Diversity Similarity Join for Big Data

Yasin N. Silva[1], Juan Martinez[1], Pedro Castro Cea[1], Humberto Razente[2], Maria C. Nardini Barioni[2]

[1] Loyola University Chicago, USA
[2] Universidade Federal de Uberlandia, Brazil
[1]{ysilva1, jmartinez29, pcastro}@luc.edu, [2]{humberto.razente, camila.barioni}@ufu.br

Abstract. The Similarity Join (SJ) has become one of the most popular and valuable data processing operators in analyzing large amounts of data. Various types of similarity join operators have been effectively used in multiple scenarios. However, these operators usually generate a large output size and many similar output pairs that represent almost the same information. In previous work, a new operator called Diversity Similarity Join (DSJ) has been proposed to address these issues. DSJ generates a smaller scale output and more meaningful and diverse result pairs. This operator, however, was proposed as a single node operator crucially limiting its scalability properties. In this paper, we propose the Distributed Diversity Similarity Join (D2SJ) operator, an approach that enables SJ diversification on big datasets. We present the design guidelines and implementation details on Apache Spark, a popular big data processing framework. Our experimental results with real-world high-dimensional data show that the proposed operator has excellent performance and scalability properties.

Keywords: Diversity Similarity Join, Big Data, Performance Evaluation, Spark, MapReduce.

1 Introduction

Today, big data has unprecedentedly spread to all kinds of industries. Big Data-driven decision-making has become very popular, and many applications produce and process massive amounts of data. While operators with exact-based semantics, such as the Natural Join and grouping/aggregation operators are widely used, many application scenarios, such as social-media platforms, biomedical information processing, and sensor data processing, can significantly benefit from similarity-aware operators (operators that identify and leverage similarities in the data). One of the most useful types of similarity operators is the distance range join (or sometimes referred to simply as the similarity join). This operator finds the pairs of records from two datasets that are separated at most by a distance threshold provided as a parameter ($\mathcal{E}$) [1]. Multiple similarity join implementation algorithms have been previously proposed. Some of them rely on distributed frameworks and can process massive datasets. However, the similarity join operator can generate a massive amount of result pairs. Moreover, many of these output pairs can be very similar to others adding little value to the analysis process, and the output size can grow quickly when the distance threshold grows. These characteristics generate the need to diversify the output of this operator.

The idea of output diversification has been studied primarily in the context of other data operators such as range and k-nearest-neighbor search. To the best of our knowledge, the only paper directly addressing the problem of diversifying the output of the similarity join was proposed by Santos et al. [2]. This previous work proposed the Diversity Similarity Join algorithm which integrates two phases. In the first phase, the standard output of the similarity join between two datasets *R* and *S* is computed. In the second phase, the records from *S* that are within the distance threshold from a given record in *R* are processed to identify a diversified subset. Crucially, however, this previous algorithm was proposed for a single-node scenario and cannot directly scale to multiple nodes to process big datasets. In this paper, we propose the Distributed Diversity Similarity join (D2SJ) operator, a fully distributed approach to diversify the similarity join output that can be used with big datasets and multiple data types and distance metrics. The main contributions of this paper are:

- We introduce the design elements of D2SJ. A distributed operator to diversify the output of the similarity join suitable to process big datasets.
- We present the implementation details for Apache Spark [3], one of the most popular big data frameworks.
- We comprehensively assess the performance and scalability properties of D2SJ and a baseline solution. We study the performance of the operators when key parameters are increased (data size, number of nodes, dimensionality, and similarity distance threshold).
- The source code of our implementation is publicly available [4].

The remainder of the paper is organized as follows: Section 2 describes the related work, Section 3 presents the general D2SJ algorithm, Section 4 describes the implementation details in Apache Spark, Section 5 reports the performance/scalability evaluation results, and Section 6 presents paths for future work.

2 Related Work

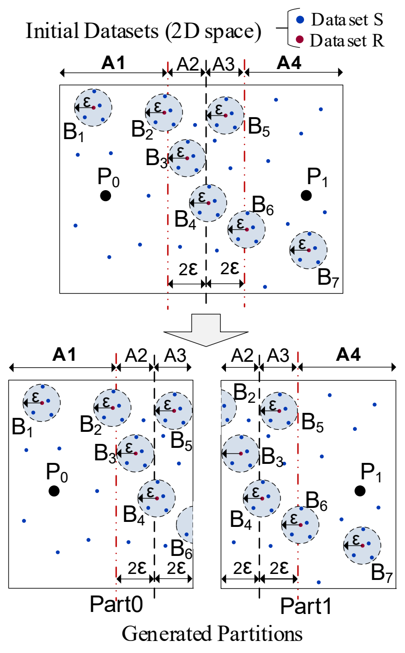
In the field of similarity-aware data processing, various types of similarity joins have been proposed. These include the distance range join, which identifies pairs with distances below a predefined threshold Ɛ [1, 5, 6, 7, 8], the k-Distance join that returns the k most-similar pairs [9], and the kNN-join which retrieves the k nearest neighbors in one dataset for each record in another dataset [10]. The distance range join has been extensively studied and is recognized as one of the most valuable similarity-aware operators. Because of this, it is usually referred to simply as similarity join.

In the realm of Big Data systems, Hadoop [11] and Spark [3] are two commonly used platforms. Hadoop, along with its programming framework MapReduce [12], facilitates two fundamental operations, namely *map* and *reduce*. The input data is divided into multiple map tasks that process the input data chunks in parallel. Each map call takes a pair (*k1*,*v1*) and produces a list of (*k2*,*v2*) pairs. The output of the map calls is subsequently transferred to reduce nodes, ensuring that all intermediate records with the same intermediate key (*k2*) are routed to the same reducer node (shuffle phase). At each reducer node, the intermediate records corresponding to a given key *k2*, are grouped and processed in a single reduce call. Spark, a more recent framework and considered a successor of Hadoop, uses Resilient Distributed Datasets (RDDs) as its fundamental data structure and supports a broader range of operations that include various types of map, reduce, grouping, filtering, and set operations. Spark operations are primarily executed in a distributed fashion utilizing the main-memory resources of a computer cluster [3].

Multiple techniques have been proposed to implement the similarity join operator on big data frameworks such as Hadoop and Spark. Several of them were experimentally compared in [13, 14]. Some of these techniques such as the Ball Hashing, Subsequence, Splitting, Hamming Code, and Anchor Points approaches [15] support string/text data and the Hamming and Edit distance functions. Other techniques such as the MRSetJoin [16] and the V-Smart-Online Aggregation [17] were proposed for set-based data and applicable distance functions such as the Jaccard and Dice Similarity. More versatile techniques such as the MRSimJoin [8, 18] and MRThetaJoin [19] can be used with a wide range of data types and distance functions.

Several approaches have also been studied to diversify the output of common data analysis operators. Most of them consider the case of the range and k-NN search operations. Drosou and Pitoura proposed DisC diversity [20], an approach to identify the representatives of a set of tuples considering coverage and dissimilarity. Both properties are defined using a distance threshold *r*. In this approach, each record *a* in the original set is represented by a record *d* in the diverse set, i.e., $dist(a, d) \leq r$). Also, the objects in the diverse set should be dissimilar to each other, i.e., for every pair of records d_1 and d_2 in the diverse set, $dist(d_1, d_2) > r$. Vieira et al. proposed two approaches to diversify k-NN search queries [21]. These approaches are based on the use of a ranking framework that includes a component measuring the level of relevance (with respect to the query) of the selected *k* records and another one measuring the diversity (distance) among these records. The framework allows the user to set a parameter to specify the relative importance of each component. More recently, Ge and Chrysanthis proposed PrefDiv [22], a technique that aims at identifying a set of dissimilar records based on user-provided distance functions and diversity thresholds on specific attributes. For instance, given two records *a* and *b*, two specified attributes A_1 and A_2, their corresponding distance functions, f_1 and f_2, and thresholds, t_1 and t_2, *a* and *b* are considered diverse if $f_1(a.A_1, b.A_1)>t_1$ and $f_2(a.A_2, b.A_2)>t_2$. This approach also aims to maximize relevance using a utility function that measures the benefit of selecting a certain record. While these diversification approaches can be used in top-k search and range search operators, the authors did not explore how these techniques can be applied to the case of similarity join operators which link elements of two sets.

To the best of our knowledge, the only previous work directly addressing the problem of diversification in the context of similarity join was proposed by Santos et al. [2]. In this approach, given two sets of records *R* and *S*, for every record $r \in R$, the algorithm identifies all records $s \in S$ that are within $\mathcal{E}$ from *r*. Then, all identified



D2SJ Partitioning and Output Generation

Strategy

- Partition the input into two partitions such that we can still identify all the *similarity balls* (B_1-B_7) (each ball has all the points in S within ε from a point in R)
- Each ball should be finally processed in only one partition producing the diverse pairs in the ball

Solution (using two pivots/partitions)

- Partition the input using two pivots (P_0 and P_1) such that each point belongs to the partition of its closest pivot
- Additionally, duplicate the points in the windows regions (A2, A3), generating: **Part0** = A1+A2+A3, **Part1** = A2+A3+A4
- Each ball is processed in a single partition, the one corresponding to its smallest closest-pivot (using index): Balls B_1, B_2, B_3, and B_4 are processed in Part0 while B_5,B_6, and B_7 in Part1
- Processing a ball (S-points around point r) identifies the subset of diverse pairs (r, s')

Fig. 1. Example of D2SJ partitioning and output generation using two pivots.

records *s* are processed one by one in order of distance from *r*. At this point, each record *s* is added to the diverse set of records in connection to *r*, denoted as *DivSet(r)*, if *s* does not belong to the area of influence (neighborhood) of any previously added record. This ensures that any added record *s* is sufficiently different than the previously added records. In our work, we use a similar notion of diversity but propose a fully distributed algorithm that is suitable to process big datasets.

3 Distributed Diversity Similarity Join (D2SJ) Algorithm

The distributed diversity similarity join (D2SJ) algorithm presented in this section addresses the problem of generating a diversified subset of the similarity join output. D2SJ adopts a similar notion of diversity as the work in [2] but uses a fully distributed and parallelized approach that enables it to process very large datasets. D2SJ can be used with any data type and metric-space distance function, and is deterministic (multiple executions with the same input data and Ɛ generate the same output).

Given two datasets that are joined (*R* and *S*), D2SJ identifies first the records in *S* that are located withing the distance threshold (Ɛ) from the records in *R*. For every record $r \in R$, the algorithm identifies the *similarity ball* around it (set of records $s \in S$ that are within Ɛ from *r*, i.e., *dist(s, r)* ≤ Ɛ). This process is performed in a distributed fashion but ensuring that each ball is eventually processed on a single node (to avoid duplicates in the output). In a second stage, the algorithm processes each ball (also in a parallel fashion) identifying the diverse set of records *s'* around each record $r \in R$.

For D2SJ to function across a cluster of computers to process vast amounts of data, the algorithm uses pivot-based partitioning to evenly distribute and parallelize the workload. Similar pivot-based partitioning was used in previous distributed algorithms, e.g., [7, 8, 23]. To make sure the algorithm identifies all the records in each similarity ball, in stage one, it duplicates some records from neighboring partitions.

Fig. 1 shows an example of how D2SJ partitions and identifies the diversified similarity join pairs using 2D data and two pivots (P_0 and P_1). The top-left image represents the input datasets (*R* and *S*). This image shows the similarity ball around each element in *R* (B_1 to B_7). The bottom-left image represents the two generated partitions (*Part0* and *Part1*). Observe that regions *A1* and *A2* contain the records that are closer to P_0 than to P_1, while *A3* and *A4* the ones that are closer to P_1 than to P_0. The regions *A1+A2* and *A3+A4* are referred to as *base regions*. Observe that the records in *A1+A2* and *A3+A4* are assigned to partitions *Part0* and *Part1*, respectively. Additionally, the regions at the boundary between the two base regions (points within 2Ɛ from the boundary) are replicated. In this example, region *A3* is added to *Part0* and *A2* to *Part1*. Regions *A2* and *A3* are referred to as *window regions*. The final content of *Part0* and *Part1* are *A1+A2+A3* and *A2+A3+A4*, respectively. Observe that all the similarity balls (B_1 to B_7) are fully contained in at least one of the partitions. The two partitions could now be sent to and processed by two different computers. The only problem is that some of the balls (B_2 to B_6) are partially or fully contained in both partitions. The approach needs a mechanism to process each similarity ball only once and ensure that a similarity ball is processed in the partition that contains the entire ball. To this end, D2SJ applies the following guidelines: (1) during partitioning, each record *x* in partition *P* is augmented with information of its closest pivot (*cPiv*) and assigned partition (*P*), and (2) given any generated similarity ball *B*, the ball will be processed only in the partition corresponding to the smallest *cPiv* among all the records in *B*. In Fig. 1, B_4 (which appears in *Part0* and *Part1*) contains some records that have P_0 as their closes pivot and others that have P_1 as their closest one. Since the smallest one (based on index) is P_0, B_4 is processed in the partition linked to this pivot (*Part0*). Observe that while D2SJ requires replicating the records in the window regions, most useful queries involve a small Ɛ with a small effect on performance.

Alg. 1 presents D2SJ's main algorithmic steps. Two sets of input data, *R* and *S*, are merged into one dataset (line 1), with pivots being selected from this combined set (line 2). After selecting the pivots, the algorithm partitions the data (lines 3-12), allowing for an even distribution of the data to be processed in each cluster node. Every input record *rec* is assigned to the partition of its closest pivot p_c (lines 5-6) and all the partitions of pivots *p* where *rec* belongs to the window regions between the partitions of *p* and p_c (lines 7-11). In general, the records in the window regions between two partitions (corresponding to pivots *p1* and *p2*) should be a superset of all the records within 2Ɛ from the hyperplane that separates the partitions. However, this hyperplane does not always explicitly exist in a metric space. Instead, it is implicit and known as the generalized hyperplane. Since the distance of a record *rec* to the generalized hyperplane between two partitions for pivots *p1* and *p2* cannot always be computed exactly, a lower bound is used following [25] (line 8): *genHyperplaneDist*(*rec*, *p1*, *p2*) = (*dist*(*rec*, *p1*) - *dist*(*rec*, *p2*)) / 2. This expression can be replaced by the exact

Algorithm 1: *DistDivSimJoin_Main*

Input: *input_R* (input dataset *R*), *input_S* (input dataset *S*), *eps* (radius), *part_num* (number of partitions), *mem_T* (memory threshold)

Output: diversified set of similaritry join pairs

```
1   $input = input\_R \cup input\_S$
2   $pivots$ = selectPivots($part\_num$, $input$)
3   //Partitioning - // $rec$: ⟨ID, dataset, value, assignedPartitionSeq, closestPivotSeq⟩
4   for each record $rec$ in $input$ do
5       $p_c$ = getClosestPivot($rec$, $pivots$)
6       output ⟨$p_c$, $rec$⟩ //intermediate output – base region
7       for each pivot $p$ in $pivots$ do
8           if ((distance($rec$, $p$) - distance($rec$, $p_c$)) / 2 $\leq 2eps$) then
9               output ⟨$p$, $rec$⟩ //intermediate output - window region
10          end if
11      end for
12  end for
13  //Shuffle: all the records sharing the same key will form a partition
14  //Similarity ball generation and diversification
15  for each partition $P_i$ do //each partition may be processeed in a different node
16      if ($P_i$.memSize() > $mem\_T$) do
17          store $P_i$ for processing in subsequent round
18      else
19          $B_i$ = IdentifySimBalls($P_i$, $eps$) //$B_i$ (ball set) format: {$B_{i\_k}$}
20                      //$B_{i\_k}$: ⟨centerPoint, records, flags⟩, flags contains partitioning data
21          //Output Generation (preventing duplication)
22          for each similarity ball $B_{i\_k}$ in $B_i$ do
23              generate $minFlags$ for $Ball_{i\_k}$ //$minFlags[q]$ = {index of first element in
24                                                  //$Ball_{i\_k}.flags[q]$ equal to 1}
25              $aPartitioningSeq = s$.assignedPartitionSeq() //$s$ is any record in $B_{i\_k}$
26              if ($\forall q$, $minFlags[q] = aPartitioningSeq[q]$) then //if we are in the
27                                          //selected partition to process this similarity ball
28                  $divB_{i\_k}$ = Diversify($B_{i\_k}$) //diversify this similarity ball
29                  output $divB_{i\_k}$ //final output
30              end if
31          end for
32      end if
33  end for
```

Alg. 1. Main D2SJ algorithm.

distance when this can be computed, e.g., for the Euclidean distance, *genHyperplaneDist* can be replaced by $euclideanHyperplaneDist(rec, p1, p2) = |(dist(rec, p1)^2 - dist(\text{rec}, p2)^2| / (2 \times dist(p1, p2)$. The partitioning phase also records the information of the closest pivot and assigned partition of each record (sequence of closest pivots and partitions if the execution requires multiple rounds). This information is used later in the process. The partitioning phase of D2SJ can be implemented using the map operations in Spark or Hadoop.

The intermediate records generated in the partitioning phase are grouped in the shuffle phase (line 13) such that all the records that belong to the same partition will form a single group. This task is implemented using the grouping operator in Spark and would be automatically performed in the shuffle phase of a Hadoop job. In the next phase, partitions are processed (1) identifying the similarity balls contained in each partition, (2) determining if a similarity ball should be processed on a given cluster node, and (3) diversifying and outputting the selected similarity ball (lines 14-33). Different partitions could be processed on different nodes. For a given partition, the algorithm first checks if the partition is small enough to be efficiently processed in a single node (line 16). If this is not the case, the partition is stored for further processing using the same D2SJ algorithm but applied to this single partition (line 17). This feature makes D2SJ a multi-round algorithm where at every round the small partitions are directly processed, and the large partitions are stored for processing in subsequent rounds. It is important to observe, however, that while D2SJ can be executed in multiple rounds, the best execution times in our experimental tests were obtained by increasing the number of pivots to generate a single round (with smaller partitions). When the partition is small enough to be processed in the current round, the algorithm identifies first the similarity balls contained in this partition (line 19). The details of this process are described later (Algorithm 2: *IdentifySimBalls*). Each similarity ball contains the records s within *eps* (Ɛ) from a given record r used as a center point. The output of *IdentifySimBalls* is a set of similarity balls where each ball is composed of a center point (from R), the data records (from S), and information needed to ensure non-duplicated ball processing (*flags*). The *flags* component of a given ball B contains a sequence of flag arrays (one array per round that processed data that included this ball). This component is used to determine if the ball should be processed in the node processing the current partition or not (lines 23-30). For example, if four pivots are being used (p_0, p_1, p_2, p_3) and a single round is needed, *B.flags* has the form $\{[f_0, f_1, f_2, f_3]\}$ and the content could be {[0, 0, 1, 1]}. A value of 1 at index i indicates that ball B contains at least one record whose closest pivot is p_i. In this example, B contains records with base region equal to p_2 and others with base region equal to p_3. A given ball of partition P_i (corresponding to pivot p_i) will be processed in the current node only if the minimum index of 1 in the flag array of this ball matches i. In the example, ball B will be processed only when this ball is detected in the partition of p_2 (because the smallest index with a value of 1 is 2). If a ball should be processed in the current node, the algorithm applies the diversification method (Algorithm 3: *Diversify*). This method processes a similarity ball and generates the subset of diversified similarity join pairs, where each pair is composed of the center point and one of the (S) records in the ball. The set of diverse SJ pairs is then added to the final output of D2SJ (line 29). The similarity ball generation and diversification phase can be performed using the reduce operations in Spark or Hadoop.

The details of *IdentifySimBalls* are presented in Alg. 2. This algorithm identifies the similarity balls in an input partition. Each similarity ball is composed of a center point r (a record in R), a set of records (records in S within Ɛ from r) and a set of flags which contain partitioning information. The algorithm separates first the records from R and S (lines 4-11). In our implementation, we use an initial algorithm to identify the

Algorithm 2: ***IdentifySimBalls***

Input: P_i (a partition), *eps* (radius)

Output: B_i (similarity balls in partition P_i, ball structure: ⟨*centerPoint*, *records*, *flags*⟩)

```
1  inputR = {}
2  inputS = {}
3  B_i = {}
4  for each record rec in P_i do
5      if (rec.dataset = 0) then //dataset
6                         //values: 0 (R), 1(S)
7          inputR.add(rec)
8      else
9          inputS.add(rec)
10     end if
11 end for
12 //Generation of similarity join balls
13 for each record r in inputR do //creates a
14               //ball around each records in R
15     b = {r, [], []}//r is the center point
16     for each record s in inputS do //find
17                //the records in S similar to r
18         if distance(r, s) ≤ eps then
19             b.records.add(s)
20         end if
21     end for
22     generateFlags(b) //updates b.flags
23     B_i.add(b) //adds the ball to the set
24 end for
25 return B_i
```

Alg. 2. Identification of Similarity Balls.

Algorithm 3: ***Diversify***

Input: b (a similarity ball)

Output: b' (diversified similarity ball)

```
1  c = b.centerPoint
2  f = b.flags
3  sort(b.records, c) //sort the records in the
4      //ball in increasing distance from the
5      //center point c
6  b' = {c, [], f} //initializing the
7                 //diversified ball
8  for every record s in b.records do
9      isDiverse = True
10     for each record d in b'.records do //this
11       //will be empty initially, but will
12       //get filled as diverse elements
13       //are discovered
14       if inInfluenceArea(s, d, c) then //if
15       //s is too similar to the diversified
16       //record d
17           isDiverse = False
18           break
19       end if
20     end for
21     if (isDiverse = True) then
22       b'.records.add(s)
23     end if
24 end for
25 return b'
```

Alg. 3. Diversification of Similarity Balls.

similarity balls as this will be executed on a relatively small set of records and on a single node. Any other single-node similarity join algorithm could be integrated to identify the balls. In our case, a data structure for a ball is initialized in line 15. For each record r (from R) in the partition, the algorithm checks if the available records from S are within Ɛ from r. All the qualifying records are added the ball of r (lines 16-21). The algorithm, then, generates the *flags* component of the ball using the closet-pivot information of the records in the ball (line 22). After this, the generated ball is added to the set of balls identified in the current partition (line 23).

Alg. 3 presents the details of the *Diversify* subroutine. The goal of this algorithm is to diversify the records of a similarity ball using a similar notion of diversity as in [2]. This algorithm receives a similarity ball (b) and returns a similarity ball (b') that contains a diverse subset of the records. The algorithm sorts first the records in the input ball as they will need to be processed in increasing distance from the center point (line 3). The structure for the diversified ball is initialized in line 6. The set of diverse records (*b'.records*) is initially empty. Then, the algorithm processes each record s from the input ball (lines 8-24). In each iteration, the algorithm verifies that s is diverse from every other record already in the diverse set (lines 10-20). If this is the case, s is

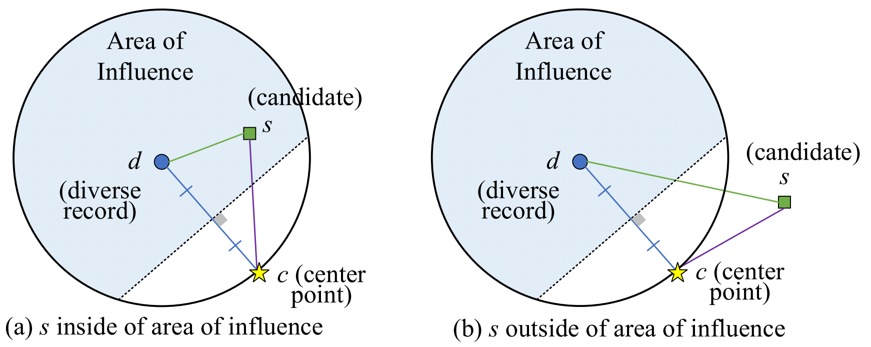


Fig. 2. Examples of different outcomes of *inInfluenceArea* with 2D data.

added to the diverse set *b'.records* (lines 21-23). Observe that if *s* fails the diversity test with an already added diverse record *d* in line 14, *s* is considered not diverse enough and the algorithm stops the process of checking with additional diverse records (lines 17-18). At the end, the method returns the diversity similarity ball *b'*.

A key aspect of this algorithm is checking if a record *s* (that belongs to a ball centered in *c*) is contained in the area of influence of an already added diverse record *d*. This check is performed by *inInfluenceArea*(*s,d,c*). Building on the work in [2], *inInfluenceArea*(*s,d,c*) returns True if $I(d,s) \geq I(d,c)$ and $I(d,s) \geq I(s,c)$, where I is the inverse of the distance function. Thus, *inInfluenceArea*(*s, d, c*) returns True if:

$$\left(\frac{1}{dist(d,s)} \geq \frac{1}{dist(d,c)}\right) \land \left(\frac{1}{dist(d,s)} \geq \frac{1}{dist(s,c)}\right),$$

or, equivalently, if $(dist(d,c) \geq dist(d,s) \land dist(s,c) \geq dist(d,s))$.

The intuition is that this check will return true if *s* (a record of a ball centered in *c*) belongs to the neighborhood of *d*. Figures 2.a and 2.b show examples of the two different outcomes for the case of 2D data and the Euclidean distance. Observe that in this case, the first condition $(dist(d,c) \geq dist(d,s))$ checks if *s* is contained in the circle centered in *d* with radius $\overline{dc}$, and the second one $(dist(s,c) \geq dist(d,s))$ checks if *s* is closer to *d* than to *c*. The shaded area in both images is the area that would be considered the area of influence of record *d*. In Fig. 2.a, *s* belongs to this area and will be considered not diverse enough. In Fig. 2.b, *s* does not belong to the area of influence

4 Implementation

Section 3 presented the algorithmic steps of D2SJ. This algorithm could be implemented on any MapReduce-based framework, e.g., Hadoop and Spark. Section 3 also indicated the main Spark and Hadoop operations for the core phases of the algorithm. As Spark is broadly considered a more efficient successor of Hadoop, we implemented the algorithm in Spark. The source code is available in [4].

In this section, we provide some additional implementation details. The implementation in Spark uses the RDD API. Spark's robust array of data processing operations enables a compact implementation. The *takeSample* operation is used to randomly se-

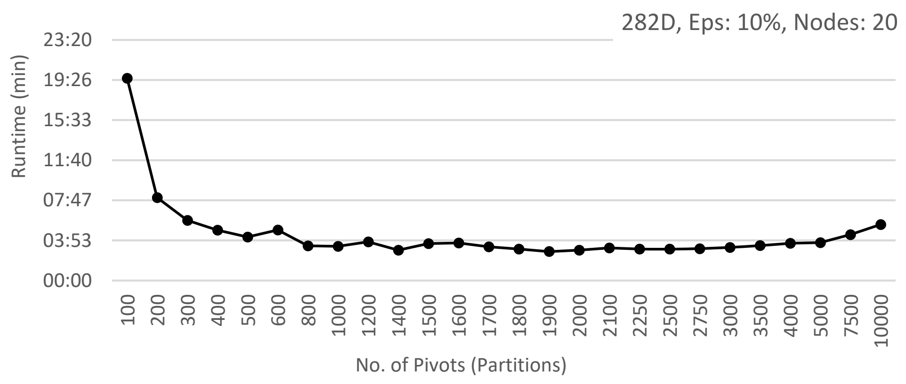


Fig. 3. Effect of the number of pivots (partitions) on execution time.

lect the pivots. Then, the *flatMapToPair* operation is used to implement the partitioning phase and the *partitionBy* operation to group the records that belong to the same partition (Shuffle phase). After this, *mapPartitionsToPair* is used in the implementation of *IdentifySimBalls*, which identified the similarity balls in a partition. Finally, *map* and *saveAsTextFile* are used to generate the final output.

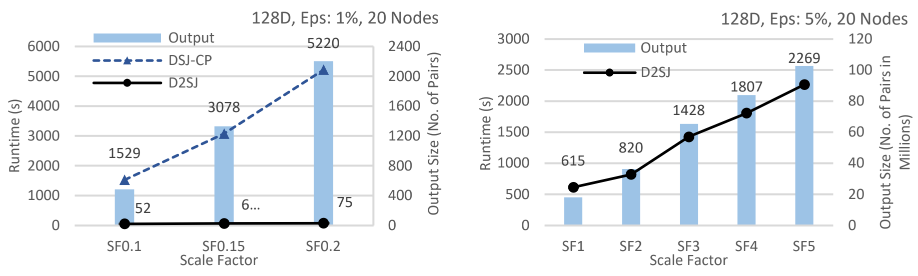
5 Performance Evaluation

5.1 Test Configuration

In this section we evaluate the performance and scalability properties of D2SJ. We also compare D2SJ with DSJ-CP, a direct Spark extension of the single-node algorithm presented in [2] (which uses a cartesian product to perform the similarity join). Both algorithms were implemented in Spark 3.0. Unless otherwise stated, all tests were executed using a cluster composed of 1 master and 20 worker nodes on the Google Cloud Platform. Each node used the Cloud Dataproc 2.0 image and had 4 virtual CPUs, 15GB of memory, and 500GB of disk space. The number of splits per Spark job was set to 2 × (# of worker nodes) × (# of vCPUs).

We used real data to perform our experiments. Specifically, we used the CoPhIR data collection [24], which is composed of visual descriptors extracted from 100 million images from Flickr. We used the following collections: Color Structure (CS, 64D), Scalable Color (SC, 64D), Edge Histogram (EH, 80D), Color Layout (CL, 12D), and Homogeneous Texture (HT, 62D). The datasets for different dimensionalities were generated as follows: 16D, 32D, and 64D: first 16, 32, and 64 attributes of CS, 128D: CS+SC, 208D: CS+SC+EH, and 282D: CS+SC+EH+CL+HT. The dataset for scale factor N (SFN) had 1,000,000 × N records. These records were equally divided to form the R and S datasets. The value of ε is expressed as the percentage of the maximum potential distance between two records.

Next, we compare how the various parameters affect D2SJ and DSJ-CP (except for varying the pivot count which is only applicable to D2SJ). Since DSJ-CP does not



(a) D2SJ vs DSJ-CP (b) D2SJ with larger datasets

Fig. 4. Execution time when increasing dataset size.

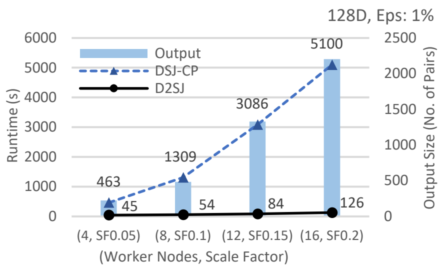
scale as well as D2SJ, we provide two graphs in each case. In the first, we scale down the experimental settings to maintain execution times under 10h and avoid stack overflow errors in DSJ-CP's cartesian product. In the second, we evaluate D2SJ under larger workloads. The settings of each test appear in the top-right label of the figure.

5.2 Performance Evaluation Results

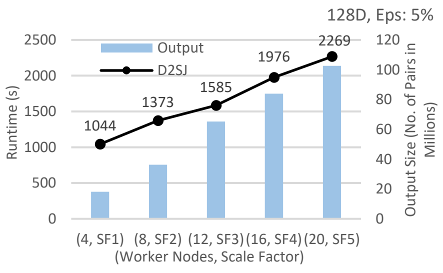
Optimal Pivot Count. Fig. 3 shows how D2SJ's execution time changes when the number of pivots increases (since a partition is generated for each pivot, this is equal to the number of partitions). This test uses the 282D SF5 dataset, a cluster with 20 worked nodes, and a large distance threshold (Ɛ = 10%). We observe that the execution time quickly decreases when the number of pivots increases initially. In general, larger numbers of pivots generate smaller execution times. However, exceeding 4,000 pivots leads to excessive replication in the window regions and increased execution times. The optimal pivot count is between 800 and 3500. These pivot counts solve the job in a single round. We use *numPivots*=400×SF in the remaining tests.

Increasing Scale Factor. Fig. 4.a shows how the execution times of D2SJ and DSJ-CP (lines) and the output size (bars) vary when the scale factor (data size) increases. These tests used scaled-down parameters to enable the comparison (Ɛ=1%, SF:[0.1-0.2]). We can observe that the execution times of D2SJ increase slowly as the scale factor increases. DSJ-CP's execution times, on the other hand, are significantly larger than those of D2SJ and grow rapidly. In fact, the execution time of DSJ-CP grows from being 29 times the execution time of D2SJ for SF 0.1 to 70 times for SF 0.2. Fig. 4.b shows D2SJ's execution times with heavier settings (Ɛ=5% and SF:[1-5]). D2SJ's execution time grows gracefully following a semi-linear pattern. In this case, none of the DSJ-CP jobs were able to finish in under 10 hours.

Increasing Scale Factor and Number of Cluster Nodes. A desired property in distributed algorithms is to have good scalability when the data size and number of nodes increase proportionally. While some overhead is expected with larger loads, a reduced overhead is desired. Fig. 5.a shows the execution times of D2SJ and DSJ-CP as the data size and number of worker nodes increase from (4 nodes, SF 0.05) to (16 nodes, SF 0.2). The results with these scale-down settings show that D2SJ scales

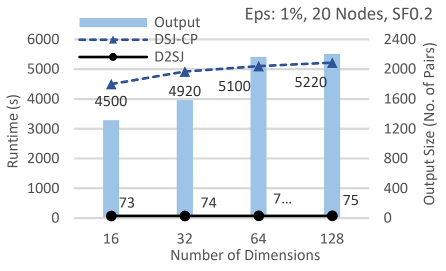


(a) D2SJ vs DSJ-CP

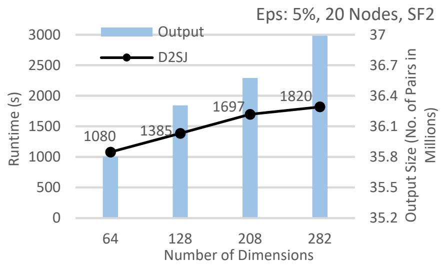


(b) D2SJ with larger clusters and datasets

Fig. 5. Execution time when increasing dataset size and number of worker nodes.



(a) D2SJ vs DSJ-CP



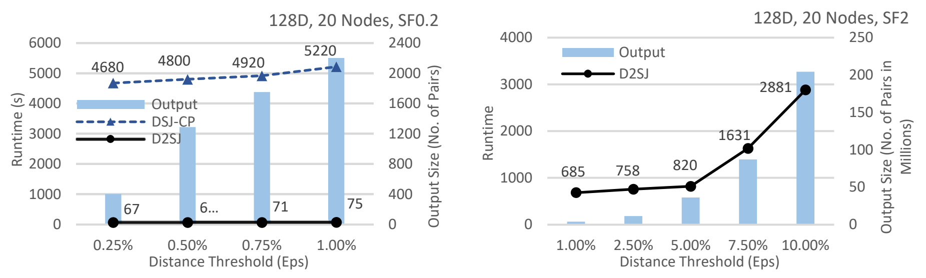
(b) D2SJ with higher dimensionality

Fig. 6. Execution time when increasing the number of dimensions.

significantly better than DSJ-CP. The execution time of DSJ-CP with the largest SF is 11 times the one with the smallest SF. In the case of D2SJ, the increase is only 2.8 times. Fig. 5.b presents D2SJ's execution time with larger workloads increasing from (4 nodes, SF 1) to (20 nodes, SF 5). We can observe that D2SJ scales well producing an execution time that is linear with a relatively small slope. D2SJ's execution time with SF5 (and 20 nodes) is only 2.2 times its execution time with SF1 (and 4 nodes).

Increasing Number of Dimensions. To evaluate the effect of data dimensionality on execution time, we executed each algorithm with datasets of varying dimensionality while fixing the scale factor. Fig. 6.a shows the execution time of both algorithms using SF 0.2 and 16D-128D datasets. In general, the execution time of both algorithms increases when dimensionality increases. D2SJ, however, has better scalability. While DSJ-CP's execution time with 128D represents an increase of 16% with respect to the 16D dataset, the increase is only of 2% for D2SJ. Fig. 6.b shows the execution time of D2SJ with larger workloads (SF2 and 64D-282D). This figure shows that the execution time of D2SJ increases sublinearly in this dimensionality range.

Increasing Distance Threshold (Ɛ). The use of diversification was motivated in part due to the large number of output records generated by traditional similarity join operations (with many output records being very similar to others). The work in [2] showed that the output of the diversified similarity join returns a very small fraction of the output of the standard similarity join. However, increasing the distance thresh-



(a) D2SJ vs DSJ-CP (b) D2SJ with larger distance thresholds

Fig. 7. Execution time when increasing the similarity join distance threshold.

old (Ɛ), still has a significant effect on the overall output size and execution time. In this experiment, we evaluate the execution time of both algorithms when Ɛ increases. Fig. 7.a shows the execution time of D2SJ and DSJ-CP when Ɛ increases from 0.25% to 1%. We can observe that the execution time of D2SJ is significantly better than that of DSJ-CP. D2SJ's execution time grows from 67s (Ɛ=0.25%) to 75s (Ɛ=1%), while the growth for DSJ-CP is from 4680s to 5220s. Fig 7.b shows D2SJ's execution time with larger workloads (Ɛ:[1%-10%]). In this case, we observe that D2SJ's execution time for Ɛ=10% is 4.2 times the one for Ɛ=1% while the output size for Ɛ=10% is 51.7 times the one for Ɛ=1%.

6 Conclusion and Future Work

Many organizations are collecting vast amounts of data that often include very similar data items. When data operators such as the Similarity Join, are executed on these datasets, the results include many similar output pairs that do not add much value to the understanding of data patterns. To address this problem in the case of similarity joins, previous work explored the integration of a diversification step. This previous work, however, was proposed for small data on a single computer. In this paper, we present D2SJ, a distributed approach to solve the diversity similarity join problem with big data. D2SJ can be used with multiple data types and distance functions. We present a detailed description of D2SJ as well as implementation details in Apache Spark. Moreover, we also present experimental results with real datasets that show strong performance and scalability properties. Future areas of research building on the results of this work could include (1) the comparative study of additional ways to diversify the output of different types of similarity join for big data and (2) the development of efficient distributed algorithms supporting these new notions of diversity.

Acknowledgments

This project was supported by an award from the Google Cloud Research program. The authors would like to thank Steven Hu, Timothy Raymer, and Steven Anderson for their contributions in the preliminary stages of this project.

References

1. 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 SIGMOD, 2001.
2. L. F. D. Santos, L. O. Carvalho, W. D. Oliveira, A. J. M. Traina, and C. Traina Jr. Diversity in Similarity Joins. In SISAP, 2015.
3. Apache. Spark. https://spark.apache.org/.
4. SimCloud Research Team. D2SJ Source Code. https://ysilva.cs.luc.edu/SimCloud/downloads.html.
5. V. Dohnal, C. Gennaro, and P. Zezula. Similarity Join in Metric Spaces Using ED-Index. In DEXA, 2003.
6. V. Dohnal, C. Gennaro, P. Savino, and P. Zezula, Similarity Join in Metric Spaces. In ECIR, 2003.
7. E. H. Jacox and H. Samet. Metric Space Similarity Joins. TODS, 33: 7:1–7:38, 2008.
8. Y. N. Silva, J. M. Reed, and L. M. Tsosie. MapReduce-based Similarity Join for Metric Spaces. In VLDB/Cloud-I, 2012.
9. G. R. Hjaltason and H. Samet. Incremental Distance Join Algorithms for Spatial Databases. In SIGMOD, 1998.
10. C. Böhm and F. Krebs. The k-Nearest Neighbour Join: Turbo Charging the KDD Process. KAIS, 6:728–749, November 2004.
11. Apache. Hadoop. https://hadoop.apache.org/.
12. J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In OSDI, 2004.
13. Y. N. Silva, J. M. Reed, A. Wadsworth, K. Brown, and C. Rong, An experimental survey of MapReduce-based similarity joins. In SISAP, 2016.
14. F. Fier, N. Augsten, P. Bouros, U. Leser, J-C. Freytag. Set Similarity Joins on MapReduce: An Experimental Survey. In VLDB, 2018.
15. F. N. Afrati, A. D. Sarma, D. Menestrina, A. Parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. In ICDE, 2012.
16. R. Vernica, M. J. Carey, and C. Li. Efficient Parallel Set-similarity Joins using MapReduce. In SIGMOD, 2010.
17. A. Metwally and C. Faloutsos. V-SMART-join: a Scalable MapReduce Framework for All-pair Similarity Joins of Multisets and Vectors. In VLDB, 2012.
18. Y. N. Silva and J. M. Reed. Exploiting MapReduce-based Similarity Joins. In SIGMOD, 2012.
19. A. Okcan and M. Riedewald. Processing Theta-joins using Mapreduce. In SIGMOD, 2011.
20. M. Drosou and E. Pitoura. DisC Diversity: Result Diversification based on Dissimilarity and Coverage. In CIKM, 2010.
21. M. R. Vieira, H. L. Razente, M. C. N. Barioni, M. Hadjieleftheriou, D. Srivastava, C. Traina Jr., and V. J. Tsotras. On query result diversification. Information Systems, 42: 57-77, 2014.
22. X. Ge and P. K. Chrysanthis. PrefDiv: Efficient Algorithms for Effective Top-k Result Diversification. In EDBT, 2020.
23. Y. N. Silva, M. Sandoval, D. Prado, X. Wallace, and C. Rong. Similarity Grouping in Big Data Systems. In SISAP, 2019.
24. P. Bolettieri, A. Esuli, F. Falchi, C. Lucchese, R. Perego, T. Piccioli, and F. Rabitti. CoPhIR: a Test Collection for Content-Based Image Retrieval. arXiv:0905.4627, 2009.
25. G. R. Hjaltason and H. Samet. Index-driven similarity search in metric spaces (survey article). TODS, 28: 517–580, 2003.