Cluster-and-Conquer: When Randomness Meets Graph Locality

George Giakkoupis, Anne-Marie Kermarrec, Olivier Ruas, François Taïani

Cluster-and-Conquer: When Randomness Meets Graph Locality

George Giakkoupis
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
george.giakkoupis@inria.fr

Anne-Marie Kermarrec
EPFL
Lausanne, Switzerland
anne-marie.kermarrec@epfl.ch

Olivier Ruas
Inria, Univ. Lille
Lille, France
olivier.ruas@inria.fr

François Taïani[†]
Univ Rennes, Inria, CNRS, IRISA
Rennes, France
francois.taiani@irisa.fr

***Abstract*—K-Nearest-Neighbors (KNN) graphs are central to many emblematic data mining and machine-learning applications. Some of the most efficient KNN graph algorithms are incremental and local: they start from a random graph, which they incrementally improve by traversing neighbors-of-neighbors links. Unfortunately, the initial random graph exhibits a poor graph locality, leading to many unnecessary similarity computations. In this paper, we remove this drawback with Cluster-and-Conquer (C^2 for short). Cluster-and-Conquer boosts the starting configuration of greedy algorithms thanks to a novel lightweight clustering mechanism, dubbed FastRandomHash. FastRandomHash leverages randomness and recursion to pre-cluster similar nodes at a very low cost. Our extensive evaluation on real datasets shows that Cluster-and-Conquer significantly outperforms existing approaches, including LSH, yielding speed-ups of up to $\times 4.42$ and even improving the KNN quality.**

***Index Terms*—KNN graph, Big Data**

I. Introduction

k-Nearest-Neighbors (KNN) graphs[1] play a fundamental role in many emblematic data-mining and machine-learning applications ranging from classification [1], [2] to recommender systems [3]–[5]. A KNN graph connects each node of a dataset to its k closest counterparts (its *neighbors*), according to some application-dependent similarity metric. In many applications, this similarity is computed from a second set of entities, termed *items*, associated with each node. (For instance, if nodes are users, items might represent the websites they have visited.) Despite being one of the simplest models in data analysis, computing an exact KNN graph remains extremely costly, incurring, for instance, a quadratic number of similarity computations under a brute-force strategy.

Many applications, however, only require a reasonable approximation of a KNN graph, as long as this approximation can be produced rapidly. This is, for instance, true of online news recommenders, in which the use of fresh data is of utmost importance, or of machine learning techniques that use the KNN graph as a first preliminary step [6], [7]. Existing approximate KNN graph algorithms essentially fall into two families, that each use different strategies to drastically reduce the number of similarities they compute: *greedy incremental solutions* [3], [8]–[10], and *partition-based techniques* (that include the popular LSH algorithm [11], [12]).

Greedy incremental solutions are currently among the best performing KNN graph construction algorithms, and exploit a local incremental search: they start from an initial random k-degree graph, which they greedily improve by traversing neighbors-of-neighbors links. Although they generally perform best, greedy approaches are critically hampered by their initial random start: similar nodes that lie close to one another in the *final KNN graph* are connected in the *initial random graph* to dissimilar nodes. In other words, these greedy approaches present a poor *initial graph locality*: nodes that are similar tend to be separated by long paths in the initial graph. As a result, greedy algorithms must initially compute many similarities between unrelated nodes, which adds a costly overhead for little to no benefit.

Partition-based algorithms [6], [11], [12] avoid this problem by clustering nodes before solving locally the problem, under a classic divide-and-conquer strategy. Unfortunately, a clustering that is both good and efficient is difficult to achieve. LSH [11], [12] for instance relies on hash functions that tend to fragment sparse datasets with large dimensions (from $\sim 10^3$ to $\sim 10^5$ in our experiments), which are typical of many on-line applications manipulating users and items. Traditional clustering techniques such as k-means [13] are similarly ill-fitted, as they require many similarity computations, which are precisely what we seek to avoid.

In this paper, we reconcile both perspectives with **Cluster-and-Conquer** (C^2 for short), a KNN graph algorithm for item-based datasets that boosts its initial graph locality by exploiting a novel, fast and accurate clustering scheme, dubbed **FastRandomHash**. FastRandomHash does not require any similarity computations (similarly to LSH) while avoiding fragmentation (similarly to k-means). FastRandomHash leverages **fast random hash functions** and employs a **new recursive splitting mechanism** to balance clusters, for optimal parallelism, and minimal synchronization between the involved threads in a parallel implementation.

Our extensive evaluation of Cluster-and-Conquer shows that our approach significantly outperforms existing approaches, including LSH, yielding speed-ups up to $\times 4.42$ and even improving the KNN quality.

[†]Authors are listed in alphabetical order.

[1]Note that the problem of computing a complete KNN graph (which we address in this paper) is related but different from that of answering a sequence of KNN queries.

II. Cluster-and-Conquer

For ease of exposition, we consider in the following that nodes are *users* associated with *items* (e.g. web pages, movies).

A. Notations and problem definition

We note $U = \{u_1, ..., u_n\}$ the set of all users, and $I = \{i_1, ..., i_m\}$ the set of all items. The subset of items associated with user u (a.k.a. her *profile*) is noted $P_u \subseteq I$. P_u is generally much smaller than I (the universe of all items).

Our objective is to approximate a k-nearest-neighbor (KNN) graph over U (noted G_{KNN}) according to some similarity function $sim \in \mathbb{R}^{U \times U}$ computed over user profiles: $sim(u, v) = f_{sim}(P_u, P_v)$ where f_{sim} may be any similarity function over sets that is positively correlated with the number of common items between the two sets, and negatively correlated with the total number of items present in both sets. These requirements cover some of the most commonly used similarity functions in KNN graph construction applications, such as cosine or the Jaccard similarity. We use the Jaccard similarity in the rest of the paper [14]: $sim(u, v) = J(P_u, P_v) = \frac{|P_u \cap P_v|}{|P_u \cup P_v|}$

A KNN graph G_{KNN} connects each user $u \in U$ with a set $knn(u)$ (the 'KNN' of u for short) which contains the k most similar users to u, with respect to the similarity function $sim(u, -)$.

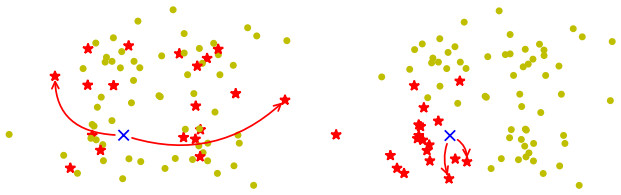
Computing an exact KNN graph is particularly expensive: a brute-force exhaustive search requires $O(|U|^2)$ similarity computations. Many scalable approaches, therefore, seek to construct *an approximate KNN graph* $\widehat{G}_{\text{KNN}}$, i.e., to find for each user u a neighborhood $\widehat{knn}(u)$ that is as close as possible to an exact KNN neighborhood [3], [9], [10]. The meaning of 'close' depends on the context, but in most applications, a good approximate neighborhood $\widehat{knn}(u)$ is one whose aggregate similarity (its *quality*) comes close to that of an exact KNN set $knn(u)$.

We capture how well the average similarity of an approximated graph $\widehat{G}_{\text{KNN}}$ compares against that of an exact KNN graph G_{KNN} with the *average similarity* of $\widehat{G}_{\text{KNN}}$, defined as the average similarity observed on the graph's edges. We then define the *quality* of $\widehat{G}_{\text{KNN}}$ as the ratio between its average similarity and the average similarity of an exact KNN graph G_{KNN}. A quality close to 1 indicates that the approximate KNN graph can replace the exact one with little impact in most applications.

With the above notations, we can summarize our problem as follows: for a given dataset $(U, I, (P_u)_{u \in U})$ and item-based similarity f_{sim}, we wish to compute an approximate $\widehat{G}_{\text{KNN}}$ in the shortest time with the highest overall quality.

B. Intuition

Some of today's best performing approaches for KNN graphs use a greedy strategy [3], [9], [10]: starting from a random initial k-degree graph, those algorithms incrementally seek to improve each user's neighborhood by exploring neighbors-of-neighbors links. Unfortunately, this random start tends to separate similar nodes by long paths in the initial graph. This disconnect between *similarity* (which captures 'closeness' from the application's point of view), and *initial graph-distance* (in terms of shortest path between two nodes), means greedy approaches initially suffer from a poor *graph locality*. Cluster-and-Conquer substantially improves the graph locality of its initial configuration, using an approximate, yet cheap and fast procedure. This basic principle is illustrated in Figure 1. Whereas standard greedy approaches (left) initially connect each user (in blue) to k random neighbors (in red), Cluster-and-Conquer partitions users into small sub-datasets (also called clusters in the following), in which similar neighbors can be selected (Fig. 1b), leading to much faster computation times. We then assign each cluster to a dedicated thread, that computes its (partial) KNN graph in isolation, improving parallelism. Merging all resulting partial graphs produces the final global KNN graph.



(a) Traditional greedy approaches **(b)** Cluster-and-Conquer

Fig. 1: Graph locality on a toy dataset (shown in 2D). A given user (in blue) starts with unrelated neighbors (in red) with traditional approaches (a). Cluster-and-Conquer ensures a much higher initial graph locality (b).

C. Cluster-and-Conquer: Overview

Building on the previous intuitions, Cluster-and-Conquer works in three steps, which we present in detail in the remaining of this section:

- **Step 1**: **Clustering**. The dataset is clustered in $t \times b$ clusters, using FastRandomHash functions, where t is the number of hash functions, and b the number of clusters per hash function. Clusters whose size exceeds N are recursively split.
- **Step 2**: **Scheduling and KNN graph computation**. The clusters are processed in parallel to produce a partial KNN for each of them. The parallel computation uses a greedy scheduling heuristic to balance work between computing cores.
- **Step 3**: **Merging**. The resulting partial KNN graphs are merged.

The resulting KNN graph is returned as the KNN graph of the whole dataset.

D. Step 1: Clustering with FastRandomHash

The FastRandomHash scheme first projects each item $i \in I$ onto a hash value $h(i)$ using a generative hash function $h : I \to [\![1, b]\!]$. The hash $H(u)$ of a user u is then taken as the minimum hash value among u's items:

$$H(u) = \min_{i \in P_u} h(i). \tag{1}$$

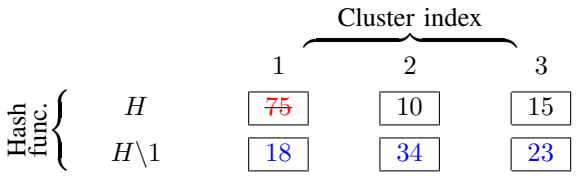


Fig. 2: Recursive splitting of clusters ($b = 3$). In the initial clustering (first line), obtained with H, the first cluster $C_H[1]$ exceeds the threshold of $N = 40$ users. It is split by applying $H\backslash 1$, which only keeps item hashes higher than 1. Users with no items being hashed to 2 or 3 remain in the first cluster.

$H(u)$ determines u's cluster under h, resulting in b clusters for each generative function, what we have termed a clustering configuration. We use t distinct generative functions, to produce t clustering configurations and a total of $t \times b$ clusters. The use of multiple hash functions reduces the risk that two similar nodes are never hashed into the same cluster, an event whose probability decreases exponentially with the number of hash functions t.

Balancing large clusters through recursive splitting. Using a minimum in the FastRandomHash function, unfortunately, introduces a bias towards the clusters of low indices. This bias is pervasive but particularly marked if highly popular items are hashed into one of the first clusters. In such a case, this cluster is likely to end up being much larger than the others in the same clustering configuration, many of which will be empty.

Highly unbalanced clusters tend to self-defeat the very purpose of the clustering step. This is because computing the partial KNN graph of a very large cluster can be almost as costly as that of the whole dataset. When this happens, the whole parallel computation is delayed, limiting the benefits of multi-threading. To avoid this situation, we recursively split overlarge clusters, by using the fact that FastRandomHash functions are extracted from hash values initially computed on items. More specifically, if a cluster C with index η_C is larger than a threshold parameter N, we compute a second FastRandomHash value $H\backslash\eta_C(u)$ for each of its users $u \in C$, by ignoring C's index η_C (i.e. the hash value that produced C) when computing the hash values of u's items:

$$H\backslash\eta_C(u) = \min_{i \in P_u,\, h(i) > \eta_C} h(i)$$

where h is the generative hash function that underlies the clustering configuration containing C. The users of C are then distributed among new clusters, one for each new hash value, with two exceptions. Users who have a single item (for whom $H\backslash\eta_C$ is undefined) and users who are alone in their new cluster remain in C. The resulting clusters are again recursively split if their size exceeds N. In summary, we split the users of a large cluster according to a second item, producing smaller and more refined clusters.

Figure 2 illustrates this recursive splitting strategy on a cluster configuration of $|U| = 100$ users, $b = 3$ and $N = 40$. The first line represents the initial clustering, obtained with H, *before* the splitting. The boxes represent the clusters and the number in each box the size of the cluster. This initial clustering is highly unbalanced: the first cluster contains most of the users (75) while the others are nearly empty. Since its size is higher than $N = 40$, the first cluster is split into new clusters, shown on the second line. The clusters of the second line are obtained using $H\backslash 1$, the FastRandomHash H keeping only hashes higher than 1, on the 75 users. The first cluster is composed of users with no items being hashed to 2 or 3. As none of the new clusters contains more than 40 users, the splitting stops. With the new clusters (shown in the second line), the new clustering configuration is more balanced, at the cost a few more clusters (5 instead of 3).

The lower the threshold size N, the more balanced the final t cluster configurations, thus the faster the computation. Still, small clusters increase the chance of similar users never appearing in the same cluster, potentially hurting the quality of the final global KNN graph. In practice, we choose $N = 2000$.

E. Step 2: Scheduling and local computation

Recursively splitting clusters reduces gross discrepancies between cluster sizes, but the final clusters might still remain unbalanced. To prevent an unbalanced workload among the threads of a parallel architecture, we apply some light-weight work scheduling. The clusters are stored in a synchronized, decreasing priority queue, ordered according to their size. We then use a basic thread pool to computes the KNN graph of each cluster in the queue, starting with the largest clusters and working down the priority queue until it becomes empty.

The partial KNN graph of each cluster C can be computed using any approach and does not need to be synchronized with any other computation. In our prototype, we use a hybrid solution that switches between a brute force approach and a greedy KNN graph algorithm depending on the number of users $|C|$ in the cluster. (In practice we use Hyrec [3], see Sec. IV-B, but any other KNN graph algorithm can be used.) To determine a threshold value for the switch, we estimate the expected number of similarity computations for each approach and pick the least expensive. The brute force approach computes $\frac{|C|\times(|C|-1)}{2}$ similarities, while Hyrec's number of similarities is bounded by $\frac{\rho\times k^2\times|C|}{2}$, where ρ is the number of iterations. As a result, if $|C| < \rho \times k^2$ we choose the brute force approach, Hyrec otherwise. In practice, we take $\rho = 5$. To further speed-up the computation, we use optimized versions of these algorithms that leverage a compact data structure [15] to provides a *fast-to-compute* estimation of Jaccard similarity values. In practice, this data structure, dubbed *GoldFinger* [15], summarizes each user's profile into a 64- to 8096-bit vector, which is then used to estimate Jaccard similarity values.

F. Step 3: Merging the KNN graphs

Once the cluster phase is over, we merge the partial KNN graphs obtained for each cluster, one by one, into a unique KNN graph knn. Merging is performed at the granularity of individual users. Each user appears in t different clusters and

is connected to up to $t \times k$ neighbors. For each cluster C, and each user u of C, the KNN neighborhood $knn'(u)$ of u in C's partial KNN graph is added to u's final neighborhood $knn(u)$, while only keeping the k best neighbors so far in $knn(u)$. While doing so we are careful to reuse similarity values, to avoid redundant computations. The resulting KNN graph knn is returned.

III. Theoretical Properties

The final neighbors of a user are selected from within the clusters in which this user appears. The quality of the final KNN graph is thus highly dependent on the ability of the hashing scheme to group similar users together. We now present two theorems that show that the probability of two users being hashed into the same bucket, before any splitting occurs, is proportional to their Jaccard similarity up to some small error introduced by collisions. The corresponding proofs can be found in the accompanying technical report [16].

Theorem 1. *The probability $\mathbb{P}[H(u_1) = H(u_2)]$ that two users u_1, u_2 obtain the same FastRandomHash value $H(\cdot)$ is lower bounded by the following inequality*

$$J_{1,2} - \frac{\kappa}{\ell} \leq \mathbb{P}[H(u_1) = H(u_2)], \tag{2}$$

where $J_{1,2} = J(P_1, P_2)$ is the Jaccard similarity between u_1's and u_2's profiles, P_1 and P_2; $\ell = |P_1 \cup P_2|$ is the joint size of the two profiles; $\kappa = \ell - |h(P_1 \cup P_2)|$ is the overall number of collisions occurring when projecting the two profiles onto $[\![1, b]\!]$, and $h(\cdot)$ is the generative hash function underpinning $H(\cdot)$. If we further assume $\kappa \leq \ell/2$, then we have

$$\mathbb{P}[H(u_1) = H(u_2)] \leq J_{1,2} + 3\frac{\kappa}{\ell} + O\left(\frac{\kappa}{\ell}\right)^2. \tag{3}$$

Theorem 1 states that FastRandomHash will tend to allocate the same hash to similar users, modulo some noise introduced by collisions. In other words, **the more similar two users are, the more likely they are to be allocated in the same clusters**. We now bound the effect of collisions with the following concentration bound.

Theorem 2. *The collision density κ/ℓ is upper bounded by the value $(1+d)\frac{\ell-1}{2b}$ with a probability bounded by the following formula*

$$\mathbb{P}\left[\frac{\kappa}{\ell} < (1+d)\frac{\ell-1}{2b}\right] \geq 1 - \left(\frac{e^d}{(1+d)^{(1+d)}}\right)^{\frac{\ell(\ell-1)}{2b}}, \tag{4}$$

where $d > 0$ is a real positive value, and the other variables are defined as in Theorem 1.

As an example, if we apply Theorems 1 and 2 to the case of $\ell = 256$, $b = 4096$ (some typical values of our experiments), and set $d = 0.5$, we obtain that

$$J_{1,2} - 0.078 \leq \mathbb{P}[H(u_1) = H(u_2)] \leq J_{1,2} + 0.234$$

with probability 0.998 over the space of all hash functions h. The left-hand side of the above equation impacts the quality of our approximation, by ensuring that pairs of similar users tend to be compared: such users show a high $J_{1,2}$ value, and have therefore a high probability to be hashed to the same bucket, and to be compared, since this probability is lower-bounded by a value which is close to their Jaccard similarity. Conversely, the right-hand side controls the performance of our approach, by lowering the chances of comparing dissimilar users (and thus performing superfluous computations): the probability of such a comparison taking place is upper-bounded by $J_{1,2}$ (which is low for dissimilar users) plus a constant due to collisions that depends on ℓ and b (0.234 in our example).

TABLE I: Description of the datasets used in our experiments

Dataset	Users	Items	Ratings > 3
MovieLens (ml)	138,362	22,884	12,195,566
AmazonMovies (AM)	57,430	171,356	3,263,050
DBLP	18,889	203,030	692,752
Gowalla (GW)	20,270	135,540	1,107,467

IV. Experimental Setup

A. Datasets

We use four publicly available users/items datasets (Table I) in which we keep only ratings higher than 3 in order to compute the Jaccard similarity. Furthermore, we restrict our study to users with at least 20 ratings (positive and negative ratings) to avoid dealing with users with not enough data (this problem, called the *cold start problem*, is generally treated separately [17]).

1) The MovieLens dataset: MovieLens (ml) [18] is an anonymous dataset containing movie ratings collected on-line between 1995 and 2015 by GroupLens Research [19]. The datasets contain movie ratings on a 0.5-5 scale by users who have at least performed more than 20 ratings.

2) The AmazonMovies dataset: AmazonMovies [20] (AM) is a dataset of movie reviews from Amazon collected between 1997 and 2012. Ratings range from 1 to 5.

3) DBLP: DBLP [21] is a dataset of co-authorship from the DBLP computer science bibliography. In this dataset, both the user set and the item set are subsets of the author set. If two authors have published at least one paper together, they are linked, which is expressed in our case by both of them appearing in the profile of the other with a rating of '5'.

4) Gowalla: Gowalla [22] (GW) is a location-based social network. As DBLP, both user set and item set are subsets of the set of the users of the social network. The undirected friendship link from u to v is represented by u rating v with a 5.

B. Baseline algorithms and competitors

We compare our approach against four competitors: a naive brute-force solution for reference, two state-of-the-art greedy KNN-graph algorithms (NNDescent [9], [10] and Hyrec [3]), and LSH [11]. On each dataset, we use the fastest competitor as our main baseline (termed 'baseline' or underlined in the following).

1) Brute force: The Brute Force competitor simply computes the similarities between every pair of profiles. This algorithm suffers from a high complexity, but produces an exact KNN graph.

2) Greedy approaches: NNDescent and Hyrec [3], [9], [10] are state-of-the-art greedy approaches that construct an approximate KNN graph by exploiting a local search strategy and by limiting the number of similarities computations. They start from an initial random graph, which is then iteratively refined until convergence. NNDescent [9], [10] and Hyrec [3] mainly differ in their iteration procedure. NNDescent compares all pairs (u_i, u_j) among the neighbors of u, and updates the neighborhoods of u_i and u_j accordingly. By contrast, Hyrec compares all the neighbors' neighbors of u with u, rather than comparing u's neighbors between themselves. Both algorithms stop either when the number of updates during one iteration is below the value $\delta \times k \times |U|$, with a fixed δ, or after a fixed number of iterations.

3) LSH: Locality-Sensitive-Hashing (LSH) [11] reduces the number of similarity computations by hashing each user into several buckets. The neighbors of a user u are then selected among the users present in the same buckets as u. To ensure that similar users tend to be hashed into the same buckets, LSH uses min-wise independent permutations of the item set as its hash functions, similarly to the MinHash algorithm [23].

C. Parameter setup

We compute KNN graphs with neighborhoods of size $k = 30$, a standard value [9]. By default, when using Cluster-and-Conquer, the number of clusters per hash functions b is set to 4096 and the number of hash functions t to 8, except for DBLP and GW for which the number of hash functions is 15. The maximum size of clusters for the recursive splitting procedure is set to $N = 2000$, except for MovieLens for which it is $N = 4000$. Both maximum sizes for clusters are below the threshold that determines whether BruteForce or Hyrec is used ($\rho \times k^2 = 4500$) in order to privilege Brute Force which tends to deliver better sub-KNNs than Hyrec. While locally computing the KNN graphs in clusters, we use 1024-bit GoldFinger vectors (See Section II-E).

The parameter δ of Hyrec and NNDescent is set to 0.001, and their maximum number of iterations to 30. The number of hash functions for LSH is 10.

D. Evaluation metrics

We measure the performance of Cluster-and-Conquer and its competitors along two main metrics: *(i)* their computation *time*, and *(ii)* the *quality* ratio of the resulting KNN (Sec. II-A). Throughout our experiments, we use a 5-fold cross-validation procedure and average our results over the 5 resulting runs.

E. Implementation details and hardware

We have implemented Brute Force, Hyrec, NNDescent, LSH, and Cluster-and-Conquer in Java 1.8. Our FastRandomHash functions are computed using Jenkins' hash function [24]. Our experiments run on a 64-bit Linux server with two Intel Xeon E5420@2.50GHz, totaling 8 hardware threads, 32GB of memory, and a HHD of 750GB. We use all 8 threads. Our code is available online[2].

TABLE II: Computation time and KNN quality. Speed-ups are computed against the best baseline (underlined). Cluster-and-Conquer clearly outperforms all competitors, yielding speed-ups of up to $\times 4.42$ against the state of the art.

datasets	Algo	**Time (s)**	**Gain** (%)	**Quality**	Δ
ml	Hyrec	<u>289.23</u>	-	<u>0.88</u>	-
	NNDescent	383.21	-	0.92	-
	LSH	1060.76	-	0.93	-
	C^2	**106.25**	63.26	0.89	+0.01
AM	Hyrec	<u>62.41</u>	-	<u>0.93</u>	-
	NNDescent	91.24	-	0.95	-
	LSH	140.53	-	0.96	-
	C^2	**14.11**	77.39	0.95	+0.02
DBLP	Hyrec	26.84	-	0.81	-
	NNDescent	<u>24.43</u>	-	<u>0.82</u>	-
	LSH	37.80	-	0.86	-
	C^2	**6.54**	73.27	0.84	+0.02
GW	Hyrec	<u>21.88</u>	-	<u>0.78</u>	-
	NNDescent	26.05	-	0.79	-
	LSH	26.91	-	0.82	-
	C^2	**8.38**	61.70	0.82	+0.04

V. Evaluation

The performances of Cluster-and-Conquer are summarized in Table II over the four datasets. In addition to those of Cluster-and-Conquer (noted C^2), the performances of Hyrec, NNDescent, and LSH are also displayed for each dataset. The best computation time is shown in bold, the time of the best baseline is underlined, and the speed-up is computed w.r.t. this best baseline.

Cluster-and-Conquer provides the best computation time on all the datasets. Cluster-and-Conquer clearly outperforms all the approaches, providing speed-ups from $\times 2.61$ (-61.70% on GW) to $\times 4.42$ (-77.39% on AM) compared to the best baselines. The KNN quality provided by Cluster-and-Conquer is similar to the one provided by the fastest approaches: the gain goes from 0.01 (on ml) to 0.04 (on GW).

VI. Related Work

KNN graphs are a key mechanism in many problems ranging from classification [1], [2] to item recommendation [3]–[5]. To speed-up the computation of the KNN graph in high dimension, recent approaches decrease the number of similarities computed. In LSH [11], [12], each user is placed into several buckets, depending on their profiles. Unfortunately, LSH tends to scale poorly when applied to datasets with large item sets.

Greedy approaches [3], [9], [10] reduce the number of computing similarities by performing a local search: they assume that neighbors of neighbors are also likely to be neighbors. They start from a random graph and then iteratively refine each

[2]https://gitlab.inria.fr/oruas/SamplingKNN

neighborhood by computing similarities among neighbors of neighbors. These approaches are the most efficient so far on the datasets we are working on. Still, they spend most of the total computation time computing similarities [8].

Sampling [25] and compact representations of the data can be used to speed-up similarity computations. Several estimators [26], [27], including the popular MinHash [23], [28], rely on compact datastructures to provide a quick yet accurate estimate of the Jaccard similarity. GoldFinger [29] is a fast-to-compute compact data structure designed to speed up the Jaccard similarity, which has shown good results across a range of datasets and algorithms [15].

Among the classical clustering techniques, k-means [13] relies on similarities to provide an efficient clustering: [30] uses a k-means to cluster the users before computing locally the KNN graph. Unfortunately, it requires to compute many similarities while our main purpose is to limit as much as possible the number of similarities computed. On the other hand, LSH clusters users without computing any similarity. The work presented in [31] uses LSH to cluster users before computing local KNN graphs, as we do, but it does not scale in high dimension. Similarly, the use of recursive Lanczos bisections [6], [32] leverages a similar strategy but the divide step has a complexity that makes it more expensive than the brute force approach when used with high dimensional datasets such as the ones we consider.

VII. Conclusions

In this paper, we have proposed Cluster-and-Conquer, a novel algorithm to compute KNN graphs. Cluster-and-Conquer accelerates the construction of KNN graphs by approximating their graph locality in a fast and robust manner. Cluster-and-Conquer relies on a divide-and-conquer approach that clusters users, computes locally the KNN graphs in each cluster, and then merges them. The novelties of the approach are the FastRandomHash functions used to pre-cluster users, their recursive splitting mechanism to produced more balanced clusters, and the fact that the clusters are computed independently, without any synchronization. Although we have only considered standalone experiments, the general structure of Cluster-and-Conquer further makes is particularly amenable to large-scale distributed deployments, in particular within a map-reduce infrastructure.

We extensively evaluated Cluster-and-Conquer on real datasets and conducted a sensitivity analysis. Our results show that Cluster-and-Conquer significantly outperforms the best existing approaches, including LSH, on all datasets, yielding speed-ups ranging from $\times 1.12$ (against Hyrec on ml1M) up to $\times 4.42$ (against Hyrec on AM), while incurring a slight increase in KNN quality. Finally, we showed that the obtained graphs can replace the exact ones when performing recommendations with almost no discernible impact on recall.

References

[1] M. Gorai, K. Sridharan, T. Aditya, R. Mukkamala, and S. Nukavarapu, "Employing bloom filters for privacy preserving distributed collaborative knn classification," in *WICT*, 2011.

[2] N. Nodarakis, S. Sioutas, D. Tsoumakos, G. Tzimas, and E. Pitoura, "Rapid aknn query processing for fast classification of multidimensional data in the cloud," *CoRR*, 2014.

[3] A. Boutet, D. Frey, R. Guerraoui, A.-M. Kermarrec, and R. Patra, "Hyrec: leveraging browsers for scalable recommenders," in *Middleware*, 2014.

[4] J. J. Levandoski, M. Sarwat, A. Eldawy, and M. F. Mokbel, "Lars: A location-aware recommender system," in *ICDE*, 2012.

[5] G. Linden, B. Smith, and J. York, "Amazon. com recommendations: Item-to-item collaborative filtering," *Internet Computing, IEEE*, 2003.

[6] J. Chen, H.-r. Fang, and Y. Saad, "Fast approximate knn graph construction for high dimensional data via recursive lanczos bisection," *Journal of Machine Learning Research*, 2009.

[7] N. Tremblay, G. Puy, P. Borgnat, R. Gribonval, and P. Vandergheynst, "Accelerated spectral clustering using graph filtering of random signals," in *ICASSP*, 2016.

[8] A. Boutet, A.-M. Kermarrec, N. Mittal, and F. Taïani, "Being prepared in a sparse world: the case of knn graph construction," in *ICDE*, 2016.

[9] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *WWW*, 2011.

[10] B. Bratić, M. E. Houle, V. Kurbalija, V. Oria, and M. Radovanović, "NN-Descent on high-dimensional data," in *WIMS*, 2018.

[11] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *STOC*, 1998.

[12] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, 1999.

[13] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Fifth Berkeley Symp. on Math. Statistics and Prob.*, 1967.

[14] C. J. van Rijsbergen, *Information retrieval*. Butterworth, 1979.

[15] R. Guerraoui, A.-M. Kermarrec, O. Ruas, and F. Taïani, "Smaller, faster & lighter knn graph constructions," in *WWW*, 2020.

[16] G. Giakkoupis, A.-M. Kermarrec, O. Ruas, and F. Taïani, "Cluster-and-conquer: When randomness meets graph locality," *arXiv preprint arXiv:2010.11497*, 2020.

[17] X. N. Lam, T. Vu, T. D. Le, and A. D. Duong, "Addressing cold-start problem in recommendation systems," in *2nd Int. Conf. on Ubiquitous Information Management and Comm.*, 2008.

[18] F. M. Harper and J. A. Konstan, "The movielens datasets: History and context," *ACM Trans. Interact. Intell. Syst.*, 2015.

[19] P. Resnick, N. Iacovou, M. Suchak, P. Bergstrom, and J. Riedl, "Grouplens: an open architecture for collaborative filtering of netnews," in *ACM Conf. on Computer Supported Cooperative Work*, 1994.

[20] J. J. McAuley and J. Leskovec, "From amateurs to connoisseurs: modeling the evolution of user expertise through online reviews," in *WWW*, 2013.

[21] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *CoRR*, vol. abs/1205.6233, 2012.

[22] E. Cho, S. A. Myers, and J. Leskovec, "Friendship and mobility: User movement in location-based social networks," in *KDD*, 2011.

[23] A. Z. Broder, "On the resemblance and containment of documents," in *Compression and Complexity of Sequences*, 1997.

[24] B. Jenkins, "Hash functions," *Dr Dobbs Journal*, 1997.

[25] A. Kermarrec, O. Ruas, and F. Taïani, "Nobody cares if you liked star wars: KNN graph construction on the cheap," in *Euro-Par'18*, 2018.

[26] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, "Fast similarity sketching," in *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, 2017.

[27] T. Christiani, R. Pagh, and J. Sivertsen, "Scalable and robust set similarity join," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, 2018.

[28] P. Li and A. C. König, "Theory and applications of b-bit minwise hashing," *Communications of the ACM*, 2011.

[29] R. Guerraoui, A. Kermarrec, O. Ruas, and F. Taïani, "Fingerprinting big data: The case of knn graph construction," in *ICDE*, 2019.

[30] G.-R. Xue, C. Lin, Q. Yang, W. Xi, H.-J. Zeng, Y. Yu, and Z. Chen, "Scalable collaborative filtering using cluster-based smoothing," in *SIGIR*. ACM, 2005.

[31] Y. Zhang, K. Huang, G. Geng, and C. Liu, "Fast kNN graph construction with locality sensitive hashing," in *ECML/PKDD*, 2013.

[32] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office, Los Angeles, CA, 1950.