PACk: An Efficient Partition-based Distributed Agglomerative Hierarchical Clustering Algorithm for Deduplication

Yue Wang
Microsoft Research
wang.yue@microsoft.com

Vivek Narasayya
Microsoft Research
viveknar@microsoft.com

Yeye He
Microsoft Research
yeyehe@microsoft.com

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

The Agglomerative Hierarchical Clustering (AHC) algorithm is widely used in real-world applications. As data volumes continue to grow, efficient scale-out techniques for AHC are becoming increasingly important. In this paper, we propose a Partition-based distributed Agglomerative Hierarchical Clustering (PACk) algorithm using novel distance-based partitioning and distance-aware merging techniques. We have developed an efficient implementation of PACk on Spark . Compared to the state-of-the-art distributed AHC algorithm, PACk achieves 2× to 19× (median=9×) speedup across a variety of synthetic and real-world datasets.

PVLDB Reference Format:
Yue Wang, Vivek Narasayya, Yeye He, and Surajit Chaudhuri. PACk: An Efficient Partition-based Distributed Agglomerative Hierarchical Clustering Algorithm for Deduplication. PVLDB, 15(6): 1132 - 1145, 2022.
doi:10.14778/3514061.3514062

1 INTRODUCTION

Agglomerative Hierarchical Clustering (AHC) is a widely-used clustering algorithm. As noted in the survey article [40], AHC finds applications in different problems including deduplication and record linkage [7, 37, 54, 57], recommender systems [47], bioinformatics [11, 16, 52], computational chemistry [14], environmental science [20], and astronomy [59]. Given an undirected weighted graph $G = (C, W)$, where C is a set of items and W is a set of weighted edges indicating the distances between pairs of items in C, AHC initializes each item into its own cluster and repeatedly merges the next pair of clusters with the smallest distance until no pair of clusters have a distance below a given threshold.

When two clusters are considered for merging in AHC, the distance between the clusters is defined by a *linkage criterion*. A commonly used [40, 62] linkage criterion in practice is the *average* distance over all pairs of edges across items in the two clusters. Other linkage criteria such as minimum (resp. maximum) distance are also used and results in more aggressive (resp. conservative) merging of clusters compared to average distance. Some specialized and efficient algorithms [3, 25, 42, 45, 56] *only* focus on min-linkage which reduces AHC to the simpler minimum spanning tree problem. For our primary motivating scenario of fuzzy deduplication, average-linkage is the most appropriate criterion. Min-linkage is too aggressive and leads to clustering very dissimilar items, and max-linkage tends to be too conservative and results in detection of too few duplicates. Thus, in this paper we focus on the case of average-linkage, which is also a more challenging problem.

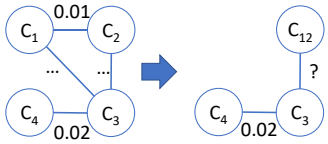

Figure 1: The merge of C_1 and C_2 determines the merge of C_3.

Due to increasing data volumes in real-world applications, the need to support AHC on Big Data platforms such as Spark is growing. For example, the Microsoft Dynamics 365 service applies clustering to find duplicates in their customer profile databases, which can have 100s of millions of records. Centralized (aka single-node) AHC algorithms, which work effectively on relatively small datasets by operating on the data in-memory, are however impractical on large datasets due to their high space complexity ($O(|C|^2)$) and time complexity ($O(|C|^2 \log |C|)$) that lead to memory and CPU bottlenecks on a single machine.

A straightforward adaptation of the centralized AHC algorithms to a scale-out, distributed setting does not perform well because the cost of accessing edges to neighboring nodes in the algorithm becomes excessively high. For example, when the AHC algorithm merges a pair of clusters, it must update distances between the newly merged cluster and all other clusters. In centralized AHC, the update is achieved via a set of relatively cheap writes in memory. However, in a distributed setting with multiple compute nodes (e.g., VMs) working on partitions of the data, this update requires data shuffles across partitions, which is significantly slower. Furthermore, in the distributed setting, multiple iterations of re-partitioning and clustering may be needed thereby amplifying the data shuffle cost.

The natural idea of parallelizing the merge operations holds promise, but is challenging to achieve since merges may have dependencies as illustrated in Figure 1. After the merge of C_1 and C_2, the distance between C_{12} and C_3 determines whether C_3 should be merged with C_{12} or C_4. In other words, the merge of C_3 depends on the merge of C_1 and C_2, thereby introducing difficulty in parallelizing merges when the graph is distributed.

The state-of-the-art distributed AHC algorithm is based on [13]. The authors show that when the linkage criterion satisfies a property called "cluster aggregate inequality" [38], we can concurrently merge all *mutual nearest neighbor pairs* in the graph without affecting correctness of the result. A mutual nearest neighbor pair is a pair of nodes (A, B) such that A is B's nearest neighbor *and* B is A's nearest neighbor. Importantly, they show that this property is

Proceedings of the VLDB Endowment, Vol. 15, No. 6 ISSN 2150-8097.
doi:10.14778/3514061.3514062

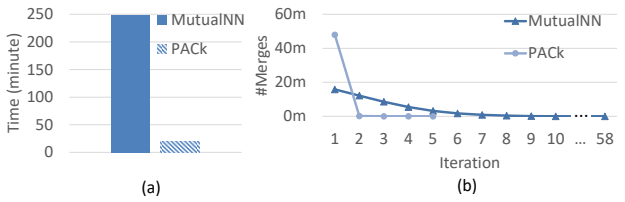

Figure 2: An example real-world dataset with injected duplicates: (a) Our proposed PACk achieves 12× speed-up compared to MutualNN. (b) PACk finishes in 5 iterations, while MutualNN takes 58 iterations with a long tail of fewer merges.

satisfied by linkage criteria including average, max and min – and hence is applicable for our motivating scenarios. An algorithm that exploits this observation, which we refer to as MutualNN is relatively straightforward to implement in a distributed, map-reduce platform using traditional relational operators such as aggregation and join. However, as we show in this paper, this algorithm is inefficient because the number of mutual nearest neighbors is often limited in real-world datasets. Therefore, it often requires multiple (10's) of iterations, with only a few cluster merges possible per iteration as shown in Figure 2. Consequently, MutualNN performs many data scans and shuffles, which lead to large execution time.

In this paper we present the PACk algorithm that builds on the above idea by introducing two novel techniques: distance-based partitioning and distance-aware merging within a partition. Intuitively, the distance-based partitioning algorithm aims to include a set of nearest neighbors in the same partition for each item. It thereby allows more merges to happen within each partition. The distance-aware merging algorithm computes distance bounds to safely merge as many mutual nearest neighbors as possible inside a partition. We show that PACk always produces the same result as centralized AHC. In addition, PACk can perform merges having dependencies in one iteration with guaranteed correctness, which MutualNN cannot do. Our approach parallelizes merges much more effectively and can sharply reduce the number of iterations required, and therefore the overall running time – see Figure 2 for an example on a real-world dataset.

The contributions of this paper are: (1) We present PACk, an efficient distributed clustering algorithm for agglomerative hierarchical clustering. We prove the correctness of PACk, and we have developed an efficient implementation of PACk on Spark. (2) We provide an analytical performance analysis of PACk. We show that PACk is more efficient and needs fewer iterations than MutualNN. (3) We present extensive experimental results comparing the performance and scalability of PACk with MutualNN on a variety of real-world and synthetic graph datasets. PACk consistently outperforms MutualNN with speed-ups ranging from 2× to 19× (median=9×). Its compute resources including CPU and memory are comparable to MutualNN and modestly higher. PACk also scales well to relatively large graphs. For example, on a real-world graph evaluated by the Dynamics 365 service in Microsoft for the task of fuzzy deduplication containing over 250 million items and 680 million edges, PACk finishes in 40 minutes using 16 commodity eight-core VMs, achieving 5× speed-up compared to MutualNN.

We organize the paper as follows: we present the background for AHC in Section 2. We introduce PACk in Section 3. We present the correctness proof and analytical performance analysis in Section 4. We present our experimental evaluation result in Section 5. Finally, we discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

2.1 Cluster Labeling and Distance Comparison

When two pairs of clusters have the same distance, we must break ties deterministically to ensure that the result of clustering is deterministic regardless of whether a centralized or distributed AHC algorithm is used. Therefore, in addition to the scalar distance between clusters, we take the cluster labels into comparison.

We assume each initial item $c \in C$ has an associated label denoted by $label(c)$ (e.g., integer, string, etc.). The labels form a totally ordered set. We further assume that the cluster label is the maximum label of the cluster's items:

$$label(C) = \max_{c \in C} \big(label(c)\big) \tag{1}$$

Let $dist(C_i, C_j)$ be the scalar distance between cluster C_i and C_j, so that we can define weight $w(C_i, C_j)$ as a three-element tuple:

Definition 1 (Cluster Pair Edge Weight).

$$w(C_i, C_j) = \big(dist, label1, label2\big)$$

where $dist = dist(C_i, C_j)$, $label1 = \min(label(C_i), label(C_j))$, *and* $label2 = \max(label(C_i), label(C_j))$.

When we compare two edges, we always compare the three-element weight tuples $w(C_i, C_j)$ instead of only the scalar distances $dist(C_i, C_j)$.

Example 1 (Labeling). *Assume we use integer labels in Figure 3 such as* $label(C_1) = 1$, $label(C_2) = 2$, $label(C_{13}) = 3$, *and so on. Two weight examples are* $w(C_1, C_2) = (0.05, 1, 2)$ *and* $w(C_3, C_2) = (0.05, 2, 3)$, *so* $w(C_1, C_2) < w(C_3, C_2)$ *because tuple* $(0.05, 1, 2) < (0.05, 2, 3)$.

2.2 Agglomerative Hierarchical Clustering

Given an undirected weighted graph $G(C, W)$, where C is a set of items and W is a set of weights indicating the distances between pairs of items in C, and a threshold $\theta > 0$, Agglomerative Hierarchical Clustering (AHC) algorithm starts by treating each item as a singleton cluster, iteratively merges nearest cluster pairs, and stops when no two clusters are less than distance θ. Algorithm 1 shows the centralized AHC algorithm, which is straightforward when the entire graph fits in memory.

There exist several linkage criteria to compute the distance function between *clusters* (Line 5) in AHC. Here we list a few:

- Max-linkage (complete-linkage) [12]:
  $dist(C_{ij}, C_x) = \max\big(dist(C_i, C_x), dist(C_j, C_x)\big)$
- Min-linkage (single-linkage) [22, 48]:
  $dist(C_{ij}, C_x) = \min\big(dist(C_i, C_x), dist(C_j, C_x)\big)$
- Average-linkage [50]:
  $dist(C_{ij}, C_x) = \frac{1}{|C_{ij}| \cdot |C_x|} \cdot \sum_{c \in C_{ij}, c' \in C_x} dist(c, c')$

Given its suitability for the fuzzy deduplication problem as noted earlier, in the rest of this paper, we focus on **Average-linkage**.

Algorithm 1: Centralized AHC

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*

while *there exists* $dist(C_i, C_j) \leq \theta$ **do**
  $(C_i, C_j) \Leftarrow \underset{C_i, C_j \in C \wedge C_i \neq C_j}{arg\,min} \left(w(C_i, C_j)\right)$
  $C_{ij} \Leftarrow C_i \cup C_j$
  $C \Leftarrow C \cup \{C_{ij}\} - C_i - C_j$
  Compute $w(C_{ij}, C_x)$ for each $C_x \in C$
$C^* \Leftarrow C$

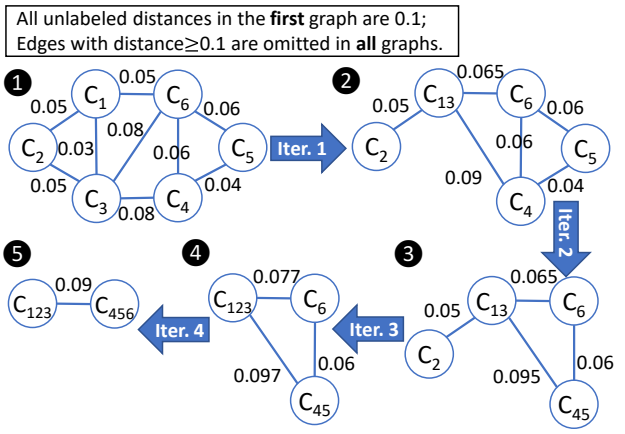

Figure 3: An example that applies Algorithm 1 to six items. θ = 0.08.

Example 2 (Centralized AHC). *Figure 3 is an example that applies Algorithm 1 to six items with threshold $\theta = 0.08$. Edges are labelled with distances. In the* first *graph, unlabeled distances are all 0.1. In* all *the graphs, edges with distance≥0.1 are omitted. The algorithm keeps merging nearest pairs until the remaining edges are greater than* 0.08.

2.3 Distributed AHC

Although the centralized AHC algorithm is straightforward, developing an efficient distributed AHC algorithm is challenging. The efficiency of distributed depends primarily on two factors. The first factor is the number of iterations. Similar to the centralized AHC, a distributed AHC usually takes multiple iterations to finish. In distributed AHC however, each iteration has certain costs such as scanning the graph and writing the intermediate result to persistent storage at the end of the iteration. The second factor is data shuffle. In each iteration, every compute node (VM) works on a partition of a large graph. Since edges can span across partitions, VMs have to shuffle data to find neighbors and update distances. Therefore, techniques that reduce number of iterations and data shuffle cost can lead to greater efficiency and improved performance.

The state-of-the-art distributed AHC is `MutualNN` [13], which parallelizes merges to reduce number of iterations and data shuffle. `MutualNN` is based on the *Cluster Aggregate Inequality* property [13, 38] that makes parallel merges possible. Specifically, in each iteration, `MutualNN` merges all *mutual nearest* neighbor pairs (C_x, C_y) where C_x's nearest neighbor is C_y and vice versa. `MutualNN` guarantees that its result is the same as the centralized AHC as long as the linkage satisfies Cluster Aggregate Inequality:

$$\forall C_i, C_j, C_x : dist(C_{ij}, C_x) \geq \min\left(dist(C_i, C_x), dist(C_j, C_x)\right) \quad (2)$$

The intuition behind Cluster Aggregate Inequality is: if C_x has a unique nearest neighbor C_y, $dist(C_y, C_x)$ must be smaller than any other $dist(C_i, C_x)$ or $dist(C_j, C_x)$, i.e. $dist(C_y, C_x) < \min\left(dist(C_i, C_x), dist(C_j, C_x)\right)$. Hence, merging any other clusters C_i and C_j cannot generate a new cluster C_{ij} whose distance to C_x is closer than $dist(C_y, C_x)$. Therefore, when C_x and C_y are *mutual* nearest neighbors, we can safely merge them. The min-, max-, and average-linkage all satisfy this inequality. The detailed proof of `MutualNN` correctness can be found in [13]. We show why Average-Linkage satisfies the inequality in [55].

Algorithm 2: `MutualNN`

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*
`/* Compute in parallel */`

while *there exists* $dist(C_i, C_j) \leq \theta$ **do**
  $NN \Leftarrow$ For each C in G, compute its nearest neighbor
  $MNN \Leftarrow$ Find mutual nearest neighbor pairs by self-join on NN
  $G \Leftarrow$ Merge mutual nearest neighbors and their edges by join and aggregation

Algorithm 2 shows how `MutualNN` works. In each iteration, it finds all mutual nearest neighbor pairs, merges them, and computes the new weights for newly merged clusters. However, `MutualNN` is inefficient as we see in Figure 2, because the number of mutual nearest neighbors is often limited in real-world datasets and `MutualNN` still takes too many iterations. Therefore, we propose a Partition-based distributed Agglomerative hierarchical Clustering (`PACk`) algorithm which significantly increases the number of merges in each iteration to improve the efficiency.

3 PACk ALGORITHM FOR DISTRIBUTED AHC

3.1 Intuition

`PACk` achieves its efficiency using two novel algorithms: distance-based partitioning of the graph, and distance-aware merging within each partition. When partitioning the graph, `PACk` places clusters with their top nearest neighboring clusters together. To limit the size of each partition, for each cluster in a partition, we only include a list of edges with the shortest distances to it, and represent all ignored edges as a lower bound b_L indicating that their distances are greater than b_L. The distance-aware merging algorithm works on each partition and performs merges locally. Whenever it merges a cluster pair, it always ensures that the two clusters are mutual nearest neighbors by checking the distance bounds, which guarantees the correctness of the result.[1] Compared to `MutualNN`, `PACk` performs

[1] Similar to `MutualNN`, `PACk` also works for Max-, Min-, Average-, and any other linkage criterion that satisfies Cluster Aggregate Inequality. The proof of correctness is in Section 4.1.

many more merges in each iteration, thereby reducing the total number of iterations required. Although the shuffle cost for one iteration of PACk could exceed that of MutualNN, since the number of iterations are significantly reduced (Figure 2), the overall shuffle cost of PACk is also much less compared to MutualNN.

Below we provide intuition on why performing more merges in each iteration can improve performance. Observe that for a given input graph, the total number of pair-wise merges done is the same regardless of the specific AHC algorithm used. For instance, given the input in Figure 3, we need four merges to get C_{13}, C_{123}, C_{45}, and C_{456}. Performing more merges in each iteration reduces running time for three reasons: (1) Merges are performed in parallel, which can take less time compared to sequential execution. (2) More merges per iteration reduces the number of iterations, thereby saving the fixed overheads incurred for each iteration. (3) More merges in one iteration reduces the shuffle cost of intermediate results in the following iterations. For example, assume a distributed algorithm finishes in four iterations as shown in Figure 3. It has to generate Graph 2 (resp. Graph 3 and 4) after Iteration 1 (resp. Iter. 2 and 3), and shuffle the graphs' weights to compute nearest neighbor etc. for the next Iteration 2 (resp. Iter. 3 and 4). In comparison, PACk requires one iteration as shown in Figure 4, so we save the shuffle cost of three intermediate graphs, Graph 2, 3, and 4, in Figure 3. Note that the intermediate weights such as $w(C_{13}, C_2)$, $w(C_{123}, C_6)$, etc. are still generated *locally* within each partition, but they are discarded once merging is done for each partition. So these intermediate weights are never shuffled after local merging.

Figure 2b shows an example illustrating how PACk can perform much more merges in one iteration than MutualNN does. It plots the number of merges done by both algorithms on one of our experimental datasets. In the first iteration, PACk completes 99% merges, which is much more than MutualNN's 32%. Moreover, in each of the following iterations, PACk still completes the majority of the *remaining* merges, while MutualNN does only a much smaller percentage. For instance, in the second iteration, PACk does 98% of its remaining merges, while MutualNN does only 37% of its remaining ones. PACk's ability to perform the majority of remaining merges in each iteration significantly reduces the number of iterations and cost of data shuffle, which shortens the running time.

3.2 Overview

Algorithm 3 describes PACk. We assume the input is a graph $G = (C, W)$ where C is the initial item set and W is the weights defined in Section 2.1[2]. PACk keeps merging clusters in iterations as long as the graph has weights that are below the distance threshold. Each iteration consists of four steps:

(1) Partitioning. We partition the graph by putting clusters with their top nearest neighbors together, so that multiple merges have a chance to happen within each partition.

(2) Distance-aware Merging. Within each partition, we merge as many mutual nearest neighbor pairs as possible. For each merge, we track the distance bounds between the newly merged clusters and the other clusters. By comparing the distance bounds, we guarantee to always merge mutual nearest pairs.

(3) Integration. The output clusters of the distance-aware merging need to be integrated because there may be overlapping clusters. As we prove in Theorem 2, every cluster in the output of Algorithm 4 is a correct merge of a mutual nearest pair. Therefore, if two clusters in the output overlap, one must be the superset of the other. Then the integration algorithm keeps the maximal clusters, i.e. those that are not a strict subset of any other cluster.

(4) Graph Update. For each merged cluster, we assign the new label to all its members using a join. Then we aggregate the edges between any two clusters to calculate sum and average distances.

Figure 4 is an example that applies PACk to the same input of Example 2. It finishes in one iteration as we will see in the remainder of Section 3.

Next, we start with the distance-aware merging in Section 3.3, which is a natural extension of the centralized AHC algorithm and allows us to do merge on a partial graph (i.e., a partition). Then, we describe the partitioning algorithm to partition a given graph in Section 3.4.

Algorithm 3: PACk

Input: Cluster graph $G = (C, W)$; Threshold θ
Output: Clusters C^*

```
while there exists dist(C_i, C_j) ≤ θ do
    P ⇐ Partition(G)                          // Algorithm 5 or 6
    C' ⇐ {LocallyMerge(p) | p ∈ P}            // Algorithm 4
    C ⇐ Integrate C'
    W ⇐ Merge weights based on C
C* ⇐ C
```

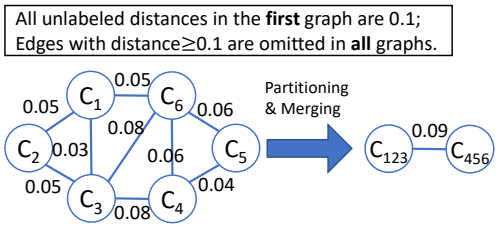

Figure 4: An example that applies PACk to six items. θ = 0.08. PACk finishes in one iteration.

3.3 Distance-Aware Merging

We start with the distance-aware merging algorithm that takes a partition as input and outputs merged clusters. It requires distance bounds as input for each partition, which will be explained in Section 3.4. Developing a merging algorithm that works for a partition and guarantees correctness is challenging because (a) a partition is usually limited by size to fit into a VM's memory, and (b) in one iteration, each partition cannot know the change outside this partition.

We develop a Distance-Aware Merging algorithm that tracks distance bounds to address the above challenges. First, for each cluster C_i in a partition, instead of requiring that all its edges reside in memory, we only require its nearest neighbors to form an edge list $\mathcal{L}(C_i)$ defined below, so that the memory size per partition

[2]In practice, W usually contains only the pairs with meaningful distance (e.g., two strings share at least one token) so that $|W| \ll |C|^2$. Various indexing techniques are used to efficiently retrieve close pairs in different scenarios (e.g., Locality Sensitive Hashing for Jaccard distance, space-partitioning trees for Euclidean distance, n-gram for edit distance, and so on). They are orthogonal to our contribution in this paper.

can be limited. Second, we convert each distance from a scalar to a range, so that the edges outside the partition (i.e. $\notin \mathcal{L}(C_i)$) can be represented by a wildcard edge indicating the lower bound of their distances to C_i. By leveraging the bounds, we are able to safely detect when two clusters are mutually nearest.

Specifically, we define $\mathcal{L}(C_i)$ as the list of nearest neighbors of C_i, whose size limit is a configurable parameter. For each $C_j \in \mathcal{L}(C_i)$, we define $b_L(C_i, C_j)$ and $b_U(C_i, C_j)$ as the lower and upper bounds of $dist(C_i, C_j)$ respectively. $b_L(C_i, C_j)$ and $b_U(C_i, C_j)$ are initialized to $dist(C_i, C_j)$. In addition to the above bounds, we always automatically attach a special wildcard C_i^* into $\mathcal{L}(C_i)$. Its lower bound $b_L(C_i, C_i^*)$ indicates that all remaining neighbors are beyond distance $b_L(C_i, C_i^*)$. Its $b_U(C_i, C_i^*)$ is an application-specific large value (e.g., ∞) indicating the upper bound.

By the definition of Average-linkage in Section 2.2, we can compute the distance between C_{ij} and any other C_x as

$$dist(C_{ij}, C_x) = \frac{dist(C_i, C_x) \cdot |C_i||C_x| + dist(C_j, C_x) \cdot |C_j||C_x|}{(|C_i| + |C_j|) \cdot |C_x|} = \frac{dist(C_i, C_x) \cdot |C_i| + dist(C_j, C_x) \cdot |C_j|}{|C_i| + |C_j|}$$

Similarly, we compute the bounds in three cases:

(1) If a neighbor C_x exists in both $\mathcal{L}(C_i)$ and $\mathcal{L}(C_j)$, we can precisely compute the bounds as:

$$b_L(C_{ij}, C_x) = \frac{b_L(C_i, C_x)|C_i| + b_L(C_j, C_x)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_x) = \frac{b_U(C_i, C_x)|C_i| + b_U(C_j, C_x)|C_j|}{|C_i| + |C_j|}$$

(2) If a neighbor C_x exists in only one edge list, say, $\mathcal{L}(C_i)$, we use the wildcard C_j^* for C_j:

$$b_L(C_{ij}, C_x) = \frac{b_L(C_i, C_x)|C_i| + b_L(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_x) = \frac{b_U(C_i, C_x)|C_i| + b_U(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$

(3) If a neighbor C_x does not exist in any edge list, we use the wildcard edge (C_{ij}, C_{ij}^*) to represent it. The bounds can be derived from edge (C_i, C_i^*) and (C_j, C_j^*):

$$b_L(C_{ij}, C_{ij}^*) = \frac{b_L(C_i, C_i^*)|C_i| + b_L(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$
$$b_U(C_{ij}, C_{ij}^*) = \frac{b_U(C_i, C_i^*)|C_i| + b_U(C_j, C_j^*)|C_j|}{|C_i| + |C_j|}$$

We keep merging clusters within the partition as long as (1) we can find a pair of mutual nearest neighbor (C_i, C_j); and (2) the upper bound of $dist(C_i, C_j)$ (i.e., $b_U(C_i, C_j)$) is no greater than θ. Algorithm 4 shows more detail. It takes a partition P_h and a threshold θ as input. P_h consists of a set of clusters $\mathscr{C}_h$ and the edge lists of the clusters $\{\mathcal{L}(C_i)|C_i \in \mathscr{C}_h\}$. It generates a set of clusters C_{out} as output.

Algorithm 4: Distance-aware Merging for Each Partition

Input: Single partition $P_h = (\mathscr{C}_h, \{\mathcal{L}(C_i)|C_i \in \mathscr{C}_h\})$; Threshold θ
Output: Clusters C_{out}
/* Compute in memory */
1 $G \Leftarrow$ Build a graph from P_h
2 **for** $C_i \in \mathscr{C}_h$ **do**
3 $NN(C_i) \Leftarrow C_i$'s nearest neighbor whose upper bound is smaller than the lower bounds of other C_i's neighbors; *null* if non-existent
4 **while** *true* **do**
5 **if** $\exists(C_i, C_j) : (NN(C_i) = C_j) \wedge (NN(C_j) = C_i) \wedge (b_U(C_i, C_j) \leq \theta)$ **then**
6 Merge C_i with C_j, and update G and $NN(\cdot)$
7 **else**
8 **break**
9 $C_{out} \Leftarrow$ Merged clusters in G

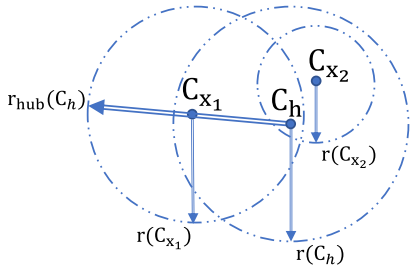

Figure 5: Example of hub radius, in which $r_{hub}(C_h) = r(C_{x_1}) + dist(C_{x_1}, C_h)$.

3.4 Partitioning

In Section 3.3, we show how distance bounds work in distance-aware merging. Next we show how to partition a graph and obtain distance bounds. A partitioning algorithm such as a random partitioning, puts random clusters together in each partition, which likely cannot be merged, thereby rendering it ineffective. Therefore, an effective partitioning algorithm must be carefully designed to let the distance-aware merging perform as many merges as possible for each partition. Intuitively, we want to place clusters with their nearest neighbors together in the same partition, so that more merges can happen locally within that partition. We show in Section 4 that a carefully designed partitioning algorithm needs no more than half the number of iterations of `MutualNN`, which significantly improves the efficiency. We first present the distance-based partitioning Algorithm 5 to illustrate the key idea, and then we present Algorithm 6 that refines Algorithm 5 to allow it to work in practice with the memory constraint of a compute node.

Intuitively, Algorithm 5 puts clusters with their nearest neighbors together to let the merging algorithm perform as many merges as possible. Specifically, Algorithm 5 focuses on clusters that have mutual nearest neighbors (i.e., *hubs*). It creates a partition for each hub by choosing an appropriate *radius* that covers many nearest neighbors but not overlaps other partitions too much.

Algorithm 5 describes the partitioning algorithm. First, it gets a radius for each cluster C_x (Line 11 to 13). The radius is 10 times the distance between C_x and its *second* nearest neighbor (the number 10 helps us reduce number of iterations as we will see in Section 4.3). Now, if we create a partition for each cluster with the radius, the partitions may overlap heavily, and some merges may redundantly

Algorithm 5: Distance-based Partitioning

Input: Cluster graph $G = (C, W)$; Bivariate distance function $dist(\cdot, \cdot)$
Output: Partitions P

/* Compute in parallel */
1 $Hub \Leftarrow \{C_h | C_h$ has a *mutual* nearest neighbor C'_h and $label(C_h) < label(C'_h)\}$
2 $r_{hub}(\cdot) \Leftarrow$ CalcRadius $(G, Hub, dist(\cdot, \cdot))$ // Defined below
3 **foreach** $C_h \in Hub$ **do**
4 $\quad \mathscr{C}_h \Leftarrow \{C_h\} \cup \{C_x | w(C_h, C_x) \in W \ \& \ dist(C_h, C_x) \leq r_{hub}(C_h)\}$
5 $\quad$ **foreach** $C_x \in \mathscr{C}_h$ **do**
6 $\quad\quad \mathcal{L}(C_x) \Leftarrow \{w(C_y, C_x) | dist(C_x, C_y) \leq r_{hub}(C_h)\} \cup \{$wildcard edge for $C_x\}$
7 $\quad P_h \Leftarrow (\mathscr{C}_h, \{\mathcal{L}(C_x) | C_x \in \mathscr{C}_h\})$
$\quad\quad$ // Each partition P_h is a tuple of cluster set and edge list set
8 $P \Leftarrow \{P_h | C_h \in Hub\}$
9 **return** P

10 **Function** CalcRadius $(G = (C, W), Hub, dist(\cdot, \cdot))$
11 $\quad$ **foreach** $C_x \in C$ **do**
12 $\quad\quad C_y \Leftarrow$ the 2nd nearest neighbor of C_x
13 $\quad\quad r(C_x) \Leftarrow 10 \cdot dist(C_x, C_y)$
14 $\quad$ **foreach** $C_h \in Hub$ **do**
15 $\quad\quad Neighbor(C_h) \Leftarrow \{C_h\} \cup \{C_x | w(C_h, C_x) \in W\}$
16 $\quad\quad r_{hub}(C_h) \Leftarrow \max_{\substack{C_x \in Neighbor(C_h) \\ \& dist(C_h, C_x) \leq r(C_x)}} \big(r(C_x) + dist(C_x, C_h)\big)$

Algorithm 6: Partitioning with Size Limit

Input: Cluster graph $G = (C, W)$; Bivariate distance function $dist(\cdot, \cdot)$; Neighbor limit k_N; Edge list limit k_L
Output: Partitions P

/* Compute in parallel */
1 $Hub \Leftarrow \{C_h | C_h$ has a *mutual* nearest neighbor C'_h and $label(C_h) < label(C'_h)\}$
2 **foreach** $C_h \in Hub$ **do**
3 $\quad \mathscr{C}_h \Leftarrow \{C_h\} \cup \{$top k_N neighbors in $W\}$
4 $\quad$ **foreach** $C_x \in \mathscr{C}_h$ **do**
5 $\quad\quad \mathcal{L}(C_x) \Leftarrow \{$top k_L neighbors' weights in $W\} \cup \{$wildcard edge for $C_x\}$
6 $\quad P_h \Leftarrow (\mathscr{C}_h, \{\mathcal{L}(C_x) | C_x \in \mathscr{C}_h\})$
$\quad\quad$ // Each partition P_h is a tuple of cluster set and edge list set
7 $P \Leftarrow \{P_h | C_h \in Hub\}$

happen in many partitions. So we control the number of partitions by focusing on the "hubs" (Line 1[3]). A hub is a cluster that has a *mutual* nearest neighbor in G, which will be merged because its radius covers its nearest neighbor. In other words, a partition including a hub and its mutual nearest neighbor ensures at least one merge. Therefore, we keep only hubs' partitions but increase the hub radius to cover clusters in other partitions. Specifically, when a hub C_h is covered by another cluster C_x's radius, we increase the hub's radius to cover C_x's radius (Line 14 to 16). Then we can safely ignore all non-hub's partitions. Note that the hubs' partitions may still overlap, but the overlapping space is much smaller and will not hurt efficiency much in practice. Figure 5 shows an example of calculating the hub's radius, in which $r_{hub}(C_h) = r(C_{x_1}) + dist(C_{x_1}, C_h)$ turns out to be the maximum radius for C_h. Finally from Line 3 to 7, for each hub, we collect the clusters within its radius and those clusters' nearest neighbors to form their edge lists.

In theory, Algorithm 5 might create very large partitions that exceeds the memory available on a VM. In order to limit the size of each partition, we use k_N to limit the number of nearest neighbors of hubs, and k_L to limit the size of edge lists. As a result, we simplify Algorithm 5 to get the version with size limit (Algorithm 6). For each hub, it simply gets k_N nearest neighbors and top k_L edges.

Example 3 (Partitioning with Size Limit). *Assume $k_N = k_L = 4$ in Algorithm 6. Given the input graph in Figure 4, we find 2 hubs $\{C_1, C_4\}$.*

C_1 forms a partition P_1 with its nearest 4 neighbors C_3, C_2, C_6, and C_4. When P_1 is passed to Algorithm 4, C_1 and C_3 are firstly merged to get C_{13}. Then C_{13} and C_2 become mutual nearest and are merged into C_{123}.

C_4 forms a partition P_4 with its nearest 4 neighbors C_5, C_6, C_3, and C_1. When P_4 is passed to Algorithm 4, C_1 and C_3 are merged first and $dist(C_{13}, C_6)$ are updated to 0.065. So C_6's nearest neighbor becomes C_4 and is no longer blocked by C_1. Then C_4 and C_5 are merged to C_{45}. Finally C_6 and C_{45} are merged to C_{456}.

In summary, Algorithm 6 creates two partitions P_1 and P_4, which are passed to Algorithm 4 to generate C_{123} and C_{456} respectively. The whole process ends in one iteration.

Discussion of k_N and k_L. In practice, moderate k_N and k_L in a wide range like $[50, 500]$ should work reasonably well as we will see in experiments in Section 5.3. If k_N and k_L are too large, it can adversely affect performance because too many distant neighbors are scanned and shuffled without increasing the number of merges in each iteration. Another benefit of using moderate k_N and k_L is to balance the load. For instance, when $k_N = k_L = 500$, the worst-case space of each partition is only around 10 MB ($O(k_N k_L \cdot sizePerEdge)$), and the worst-case time to cluster a partition is only around 10 milliseconds ($O(k_N k_L \cdot \log(k_N k_L))$). In such case, no partition can become a straggler.

4 ANALYSIS OF CORRECTNESS AND PERFORMANCE

In this section we first prove the correctness of PACk and then analyze the performance of the algorithms. Specifically, We propose a Cluster Directed Acyclic Graph (DAG) in Section 4.1 to prove the correctness of PACk. In Section 4.2 and 4.3, we use the DAG to prove that the number of iterations of PACk is half of MutualNN's. In Section 4.4, we use a simplified cost model to show the performance of PACk is better than MutualNN in an example deduplication scenario.

4.1 Cluster DAG and Correctness

Intuitively, we prove the correctness by showing that every cluster we generate must be a merge of two mutual nearest neighbors. It is consistent with the Centralized AHC (Algorithm 1), which also always merges two mutual nearest neighbors.

[3]Hubs can be efficiently detected through a relational group-by query and then a self-join. The group-by scans the graph whose size is $|W|$. Again, $|W| \ll |C|^2$ in practice because users usually remove pairs with long distances. The self-join only joins a table of nearest neighbors of size $|C|$.

We propose a Cluster Directed Acyclic Graph (DAG) that helps us model the number of iterations, which is necessary for estimating the data shuffle cost and running time. **Note** that our algorithm *never* explicitly constructs the DAG during execution. The DAG below is only *conceptual* and for our performance analysis.

Given the set of initial singleton clusters C, the function $dist(\cdot,\cdot)$, and the threshold θ, the DAG $D = (\mathscr{C}, E)$ is constructed based on the execution of Algorithm 1. Specifically, we define *Initial Clusters* as the clusters given in the input graph, define *Merged Clusters* as those merged by Algorithm 1 in all iterations (i.e. C_{ij} in Line 3 of Algorithm 1), and define set $\mathscr{C}$ as the union of all *Initial Clusters* and all *Merged Clusters*. We further define the following functions to help our presentation below: (i) For each Merged cluster C_x, we denote its two direct subclusters by $C^L(C_x)$ and $C^R(C_x)$. Also, we call C_x as the "parent" $C^P(.)$ of its two direct subclusters. E.g., $C^P(C^L(C_x)) = C_x$ and $C^P(C^R(C_x)) = C_x$. (ii) If a cluster is merged with another, we define them as "siblings" $C^S(.)$. For example, $C^S(C^L(C_x)) = C^R(C_x)$ and $C^S(C^R(C_x)) = C^L(C_x)$. For simplicity, we let $C^L(C_x) = C^L_x$, $C^R(C_x) = C^R_x$, $C^P(C_x) = C^P_x$, and $C^S(C_x) = C^S_x$ **hereafter** when the context is clear. Also, we abbreviate nested functions such as $C^L(C^R(C_x)) = C^{LR}_x$ **hereafter**.

The edge set E captures all the dependencies of merges. Specifically, a directed edge (C_x, C_y) means that C_x must be generated before C_y is generated. There are two types of edges: "Subset Dependency" and "Weight Dependency".

Definition 2 (Subset Dependency Edge). *For each Merged cluster C_x, we define two subset dependency edges (C^L_x, C_x) and (C^R_x, C_x).*

Intuitively, Subset Dependency means that C^L_x and C^R_x must be the prerequisites of C_x.

Definition 3 (Weight Dependency Edge). *For each pair of Merged clusters C_x and C_y that satisfies $C_x \cap C_y = \emptyset$, we build an edge (C_x, C_y) if and only if:*

$$\exists C'_y \in \{C^L_y, C^R_y\}, C'_x \in \{C^L_x, C^R_x\} :$$
$$w(C'_y, C'_x) < w(C^L_y, C^R_y)$$

Each Weight Dependency (C_x, C_y) means that C_y cannot be generated yet because C'_y's nearest neighbor is C'_x instead of its sibling $C^S(C'_y)$.

One can view the DAG as one or more binary trees (i.e. dendrograms) plus extra edges: all Initial/Merged clusters and the Subset Dependency edges form one or more binary trees, and the Weight Dependency are the extra edges.

Example 4 (Cluster DAG). *Figure 6 is the Cluster DAG of Example 2. C_1 to C_6 are the Initial singleton clusters. C_{13}, C_{45}, C_{123}, and C_{456} are Merged clusters.*

The solid lines represent Subset Dependency, which in fact form two binary trees (i.e., dendrograms) of the clustering process. For instance, $C^L(C_{13}) = C_1$, $C^R(C_{13}) = C_3$, $C^P(C_{13}) = C_{123}$, and $C^S(C_{13}) = C_2$.

The dashed lines represent Weight Dependency. Weight Dependency (C_{13}, C_{456}) is because C_6's nearest neighbor has been C_1 until the merge (i.e., generation) of C_{13}.

Theorem 1. *The constructed DAG D does not have cycles.*

In order to prove Theorem 1 and to facilitate our following analysis, we define a few concepts.

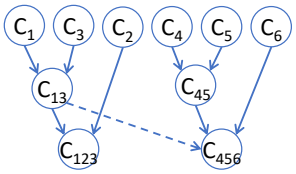

Figure 6: The Cluster DAG of Example 2. Solid lines are subset dependency. Dashed lines are weight dependency.

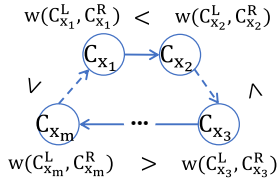

Figure 7: Illustration of the proof of Theorem 1. A cycle will lead to $w(C^L_{x_1}, C^R_{x_1}) < w(C^L_{x_2}, C^R_{x_2}) < ... < w(C^L_{x_m}, C^R_{x_m}) < w(C^L_{x_1}, C^R_{x_1})$, contradiction.

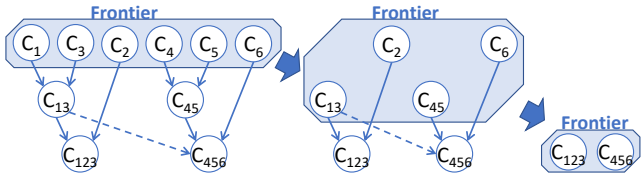

Figure 8: The change of Frontier in Example 5.

Definition 4 (Generated/Ungenerated Clusters). *At the beginning of an algorithm's iteration (i.e. Line 1 of Algorithm 1 or Line 1 of Algorithm 3), a cluster in $\mathscr{C}$ is a Generated Cluster if it is an initial cluster or is already generated through merging by the algorithm. Otherwise, it is an Ungenerated Cluster.*

Definition 5 (Frontier). *At the beginning of an algorithm's iteration, a Frontier F is the set of clusters such that each cluster $C_x \in F$ satisfies* both *conditions below:*

- *C_x is a Generated Cluster.*
- *C_x does not have parent C^P_x,* ***or*** *C^P_x is an Ungenerated Cluster.*

In other words, the frontier is the "snapshot" of the clusters at the beginning of each iteration in the algorithm.

Example 5 (Generated/Ungenerated Clusters and Frontier). *Given Example 2, suppose a clustering algorithm finishes in two iterations. The first iteration generates C_{13} and C_{45}. The second iteration generates C_{123} and C_{456}.*

Figure 8 shows how the frontier changes.

Initially, only $\{C_1, C_2, ..., C_6\}$ are Generated, which form the frontier. After the first iteration, C_{13} and C_{45} are Generated. So the frontier becomes $\{C_{13}, C_2, C_{45}, C_6\}$. After the second iteration, all clusters are Generated. The frontier becomes $\{C_{123}, C_{456}\}$.

Now we can prove Theorem 1 by contradiction (Figure 7). The idea is to show that (1) Any cluster C_x in the cycle must be a Merged cluster; (2) Each edge (C_x, C_y) in a cycle satisfy $w(C^L_x, C^R_x) < w(C^L_y, C^R_y)$, leading to a contradiction that $w(C^L_x, C^R_x) < w(C^L_x, C^R_x)$ as we go through the cycle. The detailed proof is in [55].

4.1.1 Correctness. After defining the set of all clusters $\mathscr{C}$, we can prove the correctness of our algorithm.

Theorem 2. *The output of Algorithm 3 is the same as that of Algorithm 1.*

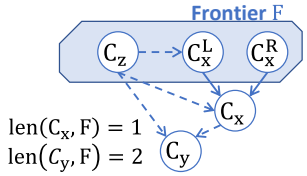

Figure 9: $len(\cdot, F)$ Examples: $len(C_x, F) = 1$ and $len(C_y, F) = 2$.

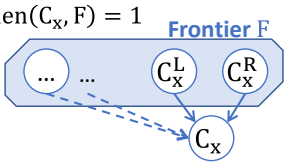

Figure 10: When $len(C_x, F) = 1$, C_x^L and C_x^R must be mutual nearest neighbors in F.

Proof. (Sketch) We prove that (A) every Merged cluster in Algorithm 3 is in the DAG, and that (B) every cluster with 0 out-degree in the DAG is generated by Algorithm 3.

(A) All Merged clusters are generated in Line 6 of Algorithm 4, which guarantees that C_i and C_j are mutual nearest. Therefore, C_i and C_j can be safely merged, because any merge of other clusters won't change the fact that C_i and C_j are mutual nearest neighbors [13, 38]. In addition, when integrating generated clusters (Line 4 in Algorithm 3), we only remove cluster and do not generate extra clusters. So every Merged cluster in Algorithm 3 is in $\mathscr{C}$.

(B) We prove by contradiction. Suppose there exists a cluster with 0 out-degree in the DAG and it is not generated by Algorithm 3. We check all its direct dependents. There are two cases:

(1) All its dependents are generated.
(2) At least one dependent is ungenerated. Let it be C_x. Then we check the dependents of C_x.

We keep checking the dependents of the ungenerated cluster until all dependents are generated. This process should always stop because all Initial clusters are Generated by definition.

When we find the ungenerated C_x whose dependents are all generated, its direct children C_x^L and C_x^R must be mutual nearest in the graph because C_x's dependents are all generated (by Definition 3). Then the partitioning algorithm should build a partition for C_x^L and C_x^R, and Line 6 of Algorithm 4 should generate $C_x = C_x^L \cup C_x^R$. **Contradiction**.

So every cluster with 0 out-degree in the DAG is generated by Algorithm 3. □

4.2 Number of Iterations of MutualNN

We define length to facilitate our proofs below. Note that length is defined only on the DAG, which is irrelevant to the distance function. We define $len(C_x)$ as the longest distance from any *Initial* cluster to $C_x \in \mathscr{C}$, and $len_{max} = \max_{C_x \in \mathscr{C}}(len(C_x))$.

Example 6 (Length from Initial Cluster). *In the DAG in Figure 6, $len(C_1) = len(C_2) = ... = len(C_6) = 0$, $len(C_{13}) = len(C_{45}) = 1$, and $len(C_{123}) = len(C_{456}) = 2$. So $len_{max} = 2$.*

Given a frontier F and an Ungenerated cluster C_x, we define $len(C_x, F)$ as the distance between F to C_x. Formally, let $pa = (C_z, ..., C_x)$ be a *valid* path from C_z to C_x where *only* the first cluster C_z is in F (i.e. $F \cap pa = \{C_z\}$). Let PA be the set of such paths. $len(C_x, F) = \max_{pa \in PA}(\text{length of } pa)$, i.e., the maximum length of these paths.

Example 7. *Figure 9 is an example where Frontier $F = \{C_z, C_x^L, C_x^R\}$.*

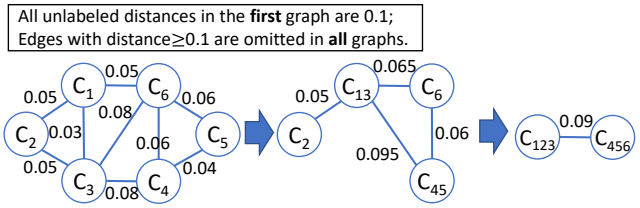

Figure 11: An example that applies MutualNN to six items. $\theta = 0.08$. MutualNN finishes in two iterations.

Regarding C_x, there are three valid paths in PA: (C_z, C_x), (C_x^L, C_x), and (C_x^R, C_x). So $len(C_x, F) = 1$. Note that (C_z, C_x^L, C_x) is not *a valid path in PA because the second cluster C_x^L is also in F, violating the definition.*

Regarding C_y, there are four valid paths in PA: (C_z, C_y), (C_z, C_x, C_y), (C_x^L, C_x, C_y), and (C_x^R, C_x, C_y). So $len(C_y, F) = 2$.

Theorem 3. *Given a dataset, the number of iterations of* MutualNN *is len_{max} of the DAG.*

The idea is to prove that each iteration of MutualNN generates all clusters C_x with $len(C_x, F) = 1$. Before that, we need to prove a lemma:

Lemma 1. *Given a Merged cluster C_x whose direct children are C_x^L and C_x^R, if there exists a C_y such that $w(C_y, C_x^L) < w(C_x^L, C_x^R)$ OR $w(C_y, C_x^R) < w(C_x^L, C_x^R)$, the DAG must have a path including two edges (C_y, C_y^P) and (C_y^P, C_x).*

The lemma above means, if C_y is closer to any of C_x's children than the child's sibling, there must be a path from C_y to its parent C_y^P to C_x. The detailed proof is in [55].

Now we can prove Theorem 3.

Proof. In each iteration, given a C_x that satisfies $len(C_x, F) = 1$, C_x^L and C_x^R must be mutual nearest in F. Otherwise, if there exists a $C_y \in F$ that is closer to C_x^L (or C_x^R), there should be a path (C^y, C_y^P, C_x) according to Lemma 1. Since $C_y \in F$, C_y^P must be Ungenerated, resulting in $len(C_x, F) \geq 2$, violating $len(C_x, F) = 1$, contradiction. So C_x will be generated when $len(C_x, F) = 1$.

So after iteration t, MutualNN generates all the clusters with $len(C_x) \leq t$. Therefore, MutualNN generates the whole DAG after len_{max} iterations, which is the length of the longest paths. □

Example 8 (MutualNN). *Figure 11 is an example that applies* MutualNN *to six items. It takes two iterations, which equals the length of the longest paths in Cluster DAG in Figure 6.*

4.3 Number of Iterations of PACk

In this section, we assume metric space in which the distance function satisfies triangle inequality, then we can prove that our algorithm takes much fewer iterations than MutualNN. But note that metric space is *not* a requirement of the correctness of PACk.

First we prove that, if the $dist(\cdot, \cdot)$ between *individual* items satisfy triangle inequality, the $dist(\cdot, \cdot)$ between *clusters* also satisfy triangle inequality in Average-Linkage.

Theorem 4. $dist(C_i, C_k) \leq dist(C_i, C_j) + dist(C_j, C_k)$

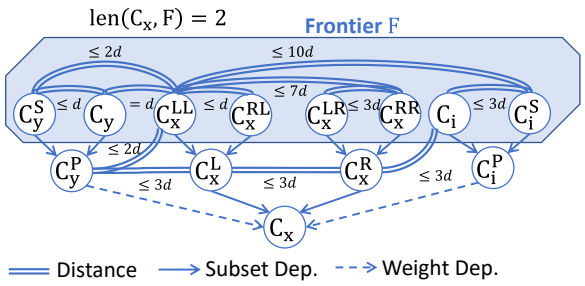

Figure 12: When $len(C_x, F) = 2$, C_x's children are not in F, and C_x^{LL}'s nearest neighbor in F is C_y, Algorithm 3 will generate C_x. (Double lines indicate distances.)

We prove it by enumerating the distances (detail in [55]).

Theorem 5. *Algorithm 3 finishes in $\lceil len_{max}/2 \rceil$ iterations when it uses the Distance-based Partitioning (Algorithm 5) and $dist(\cdot,\cdot)$ is a metric.*

Similar to the proof for `MutualNN`, we can prove that each iteration in Algorithm 3 generates all clusters C_x with $len(C_x, F) \leq 2$.

Lemma 2. *In each iteration, Algorithm 3 generates all clusters C_x that satisfy $len(C_x, F) \leq 2$ where F is the frontier.*

Now we can prove Lemma 2 as the following proof sketch. More detail is in [55].

Proof. (Sketch) When $len(C_x, F) = 1$ (Figure 10), similar to the proof of `MutualNN`, C_x^L and C_x^R must be mutual nearest neighbors in F. Then C_x will be generated in the iteration.

When $len(C_x, F) = 2$, depending on whether C_x's children are in F, there are three cases. In each case, we will prove that C_x's dependents will be in C_x's partition so that C_x's children will become mutual nearest and be merged. The detailed proof is in [55].

Here we briefly present one situation in Case (3), which shows why we set the radius to 10 times the distance between C_x^{LL} and its second nearest neighbor. In Case (3), when C_x^{LL}'s second nearest neighbor is not C_x^R's child. Let C_x^{LL}'s second nearest neighbor be C_y, and $dist(C_x^{LL}, C_y) = d$. Then we can bound many distances as in Figure 12. The longest distance is $dist(C_x^{LL}, C_i^S) \leq 10d$, where C_i^S is a cluster whose parent C_i^P has a weight dependency (C_i^P, C_x). □

Using Lemma 2, we now prove Theorem 5.

Proof. (Sketch) We can prove by induction that, in i-th iteration, Algorithm 3 generates clusters C_x with $len(C_x) \leq 2i$. So in the $\lceil len_{max}/2 \rceil$-th iteration, all clusters are generated. □

We now derive the following corollary for Algorithm 6.

Corollary 1. *If $dist(\cdot,\cdot)$ is a metric, and k_N and k_L are large enough that every $|\mathscr{C}_h|$ and $|\mathcal{L}(C_x)|$ in Algorithm 6 are no less than the corresponding $|\mathscr{C}_h|$ and $|\mathcal{L}(C_x)|$ in Algorithm 5 respectively, Algorithm 3 using Partitioning with Size Limit (Algorithm 6) finishes in $\lceil len_{max}/2 \rceil$ iterations.*

4.4 Simplified Cost Model

In this section, we develop a simplified cost model for the running time τ in the distributed system, and then compare the cost of `MutualNN` and `PACk`.

We define τ as the sum of running time of all iterations:

$$\tau = \sum_{i=1}^{\#iterations} (\tau_s(G_i) + \tau_c(G_i))$$

In Iteration i, where G_i is the input graph, the running time consists of two main parts: data shuffle time $\tau_s(G_i)$, and cpu time $\tau_c(G_i)$. Following the cost models for distributed systems like Spark [2] and Hadoop [23, 24] that assume almost uniform distribution of data[4], we further define:

$$\tau_s(G_i) = \frac{size(G_i)}{networkSpeed \cdot \#executors}$$

where $size(G_i)$ is the size of the graph in bytes, $networkSpeed$ is the number of bytes the system can shuffle in every unit time, and $\#executors$ is the number of executors; and

$$\tau_c(G_i) = \frac{oper(G_i) \cdot timePerOperation}{\#executors}$$

where $oper(G_i)$ is the number of cpu operations to process the graph and $timePerOperation$ is the time for each cpu operation. In practice, $\#executors$ is much less than the number of partitions.

Example Deduplication Scenario. Now we present the cost in a simplified scenario. Assume a set of original items S, each of which has du duplicates. In practice, each item is very similar to its duplicates (e.g., distance ≤ 0.05), but its duplicates are less similar to each other (e.g., distance > 0.05). This is a "hub-spoke" graph where each original item is a hub and its duplicates are spokes. Each item and its duplicates can be viewed as a *group*. To simplify the analysis, further assume that groups are very different from one another, which means no edge across groups as their distances are above the threshold. Let the input graph be G_1 with initial singleton clusters $C(G_1)$ and edge weights $W(G_1)$. So $|C(G_1)| = |S| \cdot (du+1)$, and $|W(G_1)| = |S| \cdot \frac{(du+1)du}{2}$.

Next, we use the Big Theta (Θ) notation to represent the asymptotic complexity.

`MutualNN`. In each iteration, all duplicates find their nearest neighbor (NN) as the original item, but the original item's NN is only one duplicate. So `MutualNN` merges only one pair within each group. Therefore, for the i-th iteration, $C(G_i) = |S| \cdot (du+2-i)$, and $|W(G_i)| = |S| \cdot \frac{(du+2-i)(du+1-i)}{2}$. It takes du iterations to finish.

Each iteration has 3 steps: (1) Find NN through an aggregation. (2) Find mutual NN through a join of the NN pairs. (3) Merge mutual NN and their edges. More detailed cost is in [55], and we only present the total costs here due to space limit:

$$size(G_i) = \Theta((du+2-i)^2|S|) \cdot sizePerEdge$$
$$oper(G_i) = \Theta((du+2-i)^2|S|)$$

[4] We empirically evaluate the performance on skewed data in Section 5.1.

Therefore,

$$\tau = \sum_{i=1}^{\#iterations} (\tau_s(G_i) + \tau_c(G_i))$$

$$= \Theta(du^3)|S| \cdot \left(\frac{sizePerEdge}{networkSpeed \cdot \#executors} + \frac{timePerOperation}{\#executors} \right)$$

Intuitively, the τ contains $\Theta(du^3)|S|$ because the graph has $\Theta(du^2)|S|$ edges initially and has $\Theta((du-i)^2)|S|$ edges after the i-th iteration. The algorithm stops after du iterations, resulting in $\sum_{i=1}^{du} \Theta((du-i)^2)|S| = \Theta(du^3)|S|$ complexity.

PACk. In practice, since *identical* items (like "Seattle" vs "Seattle") are usually aggregated together before clustering happens, so the number of different duplicates (like "Seattle" vs misspelled "Seattt**t**le") is usually very small (e.g., $du \leq 100$). Thus, k_N and k_L are very likely $\geq du$, meaning each item and its duplicates end up in the same partition. So the algorithm finishes in only 1 iteration.

The only iteration has 3 major steps on G_1 (due to space limit, the detailed minor steps are in [55]): partitioning, distance-aware merging, and cluster integration with graph update. The total cost of all steps is:

$$size(G_1) = \Theta(du^2 \cdot |S|) \cdot sizePerEdge$$

$$oper(G_1) = \Theta(|S| \cdot (du^2 \log du))$$

Therefore,

$$\tau = \tau_s(G_1) + \tau_c(G_1)$$

$$= \Theta(du^2)|S| \cdot \frac{sizePerEdge}{networkSpeed \cdot \#executors} + \Theta(du^2 \log du)|S| \cdot \frac{timePerOperation}{\#executors}$$

Comparison. PACk is less expensive than MutualNN in both data shuffle and CPU time in this example. In data shuffle, which is usually orders of magnitude slower than CPU computation, PACk's $\Theta(du^2)$ term in complexity saves much more time than MutualNN's $\Theta(du^3)$. In CPU computation, PACk's $\Theta(du^2 \log du)$ is also better than MutualNN's $\Theta(du^3)$. The $\Theta(\log du)$ term is usually very small in practice (e.g., ≤ 10), because it is bounded by k_N and k_L. So PACk is more efficient in this example.

In other real-world graphs, the graph structure is more complex than this example scenario. There could be random edges with long distances across different groups, which are difficult to be captured by the cost model. So we present experimental evaluations on the real-world data to compare the performance. As we will see in the evaluation, PACk still notably out-performs the state-of-the-art MutualNN on real-world datasets.

5 EVALUATION

In this section we evaluate PACk's (1) performance, (2) scalability and (3) sensitivity to key parameters of the algorithm and compare it with the state-of-the-art algorithm MutualNN.

Dataset. We evaluated PACk (Section 3) on six real, five modified-real and one synthetic datasets shown in Table 1.

The six real datasets are Song, Cite, LiveJ, Wiki, Urban, and Bright. Song and Cite use Jaccard distance and are from the Magellan Data Repository [8]. Song has the titles, releases, and artist

Table 1: Datasets. Numbers in parentheses are of skewed data.

Data	Type	#Items	#Edges
Song	Real	1.0M	1.1M
Cite		4.3M	1.9M
LiveJ		4.8M	69.0M
Wiki		1.8M	28.5M
Urban		0.4M	6.5M
Bright		0.8M	24.0M
Real1	Modified-Real	258.4M (35.4M)	680.4M (20.9M)
Real2		76.8M (10.5M)	358.3M (34.9M)
Real3		19.7M (2.7M)	107.7M (3.2M)
USPS		99.5M (13.6M)	427.4M (39.4M)
IMDB		11.4M (1.6M)	41.0M (1.2M)
FEBRL	Synthetic	10.0M (10.0M)	124.7M (11.6M)

names of 1.0M songs. We tokenize each string into a set of tokens, and then keep the pairs of sets with Jaccard distances ≤ 0.4 to get 1.1M edges. Cite is the union of Citeseer and DBLP paper titles containing 4.3M items. Similar to Song, we also tokenize and keep the pairs with Jaccard distance ≤ 0.4 to get 1.9M edges. LiveJ and Wiki are large graphs from Stanford Large Network Dataset Collection [29]. We assign random distances following uniform distribution in $(0, 1]$ for the edges. LiveJ [1, 30] is an online social network with 4.8M items and 69.0M edges. Wiki [27, 58] is the hyperlink network between articles in the most popular categories. It has 1.8M items and 28.5M edges. Urban and Bright use Euclidean distance. Urban is a dataset of road accidents within Great Britain urban areas from the UCI Machine Learning Repository [15]. We keep pairs within 0.5 km to get a graph with 0.4M items and 6.5M edges. Bright [5] is a location-based social network dataset from [29]. We retrieve the distinct locations and keep pairs within 0.5 km to get a graph with 0.8M items and 24.0M edges.

We additionally use six datasets to freely vary the number of duplicates, making the task more challenging. They include five modified-real datasets (Real1, Real2, Real3, USPS, and IMDB) and one synthetic dataset (FEBRL). The Real1, Real2, and Real3 are proprietary datasets used by three different applications in Microsoft. They include names, addresses and other contact information of organizations. USPS is a dataset of addresses in the United States, from which we extract distinct concatenation of street address, city, state and zip code. IMDB contains movie data from the Internet Movie Data Base, in particular the Title, Directors and Genres columns. FEBRL [6] is a synthetic dataset generated using an open source tool. We extract person name, address, suburb, state, and postcode columns from it.

We generate duplicates for these six datasets following uniform and skewed distributions. In the uniform setting, we create 9 similar items for each original item by inserting or deleting random characters. Then we perform a self-join on the data and keep the pairs with Jaccard distance ≤ 0.4. As shown in Table 1, the input graphs has 10.0 to 258.4 million items, with 41.0 to 680.4 million edges. In the skewed setting, we make the number of duplicates follow the Zipfian distribution where the exponent= 3. Then we again keep the pairs with Jaccard distance ≤ 0.4. As shown in the parentheses in Table 1, the input graphs has 1.6 to 35.4 millions items, with 1.2 to 39.4 million edges. (We also evaluate the performance when Jaccard distance ≤ 0.2 in the uniform setting in [55].)

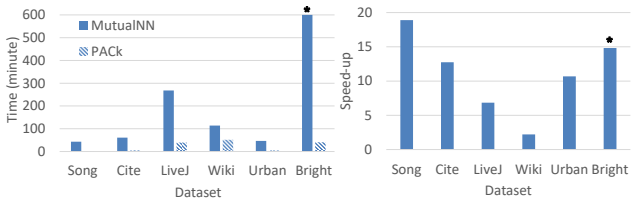

Figure 13: PACk is much more efficient than MutualNN on real datasets; The speed-up ranges from 2.2× to 18.9×. (*: MutualNN exceeds 10 hours on Bright, meaning the speed-up on Bright ≥ 14.8×.)

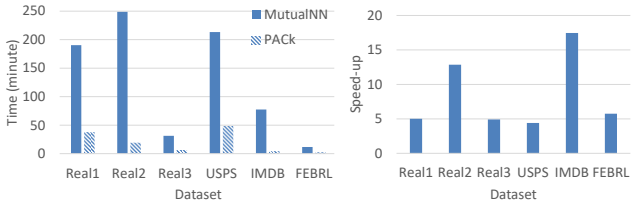

Figure 14: PACk is much more efficient than MutualNN on modified-real and synthetic datasets when number of duplicates follows uniform distribution; The speed-up ranges from 4.4× to 17.4×.

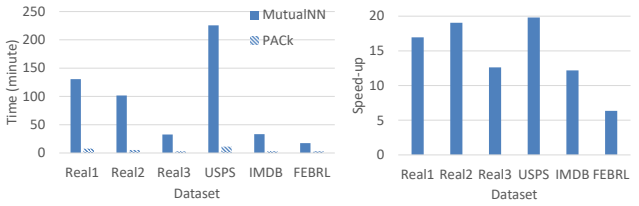

Figure 15: PACk is much more efficient than MutualNN on modified-real and synthetic datasets when number of duplicates follows Zipfian distribution; The speed-up ranges from 6.4× to 19.8×.

Baseline. We compare with the state-of-the-art algorithm that merges mutual nearest neighbors (MutualNN in Section 2.3) in each iteration. The idea was proposed in [38] and later simplified and implemented in [13].

Setting. We conduct experiments on Azure Databricks Spark clusters. The cluster has 16 D8s_v3 virtual machines. Each VM has 8 cores and 32 GB memory, running Apache Spark 2.4.3 and Scala 2.11. The default values of k_N and k_L are 500.

5.1 Performance

Our algorithm is more efficient than the state-of-the-art MutualNN on various datasets. Specifically, we see 2.2× to 18.9× speed-up on the six real datasets (Figure 13), 4.4× to 17.4× speed-up on the six modified-real and synthetic datasets in uniform setting (Figure 14), 6.4× to 19.8× speed-up on the modified-real and synthetic datasets in Zipfian setting (Figure 15).

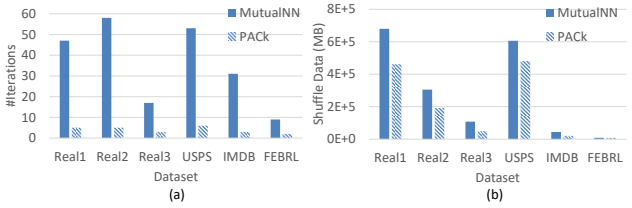

Figure 16: PACk is more efficient than MutualNN because (a) PACk takes fewer iterations; (b) PACk shuffles less data.

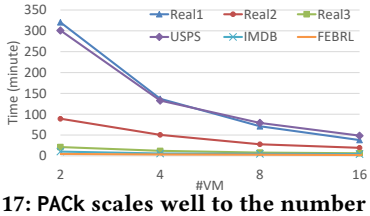

Figure 17: PACk scales well to the number of VMs.

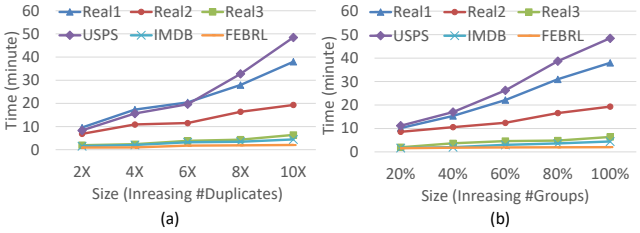

Figure 18: PACk scales almost linearly with the size of data in terms of (a) #duplicates per group; (b) #groups.

PACk is more efficient than MutualNN because PACk finishes in fewer iterations and shuffles less data. For example, on the six modified-real and synthetic datasets in uniform setting, PACk takes only 8.6% to 22.2% of iterations in MutualNN (Figure 16a), and PACk shuffles 45.2% to 88.2% of the data in MutualNN (Figure 16b).

5.2 Scalability

In this experiment we evaluate the scalability of PACk. We vary the number of VMs from 2 to 16. As Figure 17 illustrates, PACk scales well when we vary the number of VMs. For example, on Real1 dataset, the running time using 4 VMs is roughly half the time using 2 VMs. As we increase the VMs, the curve of running time flattens. It is because the merges within each iteration are almost exhaustively parallelized, while the data shuffle between iterations gradually dominates the running time.

Next, we vary the number of items per duplicate group from 2 to 10 (i.e. duplicates from 1 to 9). As Figure 18a shows, PACk scale almost linearly with the number of duplicates.

Next, we vary the number of duplicate groups from 20% to 100% of the original input (fixing the number of items per group at 10). PACk scale almost linearly with the number of groups (Figure 18b).

5.3 Parameter Sensitivity

In this experiment we study how parameters in PACk impact performance. Recall that, in Algorithm 6, k_N controls the size of each

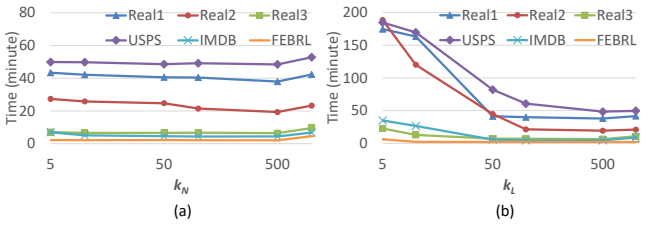

Figure 19: (a) When $k_L = 500$, time slightly decreases as k_N grows because more neighbors are included in a partition; but increases for overly large $k_N = 1000$. (b) When $k_N = 500$, time decreases as k_L grows because more edges are included in a partition; but increases for overly large $k_L = 1000$.

partition, and k_L controls the size of each edge lists. We begin by varying the parameters to see how running time changes.

We first fix $k_L = 500$ and vary k_N in $\{5, 10, 50, 100, 500, 1000\}$. As Figure 19a shows, the running time decreases slightly as k_N grows, because more neighbors are included in a partition and more merges can be done. The time then increases slightly for overly large $k_N = 1000$ as too many neighbors only adds the shuffle cost.

We then fix $k_N = 500$ and vary k_L in $\{5, 10, 50, 100, 500, 1000\}$. As Figure 19b shows, the running time decreases as k_L grows, because more edges are included in a partition and more merges can be done. The time then increases slightly for overly large $k_L = 1000$ because too many edges only adds the shuffle cost.

6 RELATED WORK

The study of Agglomerative Hierarchical Clustering (AHC) dates back to 1950s with a focus on centralized algorithms [21, 26, 28, 39, 40, 49]. The idea is to initially treat each node as a singleton cluster, and then iteratively merge small clusters into bigger clusters based on their pair-wise distances. When two clusters are merged, their distance to a third cluster is updated according to their individual distances. There exist several strategies. For example, Single-linkage [22, 48] takes the minimum distance; Complete-linkage [12] takes the maximum distance; Average-linkage [50] uses the unweighted or weighted average distance. Others strategies include Minimax [4]. These papers assume a centralized AHC that stores the graph in memory. They do not scale to large datasets since they are limited by the compute resources (CPU and memory) available on a single machine.

Researchers have also developed distributed AHC algorithms. In 2005, Ding and He [13] proposed multi-level hierarchical clustering (`MutualNN`), which merges mutually nearest cluster pairs concurrently. They proved that `MutualNN` generates the same result as centralized AHC as long as the distance function satisfies Cluster Aggregate Inequality, which is a stronger version of "reducibility property" [38] proposed in 1980s. Sun et al. [51] implemented AHC using Map-Reduce. Their algorithm collects top K edges with maximal weights from worker nodes to the driver node and merges as many pairs as possible in the centralized driver node. Its scalability is limited to the data size that can fit in the driver node. As a comparison, `PACk` performs clustering distributedly in worker nodes, which has better scalability. Zhang et al. [60, 61] solved AHC under Euclidean distance. They utilize a quad-tree or kd-tree to partition vertices in the Euclidean space, cluster them within each partition, and finally merge clusters. Their technique cannot be generalized to other distance functions like cosine, Jaccard, etc.

Some *approximate* algorithms reduce the running time by sacrificing accuracy. Ma et al. [35] merge multiple clusters in each iteration as long as their distances are within a predefined threshold. A few papers [9, 10, 31] merge edges whose distances are less than an increasing threshold in iterations. Tanaseichuk et al. [52] applies K-means to group items into clusters first and then uses AHC within each cluster. Gilpin et al. [17] group items in Euclidean space into buckets and then apply AHC within each bucket. These approximate algorithms produce different clustering results than conventional AHC does.

Another line of work focuses on special cases of AHC or special computational settings. Single-linkage as a special case of AHC is similar to the typical minimum spanning tree problem. Several efficient distributed single-linkage algorithms have been proposed [3, 25, 42, 45, 56]. Dash et al. [10] proposed an algorithm using shared memory architecture. Some researchers developed a parallel AHC on shared-memory [43] or SIMD machines [32, 33, 46].

Other partitioning strategies exist in some graph systems. For example, Pregel [36] performs message passing between vertices to perform computation on a graph. Each vertex and its neighbors can be viewed as a trivial partition, and its message passing can be supported as data shuffle on Spark using GraphX [19]. In addition, Distributed GraphLab [34] performs edge-cut and PowerGraph [18] performs vertex-cut to partition graphs. These strategies do not leverage the domain knowledge for AHC such as mutual nearest neighbors and distance bounds. In comparison, `PACk` is particularly designed for AHC and has better performance both analytically and in practice.

Many academic and industrial tools support centralized AHC. The examples include MATLAB, R [41], ScikitLearn [44] and SciPy [53] in Python, etc. Similar to the centralized AHC in the papers above, the scalability of these tools is limited to the size of data that can fit in a single node.

7 CONCLUSION

We propose an efficient, distributed agglomerative hierarchical clustering (AHC) algorithm `PACk` that scales well to large data sets. `PACk` derives its efficiency from novel distance-based partitioning and distance-aware merging techniques that enable significantly more merges to be performed in parallel, thereby reducing the number of iterations required as well as the data shuffle cost. We implement `PACk` on Spark, and compare it to the state-of-the-art approach. Our evaluation on several synthetic and real-world datasets including Microsoft Dynamics 365 shows that `PACk` achieves consistently large speedups ranging from 2× to 19× with a median of 9×.

ACKNOWLEDGMENTS

We thank Silu Huang, Wentao Wu, Chi Wang, and Arnd Christian König for their insightful comments on the paper, and Swapna Akula, Katchaguy Areekijseree, Meiyalagan Balasubramanian, and Lengning Liu for their design, implementation, and optimization in Microsoft Dynamics 365 Customer Insights.

REFERENCES

[1] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, page 44–54, New York, NY, USA, 2006. Association for Computing Machinery.

[2] L. Baldacci and M. Golfarelli. A cost model for spark sql. *IEEE Transactions on Knowledge and Data Engineering*, 31(5):819–832, 2019.

[3] M. Bateni, S. Behnezhad, M. Derakhshan, M. Hajiaghayi, R. Kiveris, S. Lattanzi, and V. Mirrokni. Affinity clustering: Hierarchical clustering at scale. In *Advances in Neural Information Processing Systems*, pages 6864–6874, 2017.

[4] J. Bien and R. Tibshirani. Hierarchical clustering with prototypes via minimax linkage. *Journal of the American Statistical Association*, 106(495):1075–1084, 2011.

[5] E. Cho, S. A. Myers, and J. Leskovec. Friendship and mobility: user movement in location-based social networks. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1082–1090, 2011.

[6] P. Christen. Febrl - an open source data cleaning, deduplication and record linkage system with a graphical user interface. In *Proceedings of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, page 1065–1068, 2008.

[7] W. W. Cohen and J. Richman. Learning to match and cluster large high-dimensional data sets for data integration. In *Proc. SIGKDD*, page 475–480, 2002.

[8] S. Das, A. Doan, P. S. G. C., C. Gokhale, P. Konda, Y. Govind, and D. Paulsen. The magellan data repository. https://sites.google.com/site/anhaidgroup/useful-stuff/data.

[9] M. Dash, H. Liu, P. Scheuermann, and K. L. Tan. Fast hierarchical clustering and its validation. *Data Knowl. Eng.*, 44(1):109–138, Jan. 2003.

[10] M. Dash, S. Petrutiu, and P. Scheuermann. Ppop: Fast yet accurate parallel hierarchical clustering using partitioning. *Data Knowl. Eng.*, 61(3):563–578, June 2007.

[11] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1370–1386, 2004.

[12] D. Defays. An efficient algorithm for a complete link method. *The Computer Journal*, 20(4):364–366, 1977.

[13] C. Ding and X. He. Cluster aggregate inequality and multi-level hierarchical clustering. In *Knowledge Discovery in Databases: PKDD*, pages 71–83, 2005.

[14] G. M. Downs and J. M. Barnard. Clustering methods and their uses in computational chemistry. *Reviews in computational chemistry*, 18:1–40, 2002.

[15] D. Dua and C. Graff. UCI machine learning repository, 2017.

[16] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein. Cluster analysis and display of genome-wide expression patterns. *Proceedings of the National Academy of Sciences*, 95(25):14863–14868, 1998.

[17] S. Gilpin, B. Qian, and I. Davidson. Efficient hierarchical clustering of large high dimensional datasets. In *Proc. CIKM*, page 1371–1380, 2013.

[18] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 17–30, 2012.

[19] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613, 2014.

[20] P. Govender and V. Sivakumar. Application of k-means and hierarchical clustering techniques for analysis of air pollution: A review (1980–2019). *Atmospheric Pollution Research*, 11(1):40–56, 2020.

[21] J. C. Gower. A comparison of some methods of cluster analysis. *Biometrics*, 23(4):623–637, 1967.

[22] J. C. Gower and G. J. Ross. Minimum spanning trees and single linkage cluster analysis. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 18(1):54–64, 1969.

[23] H. Herodotou. Hadoop performance models. Technical report, http://www.cs.duke.edu/starfish/files/hadoop-models.pdf, 2011.

[24] H. Herodotou and S. Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proc. VLDB Endow.*, 4(11):1111–1122, Aug. 2011.

[25] C. Jin, R. Liu, Z. Chen, W. Hendrix, A. Agrawal, and A. Choudhary. A scalable hierarchical clustering algorithm using spark. In *Proc. BIGDATASERVICE*, page 418–426, 2015.

[26] S. C. Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[27] C. Klymko, D. F. Gleich, and T. G. Kolda. Using triangles to improve community detection in directed networks. In *The Second ASE International Conference on Big Data Science and Computing, BigDataScience*, 2014.

[28] G. N. Lance and W. T. Williams. A General Theory of Classificatory Sorting Strategies: 1. Hierarchical Systems. *The Computer Journal*, 9(4):373–380, 1967.

[29] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[30] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1):29 – 123, 2009.

[31] K. Li, Y. He, and K. Ganjam. Discovering enterprise concepts using spreadsheet tables. In *SIGKDD*, page 1873–1882, 2017.

[32] X. Li. Hierarchical clustering on simd machines with alignment network. In *Proceedings CVPR '89: IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 660–665, 1989.

[33] X. Li. Parallel algorithms for hierarchical clustering and cluster validity. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 12(11):1088–1092, 1990.

[34] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, Apr. 2012.

[35] X.-L. Ma, H.-F. Hu, S.-F. Li, H.-M. Xiao, Q. Luo, D.-Q. Yang, and S.-W. Tang. Dhc: Distributed, hierarchical clustering in sensor networks. *Journal of Computer Science and Technology*, 26:643–662, 07 2011.

[36] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[37] A.-A. Mamun, T. Mi, R. Aseltine, and S. Rajasekaran. Efficient sequential and parallel algorithms for record linkage. *Journal of the American Medical Informatics Association*, 21(2):252–262, 2014.

[38] F. Murtagh. Complexities of hierarchic clustering algorithms: state of the art. *Computational Statistics Quarterly*, 1(2):101–113, 1984.

[39] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2(1):86–97, 2012.

[40] F. Murtagh and P. Contreras. Algorithms for hierarchical clustering: an overview, ii. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 7(6):e1219, 2017.

[41] D. Müllner. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software, Articles*, 53(9):1–18, 2013.

[42] V. Olman, F. Mao, H. Wu, and Y. Xu. Parallel clustering algorithm for large data sets with applications in bioinformatics. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 6(2):344–352, 2009.

[43] C. F. Olson. Parallel algorithms for hierarchical clustering. *Parallel Computing*, 21(8):1313 – 1325, 1995.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] S. Rajasekaran. Efficient parallel hierarchical clustering algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 16(6):497–502, 2005.

[46] E. M. Rasmussen and P. Willett. Efficiency of hierarchic agglomerative clustering using the icl distributed array processor. *Journal of Documentation*, 45:1–24, 1989.

[47] A. Shepitsen, J. Gemmell, B. Mobasher, and R. Burke. Personalized recommendation in social tagging systems using hierarchical clustering. In *Proceedings of the 2008 ACM Conference on Recommender Systems*, RecSys '08, page 259–266, New York, NY, USA, 2008. Association for Computing Machinery.

[48] R. Sibson. SLINK: An optimally efficient algorithm for the single-link cluster method. *The Computer Journal*, 16(1):30–34, 1973.

[49] P. H. Sneath. The application of computers to taxonomy. *Microbiology*, 17(1):201–226, 1957.

[50] R. Sokal and C. Michener. *A Statistical Method for Evaluating Systematic Relationships*. University of Kansas science bulletin. University of Kansas, 1958.

[51] T. Sun, C. Shu, F. Li, H. Yu, L. Ma, and Y. Fang. An efficient hierarchical clustering method for large datasets with map-reduce. In *2009 International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 494–499, 2009.

[52] O. Tanaseichuk, A. Hadj Khodabakhshi, D. Petrov, J. Che, T. Jiang, B. Zhou, A. Santrosyan, and Y. Zhou. An efficient hierarchical clustering algorithm for large datasets. *Austin Journal of Proteomics, Bioinformatics*, 2(1):1–6, 2015.

[53] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[54] F. Wang, J. Li, J. Tang, J. Zhang, and K. Wang. Name disambiguation using atomic clusters. In *Proc. WAIM*, pages 357–364, 2008.

[55] Y. Wang, V. Narasayya, Y. He, and S. Chaudhuri. An efficient partition-based distributed agglomerative hierarchical clustering algorithm for deduplication.

Technical report, https://www.microsoft.com/en-us/research/publication/tech-report-pack/, 2021.

[56] C.-H. Wu, S.-J. Horng, and H.-R. Tsai. Efficient parallel algorithms for hierarchical clustering on arrays with reconfigurable optical buses. *J. Parallel Distrib. Comput.*, 60(9):1137–1153, Sept. 2000.

[57] W. Wu, C. Yu, A. Doan, and W. Meng. An interactive clustering-based approach to integrating source query interfaces on the deep web. In *Proc. SIGMOD*, page 95–106, 2004.

[58] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich. Local higher-order graph clustering. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 555–564, New York, NY, USA, 2017. Association for Computing Machinery.

[59] V. Zappala, A. Cellino, P. Farinella, and Z. Knezevic. Asteroid families. i-identification by hierarchical clustering and reliability assessment. *The Astronomical Journal*, 100:2030–2046, 1990.

[60] W. Zhang, G. Zhang, X. Chen, Y. Liu, X. Zhou, and J. Zhou. Dhc: A distributed hierarchical clustering algorithm for large datasets. *Journal of Circuits, Systems and Computers*, 28(04):1950065, 2019.

[61] W. Zhang, G. Zhang, Y. Wang, Z. Zhu, and T. Li. Nnb: An efficient nearest neighbor search method for hierarchical clustering on large datasets. In *IEEE ICSC 2015*, pages 405–412, 2015.

[62] Y. Zhao, G. Karypis, and U. Fayyad. Hierarchical clustering algorithms for document datasets. *Data mining and knowledge discovery*, 10(2):141–168, 2005.