Mux-Kmeans: Multiplex Kmeans for Clustering Large-scale Data Set

Chen Li, Yanfeng Zhang, Minghai Jiao, Ge Yu

Northeastern University, China
lichen640819@gmail.com; {zhangyf, mhjiao}@cc.neu.edu.cn; yuge@mail.neu.edu.cn

ABSTRACT

Kmeans clustering algorithm is widely used in a number of scientific applications due to its simple iterative nature and ease of implementation. The quality of clustering result highly depends on the selection of initial centroids. Different selections of initial centroids result in different clustering results. In practice, people run a series of Kmeans processes with multiple initial centroid groups serially and return the best clustering result among them. However, in the era of big data, a Kmeans process is implemented on MapReduce to scale to large data sets. Even a single Kmeans process on MapReduce requires considerable long runtime. This paper proposes Mux-Kmeans. Rather than executing multiple Kmeans processes serially, Mux-Kmeans launches these Kmeans processes concurrently with multiple centroid groups. In each iteration, Mux-Kmeans (i) evaluates these Kmeans processes, (ii) prunes the low-quality Kmeans processes, and (iii) incubates new Kmeans processes. After a certain number of iterations, it finally obtains the best among these local optimal results. We implement Mux-Kmeans on MapReduce and evaluate it on Amazon EC2. The experimental results show that starting from the same initial centroid groups, the clustering result of Mux-Kmeans is always non-worse than the best of a series of Kmeans processes. Mux-Kmeans also saves elapsed time than serial multiple Kmeans processes.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*

General Terms

Algorithms, Design, Performance

Keywords

Kmeans, MapReduce, clustering algorithm, big data

1. INTRODUCTION

Recent advances in data acquisition, storage, and networking technology have created huge collections of high-volume, high-dimensional data. Massive amounts of data are generated through online services like Facebook user activities, Twitter tweets, Amazon purchase records, Gmail e-mails, Flickr photos, and Youtube videos. Clustering is one of the principal tools to efficiently organize such large amounts of data and to make sense of these data. It has been used in a multitude of applications such as social network analysis, pattern recognition and market analysis [15].

Kmeans (Lloyd's algorithm [12]) is the most widely used clustering algorithm due to its simplicity and linearity. The Kmeans algorithm groups the data points into clusters in a simple and iterative manner: (1) Randomly select K points from the points set that is being clustered and take them as initial centroids for K clusters, (2) Assign each point to the cluster which has the nearest centroid, (3) When all points have been assigned, recalculate all K centroids by "averaging" the points belonging to the same cluster. (4) Repeat steps (2) and (3) until the centroids' difference between two recent iterations is smaller than a specific threshold.

However, Kmeans suffers from the drawback that it converges to different local minima with different selections of initial centroids. Consequently, the result quality greatly depends on the selection of the initial centroids. In practice, users choose multiple groups of initial centroids to perform multiple Kmeans processes serially and pick the best cluster result when all Kmeans runs are done. (Note that, some initial centroids selection techniques [16] could be used to improve result quality.) In this approach, the multiple Kmeans processes are handled one after another, and each Kmeans process is composed of a few serial iterations. The latter iteration cannot be started until the former one is finished. Further, the latter Kmeans process cannot be started until the former one is finished. This serial execution approach involves a large number of global synchronizations (between iterations and between Kmeans processes), which incurs significant performance penalties. Even worse, due to high computational complexity and multiple iterations required to execute, the standard Kmeans algorithm is slow for large data sets, which means serial multiple Kmeans processes lead to multifold increase in runtime. Even though a number of Kmeans variants in distributed environment or in a cloud are proposed recently [6, 19, 20, 21], they still cannot meet the latency needs of many applications.

This paper proposes Mux-Kmeans. Rather than handling multiple Kmeans processes serially, Mux-Kmeans starts multiple Kmeans processes with different centroid groups concurrently. In every Mux-Kmeans iteration, all Kmeans processes' intermediate results are evaluated. Based on the evaluated quality of these results, we leave the promising Kmeans processes for next iteration and prune the hopeless ones. Further, we utilize the survived Kmeans processes to incubate new Kmeans processes. The new born processes are represented by new born centroid groups. Both the new born centroid groups and the survived groups will take part in the next Mux-Kmeans iteration. By this way, the computation resources spent on hopeless Kmeans attempts are rescued. In addition, new generated Kmeans attempts can expand searching scope to avoid getting stuck in few local optima.

We implement Mux-Kmeans on Hadoop MapReduce [2]. We run a series of experiments to evaluate Mux-Kmeans with three real data sets on Amazon EC2 [1]. The results show that Mux-Kmeans outperforms serial multiple Kmeans processes on both clustering quality and runtime.

The rest of the paper is organized as follows. First, we review the background in Section 2. Then we describe Mux-Kmeans in Section 3. We outline Mux-Kmeans's implementation details in Section 4. We present the experimental results in Section 5. Finally, we give an overview of the related works in Section 6 and conclude in Section 7.

2. PRELIMINARY AND BACKGROUND

This section reviews the background of Kmeans and its MapReduce implementation. Let $X = (x_1, x_2, \dots, x_n)$ be the set of n points (vectors) and each x_i is a point (vector). The Kmeans algorithm partitions these points into K disjoint clusters: $C_1, C_2, \dots, C_K$, where $\forall i, j, C_i \cap C_j = \varnothing, i, j \in 1, \dots, K$. There is one centroid for each cluster, so there are K centroids for a points set. Suppose the clustering result is KM, which represents all points are assigned to their nearest centroids. Kmeans aims at minimizing an objective function, in this case a squared error function:

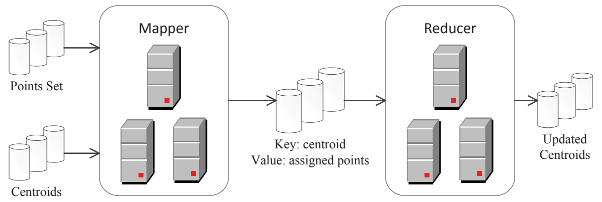
$$F(KM) = \sum_{j=1}^{K} \sum_{x_i \in C_j} ||x_i - c_j||^2, \qquad (1)$$

where x_i is assigned to jth cluster whose centroid is c_j. $||x_i - c_j||^2$ is the distance metric between point x_i and centroid c_j. $F(KM)$ is defined as Total Within-Cluster Variation (TWCV) and is used as the criteria to judge Kmeans' clustering quality.

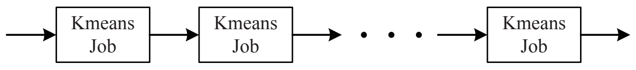
To scale Kmeans to large-scale points set, we can exploit MapReduce [2] to implement the Kmeans algorithm [22]. As shown in Fig. 1a, the MapReduce program takes the points set and the initial cluster centroids as input. Each iteration of Kmeans is implemented with a MapReduce job, and the iterative Kmeans is then implemented with a series of MapReduce jobs, as showed in Fig. 1b. The map and reduce operations are described as follows.

Map: For each point, given all K cluster centroids (mappers cached data), compute the distance from the point to all centroids, and output the closest cluster id (output key) and the point (output value).

Reduce: For each cluster id (input key), given all the points (input values) assigned to it, update the cluster centroid by "averaging" all the assigned points, and output the cluster id (output key) and the updated centroid (output value).



(a) A Kmeans iteration in MapReduce.



(b) Multiple Iterative Kmeans processes.

Figure 1: Kmeans on MapReduce

The updated centroids are used as the mapper cached data input in next MapReduce job.

The Kmeans algorithm suffers from the drawback that it converges to a local optimum result. This means that the obtained result quality highly depends on the selection of the initial K centroids. There are some efforts target on choosing a better initial centroids set by preprocessing [13]. Moreover, users start Kmeans computation from multiple initial centroids groups and run multiple Kmeans processes. Then, the best of these local optimal results is returned as the final result. Note that, some certain initial centroids selection approaches can be used to choose the multiple start points. However, these multiple Kmeans processes lead to considerable long runtime, especially for large-scale data sets. In the next section, we propose Mux-Kmeans to address this issue.

3. MUX-KMEANS

This section proposes Mux-Kmeans. Rather than performing a series of Kmeans processes serially, Mux-Kmeans performs multiple Kmeans processes concurrently as shown in Fig. 2. Suppose s initial centroid groups $(g_1, g_2, \dots, g_s)$ are selected and each centroid group g_i is composed of k initial centroids $g_i = \{c_{i,1}, c_{i,2}, \dots, c_{i,k}\}$. A Kmeans process KM_i starts with centroid group g_i and outputs the intermediate clustering result as well as the updated centroid group g'_i. These intermediate results are then *evaluated* in terms of their TWCV values $F(KM_i)$. The higher a TWCV is, the lower quality of a clustering reslut is. The Kmeans processes whose intermediate cluster results have relatively high TWCV values are *pruned* (Section 3.1). The centroid groups representing the survived Kmeans processes are adjusted in a *permute* step (Section 3.2) followed by being used to *incubate* new centroid groups (Section 3.3). The survived groups and the new born groups are used for Kmeans computation in next iteration. We make an analysis on Mux-Kmeans' computation complexity in Section 3.4. In Table 1, we summarize the related notations.

The intuition behind Mux-Kmeans is as follows. Different Kmeans processes with different initial centroid groups will converge to different clustering results (*i.e.*, local optima). These clustering results have different qualities, which can be evaluated by TWCV, but only the best one is returned. It is a waste of time and resources to launch a Kmeans processing attempt if its result is not finally returned. Based on this observation, Mux-Kmeans investigates all Kmeans

Table 1: Notations and their Meanings

Natation	Meaning
n	Number of points in X
s	Number of centroid groups in G
k	Number of centroids in a centroid group
X	Set of points; $X = \{x_1, x_2, \dots, x_n\}$
G	Set of centroid groups; $G = \{g_1, g_2, \dots, g_s\}$
g_i	ith centroid group
$c_{i,l}$	lth centroid in g_i; $g_i = \{c_{i,1}, c_{i,2}, \dots, c_{i,k}\}$
x_i	ith point in points set
$C_{i,l}$	lth cluster whose centroid is $c_{i,l}$
KM_i	A Kmeans process with centroid group g_i

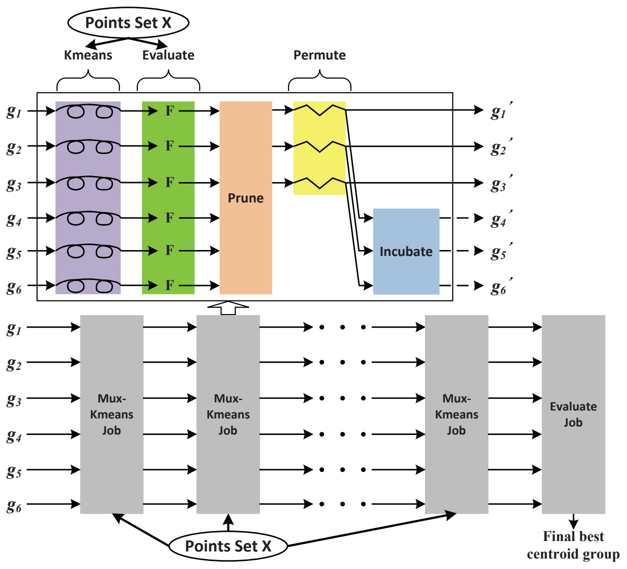


Figure 2: Mux-Kmeans process

processes' intermediate results and terminates the hopeless Kmeans processing attempts in the early stage to avoid useless computation. At meantime, in order to avoid getting stuck in a few local optima and expand the searching scale of better clustering result, our heuristic methods generate some new centroid groups with the high quality centroid groups that are survived from the prune operation. After that, new Kmeans processing attempts with the new born centroid groups are correspondingly created. By constantly repeating the prune step and incubate step, we can finally obtain a clustering result.

3.1 Kmeans, Evaluate and Prune

Mux-Kmeans concurrently starts s Kmeans processes with different initial centroid groups $(g_1, g_2, \dots, g_s)$. When all K-means processes finish one iteration, there are s intermediate clustering results $(KM_1, KM_2, \dots, KM_s)$ and s updated centroid groups $(g_1', g_2', \dots, g_s')$. We use Equation (1) to compute all intermediate results' TWCV values and then evaluate all Kmeans processes' clustering quality. As [9] says, the TWCV value is a simple and intuitive measure for evaluation, so we use every Kmeans process' TWCV value-$F(KM_i)$ to judge their clustering quality. As we have mentioned, an intermediate result has high TWCV value means that its corresponding Kmeans process has low clustering quality. Since we use different centroid groups to represent different Kmeans processes, a centroid group is relatively bad if its corresponding Kmeans process has low clustering quality. Thus, we keep the centroid groups whose corresponding Kmeans processes have high clustering quality and prune the groups with low clustering quality. We can choose a tunable parameter to decide how many Kmeans processes with high quality to continue clustering and how many Kmeans processes are pruned. In this paper, for ease of exposition, we assume 50% centroid groups survived, i.e., $s/2$ centroid groups survived.

3.2 Permute

In the following incubate step it is required to identify similar centroids who represent same cluster between different centroid groups. As we mentioned in Section 2, in every centroid group, K centroids represent K clusters. Each cluster has different amounts of points in it and there is no overlap between any two clusters. However, if we compare two different centroid groups g_i and g_j, we can find that a cluster, whose centroid is $c_{i,l}$, is more likely having an overlap with a cluster whose centroid is $c_{j,h}$. If a cluster represented by $c_{i,l}$ has the biggest overlap with a cluster represented by $c_{j,h}$ than any other centroids in g_j, we call $c_{i,l}$ and $c_{j,h}$ are similar centroids. The similar centroids can be defined as:

For $c_{i,l}$, its most similar centroid in g_j is $c_{j,h}$, where the centroid index h satisfies

$$h = \arg_{h\in\{1,\dots,k\}} \max |C_{i,l} \cap C_{j,h}| \quad (2)$$

The goal of Mux-Kmeans is finding k appropriate centroids to divide the points set. It is meaningful to use all similar centroids to help us discover better centroids for each cluster. Based on this consideration, we adjust the centroid sequence of all centroid groups so that the similar centroids have the same location in each group. If $c_{j,h}$ is similar to $c_{i,l}$, it must has the shortest distance to $c_{i,l}$ than any other centroids in g_j. In this case, we can rearrange the centroid sequence in g_i and g_j based on the distances between different centroids. Permute step adjusts the centroid sequence of all centroid groups to guarantee that for any two centroid groups g_i and g_j, centroid $c_{i,l}$ and centroid $c_{j,l}$ are similar centroids. Basically, we calculate the distances from any centroid in g_i to any centroid in g_j. Then we match two centroids $c_{i,l}$ and $c_{j,h}$ if the distance from $c_{i,l}$ to $c_{j,h}$ is the smallest among the distances from $c_{i,l}$ to all centroids in g_j. After that, we swap $c_{j,h}$'s and $c_{j,l}$'s locations so that the similar centroid has the same location in g_j as $c_{i,l}$ has in g_i. For the remaining centroids in two groups, the Permute step continues the above operations until the centroid sequence in both groups are unified. Facing with multiple centroid groups, we can take one group as standard group. Then we take its centroid sequence as the datum one. We adjust other groups' centroid sequence to match the datum sequence. The algorithm is showed as Algorithm. 1.

3.3 Incubate

On one hand the prune step eliminates the hopeless K-means processes, on the other hand the incubate step generates new centroid groups to expand the searching scale for new clustering results. Instead of blindly generating new centroid groups, we propose two heuristics to incubate new groups based on the survived centroid groups: 1) *Random Search within a Definite Scope* (**RSDS**) and 2) *Average of*

Algorithm 1 Permute Algorithm

```
 1: procedure PERMUTE($g_1, g_2, \dots, g_{s/2}$)
 2:     for all $c_{1,l} \in g_1, l \in \{1, \dots, k\}$ do
 3:         for all $c_{j,h} \in g_j$,
 4:             $j \in \{2, \dots, s/2\}, h \in \{l, \dots, k\}$ do
 5:             $c_{i,j} \leftarrow$ find the nearest centroid to $c_{1,l}$
 6:             put $c_{i,j}$ to the lth position in $g_j$
 7:         end for
 8:     end for
 9:     return $\{g_1, g_2, \dots, g_{s/2}\}$
10:     ▷ all centroid groups have the same centroid order
11: end procedure
```

Dissimilar Group Pairs (**ADGP**).

3.3.1 RSDS

The points in a cluster are not uniformly distributed. There are dense parts and sparse parts in a cluster. Points in dense part are close to each other, while the points in sparse part are far from each other. With Kmeans operation, the centroid of a cluster is more likely close to the cluster's dense part. As we mentioned in Section 3.2, among all centroid groups, there are multiple similar centroids representing each cluster. In this case, for every cluster, there are multiple centroid candidates to represent the cluster. If a centroid locates near the cluster's dense part, it is more likely to be the cluster's real centroid. However, if a centroid locates near the cluster's sparse part, it will have low possibility to be the cluster's real centroid. Based on this consideration, we can assign different weight values to a cluster's all centroids. A centroid near the dense part has high weight value while a centroid near the sparse part has low weight value. The weighted average of all centroids is likely more closer to the cluster's real centroid when compared with all existing centroids. We can use the weighted average to incubate the cluster's new centroids.

We use Fig. 3 to illustrate RSDS's method. For a cluster C, there are two centroids c_1 and c_2 in it. c_1 locates in the sparse part and c_2 locates in the dense part. The green points are assigned to c_1. The blue points are assigned to c_2. The points in purple color are the overlap part between c_1' points and c_2' points. We assign different weighted values to c_1 and c_2 based on the amounts of points assigned to them. Then we get the weighted average of c_1 and c_2. We identify it as c' in Fig. 3. We use c' as searching center. We calculate the distances from c' to c_1 and c_2 and take the minimum distance as searching radius. Then we can define a searching scope based on the searching center and searching radius. We randomly choose a point in the scope and take it as the new centroid of cluster C.

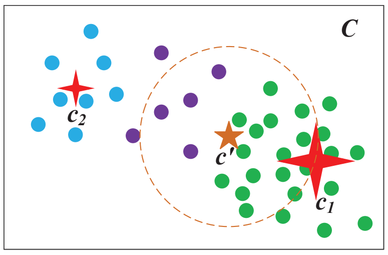


Figure 3: Random Search within a Definite Scope

After the Permute step, there are $s/2$ centroid groups: $g_1, g_2, \dots, g_{s/2}$. The centroids with same indexes $c_{1,l}, c_{2,l}, \dots, c_{s/2,l}, l \in \{1, \dots, k\}$ represent a similar cluster. We call $\{c_{1,l}, c_{2,l}, \dots, c_{s/2,l}\}$ a matched set, and there are k such sets. For set $\{c_{1,l}, c_{2,l}, \dots, c_{s/2,l}\}$, we generate a searching center c'_l:

$$c'_l = \frac{\sum_{i=1}^{s/2} c_{i,l} \cdot |C_{i,l}|}{\sum_{i=1}^{s/2} |C_{i,l}|}, l = 1, 2, \dots, k \tag{3}$$

where $|C_{i,l}|$ is the amount of points assigned to $c_{i,l}$. Thus, we can obtain k searching centers $(c'_1, c'_2, \dots, c'_k)$. As mentioned above, we define a searching scope by using center c'_l and radius d_l:

$$d_l = \min_{i \in \{1, \dots, s/2\}} \{d(c'_l, c_{i,l})\} \tag{4}$$

Then we can get k searching scopes. We randomly select a point in every scope and combine them to generate a new centroid group. For the demand of incubating $s/2$ new centroid groups, we repeat the above process for $s/2$ times. The algorithm of RSDS is showed in Algorithm. 2.

Algorithm 2 RSDS Algorithm

```
 1: procedure RSDS($g_1, g_2, \dots, g_{s/2}$)
 2:     for j = 1 to s/2 do
 3:         for l = 1 to k do
 4:             for i=1 to s/2 do
 5:                 $c'_l, d_l \leftarrow$ calculate searching center $c'_l$ and
                    radius $d_l$ using equation (3) and (4)
 6:             for
 7:             $c_l \leftarrow$ random select a new centroid around $c'_l$
                in radius $d_l$
 8:         end for
 9:         assemble all new generated centroids
10:     end for
11:     return all $s/2$ new generated centroid groups
12: end procedure
```

3.3.2 ADGP

Since Kmeans uses the distances between centroids and all points to divide the whole points set, it is more likely that a cluster's centroid locates in the central area of the cluster. However, as we start clustering with multiple randomly selected centroid groups, the centroids representing the same cluster are more likely locating near the cluster's boundary, rather than near the cluster's central area. In this case, if we generate a cluster's new centroid, we would like the new centroid near the central area of the cluster. In Fig. 4, there is a cluster with multiple centroids. Its boundary is indicated as a circle. Its centroids, which are indicated by green crosses, are all near the cluster's boundary. For any centroid, it can find a centroid who has the maximum distance to it among all centroids in the cluster. The center of the two centroids locates in/near the central area of the cluster. On the contrary, if a centroid find a centroid who has the minimum distance to it, the center of the two centroids will locate far from the cluster's central area.

As we presented in Section 3.2, we call two centroids are dissimilar centroids if the distance between them is large. So we can use the center of two dissimilar centroids to incubate a new centroid. In order to incubate a new centroid group, we can use two dissimilar groups to produce a new group.

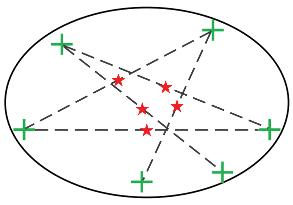


Figure 4: Average of Dissimilar Group Pairs

We calculate two centroid groups' similarity degree by the following equaton:

$$sim(g_i, g_j) = \frac{1}{\sum_{l=1}^{k} d(c_{i,l}, c_{j,l})}, i, j \in \{1, \dots, s/2\} \quad (5)$$

g_i and g_j will be more similar if they have a higher value of $sim(g_i, g_j)$. If we want to use g_i incubating a new centroid group g_m, we find the most dissimilar centroid group to g_i, which we assume it's g_j. We generate every centroid of g_m by the following equation:

$$c_{m,l} = \frac{c_{i,l} + c_{j,l}}{2}, l = 1, 2, \dots, k \quad (6)$$

The algorithm of ADGP is showed as follows:

Algorithm 3 ADGP Algorithm

```
 1: procedure ADGP($g_1, g_2, \dots, g_{s/2}$)
 2:     for $g_i, i \in \{1, \dots, s/2\}$ do
 3:         for $g_j, j \in \{1, \dots, s/2\}$ do
 4:             if $g_j == g_i$ then
 5:                 continue
 6:             else
 7:                 $g_j \leftarrow$ find the most dissimilar group to $g_i$
 8:             end if
 9:         end for
10:         $g'_i \leftarrow$ incubate new centroid group using $g_i$ and $g_j$
11:     end for
12:     return all $s/2$ new generated centroid groups
13: end procedure
```

3.4 Analysis on Computational Complexity

In this part, we assume that there is a points set which has n points. We have s centroid groups to start Mux-Kmeans and each centroid group has k centroids, which means we divide n points into k clusters. Then, for each Mux-Kmeans iteration, the Kmeans operation's computational complexity is $O(nks)$. Then the Evaluate operation needs $O(nks)$ complexity to calculate all centroid groups' TWCV. The Prune step is simply sorting all centroid groups' TWCV, judging their clustering performance based on the sorted list and pruning half groups. In this case, the Prune's complexity is $O(s \log s)$. The Permute step focuses on all centroid groups and its computational complexity is $O(sk^2)$. In Incubate step, both ADGP and RSDS methods' computational complexity is $O(s^2k)$.

4. IMPLEMENTATION

As we have analyzed in Section 3.4, Kmeans and Evaluate steps' complexities are $O(nkN)$. When dealing with a large scale points set (big n), those two steps will consume large amounts of time. In this case, we use MapReduce to do Kmeans phase and Evaluate phase. We deploy Mux-Kmeans on MapReduce in order to scale to large points sets.

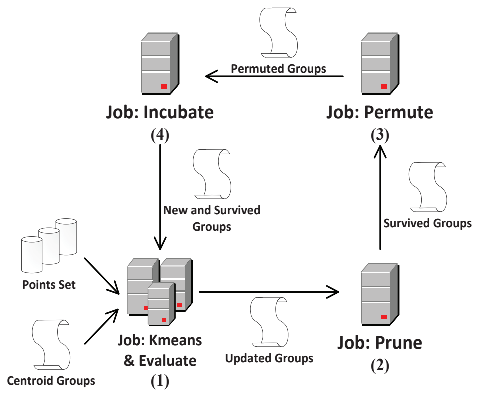


Figure 5: Mux-Kmeans Implementation

Algorithm 4 MapReduce Function

```
 1: procedure MAPPHASE($< i,\ x_i >$)
 2:     for $g_j, j \in \{1, \dots, s\}$ do
 3:         $l, d \leftarrow$ find the closest centroid $c_{j,l}$ to $x_i$
            and calculate the distance $d$ between them,
            $l \in \{1, \dots, k\}$
 4:     end for
 5:     output $< (j, l),\ (x_i, d) >$
 6: end procedure

 7: procedure REDUCEPHASE($< (j, l),\ ([x_i],\ [d]) >$)
 8:     $c_{j,l} \leftarrow$ update $c_{j,l}$
 9:     $WCV_{j,l} \leftarrow$ the sum of $[d]$
10:     output $< (j, l),\ (c_{j,l}, WCV_{j,l}) >$
11: end procedure
```

There are two kinds of data during Mux-Kmeans iterations: 1) the points set $X = \{x_1, x_2, \dots, x_n\}$ and 2) the centroid groups set $G = \{g_1, g_2, \dots, g_s\}$. In each centroid group, there are k centroids in it. Compared with the points set, the centroid groups set is small, which can be handled by a single machine, so we deploy the tasks, which only takes the centroid groups set into computation, on a single machine. The implementation framework of Mux-Kemans is showed in Fig. 5.

There are 4 jobs in one Mux-Kmeans iteration: 1) Kmeans & Evaluate, 2) Prune, 3) Permute and 4) Incubate.

1) The Kmeans & Evaluate job takes the responsibilities of Kmeans and Evaluate steps. Since this job takes the whole points set into calculation, we deploy it on MapReduce framework to process large scale points set. Since the MapReduce framework can concurrently handle all Kmeans processes, it will improve the operational efficiency than serial multiple Kmeans. In addition, during one iteration of serial multiple Kmeans, each Kmeans process needs to read the whole points set for once. In this case, n Kmeans in one iteration will have to read the points set for n times. Mux-Kmeans is different from that. During one Mux-Kmeans iteration, it only load the whole points set once and then all Kmeans processes share it. So Mux-Kmeans can save much time in loading the whole points set. The Map function and Reduce function are in Algorithm 4.

2) The Prune job only need ks values to process and we deploy it on a single machine. It calculates each centroid group's TWCV. Then it prunes $s/2$ centroid groups with low clustering quality and outputs the remaining $s/2$ centroid groups.

3) The Permute job is deployed on a single machine as it only process ks centroids. Based on standard centroid group, it rearranges all remaining centroid groups' centroid orders.

4) The Incubate job is deployed on a single machine. Using all permuted centroid groups, it generates $s/2$ new centroid groups based on ADGP or RSDS method. All new generated centroid groups and permuted centroid groups are sent to the Kmeans & Evaluate job of the next Mux-Kmeans iteration.

5. EXPERIMENTS

We evaluate the clustering performance of Mux-Kmeans with different data sets. Through all experiments, we find that Mux-Kmeans can achieve a better, or at least equivalent clustering quality when comparing with naive Kmeans. Mux-Kmeans also significantly reduces the time used by naive Kmeans when starting clustering process with multiple centroid groups.

All experiments are performed on a Amazon EC2 [1] cluster with 16 instances running Hadoop [2] 1.4.1. Each instance uses 2 ECUs and 1 CPU, 3.7 GiB of memory and 410 GiB storage. The network performance within the cluster is moderate.

We evaluate Mux-Kmeans using the following data sets.

Bio_train data set: This data set has been used in KDD Cup 2004. We remove each data point's first three inessential features and get a dataset with 145751 data points. Each point has 74 features.

Netflix data set: The Netflix data set records audience's scores for 17770 movies. However, a viewer can not score all movies. So some movies have a lot of scores while some have few scores. In this case, we convert the original Netflix data set. The converted data set contains 17770 data points and each point has 1000 scores.

Lastfm data set: The Lastfm data set records the whole listening habits for nearly 360000 users. Since the habits vary from different user, we convert the Lastfm data set and get a points set with 359330 data points. Each point has 40 features.

5.1 Clustering Quality

The first question that we ask is how well Mux-Kmeans clustering quality is.

As we have mentioned above, we use TWCV to judge both Kmeans and Mux-Kmeans' clustering quality. For all data sets, we set the number of centroid groups to 8 and the value of k to 6. Since we use two different strategies, ADGP and RSDS, to incubate new centroid groups, we show Mux-Kmeans clustering quality under those two different methods. The Kmeans' result is obtained from the centroid group that has the best clustering result among all initial centroid groups. Fig. 6 shows the variation trend of TWCV of different data sets when using Kmeans and Mux-Kmeans. The y-axis is the value of TWCV and the x-axis is the iteration. In all data sets, Kmeans and Mux-Kmeans using ADGP or RSDS show the same stable downtrend of TWCV. Moreover, when facing Lastfm dataset and Netflix dataset, Mux-Kmeans' downtrend is faster than Kmeans, no matter which incubate strategy Mux-Kmeans uses. When facing Bio_train data set, Mux-Kmeans' downtrend is about one iteration faster than Kmeans.

In addition, for Bio_train data set, we vary the number of centroid groups between 8 and 20 and the value of k between 6, 12 and 18. All results are showed in Fig. 7. Fig. 7 shows that no matter how k value and the number of centroid groups varies, Mux-Kmeans can keep a stable TWCV downtrend.

5.2 Elapsed Time

In this section, we further study the runtime of Mux-Kmeans with different incubate strategy. For this, we run Mux-Kmeans and Kmeans with different data sets. At the meantime, we vary the number of centroid groups and the value of k. Fig. 8 shows the results for Lastfm, Netflix and Bio_train data sets while the value of k varies between 6 and 12 and the number of centroid groups varies between 8, 12 and 20. We record one iteration's runtime of Mux-Kmeans and Kmeans. For Kmeans, we record the runtime of one centroid group processed by Kmeans. For Mux-Kmeans, we record the runtime of multiple centroid groups processed by Mux-Kmeans. Since Kmeans handles multiple centroid groups serially, the runtime of Kmeans handling multiple centroid groups T' is calculated as follows:

$$T' = s \cdot T \tag{7}$$

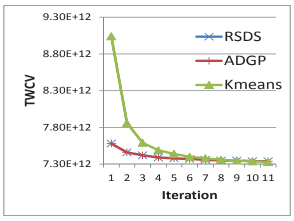
where s is the number of centroid groups and T is the runtime of Kmeans handling one centroid group.We use 'Kmeans' to represent the runtime of one centroid group handled by Kmeans. We use 'acc Kmeans' to illustrate the runtime of Kmeans handling multiple centroid groups serially.

As Fig. 8 shows, when processing multiple centroid groups, the runtimes of Mux-Kmeans using ADGP or RSDS are nearly the same. Although the runtime of naive Kmeans handling one centroid group is relatively little, naive Kmeans takes a lot of time to process multiple centroid groups. In add conditions, our experiments show that when processing multiple centroid groups, there can be at most 8X difference in runtime between Mux-Kmeans and naive Kmeans. This is due to naive Kmeans disposes all centroid groups serially while Mux-Kmeans deal with all centroid groups concurrently.

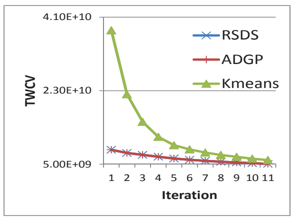
6. RELATED WORKS

A number of Kmeans variants are proposed to reduce the clustering time in a single machine environment. [7] proposes to exploit the triangle inequality between centroids to avoid the useless computations on the impossible cluster candidates for a point. [3, 10, 14] stores the points in a k-d tree and maintains a subset of candidate centroids for each point. The candidates for each point are pruned or filtered, as they propagate to the children, which avoids comparing each centroids with all the points. [18] proposes to only process those active points near cluster boundaries to improve both the efficiency and accuracy. All these efforts target on improving a single Kmeans process, which can be integrated in Mux-Kmeans.

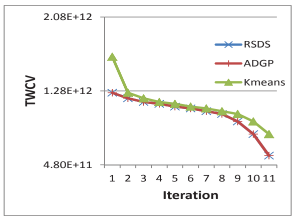
On the other hand, some researchers aim at choosing a set of representative initial centroids, expecting improving the clustering result quality. [4] proposes K-Means++, which spreads out the initial centers. It chooses the first center



(a) Lastfm Dataset k:6, s:8

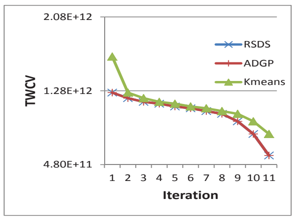


(b) Netflix Dataset, k:6, s:8

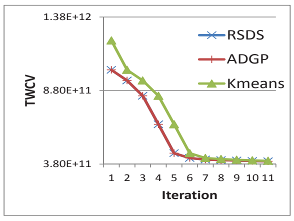


(c) Bio_train Dataset, k:6, s:8

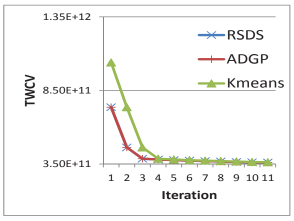
Figure 6: Lastfm, Netflix and Bio_train Clustering Quality



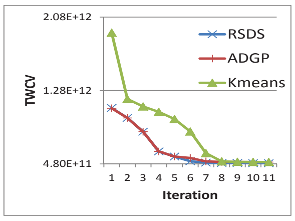
(a) k:6, s:8



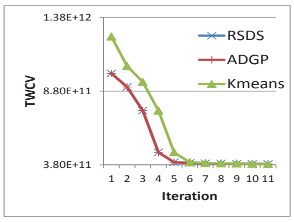
(b) k:12, s:8



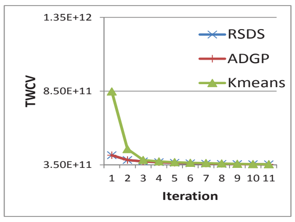
(c) k:18, s:8



(d) k:6, s:20

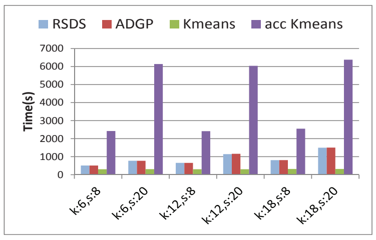


(e) k:12, s:20

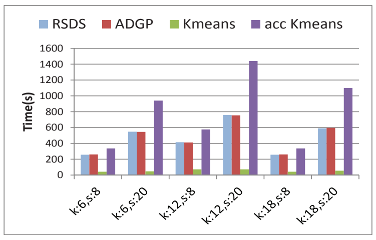


(f) k:18, s:20

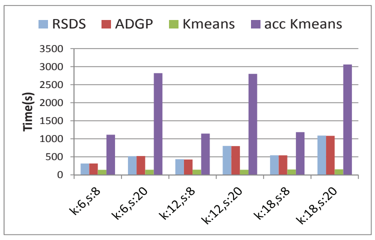
Figure 7: Bio Clustering Quality



(a) Lastfm Datset



(b) Netflix Dataset



(c) Bio Dataset

Figure 8: Elapsed Time

uniformly from the dataset. Then for the $k-1$ centers, they are sampled from the dataset based on a distribution. This method has $O(logk)$-approximation to optimum right after initialization. As the distribution in K-Means++ need to be updated after choosing each initial center, which is not scalable when facing massive data, [5] proposes K-Means||, which oversample by sampling each point independently with a larger probability. These initial centroids selection approaches can be utilized in Mux-Kmeans to choose the set of initial center groups, which could help improve the result quality.

Genetic Algorithm (GA) [8, 11, 17] is the most relevant work to Mux-Kmeans. GA uses a bunch of chromosomes to represent different solutions to a problem. It uses a fitness function to calculate each chromosome's fitness value in solving the problem. Then it selects some chromosomes with high fitness value and uses them producing new chromosomes in the mutation step in order to expand the searching scale. After several iterations, the chromosome with the best fitness value is treated as the final solution to the problem. GA and Mux-Kmeans share many similarities. They both start with many solution candidates and produce new candidates in each iteration. However, they have different optimization goals. GA aims to find solutions to optimization problems, while Mux-Kmeans aims to reduce the runtime of a specific solution Kmeans in a distributed environment. Further, Mux-Kmeans provides two novel incubation heuristics to generate new Kmeans processes rather than blindly generating new chromosomes.

7. CONCLUSION

This paper proposes the Mux-Kmeans algorithm. Mux-Kmeans takes multiple centroid groups to do clustering operation. Different from Kmeans, it handles all centroid groups concurrently. Based on Kmeans algorithm, Mux-Kmeans adds Evaluate, Prune, Permute and Incubate steps to improve clustering quality and reduce runtime. In order to handle large-scale data set, we deploy Mux-Kmeans on MapReduce framework. We apply Mux-Kmeans on Amazon EC2 and evaluate its clustering performance. The results show that, for most cases, Mux-Kmeans has better clustering quality than Kmeans. Mux-Kmeans also requires less runtime than Kmeans when processing multiple centroid groups.

8. ACKNOWLEDGEMENTS

This work was partially supported by National Natural Science Foundation of China (61300023), Fundamental Research Funds for the Central Universities (N120416001), Science and Technology Planning Project of Liaoning Province (2013217004), China Mobil Labs Fund (MCM20122051), and MOE-Intel Special Fund of Information Technology (MOE-INTEL-2012-06).

9. REFERENCES

[1] Amazon EC2, http://aws.amazon.com/ec2/, 2014.

[2] Hadoop MapReduce, http://hadoop.apache.org/, 2014.

[3] K. Alsabti. An efficient k-means clustering algorithm. In *Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.

[4] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[5] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii. Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633, 2012.

[6] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, pages 810–818. ACM, 2010.

[7] C. Elkan. Using the triangle inequality to accelerate k-means. In *ICML*, volume 3, pages 147–153, 2003.

[8] J. H. Holland. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.

[9] M. Joshi and P. Lingras. Evolutionary and iterative crisp and rough clustering ii: Experiments. In *Pattern Recognition and Machine Intelligence*, pages 621–627. Springer, 2009.

[10] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881–892, 2002.

[11] K. Krishna and M. Narasimha Murty. Genetic K-means Algorithm. *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, 29(3):433–439, 1999.

[12] S. Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[13] A. McCallum, K. Nigam, and L. H. Ungar. Efficient clustering of high-dimensional data sets with application to reference matching. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 169–178. ACM, 2000.

[14] D. Pelleg and A. Moore. Accelerating exact k-means algorithms with geometric reasoning. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 277–281. ACM, 1999.

[15] G. Punj and D. W. Stewart. Cluster analysis in marketing research: review and suggestions for application. *Journal of marketing research*, pages 134–148, 1983.

[16] M. S. Sujatha. New fast k-means clustering algorithm using modified centroid selection method. *International Journal of Engineering*, 2(2), 2013.

[17] A. Verma, X. Llora, D. E. Goldberg, and R. H. Campbell. Scaling Genetic Algorithms using Mapreduce. In *Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on*, pages 13–18. IEEE, 2009.

[18] J. Wang, J. Wang, Q. Ke, G. Zeng, and S. Li. Fast approximate k-means via cluster closures. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 3037–3044. IEEE, 2012.

[19] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. In *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, pages 10–10, 2010.

[20] Y. Zhang, Q. Gao, L. Gao, and C. Wang. Priter: a distributed framework for prioritized iterative computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 13. ACM, 2011.

[21] Y. Zhang, Q. Gao, L. Gao, and C. Wang. imapreduce: A distributed computing framework for iterative computation. *Journal of Grid Computing*, 10(1):47–68, 2012.

[22] W. Zhao, H. Ma, and Q. He. Parallel K-means Clustering Based on Mapreduce. In *Cloud Computing*, pages 674–679. Springer, 2009.