Similarity Grouping in Big Data Systems

Yasin N. Silva[1], Manuel Sandoval[1], Diana Prado[1], Xavier Wallace[1], Chuitian Rong[2]

[1]Arizona State University, USA
[2]Tianjin Polytechnic University, China
{ysilva,mosandov,daprado1,xgwallac}@asu.edu, chuitian@tjpu.edu.cn

Abstract. Distributed computing technologies have opened the door for a wide range of organizations to analyze massive amounts of data. Grouping (fast but based on exact semantics) and clustering (relatively slow but based on similarity-aware semantics) are among the most useful data analysis operations. Previous work introduced the Similarity Grouping (SG) operator, which aims to integrate the best features of grouping and clustering, i.e., fast execution times and similarity-aware grouping semantics. The SG operators, however, were proposed for single node relational database systems. This paper introduces the Distributed Similarity Grouping (DSG) operator, a highly parallel operator for identifying similarity groups in big datasets. DSG enables the identification of groups where all the elements are within a given threshold from each other. This paper presents DSG's design details, implementation guidelines on Spark and Hadoop (two important Big Data systems), and extensive performance and scalability evaluation.

Keywords: Similarity Grouping, Big Data Systems, Performance Evaluation, MapReduce, Spark, Hadoop, Clustering.

1 Introduction and Related Work

In many business and scientific scenarios, organizations accumulate large amounts of data. While organizations could gain many insights from integrating and analyzing these big datasets, in many cases their dimensions prevent their efficient processing on single-node systems. Big data systems such as Hadoop [1], Spark [2] and Bigtable [4] represent an answer to the requirement of highly distributed and parallelized data analysis of big datasets. Many of these systems are now supported on cloud-based platforms. Hadoop and Spark are two popular Big Data systems. Hadoop [1] and its programming framework, MapReduce [3], support two key operations: *map* and *reduce*. Multiple map tasks process input chunks in parallel. Each map call is given a pair (*k1*,*v1*) and produces a list of (*k2*,*v2*) pairs. The output of the map calls is transferred to the reduce nodes (*shuffle* phase) in a way that guarantees that all the intermediate records with the same intermediate key (*k2*) are sent to the same reducer node. At each reduce node, the received intermediate records are sorted and grouped. Then, each formed group is processed in a single reduce call. Spark [2] is a more recent Big Data system that uses Resilient Distributed Datasets (RDDs) as its core data structure and supports a wider range of operations (including variants of *map* and *reduce* as well as

grouping, filtering and set operations). Spark operations are performed in a distributed fashion primarily using the main-memory resources of a computer cluster.

Grouping operations are among the most useful data analytics operations. The standard grouping operator (Group-by) [5] is extensively supported in relational databases and was extended to perform subset aggregation (where grouping is performed at various levels) via the Roll-Up and Cube database operators [6]. While Group-by is very fast, its use is limited to equality-based grouping (all the records with the same grouping attribute values form the same group). Multiple sophisticated clustering operators have been proposed in data mining [11], e.g., K-Means [7] and DBSCAN [8]. They seek to find more complex patterns in data, but often at a steep increase in execution time. Single-scan versions of the well-known clustering algorithms K-Means and Cobweb are proposed in [12] and [13]. CURE [14] is an alternative algorithm based on sampling. Extensions of common clustering algorithms have also been proposed for big data platforms. The work in [15] presents an adaptation of K-Means for the Hadoop framework. K-Means is also supported in the Spark framework [16]. Considering the advantages and limitations of grouping and clustering, previous work introduced the Similarity Grouping (SGB) operator for numeric data [9, 17] to integrate the advantages of both types of operators, i.e., fast execution times and similarity-based grouping. SGB allows the specification of the desired grouping using descriptive properties such as group size and compactness. Tang et al. extended SGB operators to support multi-dimensional data [10]. While SGB operators cannot identify all the complex forms of clusters identified by clustering algorithms, they identify groups that are useful in many scenarios.

Considering that SGB operators were proposed for single-node relational databases, this paper introduces the Distributed Similarity Grouping (DSG) operator, a highly parallel and scalable approach for identifying similarity groups in big datasets. DSG identifies a particularly useful type of similarity groups where all the elements of a group are within a given threshold from each other. The proposed algorithm can be used with any distance function and data type. We also present guidelines to implement DSG in Spark and Hadoop and extensively assess its performance and scalability properties. We show that DSG performs significantly better than K-Means for identifying similarity groups and maintains execution times that are close to those of standard grouping.

2 The Distributed Similarity Grouping Algorithm

This section presents the algorithmic details of DSG. The type of similarity groups identified by this algorithm are groups where given any pair of elements (r_1, r_2) of a group, their separation is no larger than a parameter threshold (ε), i.e., distance(r_1, r_2) $\leq \varepsilon$. Like many clustering algorithms, DSG is non-deterministic, i.e., it is possible that different executions of the algorithm on the same dataset could generate slightly different solutions. Unlike K-Means, DSG does not require advance knowledge of the number of clusters. DSG uses pivot-based data partitioning to distribute and parallelize the computational tasks. The goal is to divide a large dataset into partitions that can be processed independently and in parallel. The pivots are a subset of input data records

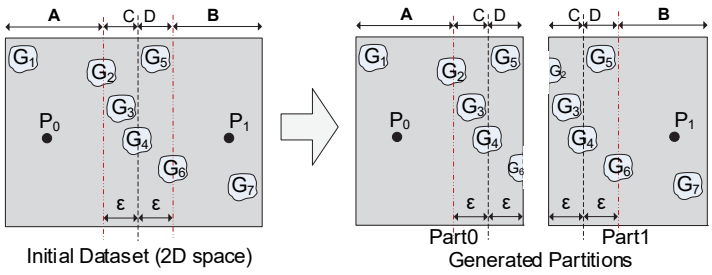

Fig. 1. Example of partitioning and similarity group generation using two pivots.

and each pivot is associated with a partition. Each input record is assigned to the partition associated with its closes pivot. In addition, DSG replicates the records at the boundary between partitions to ensure that similarity groups located in these regions are properly detected. Fig. 1 shows an example of how DSG partitions and identifies the similarity groups using two pivots (P_0 and P_1). The left image represents the initial dataset with seven clusters or similarity groups (G_1 to G_7). The right image represents the two generated partitions (*Part0* and *Part1*). Observe that regions A and C contain the records that are closer to P_0 than to P_1, while B and D the points that are closer to P_1 than to P_0. Regions A+C and B+D are referred to as *base* partitions. Observe that the records in A+C and B+D are assigned to partitions *Part0* and *Part1*, respectively. Also, the regions at the boundary between the base partitions (points within Ɛ from the boundary) are replicated, i.e., region D is added to *Part0* and C to *Part1*. Regions C and D are referred to as *window* partitions. The only problem is that some of the groups (G_2 to G_6) are partially or fully contained in both partitions. Our approach needs a mechanism to output each similarity group only once and ensure that a group is outputted in the partition that contains the entire group. To this end, DSG applies the following guidelines: (1) during partitioning, each record r in a given partition P is augmented with information of its base partition (partition of its closes pivot) and assigned partition (P), and (2) given any generated similarity group g, the group will be outputted only in the partition matching the smallest base partition among all the records in g. In Fig. 1, G_1, G_2, G_3, and G_4 are outputted in *Part0* while G_5, G_6, and G_7 in *Part1*.

Algorithm 1 presents DSG's main algorithmic steps. After pivots are generated (line 1), the data is partitioned in parallel (lines 3-11). The partitioning phase is implemented using map operations in Spark and Hadoop. Each input record r is assigned to the partition of its closes pivot P_c (lines 4-5) and all the partitions p where r belongs to the window regions between the partitions of p and P_c (lines 6-10). In general, the records in the window region between two partitions should be a superset of the records whose distance to the hyperplane that separates the partitions is at most Ɛ. 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 r to the generalized hyperplane between two partitions for pivots P_0 and P_1 cannot always be computed exactly, a lower bound is used [18] (line 7): *genHyperplaneDist*(r, P_0, P_1) = (distance(r, P_0) - distance(r, P_1))/2. This distance can be replaced by an exact distance when this can be calculated, e.g., with Euclidean distance, *genHyperplaneDist* can be

```
Algorithm 1 DistSimGrouping
Input: inputData, eps, numPivots, memT
Output: similarity groups in inputData
1  pivots = selectPivots(numPivots, inputData)
2  //Partitioning - r: ⟨ID, value,
3  assignedPartitionSeq, basePartitionSeq⟩
3  for each record r in a chunk of inputData do
4    Pc = getClosestPivot(r, pivots)
5    output ⟨Pc, r⟩ //intermediate output
6    for each pivot p in {pivots-Pc} do
7      if (dist(r, p) - dist(r, Pc))/2 ≤ eps then
8        output ⟨p, r⟩ //intermediate output
9      end if
10   end for
11 end for
12 //Shuffle: records with same key => partition
13 //Group Formation
14 for each partition Pi do
15   if size of Pi > memT then
16     store Pi for processing in subsequent round
17   else
18     Ci = findSimGroups(Pi, eps) //Ci:{Ci_k},
19     //Ci_k:⟨records, flags⟩, flags:{Fm}, Fm:{fm_n}
20     //Output Generation (without duplication)
21     for each cluster Ci_k in partition Pi do
22       generate minFlags //minFlags[o]={index
         //of 1st element in Ci.flags[o] equal to 1}
23       aPartitionSeq = r.assignedPartitionSeq
24       //r is any record in Pi
25       if ∀o,minFlags[o]=aPartitionSeq[o] then
26         output Ci_k //final output
27       end if
28     end for
29   end if
30 end for
```

Alg. 1. Main DSG algorithm.

```
Algorithm 2 findSimGroups
Input: S (data records), eps
Output: C (list of clusters or similarity groups)
1  C={}
2  for each record r in S do
3    clusterFound = False
4    for each cluster Ci in C do
5      if distance(r, Ci.getCentroid()) > eps then
6        //r cannot belong to this cluster
7      else //r may or may not belong to Ci
8        withinEps = True  // r is within eps of
         Ci's centroid, now verify all points in Ci
9        for each record x in Ci do
10         if distance(r, x) > eps then
             //r does not belong to Ci
11           withinEps = False
12           break
13       end for
14       if (withinEps =True) then//r belongs to Ci
15         Ci.add(r) //adds to Ci.records
16         Ci.updateCentroid()
17         clusterFound=True
18         break
19       end if
20   end for
21   if (clusterFound = False)
22     Create Cluster Cnew
23     Cnew.add(r)
24     Cnew.updateCentroid()
25     C.add(Cnew);
26   end if
27 end for
28 for each cluster Ci in C do
29   generateFlags(Ci) //updates Ci.flags
30 end for
31 return C
```

Alg. 2. findClusters method.

replaced by $euclideanHyperplaneDist(r, P_0, P_1) = |(\text{distance}(r, P_0)^2 - \text{distance}(r, P_1)^2| / (2 \times \text{distance}(P_0, P_1))$. The partitioning phase records the information of base and assigned partitions of each record. The intermediate records generated during partitioning are grouped in the shuffle phase (line 12) such that all the records that belong to the same partition will form a single group. This is performed automatically in the shuffle phase of Hadoop and is implemented using the grouping operator in Spark. In the similarity group formation and output generation phase (lines 14-30), similarity groups are identified and outputted in each partition and in parallel. We first check if a partition is small enough to be efficiently processed in a single node (line 15-16). If this is not the case, the partition is stored for further processing using the same SGB algorithm. While this feature guarantees that the algorithm will be able to effectively partition datasets with high concentration of records in certain regions, in practice, we can increase the number of pivots and thus decrease the size of partitions to guarantee a single round. If

a partition is small enough to be processed in a single node, the algorithm runs a single-node algorithm (*findSimGroups*) to identify the similarity groups of a single partition (line 18). The output of this algorithm is a set of clusters and each cluster is composed of its data records and information needed to ensure non-duplicated cluster generation (*flags*). The *flags* component of a given cluster C maintains a sequence of flag arrays (one array per round where the cluster was processed). For instance, if four pivots are being used (P_0, P_1, P_2, P_3) and a single round is needed, the content of $C.flags$ has the form $\{[f_{0\_0}, f_{0\_1}, f_{0\_2}, f_{0\_3}]\}$. The content of this structure could be for example {[0, 1, 0, 1]}. A value of 1 at index i indicates that cluster C contains at least one record whose base partition is the one associated with pivot P_i. The *flags* component is used to determine if a given cluster should be outputted while processing the current partition or not (lines 21-28). A given cluster of partition P_i will be outputted only if the minimum index on the corresponding flag array matches i (lines 25-27). The similarity group formation phase is performed using the reduce operations in Hadoop and Spark. The details of the *findSimGroups* method are presented in Algorithm 2. For every record r, the algorithm tries to identify a suitable cluster among the ones that were already formed (lines 2-17). If Euclidean distance is used, a centroid-based filter is used to discard non-suitable clusters (line 5). For a potentially suitable cluster, the algorithm checks the similarity group condition between r and each element of the cluster (lines 9-13). If this check is successful, r can be added to this cluster (lines 14-19). If the process ends without finding a cluster, a new cluster is created and r is added to it. After processing the input records, the method generates the *flags* components (line 28-30).

The implementation of DSG in Hadoop uses the job configuration (*jobConf*) object to distribute atomic parameters, e.g., *eps*, *memT*, and *numPivots*, to all the nodes. It also uses the random sampling and distributed cache facilities to generate the pivots and distribute them to the nodes. The partitioning and group formation phases are implemented through the *map* and *reduce* operations of Hadoop's MapReduce framework. Furthermore, customized MapReduce grouping and sorting operations were created to support the specific data structures of our solution during the shuffle stage. The Spark implementation uses the RDD API and is significantly shorter due to its robust support of data processing operations. In this case, the *sample* operation is used to select the pivots. Then, the *mapPartitionsToPair* operation is used to implement the partitioning phase and the *groupByKey* operation to group the records that belong to the same partition. Finally, the *flatMap* operation is used to perform the clustering generation phase.

3 Performance Evaluation

We implemented DSG, standard grouping (StandG), and K-Means clustering in Hadoop 2.9.1 and Spark 2.3.2. The experiments were executed on Google Cloud Platform. Unless otherwise stated, the cluster consisted of one master and ten worker nodes. Each node had 4 virtual CPUs, 15 GB of memory and 500 GB of disk space. The number of reducers per Hadoop job was set to 0.95 × (# of worker nodes) × (# of vCPUs per node - 1) and the number of splits per Spark job was 2 × (# of worker nodes) × (# of vCPUs). We implemented a parametrized synthetic dataset generator that enabled us to evaluate

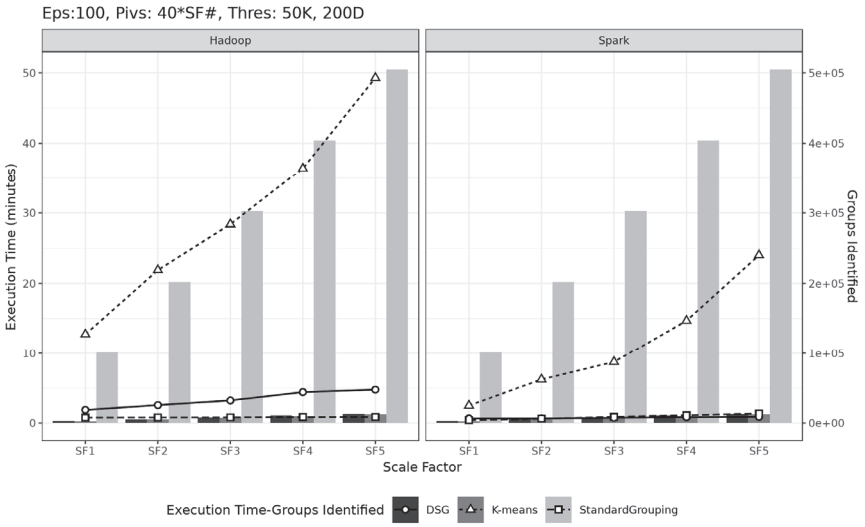

Fig. 2. Execution time when increasing dataset size.

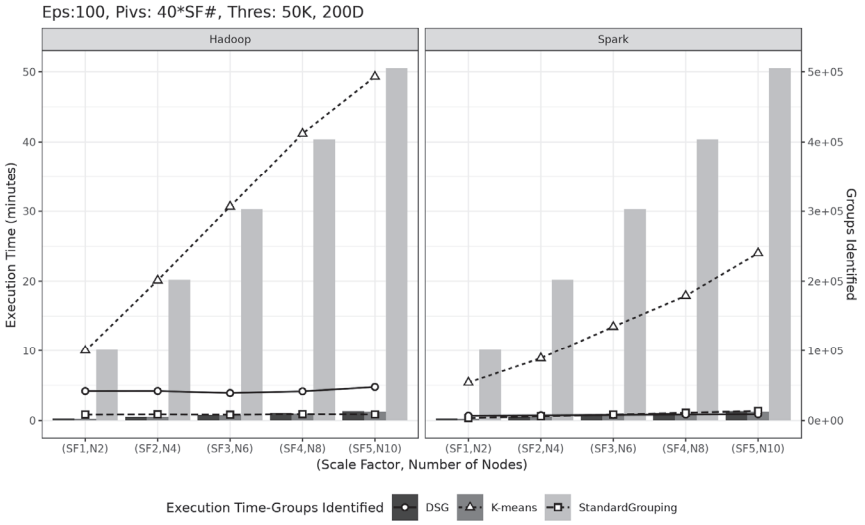

Fig. 3. Execution time when increasing dataset size and number of cluster nodes.

the algorithms under a variety of conditions. The datasets were composed of multidimensional vector-based similarity groups separated by 2Ɛ. Given this dataset, DSG and K-Means were expected to have the same output while StandG only identified equality-based groups. Each data record consisted of an ID, an aggregation attribute, and a randomly generated multidimensional vector (100D-500D). The dataset for scale factor N (SFN) had 200,000×N records. The SF1 datasets contained about 13,000 groups

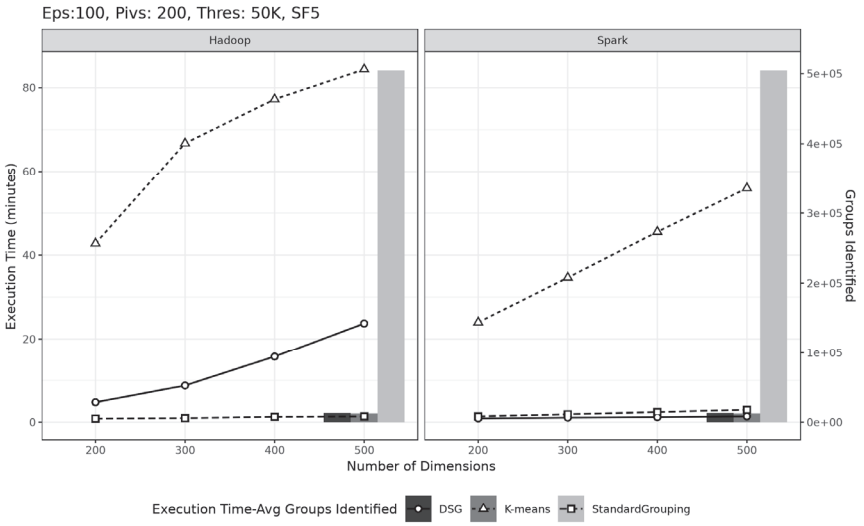

Fig. 4. Execution time when increasing the number of dimensions.

and each of them contained 50-100 records. Each record was duplicated between 1-3 times. We set DSG's *numPivots*=40×SF and *memT*=50,000 based on preliminary tests.

Increasing Scale Factor. Fig. 2 shows the execution time (lines) and the number of groups (bars) identified by DSG, StandG and K-Means as the scale factor increases. The execution times of DSG and StandG increase slowly as the scale factor increases. K-Means' execution time, on the other hand, is significantly larger than those of DSG and StandG. In Spark, DSG is about 13 times faster than K-Means while in Hadoop, about 8 times faster. While StandG generates a very large number of equality-based groups, DSG and K-Means identify the same similarity groups.

Increasing Scale Factor and Number of Cluster Nodes. Fig. 3 compares the execution time and the number of identified groups of DSG, StandG, and K-Means as the data size and number of nodes increase. In this experiment, we increase the scale factor and number of nodes available to the cluster from (SF1, 2 nodes) to (SF5, 10 nodes). DSG and StandG maintain near constant execution times while K-Means' performance increases significantly. In Hadoop, DSG's execution time for (SF5, 10 nodes) is approximately 1.25 times that of (SF1, 2 nodes). In Spark, it is practically constant.

***Increasing Number of Dimensions*.** We executed each algorithm with 200D-500D datasets while fixing the scale factor. Because SF is fixed, the number of groups in each dataset is nearly equal. Fig. 4 shows that the execution time of all algorithms increases when dimensionality increases. As expected, StandG has the best execution times, K-Means the worst ones, and DSG execution times are closer to those of StandG than to the ones of K-Means. In Spark, at 200D, DSG is 26 times faster than K-Means while at 500D, 39 times faster. The difference in execution times witnessed in Hadoop is less acute. At 200D, DSG is 9 times faster than K-Means while at 500D, 3.5 times faster.

4 Conclusions

This paper introduces the Distributed Similarity Grouping (DSG) operator to efficiently identify similarity groups in very large datasets. The paper presents the general algorithmic details of DSG and the guidelines for its implementation in two popular big data systems. An extensive performance evaluation shows that DSG is successful at identifying similarity groups identified by the K-Means clustering algorithm while having small execution times that are in general very close to those of standard grouping. Future work in this area includes (1) the study of alternative ways of selecting the pivots, (2) the development of distributed similarity algorithms for other types of similarity groups, and (3) studying optimization techniques for non-vector data, e.g., text and sets.

References

1. Apache. Hadoop. https://hadoop.apache.org/.
2. Apache. Spark. https://spark.apache.org/.
3. J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In OSDI, 2004.
4. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst., 26(2): 1–26, 2008.
5. H. Garcia-Molina, J. Ullman, and J. Widom. Database Systems: The Complete Book. Pearson, 2nd Edition.
6. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. In ICDE, 1996.
7. S. P. Lloyd. (1982). Least squares quantization in PCM. IEEE Trans. on Information Theory. 28 (2): 129–137, 1982
8. M. Ester, H.P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters. In KDD, 1996.
9. Y. N. Silva, W. G. Aref, and M. Ali. Similarity Group-by. In ICDE, 2009.
10. M. Tang, R. Y. Tahboub, W. G. Aref, M. J. Atallah, Q. M. Malluhi, M. Ouzzani, and Y. N. Silva. Similarity Group-by Operators for Multi-dimensional Relational Data. IEEE Trans. on Knowledge and Data Engineering, 28(2): 510-523, 2016.
11. P. Berkhin. Survey of clustering data mining techniques. Accrue Software, 2002.
12. M. Li, G. Holmes, and B. Pfahringer. Clustering large datasets using Cobweb and K-Means in tandem. The Australian Joint Conference on Artificial Intelligence, 2004.
13. F. Farnstrom, J. Lewis., and C. Elkan: Scalability for clustering algorithms revisited. SIGKDD Explorations Newsletter, 2 (1): 51–57, 2000.
14. S. Guha, R. Rastogi, and K. Shim. CURE: An efficient clustering algorithm for large databases. In SIGMOD Record, 27(2): 73–84, 1999.
15. P. P. Anchalia, A. K. Koundinya and S. N. K. MapReduce Design of K-Means Clustering Algorithm. In ICISA, 2013.
16. Apache. Spark Clustering. https://spark.apache.org/docs/latest/ml-clustering.html.
17. Y. N. Silva, M. Arshad, and W. G. Aref. Exploiting Similarity-aware Grouping in Decision Support Systems. In EDBT, 2009.
18. E. H. Jacox and H. Samet. Metric space similarity joins. ACM Trans. Database Syst., 33(2):7:1–7:38, 2008.