JMLR: Workshop and Conference Proceedings 41:81–96, 2015 BIGMINE 2015

Shared Execution of Clustering Tasks

Padmashree Ravindra PARAVIN@MICROSOFT.COM
Microsoft Corporation, USA

Rajeev Gupta GRAJEEV@IN.IBM.COM
IBM Research, India

Kemafor Anyanwu KOGAN@NCSU.EDU
North Carolina State University, USA

Editors: Wei Fan, Albert Bifet, Qiang Yang and Philip Yu

Abstract

Clustering is a central problem in non-relational data analysis, with k-means being the most popular clustering technique. In various scenarios, it may be necessary to perform clustering over the same input data multiple times – with different values of k, different clustering attributes, or different initial centroids – before arriving at the final solution. In this paper, we propose algorithms for parallel execution of multiple runs of k-means clustering in a way that achieves substantial savings of IO and processing resources. Proposed algorithms can easily be implemented over Hadoop/MapReduce, Spark, etc., with savings in *map* and *reduce* phases. Extensive performance evaluation using real-world datasets show that the proposed algorithms result in up to 40% savings in response times when compared to other optimization techniques proposed in literature as well as open-source implementations. The algorithms scale well with increasing data sizes, values of k, and number of clustering tasks.

1. Introduction

Clustering is a key data mining task, widely used (Aggarwal and Reddy, 2013) in many fields, including social network analysis, customer segmentation, and biological data analysis. The goal of a clustering task is to partition the data into interesting groups based on similarity of their characteristics, e.g., purchasing patterns of customers or interests of Twitter users. k-means is the most popular clustering technique, due to its simplicity and wide applicability. A popular heuristic for k-means clustering is the Lloyd's algorithm (Lloyd, 1982), that consists of two primary steps:

1) *Assignment step*: assign each data point to the *closest* of k clusters based on a user-specified distance metric.

2) *Recalculation step*: re-calculate each cluster center based on the set of data points assigned to that cluster.

The above two steps are repeated (Lloyd's iteration) sequentially until the algorithm converges, i.e., k final clusters are generated. The k-means clustering process typically involves running the algorithm multiple times with different values of k, different clustering attributes, different initial centroids, etc. Consider a scenario where a business analyst wants to cluster customers based on their age, income, interest in sports, and general health. A cluster analysis based on these attributes can be used to devise customized marketing

strategies to cater to the needs of specific groups of people such as healthy and interested in sports, senior citizens interested in specific fitness programs, etc. The analyst may try out different cluster sizes (values of k) and different clustering attributes such as:

T_1: `CLUSTER` *users* `INTO` k=2 `ON` { *age, income, bloodPressure, sportsInterest* }
T_2: `CLUSTER` *users* `INTO` k=4 `ON` { *age, income, bloodPressure, sportsInterest* }
T_3: `CLUSTER` *users* `INTO` k=2 `ON` { *age, sportsInterest* }

Task T_1 clusters *users* into 2 groups based on age, income, blood pressure, and sports interest, whereas T_2 clusters them into 4 groups (same attributes). T_3 clusters users based on a different set of attributes. After analyzing the clustering results, the analyst can decide on *reasonable* groupings based on the density of clusters, cluster sizes, and convergence criteria, and run more tasks with different input parameters. Results for the three tasks are completely different, providing different views of the same data, one of the main motivations for multi-clustering solutions[1]. Typically, multiple runs of clustering with different criteria are executed sequentially. In this paper, we consider scenarios where a number of such trial-error iterations can be performed in parallel, specifically using distributed algorithms that enable sharing of *assignment* and *recalculation* steps across clustering tasks.

1.1. Large Scale k-means Clustering

Large scale clustering solutions can be supported by leveraging various distributed data processing platforms (Bialecki et al.; Isard et al., 2007; Zaharia et al., 2010). Apache Hadoop[2], the most popular open-source distributed data processing framework, allows easy scale-out processing on large clusters of cheap commodity hardware in a fault-tolerant manner. Hadoop implements the MapReduce (Dean and Ghemawat, 2004) programming model, which allows users to encode computational tasks as *map-reduce* function pairs, that are executed in parallel across a number of machines. Input data is stored in the Hadoop Distributed File System (HDFS). During processing, chunks of data are processed by slave processes called *mappers*, which execute a user-defined *map* function on each input data record. Mappers emit *key-value* pairs that are sorted and partitioned based on keys. The partitioned data records are temporarily stored in the local disk of mappers. Reducers fetch their assigned partitions from the mappers, sort the records based on keys, and invoke a user-defined *reduce* function on each *key* partition. Though, in this paper, we illustrate implementations of the clustering solutions using Hadoop MapReduce, the algorithms can benefit from various aspects provided by systems such as Apache Spark (Zaharia et al., 2010), e.g., main-memory processing, user-controlled process execution, etc.

Let us consider the implementation of a k-means clustering task on Hadoop MapReduce and the associated IO and processing costs. Each Lloyd iteration, involving *assignment* and *recalculation* steps of a k-means clustering task can be implemented as a MapReduce job (referred as *NoShare* in Section 4), as given in literature (Pais and Rong, 2011): For each data point, mappers extract the clustering attributes, assign the data point to the nearest cluster based on a user-defined distance metric, and emit the identifier of the closest cluster ($clusId$) as the *key*, and the data point as the *value*. Reducers aggregate data points

1. http://dme.rwth-aachen.de/en/DMCS
2. https://hadoop.apache.org/

belonging to the same cluster i.e., same *clusId*, and recalculate centroids. The new set of centroids are written into HDFS. A driver program of the k-means clustering task runs the *map-reduce* functions iteratively while monitoring the convergence of data clusters. Example convergence criteria include maximum number of iterations, a threshold over the sum of distances between centroids of consecutive iterations, etc.

Each *MapReduce* iteration incurs HDFS read IO cost, processing cost for the *assignment* phase, data shuffling cost involving local disk writes, network cost to transfer data points to reducers, processing cost to recalculate centroids, and HDFS write IO costs. In our example scenario, multiple iterations of each of the clustering tasks T_1, T_2, and T_3, are executed sequentially, compounding the overall costs. Furthermore, the three tasks read the same input data (redundant scans) and perform similar processing in the *assignment-recalculation* steps, which may be avoidable.

1.2. Contributions and Outline

Several techniques have been proposed to reduce costs of MapReduce-based processing by reducing the length of MapReduce execution workflows (Afrati and Ullman, 2010; Abouzeid et al., 2009; Lee et al., 2011), sharing scans and computations across MapReduce jobs (Nykiel et al., 2010; Wang et al., 2011) and reusing intermediate data for iterative data analysis tasks (Bu et al., 2010). An important question to be answered is, *Can these techniques be applied without understanding the semantics of the underlying operations?* There is some existing work that considers relational operations (Nykiel et al., 2010). *Are the same principles applicable to non-relational operations?* In this paper, we use specific characteristics of k-means clustering to guide shared execution of multiple clustering tasks. Ours is the first work considering optimizations across multiple clustering tasks. While there exist some extensions of Hadoop for iterative tasks (Bu et al., 2010) and incremental computations (Bhatotia et al., 2011), we use clustering semantics to propose optimizations across clustering tasks. Specifically we make the following contributions:

- We identify *cost-sharing opportunities* that benefit use-cases requiring multiple runs of clustering with different criteria, e.g., value of k or clustering attributes.
- We propose a novel algorithm to improve performance of the *assignment* step when executing multiple clustering tasks in parallel. The key idea is to use the *assignment* of one clustering task to guide assignments in other clustering tasks without calculating distances between a data point and all cluster centers.
- Distributed processing on platforms like Hadoop, involves network data transfer to aggregate data points for the *recalculation* step. We propose an algorithm that allows sharing of data transfers across multiple clustering tasks, thereby reducing the number of intermediate *map* keys and *reducer* processing time.
- We present a comprehensive performance evaluation using two real-world astronomy data sets and include comparison with a Mahout-based (Pais and Rong, 2011) implementation of k-means clustering. Experiments demonstrate that the proposed algorithms require half the response time compared to other MapReduce algorithms.

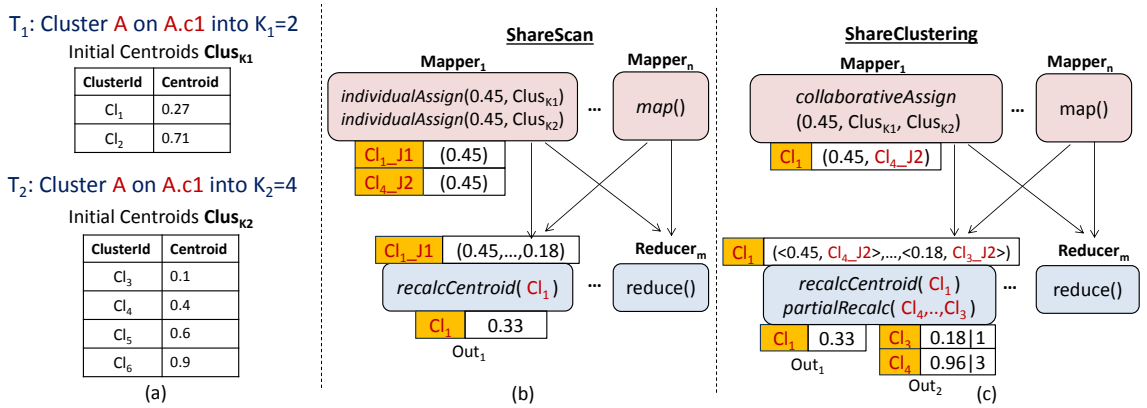


Figure 1: (a) Clustering tasks T_1 and T_2 with initial centroids $Clus_{K1}$ and $Clus_{K2}$, Shared execution of T_1 and T_2 using (b) ShareScan and (c) ShareClustering

Here is the organization of the paper: Section 2 describes the *ShareScan* algorithm for sharing of scans across tasks based on Nykiel et al. (2010). Section 3 presents clustering-specific algorithms that enable sharing of the *assignment* and *recalculation* steps across tasks, implemented as the *ShareClustering* algorithm. Section 4 presents evaluation results, followed by related work and concluding remarks in Section 5 and 6, respectively.

2. Sharing Scans Across Clustering Tasks

In this section, we describe how input data scans for multiple clustering tasks can be shared (*ShareScan*) to avoid redundant data reads. Nykiel et al. (2010) present a case for sharing scans across a number of relational `GROUP BY` queries. Consider m clustering tasks (identified by $taskId$) over the same input data each with different values of k $(k_1, k_2, ..., k_m)$ and different sets of initial centroids ($Clus_{k_1}$, $Clus_{k_2}$,...,$Clus_{k_m}$). Rather than executing the tasks independently, a *merged* task can read the input data once and perform k-means clustering as per requirements of different tasks. The mapper function of the *merged* task extracts clustering attributes for each $taskId$, uses them to compute the closest cluster (one for each $taskId$) and emits the $taskId\_clusId$ as key with data point as value. Thus, each data point leads to m emissions, one for each of the tasks. All the data corresponding to the same $taskId\_clusId$ are sent to the same reducer, which recalculates the cluster centroid. The new cluster information with the recalculated centroid is written into a HDFS file corresponding to $taskId$ which can be used by the next Lloyd's iteration. A driver program runs these steps iteratively till all tasks converge.

Consider a set of initial clusters $Clus_{k_1}$ and $Clus_{k_2}$ in Figure 1(a), corresponding to two tasks T_1 ($k_1 = 2$) and T_2 ($k_2 = 4$). Figure 1(b) illustrates the *ShareScan* algorithm. Let `individualAssign()` be the function responsible for cluster assignment. Then, for a data point (0.45), `individualAssign()` determines that the closest cluster for task T_1 is $Cl_1 \in Clus_{k_1}$ (and $Cl_4 \in Clus_{k_2}$ for task T_2). Two map output records are generated with keys $Cl_1\_T_1$ and $Cl_4\_T_2$, and assigned to reducers based on the *composite* key. Thus, each reduce recalculates the centroid for some cluster across tasks.

To summarize, the *ShareScan* implementation shares data scans, while having independent reducers (as $taskId$ is part of the intermediate key). MapReduce's *Combiner* function can be used to pre-aggregate data points assigned to the same $taskId\_clusId$, to reduce

data shuffling costs. Reducers *recalculate* centroids based on the pre-aggregated cluster information. Shared scans and combiners are well-known optimizations that can improve performance of distributed data processing. As we show in Section 4, a combination of these techniques can lead to cost savings, e.g., a combiner implementation leads to 6% performance improvement for a clustering task with $k = 200$. Clustering costs can be further reduced by using semantics of k-means clustering as shown in the next section.

3. Shared Execution of Clustering Tasks

In the *ShareScan* algorithm, though the cluster *assignment* phase for multiple tasks is done together by enabling scan-sharing, assignment of data points to clusters is done independently. We present an algorithm for a *collaborative* cluster assignment that re-uses the assignment information for one task to enable efficient assignment for other tasks. This algorithm achieves cost sharing by reducing the number of intermediate keys, i.e., *map* output of different tasks are aggregated, thus reducing the data transfer overhead. In combination, the techniques allow sharing of computations across both *map* and *reduce* phases.

3.1. Collaborative Cluster Assignment

In this section, we present a collaborative technique for assignment of data points to centroids of different clustering tasks with same set of clustering attributes. Specifically, we present a geometric argument to the effect that, *if a data point is assigned to a particular centroid in task T_1, the data point cannot be assigned to a set of centroids in a task T_2.*

Consider a data point D, and a set of initial centroids: $Clus_{k_1} = \{P_1, P_2\}$ and $Clus_{k_2} = \{S_1, S_2, ..., S_6\}$, corr. to clustering tasks T_1 ($k = 2$) and T_2 ($k = 6$), respectively. In *ShareScan*, `individualAssign`(D, $Clus_{k_1}$) calculates the distance of D from centroids P_1 and P_2. Similarly, `individualAssign`(D, $Clus_{k_2}$) calculates six distances (between D and centroids S_1 to S_6). In essence, while executing m k-means clustering tasks, we calculate the distance of each data point D from $\sum_{i=1}^{m}(k_i)$ centroids. Such processing costs may not be negligible for large values of k, large number of clustering tasks, and large data sizes. The question we need to answer is, *Can we reduce the number of distance calculations per data point?* Doing so can save cost per data point and potentially reduce processing costs.

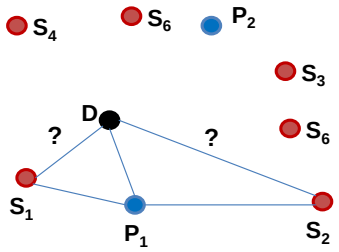


Figure 2: Is $S_1D < S_2D$?

We denote distances between centroids P_i and S_j as P_iS_j. Assume that for task T_1, `individualAssign`(D, $Clus_{k_1}$) is already computed, i.e., distances P_1D and P_2D are available. Let $S' = Clus_{k_2}$ denote the set of possible centroids in T_2 that may be closest to D. We now use distances between centroids in $Clus_{k_1}$ and $Clus_{k_2}$, along with distances of the data point with P_1 and P_2 (i.e., P_1D and P_2D), to eliminate centroids in S' that cannot be closest to D.

Consider points D, P_1, S_1, and S_2 in Fig. 2. By triangle inequality[3], we have:

$$P_1S_1 - P_1D < S_1D < P_1S_1 + P_1D$$

3. Sum of lengths of any two sides of a triangle must be greater than length of the remaining side

$$P_1S_2 - P_1D < S_2D < P_1S_2 + P_1D$$

Then, $S_1D < S_2D$ if $P_1S_1 + P_1D < P_1S_2 - P_1D$, i.e., $2P_1D < P_1S_2 - P_1S_1 = X_{112}$.

Values of X_{ijk} can be calculated for each value of P_i, S_j, and S_k. These values can be calculated once and compared with P_iD for each data value. If we find that $2P_iD$ is less than X_{i12} (*elimination criteria*), then $S_1D < S_2D$ (D is closer to S_1 than S_2). Thus, S_2 can never be closest to D and hence can be eliminated from the set of potential centroids S'. Further, we can eliminate centroids based on any X_{ijk} greater than X_{i12}. For example, if $X_{i54} > X_{i12}$, then X_{i54} also meets the elimination criteria, and hence S_4 can be eliminated.

Algorithm 1 shows the pseudocode for the proposed *collaborative cluster assignment* of tasks with common input and clustering attributes. We designate one task as the primary task whose cluster assignment is done first (without any help from other tasks). As part of offline processing (lines 1-8), we pre-compute distances between centroids in the primary task $Clus_{prim}$ and a secondary task $Clus_{sec}$, and the difference in their distances (X_{ijk} in line 3). For each value X_{ijk}, we also pre-compute a list of centroids that can be eliminated (lines 4-8). This list also includes centroids that can be eliminated by all values of $X_{ij'k'} > X_{ijk}$. During data processing, we use the value of closest primary-task centroid $clusId_p$ in *collaborativeAssign* to find the smallest X_{ijk} that meets the *elimination criteria* (lines 10-11), and retrieve the set of centroids that can be eliminated (lines 12-13). We then calculate the closest centroid to D from remaining candidate centroids S' (line 14).

3.2. Sharing Cluster Information

For a set of m clustering tasks, the intermediate *map* output and network transfer costs can be reduced by sharing data references across tasks. Consider the illustration in Fig. 1(c). Task T_2's cluster information can be embedded into task T_1's cluster information, in a way that reduces the number of *map* output records. For a set of m clustering tasks, we now have just 1 map output record per data point (as opposed to m in *ShareScan*).

Algorithm 2 provides the pseudocode for sharing cluster information, whose input is the cluster information for m clustering tasks, represented as $Clus_{taskId}$. We designate a primary task and compute the closest cluster $clusId_p$ for the data point (line 16). The closest clusters corr. to each of the $(m-1)$ secondary tasks are computed and stored in $secClus$ (lines 17-19). The key of the map output is the cluster id of the primary task, as shown in Fig. 1(c). Cluster information of secondary tasks ($secClus$) is encoded into the value part of the map output. The map output is partitioned and assigned to reducers based on the primary cluster $clusId_P$.

Recalculating Centroids: Since intermediate map output records are partitioned based on the primary (task) centroids, each reduce has complete information about all data points assigned to the same primary cluster. However, data points assigned to the same primary cluster $clusId_P$, may belong to different secondary clusters, e.g., data points (0.45) and (0.18) belong to the same primary cluster Cl_1 in Fig. 1(c), but belong to different secondary clusters Cl_4 and Cl_3, respectively. Also, the cluster information for a secondary task may be distributed across reducers, e.g., some data points belonging to secondary cluster Cl_4 may be assigned to a different primary cluster. Hence, it is only possible to partially aggregate the data points for secondary tasks. Algorithm 3 shows the pseudocode for this

Algorithm: *Shared Execution of Clustering Tasks*

```
//Offline Processing for Collaborative Cluster Assignment:
 1  foreach P_i ∈ Clus_prim do
 2      foreach (S_j, S_k) ∈ Clus_sec do
 3          X_ijk ← (P_i S_k − P_i S_j);
 4      DescX ← Sort X based on decreasing X_ijk values;
 5      elimList ← null;
 6      foreach X_ijk ∈ DescX do
 7          Add S_k to elimList;
 8          elimMap_i.put(X_ijk, elimList);

Algorithm 1: Collaborative Cluster Assignment:
collaborativeAssign (Data point D, clusId_p, Clus_sec)
 9  S'← Clus_sec;
10  ceilingVal ← 2 * distance(D, clusId_p);
11  matchedX_ijk ← Smallest X_ijk greater than ceilingVal;
12  elimList ← elimMap_i.get(matchedX_ijk);
13  S' ← S' − elimList;
14  clusId_S ← individualAssign(D, S');

Algorithm 2: Sharing Cluster Information:
shareClusInfo (dataPoint, List of Clus_taskId)
    //Clus_taskId: <clusId, centroid> for taskId
15  D ← extract clustering attributes from dataPoint;
16  clusId_p ← individualAssign(D, Clus_prim);
17  foreach secondary task id sec do
18      clusId_s ← collaborativeAssign(D, clusId_p, Clus_sec);
19      Add (sec_clusId_s, D, 1) to secClus;
    //secClus: secondary task cluster info
20  return ⟨clusId_p, secClus⟩;

Algorithm 3: Updating Cluster Information:
updateClusInfo (clusId_p, List of < secClus >);
21  Cnt_P ← 0 //taskId_clusId denoted as tid_cid
22  foreach <tid_cid, sumD, cntD> ∈ secClus do
23      Aggr_tid_cid ← aggregate(sumD, Aggr_tid_cid);
24      Cnt_tid_cid ← cntD + Cnt_tid_cid;
25  foreach cluster id i in some secondary task sec do
26      Aggr_p ← aggregate(Aggr_sec_i, Aggr_p);
27      Cnt_P ← aggregate(Cnt_sec_i, Cnt_p);
28  centroid_p ← recalcCentroid(Aggr_p, Cnt_p);
29  foreach aggregated secondary task entry for tid_cid do
30      PC_tid_cid ← partialRecalc(Aggr_tid_cid, Cnt_tid_cid);
```

process. First, for each primary cluster, the secondary cluster information in *secClus* is aggregated based on secondary task *tid* and cluster id *cid* (lines 22-24). Partial centroids PC_{tid_cid} are calculated for each secondary task using this aggregated information. Aggregates for the primary cluster can be computed based on data points assigned to any of the

secondary tasks (lines 25-27), which is then used to recalculate the centroid for the primary cluster (line 28). Once the partial centroids corr. to all secondary tasks are available, the final centroids can be computed by aggregating the partial centroids. The proposed algorithms were implemented using MapReduce, referred as *ShareClustering* for the rest of this paper. In *map*, for each data point, we invoke `shareClusInfo`(), i.e., cluster *assignment* of primary task using `individualAssign`() and *assignment* of secondary tasks using `collaborativeAssign`(). In *reduce*, we aggregate the data points corresponding to a primary centroid, and call `updateClusInfo`() to *re-calculate* final primary centroids and partial secondary centroids.

Selection of a Primary Task. Several factors impact the selection of a primary task. Number of intermediate keys in *ShareClustering* equals the number of clusters in the primary task. A primary task with very small k, leads to load balancing issues among reducers, resulting in overall increase in execution time. If convergence conditions for all tasks are specified wrt. *number of iterations*, one may select the one with maximum iterations as the primary task. Another convergence criteria specifies a limit on residual sum of squares (RSS) value of the clustering solution[4], i.e., sum of squared distance of each data point from its centroid. In such cases, one can estimate the relative number of iterations based on data statistics and the convergence condition. More detailed analysis in Section 4.

Sharing Clustering Attributes. *ShareClustering* can easily be extended for tasks with different clustering attributes. For example, clustering attributes for task T_3 is a subset of task T_1's attributes. In some cases, clustering attributes may form disjoint sets. Our solution is to maintain a union of the clustering attributes, and calculate aggregates for a particular task based on the relevant subset. A task whose clustering attributes have a maximum overlap (or coincide) with the superset of attributes across tasks, is nominated as the *leading secondary task*. A union of the required clustering attributes is extracted for each data point, and is recorded with the entry of the leading secondary task. Relevant subsets corresponding to other secondary tasks are extracted from this superset of attributes, while calculating partial aggregates. A study on the selection of the leading secondary task is included in Section 4.

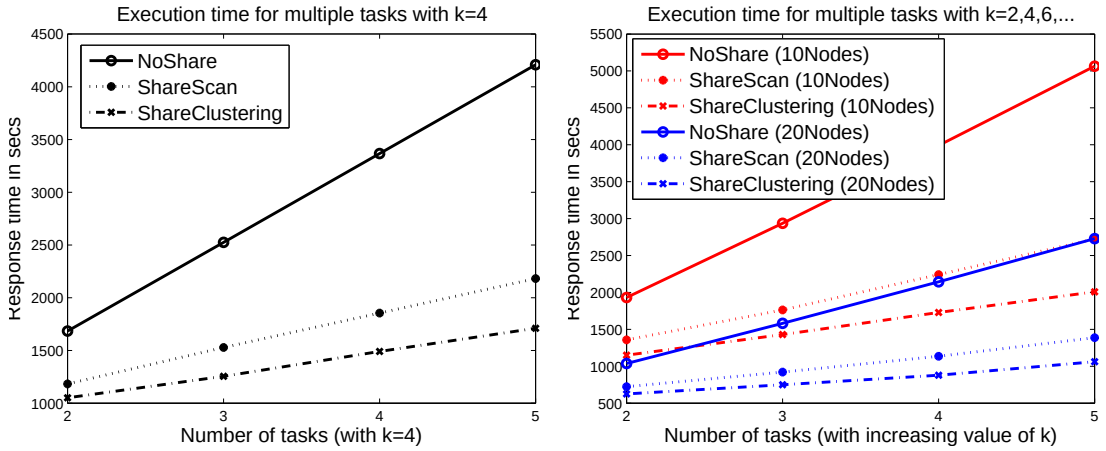
4. Empirical Evaluation

We designed a set of experiments to evaluate the various features of our algorithm, such as scalability with increasing number of clustering tasks, values of k, data sizes, and number of clustering attributes. Cost analysis of the proposed algorithms is available in Appendix A.

Experiment Setup: Experiments were conducted on 10-node and 20-node Hadoop setups, with each node being dual core Intel X86 machine with 2.33 GHz processor speed, 4GB memory, and running Red Hat Linux. The software stack comprises of Hadoop-0.20.2 with HDFS block size 256MB, replication factor 2, and heap-size for child threads set to 1024MB. All results recorded were averaged over three trials.

Datasets: We used two real-world astronomy datasets that contain snapshots of particles from a cosmological simulation[5] of the Universe. Each record is a multi-dimensional vector with 10 to 13 numeric attributes describing mass, velocity, temperature, etc. For evaluation,

4. http://nlp.stanford.edu/IR-book/html/htmledition/k-means-1.html
5. http://nuage.cs.washington.edu/benchmark/astro-nbody/



(*a*) Cosmo-Dark, k=4,4,4,...(10-nodes) (*b*) Cosmo-Gas, k=2,4,6,...(10 vs.20-nodes)

Figure 3: Performance comparison with increasing number of clustering tasks

we use *Cosmo-Dark* and *Cosmo-Gas* datasets that correspond to snapshots of *dark matter* and *gas* particles, respectively. As part of preprocessing, numeric column values of these datasets were normalized using 0-1 normalization (Wang et al., 2009) to ensure that no single column is dominant during clustering. Data sizes after normalization were 16GB for *Cosmo-Dark* and 21.6GB and 128GB for the two *Cosmo-Gas* datasets. We also used a synthetic (decision support) benchmark dataset, TPC-H, to corroborate our results (not presented for lack of space).

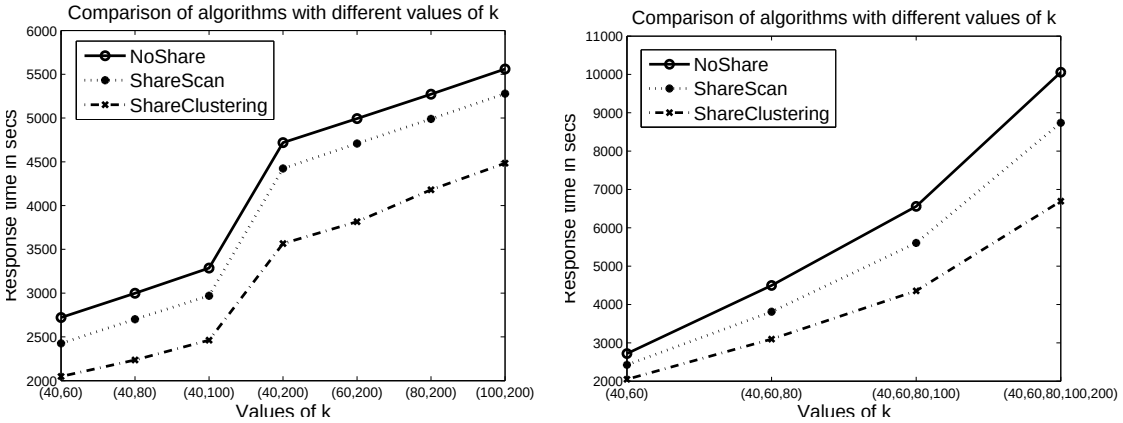
We report performance using three iterations of clustering, i.e., three MapReduce jobs with output as set of recalculated cluster centroids. We omit the last *map*-only job that assigns data points to clusters. All results are with combiner implementation. The *NoShare* implementation with combiner is similar to the k-means implementation in Mahout (described in Section 1.1). We use $k = (k_1, k_2,..)$ to represent values of k for multiple tasks, where k_1 represents the primary task for *ShareClustering*.

4.1. Scalability Analysis: Results and Discussion

In this section, we present a scalability study of *ShareClustering* with increasing values of k, number of tasks, data sizes, and number of clustering attributes.

1) Increasing Number of Clustering Tasks. Fig. 3(*a*) shows execution times for clustering tasks with k=4 but different initial centroids. For the first set with 2 tasks k=(4,4), *ShareClustering* shows 37% performance gain over *NoShare* by enabling sharing of *assignment-recalculation* steps. As we increase the number of clustering tasks to 5, *ShareScan* produces 5 tuples (one for each task) per data point while *ShareClustering* produces one tuple (all secondary task information piggybacked with primary task information) per data point. For this case, *ShareClustering* has a 59% gain over *NoShare*. Results confirm that response times of *ShareClustering* increase much slowly when compared to cases where data transfer and processing are not shared.

2) Varying Values of k. Fig. 3(*b*) shows evaluation on 10-node and 20-node Hadoop setups, with increasing number of tasks, i.e., first set with 2 tasks k=(2,4) to last set of 5 tasks with k=(2,4,6,8,10). Different values of k impact the number of map output



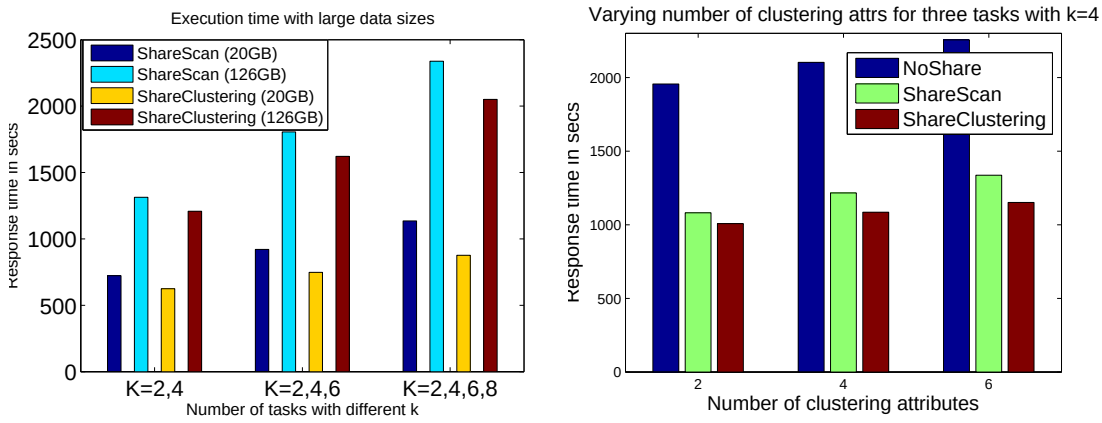
(a) Two tasks with increasing k (b) Increasing no. of tasks with increasing k

Figure 4: Study on impact of value of k (Cosmo-Gas, 24-nodes)

keys, e.g., for k=(2,4), *ShareScan*'s *combiner* will at most output 6 tuples per mapper, whereas for k=(2,4,6,8,10), the number will be 30. In case of *ShareClustering*, same number of tuples are produced per mapper but the size of tuple increases with the increase in number of (secondary) clustering tasks. Results show that *ShareClustering* improves the benefits of shared execution with a 15% gain over *ShareScan* for 2 tasks (40% over *NoShare*), and 26% with 5 tasks (60% over *NoShare*), respectively. We repeated the above set of experiments on a larger 20-node Hadoop setup to study how the proposed algorithms scale out. The increase in the availability of compute nodes allows more parallel map processing and reduces the time for input scans, map processing, and sorting. *ShareClustering* shows consistent performance across the larger Hadoop setup.

3) Impact of k on Shared Execution. Fig. 4(a) shows results for workloads (x-axis) with two tasks but with varying values of k (128GB Cosmo-Gas dataset). The first 4 tasks all have primary task with k=40, but with increasing values of k for secondary tasks. This lets us zoom into the impact of k on the map output size. The *per mapper output* in *ShareScan* increases with the increasing values of k i.e., number of tuples is at most 100, 120, 140, 240, resp. For *ShareClustering*, this value is at most 40 for all 4 workloads. For the last 4 tasks with secondary task's k=200 and varying k for primary task, the cluster information for 200 secondary clusters are piggybacked across 40, 60, 80, and 100 clusters, resp. Experiments show that our algorithms perform well even with large values of k, with 30-35% performance gain over *NoShare*. Fig. 4(b) shows additional results with increasing number of tasks and larger k values. The benefit of *ShareClustering* becomes clear with the last workload consisting of 5 tasks with k=(40,60,80,100,200), with almost 23% performance gain over *ShareScan*.

4) Increasing Data Size. Fig. 5(a) shows a scalability study of the shared execution algorithms with Cosmo-Gas simulation datasets with sizes 20GB and 126GB, resp, (with 20-node Hadoop). While the smaller dataset had approximately 147 million data points, the larger dataset had about 900 million data points. Across data sizes, *ShareClustering* showed an increased performance improvement with increasing number of clustering tasks. *ShareClustering* maintains a 8-13% performance gains over *ShareScan* for k=(2,4), which increases to 12-22% for the last workload k=(2,4,6,8). Additional experiments using



(a) (b)

Figure 5: Scalability Study with (a) Increasing size of data (Cosmo-Gas, 20-nodes) (b) Varying number of clustering attributes for tasks k=(4,4,4) (Cosmo-Dark, 10-nodes)

larger synthetic benchmark dataset, TPC-H, showed corroborative evidence of scalability of *ShareClustering* (omitted due to lack of space).

5) Varying Number of Clustering Attributes. Clustering on different subsets of attributes provide different perspectives on the same data. Fig. 5(b) shows evaluation results with increasing number of clustering attributes from 2 to 6, with time measured for completing a set of three clustering tasks, each with $k = 4$ but different initial centroids (*Cosmo-Dark*, 10-node Hadoop setup). As expected, all three algorithms have the least response time for the first set with smallest number of attributes. Increase in number of clustering attributes, impacts the size of map output for all three algorithms. Experiments show the benefit of sharing *map* output and *reduce* processing for clustering over large number of attributes (high multi-dimensionality). *ShareClustering* starts with 6% gain over *ShareScan* for 2 attributes, and increases to 13% for 6 attributes.

4.2. Impact Analysis: Results and Discussion

1) Benefit of Collaborative Cluster Assignments. The impact of the *collaborative cluster assignment* phase for the workloads in Fig. 4(a) is shown in Fig. 6. For each workload, we show the number of distance calculations per data point for the primary and secondary tasks, with and without the collaborative cluster assignment. On an average we observe that *collaborativeAssign* is able to reduce 65-80% of secondary task distance calculations using the primary task centroid information. From the last four workloads, we observe that a higher value of k in primary task can help prune more secondary centroids. However, the total distance calculations per data point is also affected by the primary task's k (additional details in Table 3). Though MapReduce costs are dominated by scan, IO, and communication between different nodes, such algorithm-specific strategies can be used to further reduce the clustering times. Analysis of collaborative cluster assignment for workloads with increasing tasks (Fig. 4(b)) also showed up to 65% reductions in distance calculations across secondary tasks. Though we considered eliminating secondary centroids based on primary-secondary centroid distances, it may be possible to exploit distances between secondary task centroids to further reduce distance calculations.

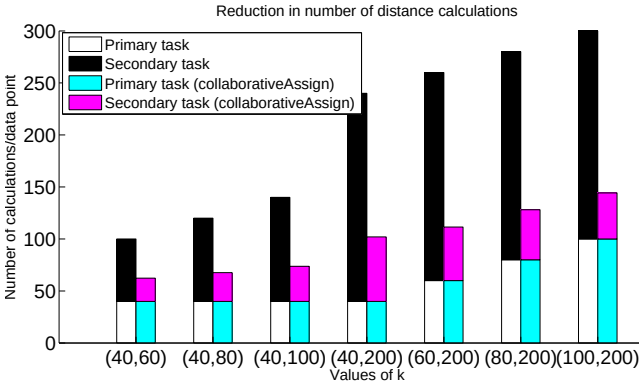
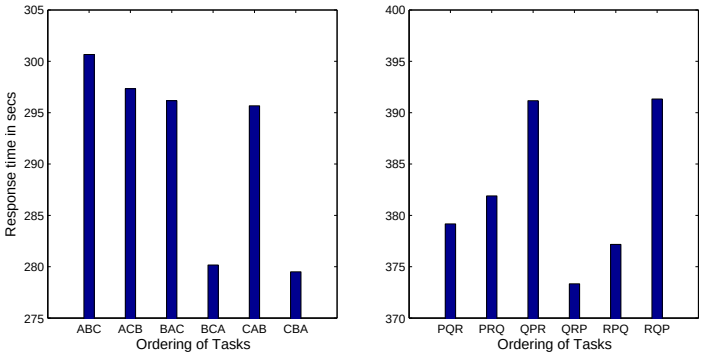


Figure 6: Benefit of collaborative cluster assignments (Cosmo-Gas, 24-nodes)

Table 1: Shared execution of clustering Tasks with k-means++ (execution time in seconds)

Approach	k**=(2,4)**	k**=(2,8)**
NoShare (random)	3936	4700
NoShare (k-means++)	3097	3884
ShareClustering (k-means++)	2628	3411



(a) Same k (b) Varying k

Figure 7: Selection of primary and leading secondary tasks (Cosmo-Gas, 20-nodes)

2) Choice of Primary and Leading Secondary Tasks. In the case of workloads with same set of clustering attributes, experiments showed that *ShareClustering* performs consistently, independent of the choice of the primary task. In this section, we consider tasks with different clustering attributes. Consider a workload with three tasks (all k=4): Task A, Task B, and Task C with 4, 6, and 12 clustering attributes, resp. Fig. 7(a) shows the response time for *ShareClustering* with different choice of primary and leading secondary task (Cosmo-Gas, 20-node Hadoop setup). Notation ABC denotes that task A is the primary task and task B is the leading secondary task. Recall that the entry for the leading secondary task includes the superset of all clustering attributes. Experiments show that tasks with clustering attributes with maximum overlap with the superset, should be chosen as the primary and leading secondary tasks, e.g., 12 attributes of task C overlap with the superset of attributes, and task B has the second maximum overlap. Orderings BCA and CBA perform the best since the clustering attributes of a primary task are implicitly encoded as part of the leading secondary task.

For workloads with different values of k, selection of the primary and leading secondary task depends on the value of k and number of clustering attributes. Consider a workload with three tasks: P (k=40), Q (k=10), and R (k=20) with 4, 6, and 12 clustering attributes, resp. Average execution times for each iteration of the different orderings are represented in Fig. 7(b) (Cosmo-Gas, 20-node Hadoop setup). If a task with high k is chosen as a leading

secondary task, recording the superset of attributes with this entry adds overhead to the intermediate writes and network transfer costs. Hence, we choose a task with maximum overlap with the superset of attributes, but as low k value as possible. Results show that the ordering QRP achieves the best performance, since task R has the maximum overlap with the superset (12 attributes) and has a low k, when compared to task P. Also, task Q with low k performs well as the primary task.

3) Benefit of Shared Execution with k-means++. We performed additional experiments to study the impact of shared execution strategies with the use of another optimization technique, i.e., *k-means++* (Arthur and Vassilvitskii, 2007) algorithm for initialization of appropriate centroids to improve clustering quality. Table 1 shows a comparison of execution times for *NoShare* and *ShareClustering* with randomly sampled centroids as well as centroids chosen using the *k-means++* algorithm. The reported execution time is the total execution time till all tasks converge (maximum iteration = 10, convergence threshold = 0.01). A subset of the data with ∼9 million data points were input to the *k-means++* algorithm. For the task with k=2, the *k-means++* centroids reduced the required number of Lloyd's iterations from 6 to 2. For individual tasks with k=4 and k=8, not much reduction was achieved. In general, while techniques such as *k-means++* reduce the required number of iterations per task, the proposed shared execution strategies can be applied across tasks to further enhance the performance improvement.

5. Related Work

Several optimization techniques have been proposed to reduce the length of MapReduce workflows (Afrati and Ullman, 2010; Abouzeid et al., 2009; Lee et al., 2011) and share scans and computations (Nykiel et al., 2010; Wang et al., 2011; Bu et al., 2010) to reduce the IO and network transfer costs. As per MRShare (Nykiel et al., 2010), two tasks can share map output if they have overlapping intermediate key-value pairs, which is applicable only when two clustering tasks have common clusterIds and data points are assigned to the same clusterId across tasks. However, *ShareClustering* can be applied to tasks with different map output keys (different clusterIds) and values (subset of clustering attributes).

Data clustering problems have been objects of study for many years by data management and data mining researchers (Zhang et al., 1996). Among the various algorithms proposed for data clustering, k-means is by far the most used algorithm. Although we have described our technique using Lloyd's algorithm as the baseline, similar ideas can be applied to other clustering techniques. Algorithms such as k-means++ (Arthur and Vassilvitskii, 2007) or its parallel version (Bahmani et al., 2012) that improve clusterings by choosing appropriate initial centroids, can be used along with *ShareClustering* (as shown in Section 4.2) to reduce the overall clustering time across multiple clustering tasks. Lv et al. (2010) proposed a parallel k-means algorithm for clustering remote sensing images using Hadoop. Apache Mahout[6] is a library of machine learning algorithms for data clustering, classification, and collaborative filtering on Hadoop. Mahout's implementation of k-means is similar to *NoShare*, and uses a mapper/combiner/reducer/driver flow to execute the k-means algorithm. Ene et al. (2011) propose approximation algorithms for k-center and k-median problems, that execute in constant number of MapReduce cycles, along with

6. http://mahout.apache.org/

an iterative sampling approach to reduce the size of input to the clustering algorithm. The authors observed that the approximation did not work well for k-center algorithm due to its sensitivity to sampling. Such sample-based clustering techniques can be integrated during shared execution of tasks using our approach (possibly on a subset of attributes). Further, since our algorithms scale well with increasing number of clustering tasks, it is a promising direction to pursue when considering ensemble clustering and multi-clustering solutions.

There has been a lot of work on multi-query optimization, such as exploiting common subexpressions (Zhou et al., 2007) in a set of relational queries to optimize query processing. Shareable sub-expressions are determined from queries involving the same database table and a transformation-based optimizer is used to rewrite queries in an optimized manner. Our approach of sharing map and reduce processing, as well as map output, is generic and can be extended to mixed workloads involving both relational (e.g., GROUP BY) and non-relational (e.g.,clustering) operations.

6. Conclusion and Future work

In this paper, we considered sharing opportunities while executing large scale clustering tasks. Specifically, we proposed algorithms that enable sharing of *assignment* and *recalculation* steps, while executing multiple k-means clustering tasks in parallel. Empirical evaluation using real-world datasets show that the algorithms perform consistently with varying values of k, clustering attributes, initial centroids, and scale well with increasing number of clustering tasks. This is especially important for scenarios that involve clustering of large datasets with unknown characteristics, requiring multiple iterations of trial and error with different clustering criteria. In future work, we will explore opportunities of shared execution across relational and non-relational tasks.

Acknowledgements: We thank Dr. Prasan Roy for discussions during initial stages of the work. Simulation Astro was graciously supplied by Tom Quinn and Fabio Governato from Department of Astronomy at University of Washington.

References

Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An Architectural Hybrid of MapReduce and DBMS Technologies for Analytical Workloads. *VLDB*, 2009.

Foto N. Afrati and Jeffrey D. Ullman. Optimizing Joins in a MapReduce Environment. In *EDBT*, 2010.

Charu C Aggarwal and Chandan K Reddy. *Data Clustering: Algorithms and Applications*. CRC Press, 2013.

David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proc. of the 18th annual ACM-SIAM symposium on Discrete algorithms*, 2007.

Bahman Bahmani, Benjamin Moseley, Andrea Vattani, Ravi Kumar, and Sergei Vassilvitskii. Scalable k-means++. In *VLDB*, 2012.

P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquini. Incoop: Mapreduce for incremental computations. *SOCC*, 2011.

A. Bialecki, M. Cafarella, D. Cutting, and O. O Malley. Hadoop: A Framework for Running Applications on Large Clusters Built of Commodity Hardware.

Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D. Ernst. Haloop: efficient iterative data processing on large clusters. *VLDB Endow.*, 3, 2010.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

Alina Ene, Sungjin Im, and Benjamin Moseley. Fast clustering using mapreduce. In *SIGKDD*, 2011.

M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3), 2007.

Rubao Lee, Tian Luo, Yin Huai, Fusheng Wang, Yongqiang He, and Xiaodong Zhang. Ysmart: Yet another sql-to-mapreduce translator. In *ICDCS*, 2011.

S Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2), 1982.

Zhenhua Lv, Yingjie Hu, Haidong Zhong, Jianping Wu, Bo Li, and Hui Zhao. Parallel k-means clustering of remote sensing images based on mapreduce. In *Web Information Systems and Mining*. 2010.

Tomasz Nykiel, Assaf Michalis, Chaitanya Mishra, George Kollios, and Nick Koudas. Mrshare: Sharing across multiple queries in mapreduce. *VLDB*, 2010.

R. Pais and C. Rong. K-means Clustering in the Cloud – A Mahout Test. In *WAINA*, 2011.

Wei Wang, Svein Knapskog, and Sylvain Gomault. Attribute normalization in network intrusion detection. *ISPAN*, 2009.

Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *SOCC*, 2011.

Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. In *USENIX HotCloud*, 2010.

T Zhang, R Ramakrishnan, and M Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD record*, 1996.

Jingren Zhou, Per-Ake Larson, Johann-Christoph Freytag, and Wolfgang Lehner. Efficient exploitation of similar subexpressions for query processing. In *SIGMOD*, 2007.

Appendix A. Analysis of Algorithms

For m k-means clustering tasks, let k_i be the value of k for the i^{th} task, and n be number of data points. Assume clustering on d dimensions each of size δ, i.e., size of a data point is $d\delta$. Both *map* key and count are integers with size ξ. Table 2 summarizes assertions regarding costs of the two algorithms. For the post-combiner analysis of *ShareClustering*, we assume that on average, data points corr. to each primary task key (k_1) will be distributed across $\lceil \frac{k_j}{k_1} + 1 \rceil$ non-primary task clusters. Based on the above assertions, we summarize the cost of MapReduce-based execution of *ShareScan* and *ShareClustering*. HDFS read / write costs are similar for both algorithms and hence not considered for this comparison.

Table 2: *Assertions regarding costs of algorithms*

Assertion	ShareScan	ShareClustering
Pre-Combiner		
A1: No. of map output keys	$m.n$	n
A2: Size of {*key*, *value*} pair	$(d\delta + 2\xi)$	$d\delta + (m-1)\ 2\xi + \xi$
Intermediate data size	$mn(d\delta + 2\xi)$	$n(d\delta + (m-1)2\xi + \xi)$
Post-Combiner		
A3: No. of keys per mapper	$\sum_i k_i$	k_1
A4: Size of {*key*, *value*} pair	$(d\delta + 2\xi)$	$(\xi + \sum_{j=2}^{m} \lceil \frac{k_j}{k_1} + 1 \rceil (d\delta + 2\xi))$
Intermediate data size	$\sum_i k_i(d\delta + 2\xi)$	$k_1(\xi + \sum_{j=2}^{m} \lceil \frac{k_j}{k_1} + 1 \rceil (d\delta + 2\xi))$

Table 3: *Distance calculations per data point in ShareClustering (Cosmo-Gas, 24-nodes)*

Values of k	Prim. Task	Sec. Tasks	α Value	Savings (%)	Values of k	Prim. Task	Sec. Tasks	α Value	Savings (%)
(40,60)	40	60	0.66	39.69	(80,200)	80	200	0.82	58.88
(40,80)	40	80	0.69	46.66	(100,200)	100	200	0.85	56.67
(40,100)	40	100	0.72	51.52	(40,60,80)	40	140	0.68	53.09
(40,200)	40	200	0.74	61.42	(40,60,80,100)	40	240	0.68	58.43
(60,200)	60	200	0.79	61.10	(40,60,80,100,200)	40	440	0.69	63.86

Map processing: *ShareScan* computes distances wrt. all centroids in all m tasks, i.e., $\sum_{i=1}^{m} k_i$ distances per data point; *ShareClustering* computes: k_1 + (1-α)$\sum_{i=2}^{m} k_i$, where the *elimination factor* α denotes the portion of secondary centroids that need not be considered while calculating distances with a data point. Table 3 represents the number of distance calculations per data point for the primary and secondary tasks, the α value for *ShareClustering*, and the percentage savings in number of computations when clustering multiple tasks using *ShareClustering* as opposed to *ShareScan*.

Intermediate data sort and shuffle: The local disk IO cost for sort and shuffle is a function of *log* of intermediate data sizes (refer to assertion $A2$).It can be seen that this cost will be much less for *ShareClustering* when compared to *ShareScan*.

Network data transfer: For tasks with the same set of clustering attributes, the intermediate data size per mapper is $\sum_i k_i(d\delta + 2\xi)$ for *ShareScan*. For *ShareClustering*, this size can be approximated as $\sum_i k_i(d\delta + 2\xi) - k_1(d\delta + \xi)$, i.e., the reduction equals *number of mappers* multiplied by $k_1(d\delta + \xi)$. For cases with varying clustering attributes, more savings in network data transfer can be achieved by maximizing the factor $k_1(d\delta + \xi)$, i.e., selecting a primary task whose attributes have maximum overlap with the superset of attributes.

Though *ShareClustering* requires additional processing to compute final centroids from partial ones, the reduced number of distance calculations and intermediate map output due to shared cluster information, achieve additional savings over *ShareScan*.