Technical Report: MapReduce-based Similarity Joins

Yasin N. Silva
Arizona State University
ysilva@asu.edu

Jason M. Reed
Arizona State University
jmreed3@asu.edu

Lisa M. Tsosie
Arizona State University
lmtsosi1@asu.edu

June 25, 2012

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 ε. 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 *MRSimJoin*, 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. The proposed algorithm is general enough to be used with data that lies in any metric space. The algorithm can also be used with multiple data types, e.g., numerical data, vector data, text, etc. We present multiple guidelines to implement the algorithm in Hadoop, a highly used open-source cloud system. The extensive experimental evaluation of the implemented operation shows that it has very good execution time and scalability properties.

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 ε. Similarity Joins have been studied and extensively used in multiple application domains, e.g., record linkage, data cleaning, multimedia applications, sensor networks, marketing analysis, etc. 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, service logs, click streams, and so on, 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 [7], 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., cluster 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 a highly parallel fashion.

Multiple Similarity Join algorithms and implementation techniques 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 computing systems. This paper focuses on the study, design and implementation techniques of MapReduce-based Similarity Joins. The main contributions of our work are:

- We present MRSimJoin, a MapReduce-based algorithm, that efficiently solves the Similarity Join problem. MRSimJoin extends the previously proposed single-node QuickJoin algorithm [16] 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 multiple 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 [19] (up to 15 times faster). MRSimJoin scales very well when important parameters like epsilon, data size, number of nodes, and number of dimensions increase.

- Our experimental analysis considers important aspects of the algorithm that were not covered in the QuickJoin paper. Particularly, we study the effect of the number of pivots on the execution time and provide an expression to compute a good value for this parameter.

The remaining part of this paper is organized as follows. Section 2 presents the related work. Section 3 describes in detail the MRSimJoin algorithm and the enhancements for the case of Euclidean distance. Section 4 describes the guidelines to implement MRSimJoin in Hadoop. The performance evaluation of the implemented Similarity Join operation is studied in Section 5. Section 6 presents the conclusions and future research directions.

2 Related Work

Most of the work on Similarity Join has considered the case of non-distributed solutions. This previous work introduced the semantics of different types of Similarity Joins and proposed techniques to implement them primarily as standalone operations. Several types of Similarity Join have been proposed in the literature, e.g., distance range join (retrieves all pairs whose distances are smaller than a predefined threshold ε) [11, 10, 3, 9, 16], k-Distance join (retrieves the k most-similar pairs) [14], and kNN-join (retrieves, for each tuple in one dataset, the k nearest-neighbors in another one) [4]. The distance range join is one of the most studied and useful types of Similarity Joins. This type of join is commonly referred to simply as Similarity Join and is also the focus of this paper. Among its most relevant implementation techniques, we find approaches that rely on the use of pre-built indices, e.g., eD-index [11] and D-index [10]. These techniques strive to partition the data while clustering together the similar objects. Several non-index-based techniques have also been proposed to solve the Similarity Join problem, e.g., EGO [3], GESS [9] and QuickJoin [16]. The Quickjoin algorithm [16], which has been shown to outperform EGO and GESS, 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). MRSimJoin uses a similar way to logically partition the data into partitions and windows. MRSimJoin also makes use of the QuickJoin algorithm to solve the Similarity Join in a single node when the data is small enough. Also of importance is the work on Similarity Join techniques in the context of database systems. Some work has focused on the implementation of Similarity Joins using standard database operators [5, 13]. These techniques are applicable only to string or set-based data. The general approach pre-processes the data and query, e.g., decomposes data and query strings into sets of grams (substrings of a string that are used as its signature), and stores the results of

this stage on separate relational tables. Then, the result of the Similarity Join can be obtained using standard SQL statements. More recently, Similarity Joins have been proposed and studied as first-class database operators [23, 22]. This work proposes techniques to implement and optimize Similarity Joins inside database query engines.

The MapReduce framework was introduced in [7]. The Map-Reduce-Merge variant [26] extends the MapReduce framework with a *merge* phase after the reduce stage to facilitate the implementation of operations like join. Map-Join-Reduce [17] 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 consider the case of equi-joins. The two main types of MapReduce-based joins are Map-side joins and Reduce-side joins. Among the Map-side joins we have Map-Merge [25] and Broadcast Join [2, 6]. The Map-Merge approach [25] has two steps: in the first one, input relations are partitioned and sorted, and in the second one, mappers merge the intermediate results. The Broadcast Join approach [2, 6] considers the case where one of the relations is small enough to be sent to all mappers and maintained in memory. The overall execution time is reduced by avoiding sorting and distributing on both input relations. Repartition join [25] is the most representative instance of Reduce-side joins. In this approach, 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 [19]. 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. In this paper, we compare MRSimJoin with an adaptation of the memory-aware algorithm in [19] to answer Similarity Joins and show that MRSimJoin performs significantly better, i.e., the execution time of MRSimJoin is up to 15 times faster.

To the best of our knowledge, the only work that specifically addresses the problem of Similarity Joins in the context of cloud systems is the one presented in [24]. The work in [24], 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 [24] 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 comparison of the MapReduce framework and parallel databases is presented in [20]. Multiple parallel join algorithms have been proposed in the context of parallel databases, e.g., [18, 21, 8]. The work in [18] presents a comparison of several hash-based and sort-merge-based parallel join algorithms. A hash-based algorithm to address the case of data skew is presented in [21]. The algorithm dynamically allocates partitions to the processing units with the goal of assigning the same data volume to each unit. Our work bears some basic resemblance with the partitioning-based parallel join algorithm in [8]. However, the work in [8] focuses on the simple case of band joins with 1D numerical data.

3 Similarity Joins Using MapReduce

This section presents some preliminary information about MapReduce, describes in detail the MRSimJoin algorithm to solve the Similarity Join problem in a parallel manner and presents the modifications to improve the algorithm's performance for the case of Euclidean distance.

3.1 A Quick Introduction to MapReduce

MapReduce is the the main software framework for distributed processing over cloud systems [7]. This framework is able to quickly process massive amounts of data and works by dividing the processing task into two phases: *map* and *reduce*. The framework user is required to provide two functions, i.e., the *map* function and the *reduce* function. These functions have key-value pairs as inputs and outputs and have the following general form:

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

The data types of the key-value pairs are also specified by the framework user. Note that the input and output types of each function can be different. However, the input of the *reduce* function should use the same types as the output of the *map* function.

The execution of a MapReduce job works as follows. The framework splits the input dataset into independent data chunks that are processed by multiple independent *map* tasks in a parallel manner. Each *map* call is given a pair `(k1,v1)` and produces a list of `(k2,v2)` pairs. The output of the *map* calls is known as the intermediate output. The intermediate data is transferred to the *reduce* nodes by a process known as the *shuffle*. Each *reduce* node is assigned a different subset of the intermediate key space; these subsets are referred as *partitions*. The framework guarantees that all the intermediate records with the same intermediate key (k2) are sent to the same reducer node. At each *reduce* node, all the received intermediate records are sorted and grouped. Each formed group will be processed in a single *reduce* call. Multiple *reduce* tasks are also executed in a parallel fashion. Each *reduce* call receives a pair `(k2,list(v2))` and produces as output a list of `(k3,v3)` pairs.

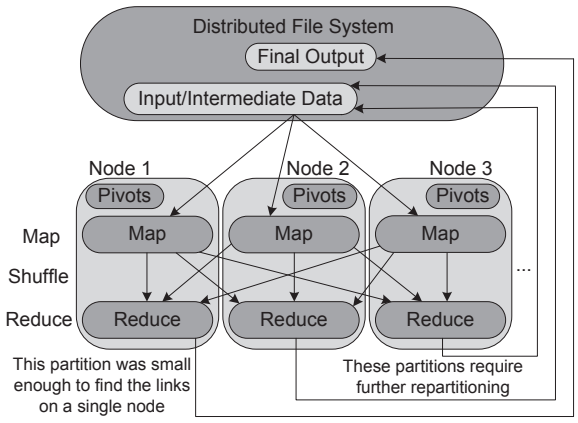

Figure 1: A MRSimJoin round.

The processes of transferring the *map* outputs to the *reduce* nodes, sorting the records at each destination node, and grouping these records are driven by the *partition*, *sortCompare* and *groupCompare* functions, respectively. These functions have the following form:

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

The default implementation of the *partition* function receives an intermediate key ($k2$) as input and generates a partition number based on a hash value for $k2$. The default *sortCompare* and *groupCompare* functions directly compare two intermediate keys ($k2_1$, $k2_2$) and return -1 ($k2_1 < k2_2$), 0 ($k2_1 = k2_2$), or $+1$ ($k2_1 > k2_2$). The result of using the default comparator functions is that all the intermediate records in a *reduce* node are sorted by the intermediate key and a group is formed for each different value of the intermediate key. Custom partitioner and comparator functions can be provided to replace the default functions.

The input and output data are usually stored in a distributed file system. The MapReduce framework takes care of scheduling tasks, monitoring them and re-executing them in case of failures.

3.2 The MRSimJoin Algorithm

The Similarity Join (SJ) operation between two datasets R and S is defined as follows:

$$R \bowtie_{\theta_\varepsilon(r,s)} S = \{\langle r, s\rangle | \theta_\varepsilon(r, s), r \in R, s \in S\}$$

where $\theta_\varepsilon(r, s)$ represents the Similarity Join predicate, i.e., $dist(r, s) \leq \varepsilon$.

The MRSimJoin algorithm presented in this section identifies all the pairs, i.e., links, that belong to the result of the Similarity Join operation. Furthermore, the algorithm can be used with any dataset that lies in a metric space. 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. 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 links for the small partitions that were processed in a single-node, and (2) intermediate data for the partitions that will require further partitioning. Fig. 1 represents the execution of a single MRSimJoin round. This figure shows that the partitioning process and the generation of results or intermediate data is performed in parallel by multiple nodes. 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, which are a 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 ε. 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 of the distance is used [15]:

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

This distance can be 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 to the one in the Quickjoin algorithm [16]. 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.

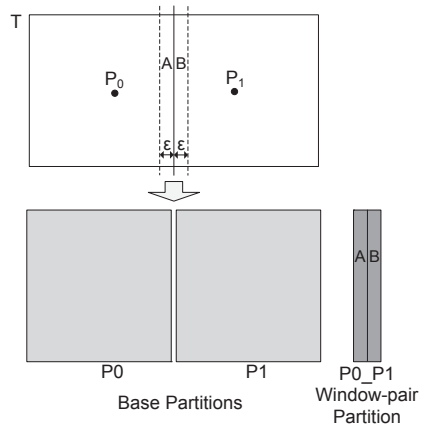

Figure 2: Partitioning a base partition.

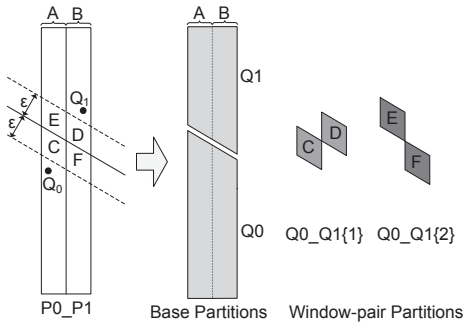

Figure 3: Partitioning a window-pair partition.

Fig. 2 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. 3 represents the repartitioning of the window-pair partition $P0\_P1$ of Fig. 2. 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 $Q1$.

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

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**

3.2.1 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. The names of intermediate directories that store base partitions have the following format:

$$\langle outDir \rangle \text{/intermediate/B\_}\langle roundNum \rangle \text{\_}\langle p \rangle$$

The names of intermediate directories storing window-pair partitions have the following format:

$$\langle outDir \rangle \text{/intermediate/W\_}\langle roundNum \rangle \text{\_[}\langle uAttr \rangle \text{]\_}\langle p1 \rangle \text{\_}\langle p2 \rangle$$

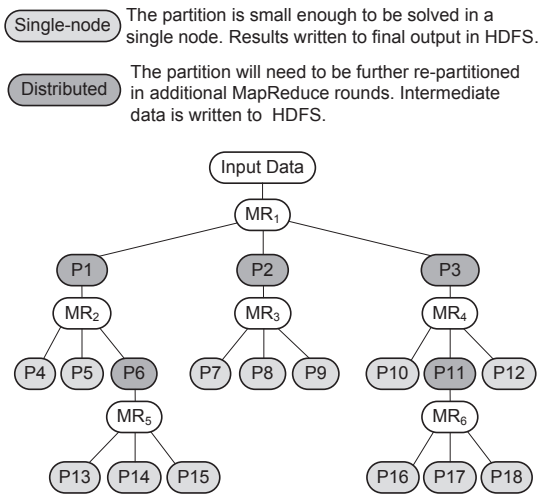

Figure 4: Example of the MapReduce rounds and partitions generated by MR-SimJoin.

p, $p1$ and $p2$ are pivot indices. $uAttr$ ensures unique directory names. The type of partition being stored in a directory can be identified from the directory name (use of B or W).

Each iteration of the while loop (lines 3 to 22) corresponds 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 [16]. Larger partitions are stored as intermediate data for further processing.

For each round, the main 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* sets up 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. *MR_Job* also makes sure that the provided atomic parameters, i.e., $outDir$, $numPiv$, eps and $memT$, are available at every node that will be used in the MapReduce job and that the *pivots* are available at each node that will execute *map* tasks.

If a round is processing a previously generated partition, after the MapReduce job finishes, the main routine renames the job input directory to relocate it under the processed directories (line 20).

Fig. 4 shows an example of the multiple rounds that are executed by the

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 ← GetClosestPivotIndx(elem, pivots)
2: output ((p, −1), (id, elem, −1))
3: for i = 0 → numPiv − 1 do
4:    if i ≠ p then
5:       if (dist(elem, pivots[i]) − dist(elem, pivots[p]))/2 ≤ eps then
6:          output ((p, i), (id, elem, p))
7:       end if
8:    end if
9: end for
```

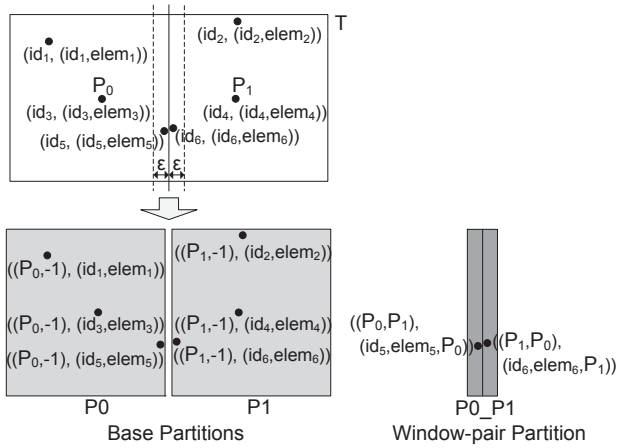

Figure 5: Example of the Map function for a base round.

main routine. Each node in the tree with name MR_N represents a MapReduce job. This figure also shows the partitions generated by each job. Light gray partitions are small partitions that are processed running the single-node Similarity Join routine. Dark gray partitions are partitions that require additional repartitioning. A sample sequence of rounds can be: MR_1, MR_2, MR_3, MR_4, MR_5 and MR_6. The original input data is always processed in the first round. Since the links of any partition can be obtained independently, the routine will generate a correct result independently of the order of rounds.

3.2.2 Base Partition Round

A base partition round processes the initial input data or a base partition previously generated by a base partition round. Each base partition round executes a MapReduce job initialized by the main routine as described in Section 3.2.1. The goal of a base partition MapReduce job is to partition its input data and

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**

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. The main routine sets up each base partition MapReduce job to use *Map_base* and *Reduce_base* as the *map* and *reduce* functions, respectively. Additionally, the default *partition* function is replaced by *Partition_base* and the default *sortCompare* and *groupCompare* functions are replaced by *Compare_base*. This section explains in detail each of these functions.

Map_base, the map function for the base partition rounds, is presented in Algorithm 2. The format of the input key-value pairs, i.e., $k1, v1$, is: $k1 = id, v1 = (id, elem)$, and the format of the intermediate key-value pairs, i.e., $k2, v2$, is: $k2 = (part, win), v2 = (id, elem, part)$. 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. 5 shows an example of the intermediate key-value pairs generated by *Map_base*. Region T contains all the key-value pairs of the job input data. Different segments of this region are processed by different *map* tasks on possibly different nodes. The overall result of the *map* phase is independent of the number or distribution of the *map* tasks. In this example, they will always generate the key-value pairs shown in partitions $P0$, $P1$ and $P0\_P1$. Each input record generates an intermediate key-value pair corresponding to its associated base partition ($P0$ or $P1$). Additionally, each record in the windows between the two base partitions, e.g., id_5 and id_6, generates a key-value pair corresponding to the window-pair partition $P0\_P1$.

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.

In the scenario of Fig. 5, *Partition_base* will generate the same partition number, i.e., $(P_0 \times C1)$ mod *NUMPARTITIONS*, for all the intermediate keys that correspond to partition $P0$. Similarly, the function will generate the same partition number, i.e., $(P_0 \times C2 + P_1 \times C3)$ mod *NUMPARTITIONS*, for all the intermediate keys that correspond to partition $P0\_P1$.

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. 6.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.

Compare_base receives as input two intermediate record keys, i.e., $k2_1, k2_2$, and returns 0 (when $k2_1$ and $k2_2$ belong to the same group), -1 (when $k2_1$ has lower order than $k2_2$), or $+1$ (when $k2_1$ has higher order than $k2_2$). The algorithm considers all the possible combinations of the intermediate keys. When both keys belong to base partitions, the algorithm orders them based on their pivot indices (lines 1 to 6). When one key belongs to a base partition and the other one to a window-pair partition, the algorithm orders them giving the base key a lower order than the window-pair key (lines 7 to 10). Finally, if both keys belong to window-pair partitions, the algorithm orders them based on the pair (minimum pivot index, maximum pivot index) using lexicographical order (lines 11 to 25). The *min* and *max* functions are used to group together all the intermediate records of a window-pair independently of the specific window they belong to.

Fig. 6.b shows the order of partitions generated by *Compare_base* for the scenario with two pivots presented in Fig. 5. Fig. 6.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 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 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) ∧ (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) ∧ (win_2 ≠ -1) then // basePart-winPart
 8:    return -1
 9: else if (win_1 ≠ -1) ∧ (win_2 = -1) then // winPart-basePart
10:    return +1
11: else // (win_1 ≠ -1) ∧ (win_2 ≠ -1), winPart-winPart
12:    min_1 ← min(part_1, win_1)
13:    max_1 ← max(part_1, win_1)
14:    min_2 ← min(part_2, win_2)
15:    max_2 ← max(part_2, win_2)
16:    if (min_1 = min_2) ∧ (max_1 = max_2) then // elements belong to the same
       window-pair
17:       return 0
18:    else // elements do not belong to the same window-pair
19:       if min_1 = min_2 then
20:          return (max_1 < max_2)? -1 : +1
21:       else
22:          return (min_1 < min_2)? -1 : +1
23:       end if
24:    end if
25: end if
```

algorithm calls a single-node Similarity Join 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 latter case, the last part of the directory name includes the indices of the two pivots associated to the window-pair partition. These values will be used in the algorithms of the window-pair rounds.

In the scenario represented in Fig. 5, the MapReduce framework calls the *Reduce_base* function for each partition of Fig. 6.b: $P0$, $P1$ and $P0\_P1$.

Algorithm 5 *Reduce_base*()

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

```
1: if sizeInBytes(v2List) ≤ 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_⟨roundNum⟩_⟨kPart⟩
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_⟨roundNum⟩_⟨kPart⟩_⟨kWin⟩
11:    end for
12:  end if
13: end if
```

3.2.3 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. Similarly to base partition rounds, window-pair partition rounds generate result links and intermediate data. However, the links generated 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.

Map_windowPair, the map function for the window-pair partition rounds,

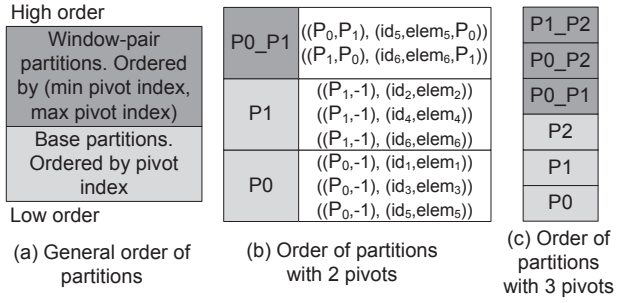

Figure 6: Group formation in a base round.

Algorithm 6 *Map_windowPair*()

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

```
1: p ← GetClosestPivotIndx(elem, pivots)
2: output ((p, −1, −1), (id, elem, −1, prevPart))
3: for i = 0 → numPiv − 1 do
4:    if i ≠ p then
5:       if (dist(elem, pivots[i]) − dist(elem, pivots[p]))/2 ≤ eps then
6:          output ((p, i, prevPart), (id, elem, p, prevPart))
7:       end if
8:    end if
9: end for
```

Algorithm 7 *Partition_windowPair*()

Input: $k2, W_1, W_2$. $k2 = (part, win, prevPart)$, W_1 and W_2 are the last two components of the job input directory name job_inDir
Output: $k2$'s partition number

```
1: if win = −1 then // base partition
2:    partition ← (part × C4) mod NUMPARTITIONS
3: else // window-pair partition
4:    minVal ← min(part, win)
5:    maxVal ← max(part, win)
6:    if (part > win ∧ prevPart = W1) ∨ (part < win ∧ prevPart = W2) then
7:       partition ← (minVal × C5 + maxVal × C6) mod NUMPARTITIONS
8:    else // (part > win ∧ prevPart = W2) ∨ (part < win ∧ prevPart = W1)
9:       partition ← (minVal × C7 + maxVal × C8) mod NUMPARTITIONS
10:   end if
11: end if
```

is presented in Algorithm 6. In this case, 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 function is similar to Map_base. The difference is in the format of the intermediate records. $Map\_windowPair$ outputs one intermediate key-value pair of the form ((p, -1, -1), (id, $elem$, -1, $prevPart$)) for the base partition with pivot p that $elem$ belongs to (line 2) and one key-value pair of the form ((p, i, $prevPart$), (id, $elem$, p, $prevPart$)) for each window-pair partition (corresponding to pivots p and i) that $elem$ belongs to (lines 3 to 9). Fig. 7 shows an example of the intermediate key-value pairs generated by $Map\_windowPair$.

The MapReduce framework partitions the intermediate data using the *Partition_windowPair* function presented in Algorithm 7. *Partition_windowPair* receives an intermediate key, i.e., $k2 = (part, win, prevPart)$, as input and gen-

Algorithm 8 *Compare_windowPair*()

Input: $k2_1, k2_2, W_1, W_2$. $k2_1 = (part_1, win_1, prevPart_1)$, $k2_2 = (part_2, win_2, prevPart_2)$, W_1 and W_2 are the last two components of the job input dir. name job_inDir

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** // baseP-baseP
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** // baseP-winP
8: return -1
9: **else if** $(win_1 \neq -1) \wedge (win_2 = -1)$ **then** // winP-baseP
10: return $+1$
11: **else** // $(win_1 \neq -1) \wedge (win_2 \neq -1)$, winP-winP
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** $\neg((min_1 = min_2) \wedge (max_1 = max_2))$ **then**
15: **if** $min_1 = min_2$ **then**
16: return $(max_1 < max_2)? -1 : +1$
17: **else**
18: return $(min_1 < min_2)? -1 : +1$
19: **end if**
20: **end if**
21: **if** $(part_1 = part_2) \wedge (prevPart_1 = prevPart_2)$ **then** // = part, = old part
22: return 0
23: **end if**
24: **if** $(part_1 = part_2)$ **then** // = part, $\neq$ old part.
25: **if** $part_1 < win_1$ **then** // $part_2 < win_2$
26: **if** $(prevPart_1 = W_1) \wedge (prevPart_2 = W_2)$ **then**
27: return -1
28: **else** // $(prevPart_1 = W_2) \wedge (prevPart_2 = W_1)$
29: return $+1$
30: **end if**
31: **else** // $part_1 > win_1 \wedge part_2 > win_2$
32: **if** $(prevPart_1 = W_1) \wedge (prevPart_2 = W_2)$ **then**
33: return $+1$
34: **else** // $(prevPart_1 = W_2) \wedge (prevPart_2 = W_1)$
35: return -1
36: **end if**
37: **end if**
38: **end if**

```
39:   if (prevPart_1 = prevPart_2) then // ≠ part, = old partition
40:     if prevPart_1 = win_1 then // prevPart_2 = win_1
41:       if part_1 < win_1 then // part_2 > win_2
42:         return -1
43:       else // (part_1 > win_1) ∧ (part_2 < win_2)
44:         return +1
45:       end if
46:     else // prevPart_1 = win_2 ∧ prevPart_2 = win_2
47:       if part_1 < win_1 then // part_2 > win_2
48:         return +1
49:       else // (part_1 > win_1) ∧ (part_2 < win_2)
50:         return -1
51:       end if
52:     end if
53:   end if
54:   return 0 // ≠ partitions, ≠ old partitions
55: end if
```

erates the corresponding partition number. The generation of the partition number is similar to the process in *Partition_base*. The difference is that *Partition_windowPair* distinguishes between the two window-pair partitions of any pair of pivots. The correct identification of the specific window-pair a record belongs to is obtained using the information of the previous partition of the record (lines 6 to 10).

In the scenario of Fig. 7, *Partition_windowPair* will generate the same partition number, i.e., $(Q_0 \times C4)$ mod *NUMPARTITIONS*, for all the intermediate keys that correspond to partition $Q0$. Similarly, the function will generate the same partition number, i.e, $(Q_0 \times C7 + Q_1 \times C8)$ mod *NUMPARTITIONS*, for all the intermediate keys that correspond to partition $Q0\_Q1\{1\}$. The partition number of the records in $Q0\_Q1\{1\}$ is generated in line 9 while the partition number of the records in $Q0\_Q1\{2\}$ is generated in line 7. We use the numbers 1 and 2 at the end of the window-pair partitions' names to differentiate between them. We reference this number as the window-pair *sequence*.

After generating the partition numbers of intermediate records, the records are send to their corresponding reduce nodes. In a window-pair partition round, the records received at each reduce node are sorted and grouped using the *Compare_windowPair* function presented in Algorithm 8. This function groups all the records that belong to the same partition establishing the order of partitions shown in Fig. 8.a.

Compare_windowPair receives as input two intermediate record keys, i.e., $k2_1, k2_2$, and returns 0, -1 or $+1$ depending on the order of the associated partitions. 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 case, *Compare_windowPair* orders them based on the tuple (minimum pivot index,

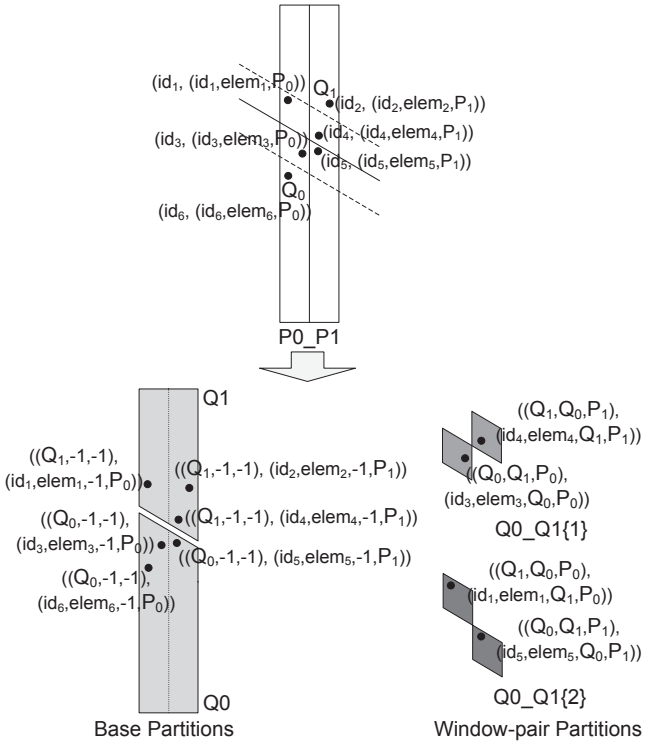

Figure 7: Example of the Map function for a window-pair round.

Algorithm 9 *Reduce_windowPair*()

Input: $(k2, v2List), W_1, W_2$. $k2 = (kPart,\ kWin,\ kPrevPart)$, $v2List =$ list(id, $elem$, $part$, $prevPart$), W_1 and W_2 are the last two components of the job input directory name job_inDir

Output: SJ matches or intermediate data. Intermediate data = list$(k3, v3)$. $k3 = id$, $v3 = (id, elem, part)$

1: **if** all the elements of $v2List$ have the same value of $prevPart$ **then**
2: return
3: **end if**
4: **if** $sizeInBytes(v2List) \leq memT$ **then**
5: $InMemorySimJoinWin(v2List, outDir, eps)$
6: **else**
7: **for** each element e of $v2List$ **do**
8: output $(e.id, (e.id, e.elem, e.prevPart))$ to $outDir$/intermediate/W_⟨$roundNum$⟩_⟨$kPart$⟩_ ⟨W_1⟩_⟨W_2⟩
9: **end for**
10: **end if**

maximum pivot index, window-pair sequence) using lexicographical order (lines 11 to 55). Several sub-cases are considered. When the keys do not belong to the windows between the same pair of pivots, they are ordered based on (minimum pivot index, maximum pivot index) (lines 14 to 20). Otherwise, the algorithm considers the following cases: (1) when the keys belong to the same partition and same old partition, they have the same order (lines 21 to 23), (2) when the keys belong to the same partition but different old partition, they are ordered by their window-pair sequences (lines 24 to 38), (3) when the keys belong to different partitions but the same old partition, they are also ordered by their window-pair sequences (lines 39 to 53), and (4) when the keys belong to different partitions and different old partitions, they have the same order (line 54).

Fig. 8.b shows the order of partitions generated by *Compare_windowPair* for the scenario with two pivots presented in Fig. 7. Fig. 8.c shows the order of partitions for the case of a window-pair round with three pivots.

After generating the groups in a reduce node, the MapReduce framework calls the *reduce* function *Reduce_windowPair* once for each group. This function is presented in Algorithm 9 and receives as input a key-value pair $(k2, v2List)$. The goal of this function is to generate the window links of the partitions that are small enough to be processed in a single node and to store the data of larger partitions for further repartitioning.

If all the records in $v2List$ belong to the same old partitioning, there is no possibility of generating window links and thus the function terminates immediately (lines 1 to 3). If the size of the list is small enough to be processed in a single node, the algorithm calls function *InMemorySimJoinWin* to get the window links in the current partition (lines 4 to 5). Otherwise, all the records of the group are stored in an intermediate directory for further partitioning

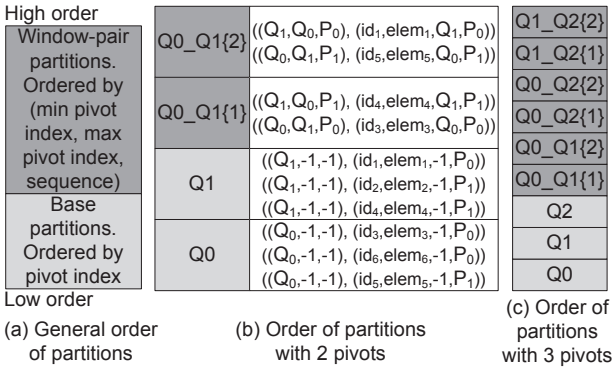

Figure 8: Group formation in a window-pair round.

(lines 6 to 10). Both the intermediate records and the directory name propagate the information of the previous partitions. This information will enable the correct generation of window links in next rounds. Note that, in the case of *Reduce_windowPair*, all partitions that are stored for further processing are set to be repartitioned by a future window-pair partition round, i.e., directory name uses "*W_*". This is the case because the links generated in a window-pair round or in any of its generated partitions should always be window links.

In the scenario represented in Fig. 7, the MapReduce framework calls the *Reduce_windowPair* function for each partition of Fig. 8.b: $Q0$, $Q1$, $Q0\_Q1\{1\}$ and $Q0\_Q1\{2\}$.

3.3 Enhancements for Euclidean Distance

Since the MRSimJoin solution presented in section 3.2 is based on the generalized hyperplane distance, it could be used with any dataset that lies in a metric space. The solution, however, could be enhanced in cases where the distance from a record to the hyperplane between two partitions can be computed exactly [16]. In the case of Euclidean spaces, the exact distance from a record t to the hyperplane that separates the partitions of two pivots P_0 and P_1 is given by:

$$hDist(t, P_0, P_1) = (dist(t, P_0)^2 - dist(t, P_1)^2)/(2 \times dist(p1, p2))$$

where $dist(a, b)$ is the Euclidean distance between a and b.

To use this exact distance, the generalized hyperplane distance should be replaced by $hDist$ in line 5 of *Map_base* and also in line 5 of *Map_windowPair*.

4 Implementation in Hadoop

The presented MRSimJoin algorithms are generic enough to be implemented in any MapReduce framework. This section presents a few additional guidelines for its implementation on the popular Hadoop MapReduce framework [1].

Distribution of atomic parameters. One of the tasks of the *MR_Job* function called in the main MRSimJoin routine is to make sure that the provided atomic parameters, i.e., $outDir$, $numPiv$, eps and $memT$, are available at every node that will be used in the MapReduce job. In Hadoop, this can be done using the job configuration $jobConf$ object and its methods *set* and *get*.

Distribution of pivots. *MR_Job* also sends the list of pivots to every node that will execute a *map* task. In Hadoop this can be done using the *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 can be done using the *rename* method of Hadoop's *FileSystem* class. The method will change the directory path in Hadoop's distributed file system without physically moving its data.

Single-node Similarity Join. *InMemorySimJoin* and *InMemorySimJoin-Win* represent single-node algorithms to get the links and window links in a given dataset, respectively. We have implemented these functions using the Quickjoin algorithm [16].

5 Performance Evaluation

We implemented MRSimJoin using the Hadoop 0.20.2 MapReduce framework. In this section we evaluate its performance with synthetic and real-world data.

5.1 Test Configuration

We performed the experiments using a Hadoop cluster running on the Amazon Elastic Compute Cloud (Amazon 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 [12]. 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 SFN are N times the number of links

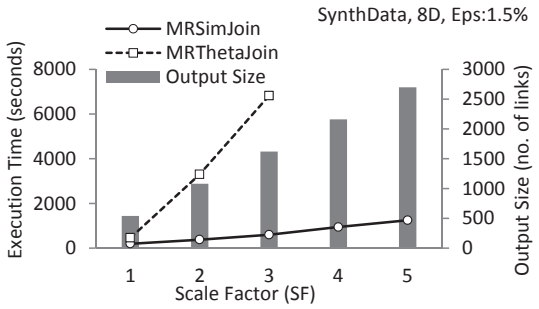

Figure 9: Increasing SF - SynthData.

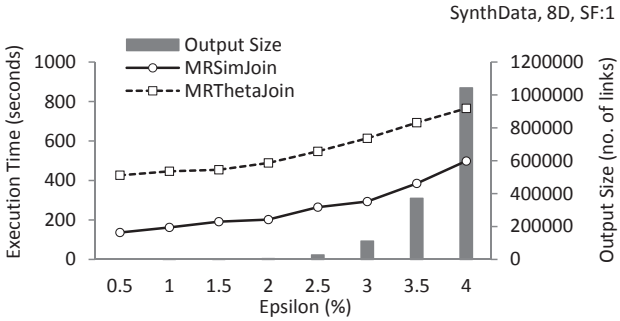

Figure 10: Increasing Epsilon - SynthData.

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 ε 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 [19] that uses the single-node QuickJoin algorithm [16] 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 Synthetic Data

Increasing Scale Factor. Fig. 9 compares the way MRSimJoin and MR-ThetaJoin scale when the data size increases (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

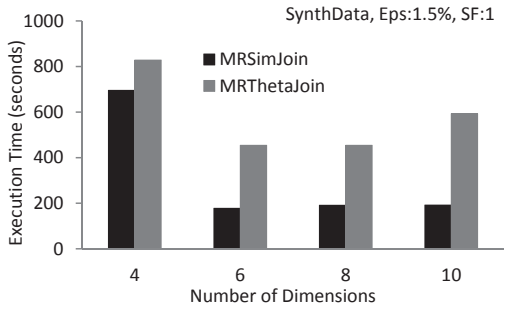

Figure 11: Increasing Dimensions - SynthData.

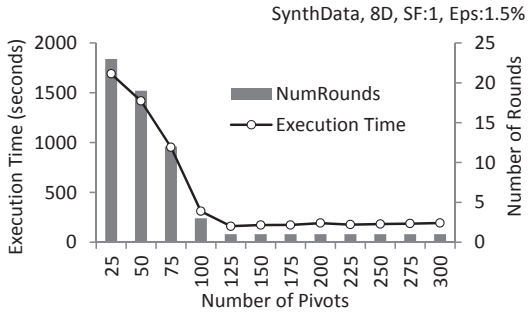

Figure 12: Increasing Num. of Pivots - SynthData.

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. 10 shows how the execution time of MRSimJoin and MRThetaJoin increase when epsilon increases (0.5%-4.0%). These tests use 8D vectors and SF1. 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. 11. This experiment uses epsilon of 1.5% and SF=1. 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.

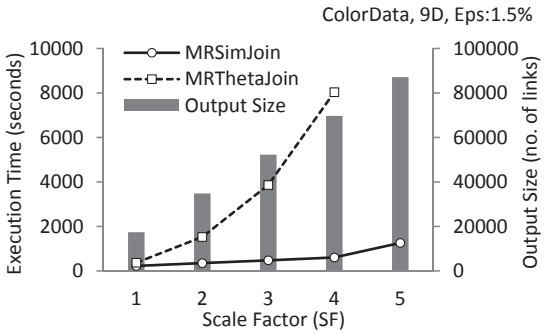

Figure 13: Increasing SF - ColorData.

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. 12. 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 relatively smaller values of epsilon, i.e., 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 [16]. 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 pivot 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 Color Data

Increasing Scale Factor. Fig. 13 compares the way MRSimJoin and MR-ThetaJoin scale when the data size increases (SF1-SF5). The results for ColorData are similar to the ones we found for the case of synthetic vector data. Specifically, 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. 14 shows how the execution times of MRSimJoin and MRThetaJoin increase when epsilon increases. The results of this test are also inline with the ones of the test with SynthData. The execution times of

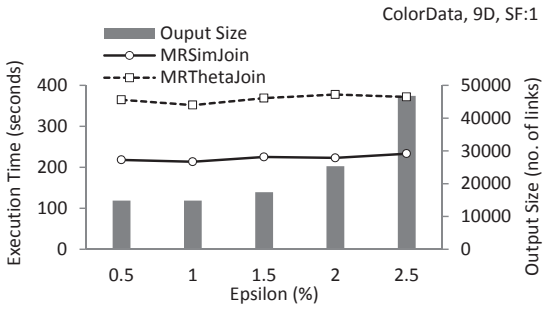

Figure 14: Increasing Epsilon - ColorData.

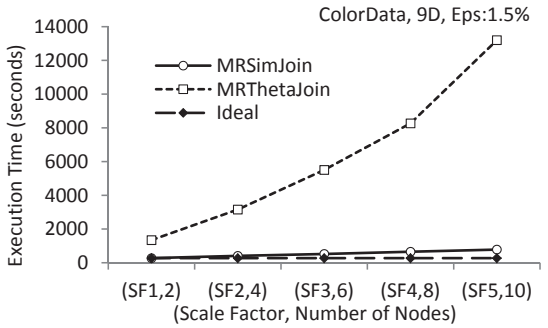

Figure 15: Increasing Number of Nodes and SF - ColorData.

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. 15 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 only 2.8 times the one for (SF1, 2) while MRThetaJoin's execution time for (SF5, 10) is 9.8 times the one for (SF1, 2). Moreover, the execution time of MRThetaJoin grows from being 4.9 times the one of MRSimJoin for (SF1, 2) to 16.9 times for (SF5, 10).

6 Conclusions and Future Work

Cloud-based systems have become a crucial component to efficiently process and analyze the large amounts of data currently available in many commercial and scientific organizations. The Similarity Join is recognized as one of the most

useful data analysis operations and has been used in many application scenarios. While multiple implementation techniques have been proposed for the Similarity Join, 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 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 like epsilon, data size, number of nodes and number of dimensions 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.

Our paths for future work include the study of: (1) other similarity-aware operators, e.g., kNN Join and kDistance Join, for cloud systems, (2) indexing techniques that can be exploited to implement Similarity Join operations, and (3) cloud queries with multiple similarity-based operators.

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] C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the kdd process. *Knowl. Inf. Syst.*, 6:728–749, November 2004.

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

[6] S. Chen. Cheetah: a high performance, custom data warehouse on top of mapreduce. *Proc. VLDB Endow.*, 3:1459–1468, September 2010.

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

[8] D. J. DeWitt, J. F. Naughton, and D. A. Schneider. An evaluation of non-equijoin algorithms. In *VLDB*, 1991.

[9] J.-P. Dittrich and B. Seeger. Gess: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *ACM SIGKDD*, 2001.

[10] V. Dohnal, C. Gennaro, P. Savino, and P. Zezula. Similarity join in metric spaces. In *25th European conference on IR research*, 2003.

[11] V. Dohnal, C. Gennaro, and P. Zezula. Similarity join in metric spaces using ed-index. In *Database and Expert Systems Applications*, volume 2736 of *Lecture Notes in Computer Science*, pages 484–493. 2003.

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

[13] 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.

[14] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. In *ACM SIGMOD*, 1998.

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

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

[17] 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.

[18] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: a new, robust, parallel hash join method for data skew in the super database computer (sdc). In *VLDB*, 1990.

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

[20] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. In *ACM SIGMOD*, 2009.

[21] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *ACM SIGMOD*, 1989.

[22] Y. N. Silva, A. M. Aly, W. G. Aref, and P.-A. Larson. Simdb: a similarity-aware database system. In *ACM SIGMOD*, 2010.

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

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

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

[26] 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.