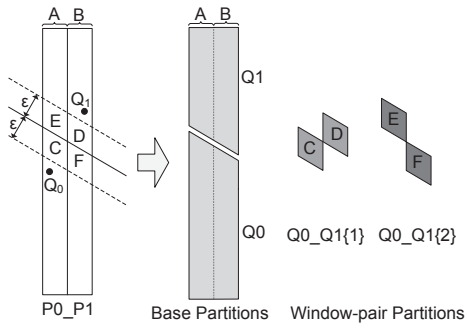


**Figure 2: Partitioning a window-pair partition.**

partitioning. The main routine of MRSimJoin executes the required rounds until all the input and intermediate data is processed.

Data partitioning is performed using a set of $K$ pivots, i.e., a random subset of the data records to be partitioned. The process generates two types of partitions: *base partitions* and *window-pair partitions*. A base partition contains all the records that are closer to a given pivot than to any other pivot. A window-pair partition contains the records in the boundary between two base partitions. In general, the window-pair records should be a superset of the records whose distance to the hyperplane that separates the base partitions is at most $\varepsilon$. Unfortunately, this hyperplane does not always explicitly exist in a metric space. Instead, the hyperplane is implicit and known as a *generalized hyperplane*. Since the distance of a record $t$ to the generalized hyperplane between two partitions with pivots $P_0$ and $P_1$ cannot always be computed exactly, a lower bound is used [8]:

$$gen\_hyperplane\_dist(t, P_0, P_1) = (dist(t, P_0) - dist(t, P_1))/2$$

This distance is replaced with an exact distance if this can be computed, e.g., in Euclidean spaces.

Processing the window-pair partitions guarantees the identification of the links between records that belong to different base partitions. A round that further repartitions a base partition or the initial input data is referred to as a *base partition round*, a round that repartitions a window-pair partition is referred to as a *window-pair partition round*. At the logical level, the data partitioning in MRSimJoin is similar

**Algorithm 1** *MRSimJoin*($inDir$, $outDir$, $numPiv$, $eps$, $memT$)

**Input:** $inDir$ (input directory with the records of datasets $R$ and $S$), $outDir$ (output directory), $numPiv$ (number of pivots), $eps$ (epsilon), $memT$ (memory threshold)
**Output:** $outDir$ contains all the results of the Similarity Join operation $R \bowtie_{\theta_\varepsilon(r,s)} S$

1. $intermDir \leftarrow outDir$ + "/intermediate"
2. $roundNum \leftarrow 0$
3. **while** true **do**
4. **if** $roundNum = 0$ **then**
5. $job\_inDir \leftarrow inDir$
6. **else**
7. $job\_inDir \leftarrow GetUnprocessedDir(intermDir)$
8. **end if**
9. **if** $job\_inDir =$ null **then**
10. break
11. **end if**
12. $pivots \leftarrow GeneratePivots(job\_inDir, numPiv)$
13. **if** $isBaseRound(job\_inDir)$ **then**
14. $MR\_Job(Map\_base, Reduce\_base, Partition\_base, Compare\_base, job\_inDir, outDir, pivots, numPiv, eps, memT, roundNum)$
15. **else**
16. $MR\_Job(Map\_windowPair, Reduce\_windowPair, Partition\_windowPair, Compare\_windowPair, job\_inDir, outDir, pivots, numPiv, eps, memT, roundNum)$
17. **end if**
18. $roundNum$++
19. **if** $roundNum > 0$ **then**
20. $RenameFromIntermToProcessed(job\_inDir)$
21. **end if**
22. **end while**

to the one in the Quickjoin algorithm [9]. The core difference, however, is that in MRSimJoin the partitioning of the data, the generation of the result links, and the storage of intermediate results is performed in a fully distributed and parallel manner. Fig. 1 represents the repartitioning of a base partition. In this case, the result of the Similarity Join operation on the dataset $T$ is the union of the links in $P0$ and $P1$, and the links in $P0\_P1$ where one element belongs to window $A$ and the other one to window $B$. We refer to this last type of links as *window links*. Fig. 2 represents the repartitioning of the window-pair partition $P0\_P1$ of Fig. 1. In this case, the set of window links in $P0\_P1$ is the union of the window links in $Q0$, $Q1$, $Q0\_Q1\{1\}$ and $Q0\_Q1\{2\}$. Note that windows $C$ and $F$ do not form a window-pair partition since their window links are a subset of the links in $Q0$. Similarly, the window links between $E$ and $D$ are a subset of the links in $Q1$. MRSimJoin inherits correctness from the Quickjoin algorithm.

The remaining part of this section presents the algorithmic details of the main MRSimJoin routine, and the base partition and window-pair partition rounds.

## 3.1 The Main Algorithm

The main routine of MRSimJoin is presented in Algorithm 1. The routine uses an intermediate directory (line 1) to store the partitions that will need further repartitioning. Each iteration of the while loop (lines 3 to 22) corresponds

**Algorithm 2** *Map_base*()

**Input:** $(k1, v1)$. $k1 = id, v1 = (id, elem)$
**Output:** list$(k2, v2)$. $k2 = (part, win)$, $v2 = (id, elem, part)$

1. $p \leftarrow GetClosestPivotIndx(elem, pivots)$
2. output $((p, -1), (id, elem, -1))$
3. **for** $i = 0 \rightarrow numPiv - 1$ **do**
4. **if** $i \neq p$ **then**
5. **if** $(dist(elem, pivots[i]) - dist(elem, pivots[p]))/2 \leq eps$ **then**
6. output $((p, i), (id, elem, p))$
7. **end if**
8. **end if**
9. **end for**

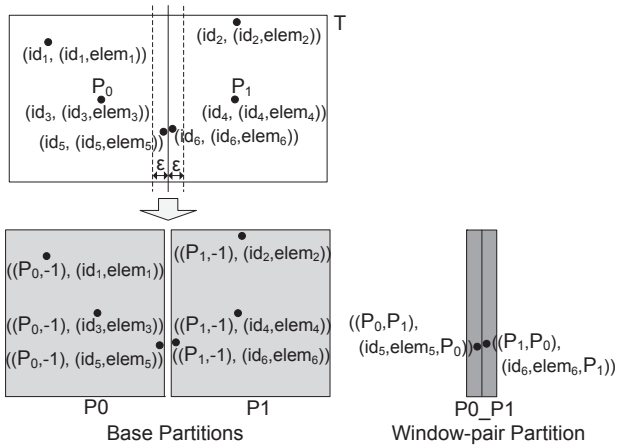


**Figure 3: Output of Map function - base round.**

to one round and executes a MapReduce job. In each round, the initial input data or a previously generated partition is repartitioned. If a newly generated partition is small enough to be processed in a single node, the Similarity Join links are obtained running a single-node Similarity Join algorithm. In our implementation we use Quickjoin [9]. Larger partitions are stored as intermediate data for further processing.

For each round, the routine sets the values of the job input directory (lines 4 to 8) and randomly selects *numPivots* pivots from this directory (line 12). Then the routine executes a base partition MapReduce job (line 14) or a window-pair partition MapReduce job (line 16) based on the type of the job input directory. The routine *MR_Job* sends the pivots to all mappers as explained in Section 4 and starts a MapReduce job that will use the provided *map*, *reduce*, *partition* and *compare* functions. The *partition* function will be used to replace the default *partition* function. The *compare* function will be used to replace the default *sortCompare* and *groupCompare* functions.

## 3.2 Base Partition Round

A base partition round processes the initial input data or a base partition previously generated by a base partition round. The goal of a base partition MapReduce job is to partition its input data and produce: (1) the result links for partitions that are small enough to be processed in a single node, and (2) intermediate data for partitions that require further processing.

**Algorithm 3** *Partition_base*()

**Input:** $k2$. $k2 = (part, win)$
**Output:** $k2$'s partition number

1. **if** $win = -1$ **then** // base partition
2. $partition \leftarrow (part \times C1) \bmod NUMPARTITIONS$
3. **else** // window-pair partition
4. $minVal \leftarrow min(part, win)$
5. $maxVal \leftarrow max(part, win)$
6. $partition \leftarrow (minVal \times C2 + maxVal \times C3) \bmod NUMPARTITIONS$
7. **end if**

*Map_base*, the map function for the base partition rounds, is presented in Algorithm 2. We use the value -1 when a given parameter is not applicable or will not be needed in the future. The MapReduce framework divides the job input data into chunks and creates *map* tasks in multiple nodes to process them. Each *map* task is called multiple times and each call executes the *Map_base* function for a given record $(id, (id, elem))$ of the input data. The *Map_base* function identifies the closest pivot $p$ to $elem$ (line 1). The function then outputs one intermediate key-value pair of the form $((p, -1), (id, elem, -1))$ for the base partition that $elem$ belongs to (line 2) and one key-value pair of the form $((p, i), (id, elem, p))$ for each window-pair partition (corresponding to pivots $p$ and $i$) that $elem$ belongs to (lines 3 to 9). Fig. 3 shows an example of the intermediate key-value pairs generated by the *map* tasks.

The MapReduce framework partitions the intermediate data generated by *map* tasks. This partitioning is performed calling the *Partition_base* function presented in Algorithm 3. *Partition_base* receives an intermediate key, i.e., $k2 = (part, win)$, as input and generates the corresponding partition number. $C1 - C3$ are constant prime numbers and *NUMPARTITIONS* is the maximum number of partitions set by the MapReduce framework. The partition number for an intermediate key that corresponds to a base partition is computed using a hash function on *part* (line 2). When the key corresponds to a window-pair partition, the partition number is computed using a hash function on $min(part, win)$ and $max(part, win)$ (line 6). This last hash function will generate the same partition number for all intermediate records of a window-pair partition independently of the specific window they belong to.

After identifying the partition numbers of intermediate records, the shuffle phase of the MapReduce job sends the intermediate records to their corresponding reduce nodes. The intermediate records received at each reduce node are sorted and grouped using the *Compare_base* function presented in Algorithm 4. The main goal of the *Compare_base* function is to group the intermediate records that belong to the same partition. The function establishes the order of partitions shown in Fig. 4.a. Base partitions have lower order than window-pair partitions. Multiple base partitions are ordered based on their pivot indices. Multiple window-pair partitions are ordered based on the two associated pivot indices of each partition. Fig. 4.b shows the order of partitions generated by *Compare_base* for the scenario with two pivots presented in Fig. 3. Fig. 4.c shows the order of partitions for the case of a base round with three pivots.

After generating the groups in a reduce node, the MapReduce framework calls the *reduce* function *Reduce_base* once

**Algorithm 4** *Compare_base*()

**Input:** $k2_1, k2_2$. $k2_1 = (part_1, win_1), k2_2 = (part_2, win_2)$
**Output:** 0 ($k2_1$ and $k2_2$ belong to the same group), $-1$ (group number of $k2_1$ < group number of $k2_2$), or $+1$ (group number $k2_1$ > group number of $k2_2$)

1. **if** $(win_1 = -1) \wedge (win_2 = -1)$ **then** // basePart-basePart
2. **if** $(part_1 = part_2)$ **then**
3. return 0
4. **else**
5. return $(part_1 < part_2)? -1 : +1$
6. **end if**
7. **else if** $(win_1 = -1) \wedge (win_2 \neq -1)$ **then** // basePart-winPart
8. return -1
9. **else if** $(win_1 \neq -1) \wedge (win_2 = -1)$ **then** // winPart-basePart
10. return $+1$
11. **else** // $(win_1 \neq -1) \wedge (win_2 \neq -1)$, winPart-winPart
12. $min_1, max_1 \leftarrow min(part_1, win_1), max(part_1, win_1)$
13. $min_2, max_2 \leftarrow min(part_2, win_2), max(part_2, win_2)$
14. **if** $(min_1 = min_2) \wedge (max_1 = max_2)$ **then** // elements belong to the same window-pair
15. return 0
16. **else** // elements do not belong to the same window-pair
17. **if** $min_1 = min_2$ **then**
18. return $(max_1 < max_2)? -1 : +1$
19. **else**
20. return $(min_1 < min_2)? -1 : +1$
21. **end if**
22. **end if**
23. **end if**

**Algorithm 5** *Reduce_base*()

**Input:** $(k2, v2List)$. $k2 = (kPart, kWin)$, $v2List$ = list$(id, elem, part)$
**Output:** SJ matches or intermediate data. Intermediate data = list$(k3, v3)$. $k3 = id$, $v3 = (id, elem[, part])$

1. **if** $sizeInBytes(v2List) \leq memT$ **then**
2. $InMemorySimJoin(v2List, outDir, eps)$
3. **else**
4. **if** $kWin = -1$ **then**
5. **for** each element $e$ of $v2List$ **do**
6. output $(e.id, (e.id, e.elem))$ to $outDir/$ intermediate/B_$\langle roundNum \rangle$_$\langle kPart \rangle$
7. **end for**
8. **else**
9. **for** each element $e$ of $v2List$ **do**
10. output $(e.id, (e.id, e.elem, e.part))$ to $outDir/$ intermediate/W_$\langle roundNum \rangle$_$\langle kPart \rangle$_$\langle kWin \rangle$
11. **end for**
12. **end if**
13. **end if**

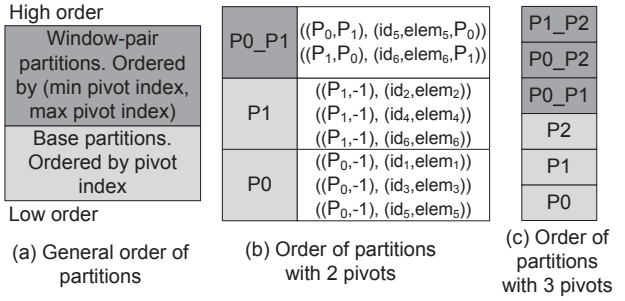


**Figure 4: Group formation in a base round.**

for each group. This function is presented in Algorithm 5. The function receives as input the key-value pair $(k2, v2List)$. $k2$ is the intermediate key of one of the records of the group being processed and $v2List$ is the list of values of all the records of the group. If the size of the list is small enough to be processed in a single node, the algorithm calls a single-node SJ routine, i.e., *InMemorySimJoin*, to get the links in the current partition (lines 1 to 2). Otherwise all the records of the group are stored in an intermediate directory for further partitioning. If the current group is a base partition, the algorithm stores its records in a directory that will be processed in a future base partition round (lines 4 to 7). Likewise, the records of a window-pair partition are stored in a directory that will be processed in a future window-pair partition round (lines 8 to 12). In the scenario represented in Fig. 3, the MapReduce framework calls the *Reduce_base* function for each partition of Fig. 4.b: $P0$, $P1$ and $P0\_P1$.

## 3.3 Window-pair Partition Round

A window-pair partition round processes a window-pair partition generated by a base round or any partition generated by a window-pair round. Like base partition rounds, window-pair partition rounds generate result links and intermediate data. However, the links generated in a window-pair round are window links, i.e., links between records of different previous partitions. A window-pair round uses the functions *Map_windowPair*, *Reduce_windowPair*, *Partition_windowPair* and *Compare_windowPair* in a similar way their counterparts are used in a base partition round. This section explains these functions highlighting the differences. Additional details appear in a technical report [14].

Like *Map_base*, *Map_windowPair* outputs intermediate records for the base and window-pair partitions that a record belongs to. The difference is in the format of the records. The format of the input key-value pair, i.e., $k1, v1$, is: $k1 = id, v1 = (id, elem, prevPart)$, and the format of the intermediate key-value pairs, i.e., $k2, v2$, is: $k2$=($part$, $win$, $prevPart$), $v2$=($id$, $elem$, $part$, $prevPart$). The information of the previous partition of a record is maintained to identify the new window-pair partitions that the record belongs to. Fig. 5 shows an example of the generated records.

*Partition_windowPair* is similar to *Partition_base* but receives an intermediate key of the form $k2 = (part, win, prevPart)$ and distinguishes between the two window-pair partitions of any pair of pivots. In the scenario of Fig. 5, *Partition_windowPair* generates the same partition number for all the intermediate keys that correspond to partition $Q0\_Q1\{1\}$. The number at the end of a window-pair partition name is referenced as the window-pair *sequence*.

*Compare_windowPair* groups all the records that belong to the same partition establishing the order of partitions shown in Fig. 6.a. The algorithm considers all the possible combinations of the intermediate keys. All the cases are processed as in *Compare_base* with the exception of the case where both keys belong to window-pair partitions. In this

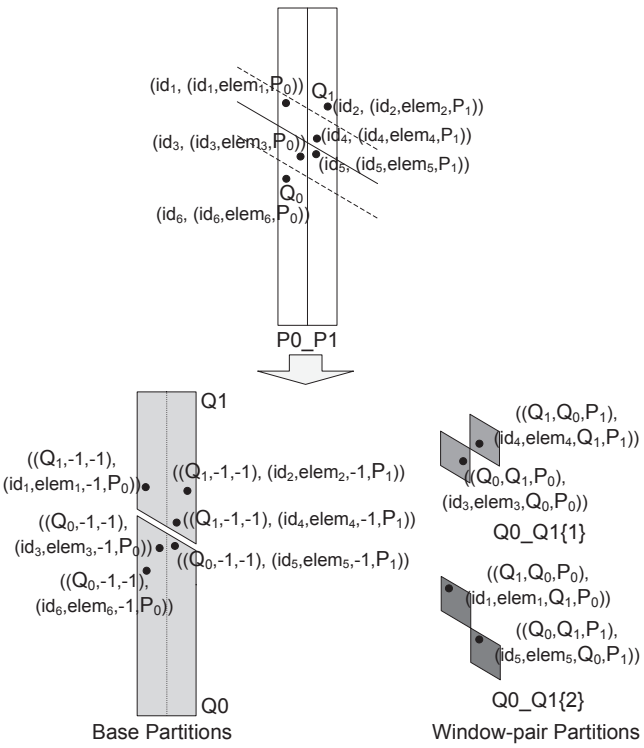


**Figure 5: Output of Map function - window-pair round.**

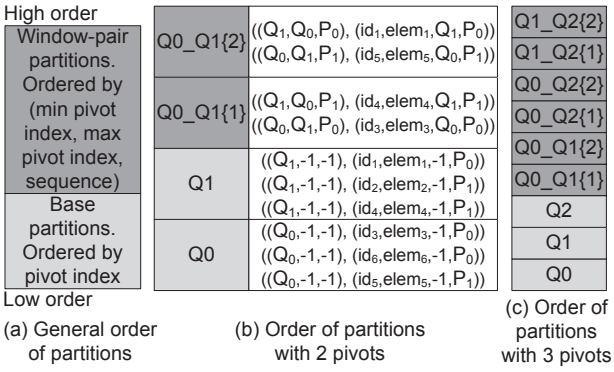


**Figure 6: Group formation in a window-pair round.**

case, *Compare_windowPair* orders them based on the tuple (minimum pivot index, maximum pivot index, window-pair sequence) using lexicographical order. Fig. 6.b shows the order of partitions generated by *Compare_windowPair* for the scenario with two pivots presented in Fig. 5. Fig. 6.c shows the order of partitions for the case of a window-pair round with three pivots.

*Reduce_windowPair* is similar to *Reduce_base*, but in this case, all partitions that need further processing are set to be repartitioned by a future window-pair partition round. This is the case because the links generated in a window-pair round or in any of its generated partitions should be window links. In the scenario of Fig. 5, the MapReduce framework calls the *Reduce_windowPair* function for each partition of Fig. 6.b: $Q0$, $Q1$, $Q0\_Q1\{1\}$ and $Q0\_Q1\{2\}$.

# 4. IMPLEMENTATION IN HADOOP

The MRSimJoin algorithm is generic enough to be implemented in any MapReduce framework. This section presents additional guidelines for its implementation on Hadoop [1].

***Distribution of atomic parameters***. *MR_Job* sends the atomic parameters, i.e., $outDir$, $numPiv$, $eps$ and $memT$, to every node that will be used in the MapReduce job. This is done using Hadoop's job configuration $jobConf$ object.

***Distribution of pivots***. *MR_Job* sends the pivots to every node that executes *map* tasks. This is done using $DistributedCache$, a facility that allows the efficient distribution of application-specific, large, read-only files.

***Renaming directories***. The main MRSimJoin routine renames a directory to flag it as already processed. This is done using the *rename* method of Hadoop's *FileSystem* class. The method will change the directory path in the distributed file system without physically moving its data.

# 5. PERFORMANCE EVALUATION

## 5.1 Test Configuration

We implemented MRSimJoin using Hadoop 0.20.2 and performed the experiments using a Hadoop cluster running on the Amazon Elastic Compute Cloud (EC2). Unless otherwise stated, we used a cluster of 10 nodes (1 master + 9 worker nodes) with the following specifications: 15 GB of memory, 4 virtual cores with 2 EC2 Compute Units each, 1,690 GB of local instance storage, 64-bit platform. We set the block size of the distributed systems to 64 MB and the total number of reducers to: $0.95 \times \langle$no. worker nodes$\rangle \times \langle$max reduce tasks per node$\rangle$. We use the following datasets:

***SynthData***. This is a synthetic vector dataset (up to 16D). The components of each vector are randomly generated numbers in the range [0 - 1,000]. The dataset for scale factor 1 (SF1) contains 5 million records (1.3 GB).

***ColorData***. This dataset contains 9D feature vectors extracted from a Corel image collection [6]. The vector components are in the range [-4.8 - 4.4]. The original dataset contains 68,040 records. The SF1 dataset contains 5 million records (390 MB) and was generated following the same process to generate datasets for higher SFs.

The datasets for SF greater than 1 were generated in such a way that the number of links of any SJ operation in SF$N$ are $N$ times the number of links of the operation in SF1. Specifically, the datasets for higher SFs were obtained adding shifted copies of the SF1 dataset such that the separation between the region of new vectors and the region of previous vectors is greater than the maximum value of $\varepsilon$ used in our tests. Half of the records of each dataset belong to $R$ and the remaining ones to $S$. We use the Euclidean distance with both datasets. The available memory to perform the in-memory SJ algorithm (QuickJoin) was 32 MB.

We compare the performance of MRSimJoin with the one of MRThetaJoin, an adaptation of the memory-aware 1-Bucket-Theta algorithm proposed in [11] that uses the single-node QuickJoin algorithm [9] in the reduce function. We did not include the execution time of MRThetaJoin when the algorithm took a relatively long time (more than 3 hours).

## 5.2 Performance Evaluation with SynthData

***Increasing Scale Factor***. Fig. 7 compares the way MRSimJoin and MRThetaJoin scale when the data size in-

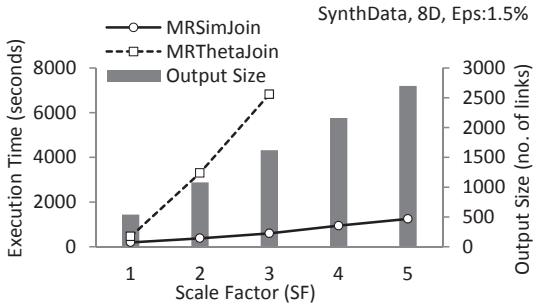


**Figure 7: Increasing SF - SynthData.**

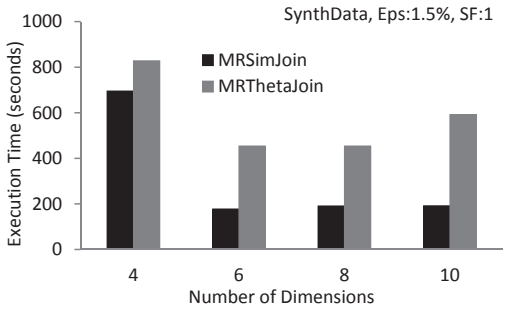


**Figure 9: Increasing Dimensions - SynthData.**

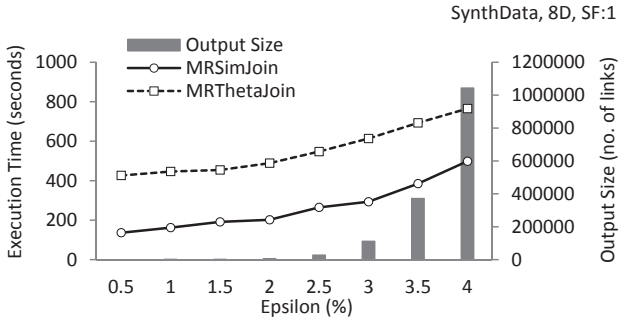


**Figure 8: Increasing Epsilon - SynthData.**

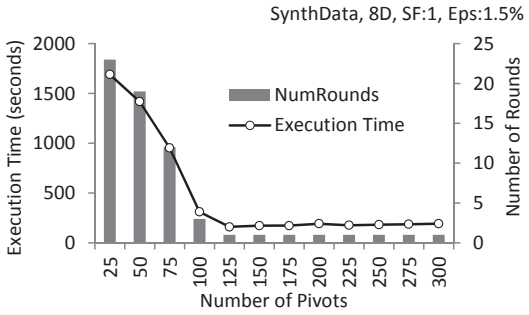


**Figure 10: Increasing # Pivots - SynthData.**

creases (SF1-SF5). This experiment uses 8D vectors and a value of epsilon of 1.5% of the maximum possible distance. The core results in this figure is that MRSimJoin performs and scales significantly better than MRThetaJoin. The execution time of MRThetaJoin grows from being 2.4 times the one of MRSimJoin for SF=1 to 11.4 times for SF=3. The execution time of MRThetaJoin is significantly higher than that of MRSimJoin because the total size of all the partitions of MRThetaJoin is significantly larger than that of MRSimJoin.

***Increasing Epsilon***. Fig. 8 shows how the execution time of MRSimJoin and MRThetaJoin increase when epsilon increases (0.5%-4.0%). The performance of MRSimJoin is better than the one of MRThetaJoin for all the values of epsilon. Specifically, the execution time of MRThetaJoin is between 1.5 to 3 times the one of MRSimJoin. We can also observe that, in general, the execution time of both algorithms grows slowly when epsilon grows. The increase in execution time is due to a higher number of distance computations in both algorithms and slightly larger sizes of window-pair partitions in the case of MRSimJoin.

***Increasing Number of Dimensions***. The execution time of MRSimJoin and MRThetaJoin for several numbers of dimensions (4D-10D) is presented in Fig. 9. The figure shows that MRSimJoin performs better than MRThetaJoin for all the numbers of dimensions considered. The execution time of MRThetaJoin is 20% higher than that of MRSimJoin for 4D and 200% higher for 10D. Observe also that, in general, the execution time of both algorithms decreases when the dimensionality increases. This is the case because the dataset gets more sparse when the number of dimensions increases (we maintain a constant number of tuples). Consequently, the number of records in the output decreases significantly in higher dimensions. The output size is about 46 million for 4D and less than 100 records for 10D.

***Increasing Number of Pivots***. The execution time and number of rounds for MRSimJoin, as the number of pivots increases, is presented in Fig. 10. The figure shows that smaller number of pivots generate higher number of rounds. We also observe that, in general, the execution time decreases when the number of rounds decreases. We found that in most of the experiments presented in this section, the best execution time is achieved using a single round. To compute the number of pivots ($P$) that will generate a single round for values of epsilon smaller than 25%, we can use the fact that the space needed for the in-memory QuickJoin algorithm is about twice the size of the input data [9]. Thus, to ensure that the average MRSimJoin base partition (and window-pair partition) can be solved using the in-memory QuickJoin we need: $P = 2 \times k \times D/T$, where $D$ is the total input size, $T$ is the available memory for QuickJoin, and $k$ is a factor that compensates the effect of data skewness on the size of partitions (we used $k = 2$). Using this expression, the value of $P$ for this experiment is 166. This value of $P$ generates a single round and an execution time that is only 8% higher than the best execution time (obtained with $P$=125).

## 5.3 Performance Evaluation with ColorData

***Increasing Scale Factor***. Fig. 11 compares the way MRSimJoin and MRThetaJoin scale when the data size increases. The results for ColorData are similar to the ones we found for the case of synthetic vector data. The execution time of MRThetaJoin grows from being 1.6 times of that of MRSimJoin for SF1 to 13.3 times for SF4.

***Increasing Epsilon***. Fig. 12 shows how the execution times of MRSimJoin and MRThetaJoin increase when epsilon increases. The results of this test are also inline with

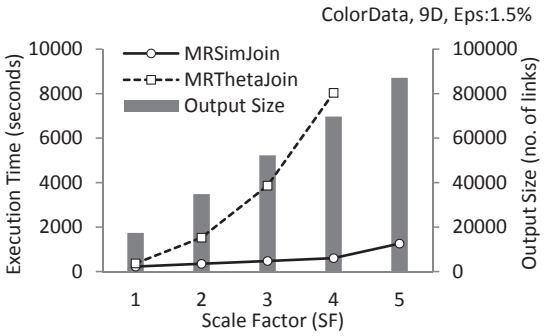


**Figure 11: Increasing SF - ColorData.**

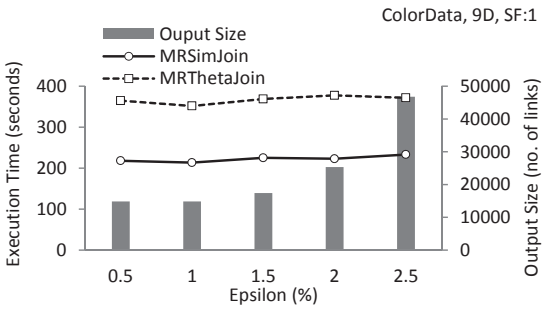


**Figure 12: Increasing Epsilon - ColorData.**

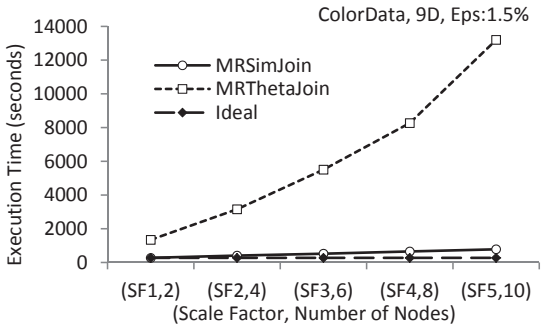


**Figure 13: Increasing # Nodes and SF - ColorData.**

the ones of the test with SynthData. The execution times of both algorithms grow slowly and in all cases the execution time of MRSimJoin is about 60% of that of MRThetaJoin.

***Increasing Number of Nodes and Scale Factor***. One of the goals of cloud-based operations is to scale efficiently when the number of nodes and the data size increase proportionally. Ideally, the execution time should remain constant. Fig. 13 shows the execution time of MRSimJoin and MRThetaJoin when the data size and the number of nodes increase from (SF1, 2 nodes) to (SF5, 10 nodes). MRSimJoin follows the ideal execution time much more closely than MRThetaJoin. MRSimJoin's execution time for (SF5, 10) is 2.8 times the one for (SF1, 2). MRThetaJoin's execution time for (SF5, 10) is 9.8 times the one for (SF1, 2).

# 6. CONCLUSIONS

We present MRSimJoin, a MapReduce-based algorithm to efficiently solve the Similarity Join problem. MRSimJoin iteratively partitions the data until the partitions are small enough to be efficiently processed in a single node. Each iteration executes a MapReduce job that processes the generated partitions in parallel. The proposed algorithm can be used with any dataset that lies in a metric space. We implemented MRSimJoin using the Hadoop MapReduce framework. An extensive performance evaluation of MRSimJoin with synthetic and real-world data shows that it scales very well when important parameters increase. Furthermore, we show that MRSimJoin performs significantly better than an adaptation of the state-of-the-art MapReduce-based algorithm to answer arbitrary joins.

# 7. REFERENCES


[1] Apache. Hadoop. `http://hadoop.apache.org/`.

[2] 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 *ACM SIGMOD*, 2010.

[3] C. Böhm, B. Braunmüller, F. Krebs, and H.-P. Kriegel. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. In *ACM SIGMOD*, 2001.

[4] S. Chaudhuri, V. Ganti, and R. Kaushik. A primitive operator for similarity joins in data cleaning. In *ICDE*, 2006.

[5] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *OSDI*, 2004.

[6] A. Frank and A. Asuncion. UCI machine learning repository. `http://archive.ics.uci.edu/ml`, 2010.

[7] L. Gravano, P. G. Ipeirotis, H. V. Jagadish, N. Koudas, S. Muthukrishnan, and D. Srivastava. Approximate string joins in a database (almost) for free. In *VLDB*, 2001.

[8] G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). *ACM Trans. Database Syst.*, 28:517–580, December 2003.

[9] E. H. Jacox and H. Samet. Metric space similarity joins. *ACM Trans. Database Syst.*, 33:7:1–7:38, June 2008.

[10] D. Jiang, A. K. H. Tung, and G. Chen. Map-join-reduce: Toward scalable and efficient data analysis on large clusters. *IEEE Trans. on Knowl. and Data Eng.*, 23:1299–1311, September 2011.

[11] A. Okcan and M. Riedewald. Processing theta-joins using mapreduce. In *ACM SIGMOD*, 2011.

[12] Y. N. Silva, W. G. Aref, and M. H. Ali. The similarity join database operator. In *ICDE*, 2010.

[13] Y. N. Silva and J. M. Reed. Exploiting mapreduce-based similarity joins. In *ACM SIGMOD*, 2012.

[14] Y. N. Silva, J. M. Reed, and L. M. Tsosie. Technical report: Mapreduce-based similarity joins. `http://www.public.asu.edu/~ynsilva/tr/MRSJTechRep.pdf`, 2012.

[15] R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using mapreduce. In *ACM SIGMOD*, 2010.

[16] T. White. *Hadoop: The Definitive Guide*. Yahoo! Press, 2010.

[17] H.-c. Yang, A. Dasdan, R.-L. Hsiao, and D. S. Parker. Map-reduce-merge: simplified relational data processing on large clusters. In *ACM SIGMOD*, 2007.