

## Towards distributed node similarity search on graphs

Zhang, Tianming; Gao, Yunjun; Zheng, Baihua; Chen, Lu; Wen, Shiting; Guo, Wei

*Published in:*  
World Wide Web

*DOI (link to publication from Publisher):*  
[10.1007/s11280-020-00819-6](https://doi.org/10.1007/s11280-020-00819-6)

*Creative Commons License*  
CC BY-NC-ND 4.0

*Publication date:*  
2020

*Document Version*  
Accepted author manuscript, peer reviewed version

[Link to publication from Aalborg University](#)

*Citation for published version (APA):*  
Zhang, T., Gao, Y., Zheng, B., Chen, L., Wen, S., & Guo, W. (2020). Towards distributed node similarity search on graphs. *World Wide Web*, 23(6), 3025-3053. <https://doi.org/10.1007/s11280-020-00819-6>

### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal -

### Take down policy

If you believe that this document breaches copyright please contact us at [vbn@aub.aau.dk](mailto:vbn@aub.aau.dk) providing details, and we will remove access to the work immediately and investigate your claim.

# Towards distributed node similarity search on graphs

Tianming Zhang<sup>1</sup> · Yunjun Gao<sup>1</sup> · Baihua Zheng<sup>2</sup> · Lu Chen<sup>3</sup> · Shiting Wen<sup>4</sup> · Wei Guo<sup>1</sup>

Received: 1 October 2018 / Revised: 15 January 2020 / Accepted: 22 April 2020 /

## Abstract

Node similarity search on graphs has wide applications in recommendation, link prediction, to name just a few. However, existing studies are insufficient due to two reasons: (i) the scale of the real-world graph is growing rapidly, and (ii) vertices are always associated with complex attributes. In this paper, we propose an efficiently distributed framework to support node similarity search on massive graphs, which considers both graph structure correlation and node attribute similarity in metric spaces. The framework consists of preprocessing stage and query stage. In the preprocessing stage, a parallel KD-tree construction (KDC) algorithm is developed to form a newly defined graph so-called *hybrid graph*, in order to integrate node attribute similarity into the original graph. To equally divide graph vertices into subsets, KDC adopts the KD-tree partitioning after the pivot mapping. In addition, two metric pruning rules and an optimized allocation strategy are presented to reduce communication and computation costs. In the query stage, based on the formed hybrid graph, we develop similarity search methods using random walk with restart (RWR) to measure node similarity. To boost efficiency, we derive tight bounds to rapidly shrink the search region. Extensive experiments with three real massive graphs are conducted to verify the effectiveness, efficiency, and scalability of our proposed techniques.

**Keywords** Graph · Node similarity search · Distributed processing · Algorithm

## 1 Introduction

Graph has been applied in diverse domains such as social network, bioinformatics and chemical datasets. Node similarity search on graphs has a wide application in recommendation [31], link prediction [11], etc. For instance, in a social network, node similarity search is able to be used to recommend like-minded friends for users. In an academic collaboration network, node similarity queries can be employed to find researchers' preferred papers.

Nowadays, the scale of the real-world graph is growing rapidly, and nodes of graphs are always associated with complex attributes such as image visual features, user profile, and paper keywords. Different types of attributes need various distance metrics [8] (e.g.,

Minkowski distance, edit distance, Jaccard distance, etc.) to measure similarity. A large number of existing efforts [15, 22, 24, 25, 34] only rely on graph structure to calculate node similarity. However, to obtain high-quality answers, it is crucial to integrate node attribute similarity into node similarity.

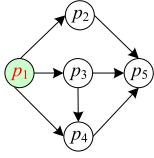
*Example 1* Figure 1a depicts a directed citation network, where a node represents a paper associated with keywords/attributes and any edge from  $p_i$  to  $p_j$  indicates  $p_i$  cites  $p_j$ . Assume that a PhD student who is interested in paper  $p_1$  would like to look for top-2 papers most relevant to  $p_1$ . Without considering paper attributes, existing methods such as RWR (restart probability  $c = 0.5$ <sup>1</sup>) locate the results purely based on graph structure. According to the similarity scores derived by RWR (listed in RWR column of Figure 1b),  $p_5$  and  $p_4$  are returned. Nonetheless, after checking, we could observe that  $p_5$  might not be the most relevant paper to  $p_1$  as it does not share any common keyword with  $p_1$ . This case shows that node attributes play an important role in evaluating node similarity. On the other hand, we can adopt Jaccard similarity to evaluate similarities between papers. As shown in the column entitled Jac. of Figure 1b, paper  $p_2$  with the highest Jaccard similarity score is the most relevant to  $p_1$ . According to our manual evaluation,  $p_2$  is more relevant to  $p_1$ , compared with  $p_5$  returned by RWR. In other words, similarity search on graphs shall encompass both node attribute similarity and graph structure correlation, and a ranking that balances these two aspects is more desirable. Users are able to set different values of a balanced factor  $\alpha$  in order to achieve satisfying results based on their own needs. If  $\alpha = 0.5$ , the scores listed in the last column of Figure 1 are the combination of both RWR scores that capture the graph structure correlation and Jaccard similarity scores that capture the node attribute similarity. Considering both scores,  $\{p_2, p_4\}$  is returned as the top-2 result.

In this paper, we aim at developing a *unified* and *scalable* framework to evaluate node similarity, not only using graph structure but also considering complex attributes of nodes in generic metric spaces so as to accommodate broad distance metrics. Given the fact that node similarity search based on graph structure or attribute values can be supported by existing approaches, a naïve solution is to compute metric scores using node attributes as well as calculate graph similarity scores (e.g., RWR scores) using the graph structure between all vertices and the query vertex, and then rank vertices based on the combined scores. Nevertheless, this method is *inefficient* due to a large number of unnecessary metric and RWR similarity computations.

In order to enable seamless search on graphs that considers both graph structure and node attributes, the challenge is *how to combine efficiently structure similarity with attribute similarity by a small number of similarity calculations*. We present an efficiently distributed framework. The key idea of the framework is to integrate important node attribute similarity into the original graph, and then perform node similarity search with pruning. The framework consists of preprocessing stage and query stage. In the preprocessing stage, we propose a *newly* defined graph so-called *hybrid graph* which integrates node attribute similarity into the original graph. To be more specific, node  $n_a$  in a hybrid graph is connected to another node  $n_b$  via an edge  $e$  because nodes  $n_a$  and  $n_b$  are physically connected or share

---

<sup>1</sup>As suggested in [13], restart probability  $c$  is empirically set as 0.5.



paper	keywords
$p_1$	metric; graph; search
$p_2$	metric; graph; search; system
$p_3$	graph; system
$p_4$	metric; space
$p_5$	classification; mining

paper pair	RWR	Jac.	balance
$(p_1, p_2)$	0.083	<b>0.75</b>	<b>0.417</b>
$(p_1, p_3)$	0.083	<b>0.25</b>	0.167
$(p_1, p_4)$	<b>0.104</b>	<b>0.25</b>	<b>0.177</b>
$(p_1, p_5)$	<b>0.115</b>	0.0	0.058

(a) Citation graph

(b) Similarity score

**Figure 1** Example of a citation graph

similar attributes, and the weight of the edge  $e$  indicates not only the structure similarity between two nodes but also the similarity between their attributes. Whereas, for a massive graph containing millions or even billions of nodes, it is *impractical* and *unnecessary* to find the attribute similarity between each pair of nodes. Consequently, the hybrid graph aims to capture similarities between important nodes (i.e., those highly similar nodes) only. In other words, hybrid graphs ignore the similarity between unimportant nodes (i.e., those highly unsimilar nodes) for efficiency reason. In our work, we adopt  $tNN$  graph [36] as a similarity graph that captures the attribute similarities between pairs of similar nodes. Although  $tNN$  graph is not a new concept, the challenge is *how to design a distributed and scalable  $tNN$  graph construction algorithm in metric spaces*. We present a KD-tree based Construction (KDC) algorithm, which is able to form  $tNN$  graphs efficiently. We also develop two metric pruning lemmas based on the triangle inequality and an optimized allocation strategy to further reduce communication and computation costs.

In the query stage, the challenge is *how to improve the search efficiency*. Based on the hybrid graph with edge weights capturing both the structure correlation and attribute similarity between nodes, we propose a RWR-related algorithm for supporting node similarity search on the hybrid graph, implemented within Apache Giraph, an open source project of Pregel system [20]. To boost efficiency, new tight bounds are derived to rapidly reduce the number of potential nodes we have to evaluate during search. In brief, the key contributions of this paper are summarized as follows:

- We identify the limitations of node similarity search on graphs that are only based on graph structure, and suggest to integrate attribute similarity into the search.
- We propose a new graph structure, i.e., hybrid graph, which captures both graph structure correlation and attribute similarity between nodes in the graph, together with an efficient construction algorithm with good scalability.
- We present a distributed algorithm on top of hybrid graphs for answering node similarity search. To boost efficiency, new tight bounds are derived to rapidly shrink the search region.
- We conduct extensive experiments using three real massive graphs to evaluate the performance of our methods, and the results demonstrate the effectiveness, efficiency, and scalability of our proposed techniques.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 formalizes our studied problem. Section 4 elaborates our distributed framework. Sections 5 and 6 detail a  $tNN$  graph construction algorithm and a node similarity search algorithm, respectively. Section 7 reports experimental results and our findings. Finally, Section 8 concludes the paper with some directions for future work.

## 2 Related work

In this section, we overview the related work on  $t$  nearest neighbor ( $t$ NN) graph, random walk with restart (RWR) computation, and distributed graph processing systems, respectively.

### 2.1 $t$ NN graph

To integrate node attribute similarity into an original graph, an attribute augmented graph, which inserts a set of attributed vertices and edges into the original graph, is used in graph clustering [7, 37]. Nonetheless, (i) the largest dataset used in [7] only has 84,170 nodes, while we focus on massive graphs with millions or even billions of nodes. (ii) The methods they proposed are just suitable for specific attribute graphs, and thus cannot be extended to general metric spaces. Therefore, we first aim to build a similarity graph based on node attribute similarity. As pointed out in [18],  $t$ NN graph tends to perform well empirically.

$t$ NN graph can be formed by Nearest Neighbor (NN) search [4], with complexity of  $O(|V|^2)$ . Obviously, this approach is impractical when it is applied to massive datasets. To reduce complexity, a number of algorithms have been proposed. For example, algorithms GBA and IPA [5] utilize dynamic disk-based metric indexes to find all- $t$ -nearest-neighbor. Approximate  $t$ NN graph construction methods using techniques such as local search [10], RMMH [29] and LSH [36] trade in the accuracy for efficiency. KIFF [3] iteratively refines the  $t$ NN approximation of every vertex with ranked candidate sets. Although these methods improve performance, they are mostly approximation based and thus cannot guarantee accuracy. In addition, a distributed approach  $t$ KNNG is presented in [23]. Nonetheless, it does not scale well, as demonstrated in Section 7. Consequently, new efficient  $t$ NN graph construction algorithms for massive graphs are required.

### 2.2 Random walk with restart computation

Among significant proximity metrics [15, 21, 22, 24, 25, 34, 35], RWR is the most popular one. It assumes a random surfer which starts at a query node  $v$ , and at each step of random walk, current node either restarts from  $v$  with a probability  $c$  or randomly selects an outgoing edge with probability  $(1 - c)$ . Existing approaches for computing RWR can be classified into three categories, namely, *initiative iterative method and its variants*, *matrix-based approaches*, and *approximation algorithms based on Monte Carlo method*. Power iteration method is advanced but time-consuming due to multiple iterations. To speed up convergence, several variations are proposed, including (i) *fast local search (FLoS)* [30], an approach to accelerate the computation for the top- $k$  query using *path-based metrics* [16]; (ii) *matrix-based approaches* that utilize the technique factorizing probability transitive matrix into canonical form; and (iii) approaches based on triangular factorization, SVD, LU, QR decomposition, and schur complement computation [12, 19, 27], etc.

To sum up, initiative iterative methods are time-consuming due to multiple iterations, matrix-based approaches are prohibitive for large graphs, and approximation algorithms cannot guarantee accuracy. In a distributed environment, the representative method presented in [16] utilizes path-based relevance metrics to detect the emergence of the top- $k$  items without ranking. However, in this paper, we focus on node similarity search with exact node ranking. Hence, the method in [16] cannot be directly applied in our studied problem. We develop  $t$ PBNN which integrates top- $k$  result ranking into the approach proposed in [16]. Nonetheless,  $t$ PBNN is inefficient, as confirmed in Section 7. In view of this, we

aim to derive tight bounds to reduce iterations, and develop efficient node similarity search approaches based on RWR in answering similarity search over massive graphs.

### 2.3 Distributed graph processing systems

Batarfi et al. [1] provide a comprehensive survey of the state-of-the-art large-scale graph processing platforms, including MapReduce [9], Pregel [20], Giraph++ [28], GraphLab [17], Trinity [26], Spark [33], GraphX [14], etc. MapReduce [9] has been adopted by corporations for big data processing. Pregel [20], which is based on bulk synchronous parallel model, is introduced by Google to process graph applications. Apache Giraph<sup>2</sup> is an open source implementation of the Pregel system. Giraph++ [28] is built on top of Giraph, and it shifts from a node-centric to a graph-centric computing system. GraphLab [17] initiates a family of related systems as an open-source project. Trinity [26] and Spark [33] are memory-based distributed processing systems. GraphX [14] is built on Spark for graph-parallel computation.

We design our algorithms based on the Pregel-like systems, because they were shown to be more suitable for iterative graph query processing. Pregel-like systems first distribute vertices of the input graph across a group of workers. Then, computation tasks are performed in a series of supersteps. During a superstep, each active vertex invokes a user-defined function, *compute()*, and vertices communicate with each other between supersteps. Program terminates when all vertices vote to halt, and there is no message in transmit. Moreover, Pregel-like systems support aggregators which enable global computation.

### 3 Problem statement

In this section, we first define the input graph  $G$ . Next, we introduce a newly defined graph structure so-called *hybrid graph* to support node similarity queries efficiently. Finally, we transform the problem of node similarity search on  $G$  into that defined on the hybrid graph. Table 1 summarizes the symbols used frequently throughout this paper.

**Definition 1 Input Graph.** An input graph (studied in our work) is denoted as  $G(V, E, A, d, w)$ , where  $V$  is a set of vertices,  $E$  is a set of edges,  $A$  is a set of metric attributes associated with the vertices in  $V$ ,  $w$  is an edge weight function, and  $d$  is a metric distance function which satisfies four properties: (1) *symmetry*, (2) *non-negativity*, (3) *identity*, and (4) *triangle inequality*.

In this paper, given an input graph  $G(V, E, A, d, w)$  and a query node, we aim to find similar nodes to the specified query node considering both graph structure correlation and node attribute similarity in metric spaces. As mentioned in Section 1, a naïve method is to compute RWR score  $Tscore$  and metric score  $Ascore$  based on graph structure and node attributes respectively, and then get the final score  $Escore = \delta \times \alpha \times Tscore + (1 - \alpha) \times Ascore$ . Here,  $\delta$  denotes a normalization factor to unify  $Tscore$  and  $Ascore$  into the same metric space, and  $\alpha \in [0, 1]$  is a balanced factor to accommodate the influence between graph structure correlation and node attribute similarity. Unfortunately, this approach is inefficient on account of many superfluous computation. Consequently, we introduce a new

<sup>2</sup>Giraph is available at <http://giraph.apache.org/>.

**Table 1** Symbols and description

Annotation	Description
$G_h(V, E_h, w_h)$	a hybrid graph $G_h$ with a set $V$ of vertices, a set $E_h$ of edges, and an edge weight function $w_h$
$G(V, E, A, d, w)$	a given graph $G$ with a set $V$ of vertices, a set $E$ of edges, a set $A$ of attributes, a metric distance function $d$ , and an edge weight function $w$
$ V $ or $ E $	the number of vertices or edges
$u$ or $v$	a vertex
$(u, v)$ or $e$	an edge
$m[u, v]$	the transition probability from node $v$ to node $u$
$\mathbf{M}$	a $ V  \times  V $ column normalized adjacent matrix for $G_h$
$N_{in}[u]$ or $N_o[u]$	the set of $u$ 's in-neighbors or out-neighbors
$\vec{q}$	a $ V  \times 1$ query vertex vector where $\sum q[v] = 1$
$\vec{s}$	a $ V  \times 1$ RWR vector where its element $s[u]$ denotes the RWR score of the vertex $u$ in $V$
$\underline{s}_i[u]$ or $\bar{s}_i[u]$	the lower or upper bound of $s[u]$ in the $i^{th}$ iteration
$M_{\max}[u]$	the maximum probability incident to vertex $u$ in the matrix $\mathbf{M}$
$c$	a restart probability in the range of $(0, 1)$
$v.tNN$	the $t$ nearest neighbors of vertex $v$
$v.d_t$	the distance from $v$ to its $t^{th}$ nearest neighbor
$BB(P_i)$	a bounding box for partition $P_i$
$MBB(P_i)$	a minimum bounding box for partition $P_i$
$v.SR$	the search region of vertex $v$
$P_i.SR$	the search region of partition $P_i$
$kNN(q, k)$	the result set of a $k$ nearest neighbour ( $kNN$ ) query on the hybrid graph w.r.t. $q$

graph structure, called *hybrid graph*, which incorporates attribute similarity between nodes into the original graph to boost search efficiency. Before introducing *hybrid graph*, we present the definition of  $tNN$  graph below.

**Definition 2  $tNN$  Graph** [36]. Given a vertex set  $V$ , a distance function  $d$ , and an integer  $t$ , a  $tNN$  graph is a directed weighted graph  $G_t(V, E_t, w_t)$  that links each vertex  $v \in V$  to its  $t$  nearest neighbors according to the distance function  $d$ , i.e.,  $E_t = \cup_{v \in V, v_i \in N_t(v, V)} (v, v_i)$  and  $w_t(v, v_i) = d(v, v_i)$ . Here,  $N_t(v, V)$  represents the  $t$  nodes in  $(V - v)$  which are most similar to  $v$ .

**Definition 3 Hybrid Graph.** Given an input graph  $G = (V, E, A, d, w)$  and a  $tNN$  graph  $G_t(V, E_t, w_t)$  constructed based on the vertex set  $V$  and the metric distance function  $d$ , a *hybrid graph*, denoted as  $G_h(V, E_h, w_h)$ , is a directed graph based on the original vertex set  $V$ , where  $E_h = E \cup E_t$ , and weight function  $w_h$  defined in (1) captures graph structure correlation and attribute similarity between nodes.

$$w_h(e) = \begin{cases} \alpha \times w(e) & e \in (E - E_t) \\ \delta \times (1 - \alpha) \times w_t(e) & e \in (E_t - E) \\ \delta \times (1 - \alpha) \times w_t(e) + \alpha \times w(e) & e \in (E \cap E_t) \end{cases} \quad (1)$$

In (1),  $\delta$  is a normalization factor to make sure  $w(e)$  and  $w_t(e)$  are in the same order of magnitude. Parameter  $\alpha \in [0, 1]$  indicates the importance of structure correlation vs. that of attribute similarity.  $w_t$  is a re-weight function, defined below.

$$\forall(u, v) \in E_t, w_t(u, v) = \frac{\sum_{s \in N_o[u]} w_t(u, s)}{w_t(u, v)} \quad (2)$$

In (2),  $N_o[u]$  is a set of  $u$ 's out-neighbors. The reason why we re-weight is that, the weight  $w_t(u, v)$  captures the metric distance from a node  $u$  to one of its  $t$ NN nodes  $v$ . The shorter the metric distance is, the more similar the two nodes are. Nonetheless, when we perform RWR calculation, the weight of an edge indicates the proximity of two nodes. The larger the weight is, the closer proximity the two nodes have. Thus, we define  $w_t$  by (2), and use it in (1) such that weights of edges in the  $t$ NN graph are consistent with weights of edges in the original graph.

Note that, hybrid graph  $G_h(V, E_h, w_h)$  is a *virtual graph* that does not need construction or physical storage. In other words, given an original graph  $G$  and its  $t$ NN graph, the corresponding hybrid graph can be generated dynamically if necessary. Based on the hybrid graph, the problem of node similarity search on  $G$  can be transformed into that defined on  $G_h$ . Next, we give the definition of  $k$  Nearest Neighbor ( $k$ NN) query on  $G_h$ .

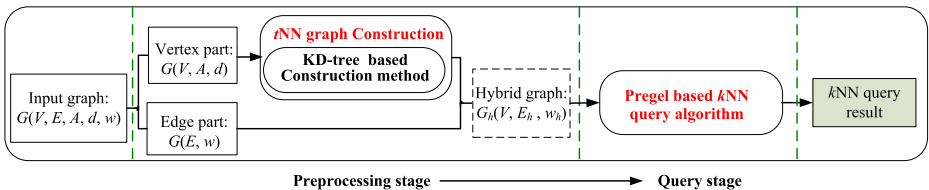
**Definition 4 ( $k$  Nearest Neighbor ( $k$ NN) Query on Hybrid Graph).** Given a hybrid graph  $G_h = (V, E_h, w_h)$ , a restart probability parameter  $c$ , an integer  $k$ , and a query node  $q$ , a  $k$ NN query on  $G_h$  finds  $k$  nodes that have the highest similarity scores w.r.t.  $q$ , sorted in descending order of their similarity scores, i.e.,  $kNN(q, k) = \{R \mid R \subseteq V \wedge |R| = k \wedge \forall v \in R, \forall o \in V - R, s[o] \leq s[v]\}$ , and let  $R = \{v_1, v_2, \dots, v_k\}$ ,  $\forall v_i \in R, \forall v_j \in R$ , if  $i > j$ , then  $s[v_i] \leq s[v_j]$ .  $s[v]$  is the RWR score of a vertex  $v$ .

It is worth mentioning that, in this paper, we focus on designing  $k$ NN query algorithms on  $G_h$  but ignoring range query algorithms. Nevertheless, our proposed  $k$ NN query algorithm can be easily extended to handle range queries.

## 4 Node similarity search framework

We present a distributed framework, as illustrated in Figure 2. Specifically, our proposed distributed framework consists of two stages, i.e., preprocessing stage and query stage, which are summarized below.

**Preprocessing stage** At this stage, we propose hybrid graph, which integrates attribute similarity between nodes into the original graph. The generation can be divided into two steps.



**Figure 2** Our distributed framework



- (i) Given an input graph  $G = (V, E, A, d, w)$ , we construct a  $t$ NN graph  $G_t(V, E_t, w_t)$ , which captures the attribute similarities between pairs of similar nodes based on vertex information  $(V, A, d)$ . In this paper, a KD-tree based construction method, which adopts the KD-tree technique after the pivot mapping to equally partition graph nodes into subsets, is presented for  $t$ NN graph construction.
- (ii) Given a parameter  $\alpha$ , the hybrid graph  $G_h$  can be created based on the  $t$ NN graph  $G_t$  and the input graph  $G$ . First, we re-weight each edge  $(u, v)$  in  $G_t$  according to (2).

Then, a hybrid graph  $G_h(V, E_h, w_h)$  is generated by combining the re-weighted  $t$ NN graph and the original edge part  $(E, w)$ . As mentioned in Section 3,  $G_h$  is a virtual graph that does not need construction or physical storage. Hence, the second step is triggered if necessary. The first step (i.e.,  $t$ NN graph construction) is the key step, which will be described in Section 5.

**Query stage** In the query stage, based on the formed hybrid graph  $G_h$ , we present a Pregel based  $k$ NN query algorithm by using RWR to support  $k$ NN queries in Pregel. To accelerate search, we derive tight similarity bounds. The query method will be elaborated in Section 6.

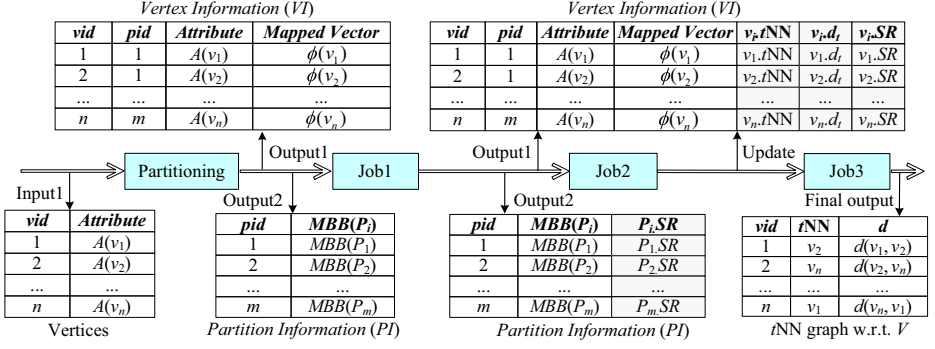
## 5 $t$ NN graph construction

Existing sequential algorithms for  $t$ NN graph construction either are specific to certain similarity measures, or are mostly approximate and thus cannot guarantee accuracy. The representative distributed algorithm is DKNNG [23].

DKNNG proceeds through three steps. (i) It partitions the whole data set into clusters using  $k$ -means, and then assigns each cluster to one processor. (ii) Each processor computes partial  $k$ NN results by constructing and querying sequential  $k$ NN data structures. (iii) Each processor gathers partial results from other processors, and then performs  $k$ NN queries for each vertex in the data set. Although DKNNG could exploit  $k$ NN data structure to reduce computation cost, it is still inefficient and has limited scalability, as to be confirmed in Section 7. The clustering result of DKNNG might be screwed (i.e., some clusters contain most of vertices), which causes load imbalance, and thus weakens the overall performance significantly. To this end, we propose a *KD-tree based Construction* (KDC) algorithm to guarantee load balancing. The basic idea of KDC is to perform KD-tree partitioning on a vertex set  $V$  to obtain evenly balanced partitions, find local  $t$ NN result for each vertex within partition and across partitions, and locate global  $t$ NN result by merging local  $t$ NN results. Moreover, two metric pruning lemmas and an optimized allocation strategy are presented to further reduce communication and computation costs. The value of  $t$  is an input for  $t$ NN graph construction, which has an impact on the queries we would like to perform on top of the hybrid graph. Theoretically, the larger the  $t$  value is, the more the nearest neighbors of each node are captured, but suffer from prohibitive computation cost. The setting of  $t$  is to be further verified in Section 7. In the following, we first describe KDC algorithm in detail. Then, we discuss how KDC is able to support dynamic update. Finally, we analyze the complexity of KDC.

### 5.1 KDC algorithm

In this subsection, we present KDC algorithm. Figure 3 shows its framework, which consists of one partitioning step and three MapReduce jobs.

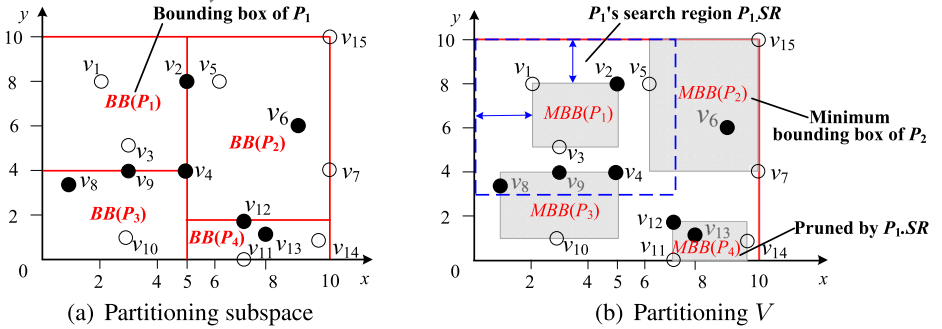


**Figure 3** Illustration of KDC framework

### 5.1.1 Partitioning

Given an input graph  $G(V, E, A, d, w)$ , a good partitioning of  $V$ , where vertices are distributed uniformly, is crucial to achieve load balancing. To do this, KDC first selects  $l (\ll |V|)$  vertices to form a pivot set  $S_p = \{s_{p_1}, s_{p_2}, \dots, s_{p_l}\}$ . Then,  $\forall v \in V$ , KDC maps  $v$  to a vector in a  $l$ -dimensional vector space, denoted as  $\phi(v) = \langle d(v, s_{p_1}), d(v, s_{p_2}), \dots, d(v, s_{p_l}) \rangle$ , based on its metric distances to pivots. Thereafter, given an integer  $m$ , KDC divides the vector space into  $m$  partitions using KD-tree partitioning technique to uniformly distribute vertices. Every partition  $P_i (1 \leq i \leq m)$  is bounded by a bounding box  $BB(P_i)$  and each vertex is located into one bounding box.

*Example 2* Figure 4 depicts an example to illustrate the partitioning process. Note that, every point  $v_i (1 \leq i \leq 15)$  in Figure 4 represents the mapped vector  $\phi(v_i)$  but not the original vertex. Using the KD-tree partitioning technique, each time we choose a dimension with the maximum variance, and divide vertices into two disjoint partitions according to the median value. Considering that we target at massive graphs, the number of vertices could be huge (e.g.,  $2 \times 10^7$  in our experiments). To guarantee the efficiency of dimension selection, we adopt sampling method. Let  $S = \{v_2, v_4, v_6, v_8, v_9, v_{12}, v_{13}\}$  be the sample set of our



**Figure 4** Illustration of KD-tree partitioning

running example. By calculation,  $x$ -dimension has its variance of 6.82 while  $y$ -dimension has its variance of 4.86. Hence, the space is first partitioned by  $x$ -dimensional median (i.e., 5), and then, it is partitioned by  $y$ -dimensional median (i.e., 4 and 2) to generate four partitions  $P_1, P_2, P_3$ , and  $P_4$ . Each red-line rectangle in Figure 4a denotes a bounding box  $BB(P_i)$ . Based on  $BB(P_i)$  that a vertex locates into, each vertex is associated with a partition  $P_i$ , and thus, have  $P_1 = \{v_1, v_2, v_3\}$ ,  $P_2 = \{v_5, v_6, v_7, v_{15}\}$ ,  $P_3 = \{v_4, v_8, v_9, v_{10}\}$ , and  $P_4 = \{v_{11}, v_{12}, v_{13}, v_{14}\}$ . For every partition  $P_i$ , we can get a tight minimum bounding box  $MBB(P_i) = \{[min_j, max_j] \mid j \in [1, n]\}$ . As an example, shadow rectangles in Figure 4b represent the minimum bounding boxes, e.g.,  $MBB(P_1) = \{[2, 5], [5, 8]\}$ .

After KD-tree partitioning, we maintain two output tables, i.e., partition information table  $PI$  and vertex information table  $VI$ , as depicted in Figure 3. Assume that there are  $m$  partitions in total, denoted as  $P = \cup_{1 \leq i \leq m} P_i$ . Specifically, the information of each partition  $P_i \in P$  maintained by  $PI$  includes a partition id  $pid$  of  $P_i$ , and the minimum bounding box  $MBB(P_i)$  of  $P_i$ . The information of each vertex  $v_i \in V$  captured by  $VI$  contains a vertex id  $vid$  of  $v_i$ , the corresponding partition id  $pid$ , the attribute  $A(v_i)$  of  $v_i$ , and the mapped vector  $\phi(v_i)$ .

### 5.1.2 The first MapReduce job

The main task of each reducer is to derive the similarity between each pair of vertices in the same partition according to a metric distance function  $d$  and to find the local  $t$ NN result for each vertex  $v \in P_i$ . When this job completes,  $VI$  appends  $v.tNN$ ,  $v.d_t$ , and  $v.SR$  to every vertex  $v \in P_i$ . Here,  $v.tNN$  represents the  $t$  nearest neighbors of  $v$ .  $v.d_t$  denotes the distance from  $v$  to its  $t^{th}$  nearest neighbor.  $v.SR = \{[d(v, s_{p_j}) - v.d_t, d(v, s_{p_j}) + v.d_t] \mid j \in [1, n]\}$  represents the search region of  $v$ , and it bounds the search region for potential vertices that may update  $v.tNN$ . In addition, by considering the search regions for all the vertices in a partition  $P_i$ , the search region of  $P_i$ , denoted as  $P_i.SR = \cup_{v \in P_i} v.SR \cup MBB(P_i)$ , has been captured by  $PI$ .  $P_i.SR$  bounds the search region for potential vertices which could update  $\cup_{v \in P_i} v.tNN$ . For instance, the blue dotted rectangle in Figure 4b denotes the search region of  $P_1$ , i.e.,  $P_1.SR = \{[0, 7], [3, 10]\}$ .

### 5.1.3 The second MapReduce job

Once the local  $t$ NN computation is finished, the second MapReduce job is launched to perform  $t$ NN computation across partitions. Then,  $v.tNN$  is updated when a vertex  $v_j \notin (v.tNN \cup v)$  having  $d(v, v_j) < v.d_t$  is found. To enable  $t$ NN search across partitions, the mapper needs to distribute partition  $P_i$  to partition  $P_j$  if  $P_i$  has a vertex that might be one potential  $t$ NN for any vertex in  $P_j$ . A brute-force method is to distribute  $P_i$  to all the other partitions that are different from  $P_i$ , but it is costly. In order to reduce communication and computation costs, we present an optimized allocation strategy. Specifically, for a partition  $P_i$ , it is sent to partition  $P_j$  based on the strategy defined in (3).

$$j \in \begin{cases} (i, \lceil m/2 \rceil + i) & i \leq \lceil m/2 \rceil \\ [1, i - \lceil m/2 \rceil] \vee (i, m] & i > \lceil m/2 \rceil \end{cases} \quad (3)$$

In (3),  $m$  is the number of partitions. If  $i \leq \lceil m/2 \rceil$ , partition  $P_i$  is sent to partition  $P_j$  where  $j > i$  and  $j - \lceil m/2 \rceil < i$ . Otherwise, partition  $P_i$  is sent to partition  $P_j$  where

$j > i$  or  $j + \lceil m/2 \rceil \leq i$ . This guarantees that (i) metric distance computation between each pair of vertices is considered. (ii) The whole communication and computation costs are reduced by 50%, compared with the above brute-force approach. Also, we develop two lemmas to enable metric pruning, which can avoid allocating unqualified vertices to partitions.

**Lemma 1** *Given a partition  $P_i$  and a vertex  $v_j \notin P_i$ , if  $\phi(v_j)$  locates outside  $P_i$ 's search region  $P_i.SR$ , then  $v_j \notin P_i.tNN$ , in which  $P_i.tNN = \cup_{v \in P_i} v.tNN$ .*

*Proof* If  $\phi(v_j)$  locates outside  $P_i.SR$ , then  $\exists s_{p_k} \in S_p, \forall v \in P_i, d(v_j, s_{p_k}) > d(v, s_{p_k}) + v.d_t$  or  $d(v_j, s_{p_k}) < d(v, s_{p_k}) - v.d_t$ , i.e.,  $\forall v \in P_i, |d(v_j, s_{p_k}) - d(v, s_{p_k})| > v.d_t$ . Based on the triangle inequality,  $\forall v \in P_i, d(v_j, v) \geq |d(v_j, s_{p_k}) - d(v, s_{p_k})| > v.d_t$ . Therefore, for any  $v \in P_i, v_j \notin v.tNN$ , i.e.,  $v_j \notin P_i.tNN$ . The proof completes.  $\square$

Back to the example shown in Figure 4b.  $P_1.SR = \{[0, 7], [3, 10]\}$  and  $MBB(P_4) \cap P_1.SR = \emptyset$ , i.e.,  $\forall v_j \in P_4, \phi(v_j)$  locates outside  $P_1.SR$ . Hence, none of vertices in  $P_4$  could be a  $tNN$  for any vertex in  $P_1$  by Lemma 1.

**Lemma 2** *Given a partition  $P_i$  and a vertex  $v \notin P_i$ , if  $MBB(P_i) \cap v.SR = \emptyset$ , then  $\forall v_k \in P_i, v_k \notin v.tNN$ .*

*Proof*  $MBB(P_i) = \{\min_j, \max_j \mid j \in [1, n]\}$  is the minimum bounding box of  $P_i$ , then,  $\forall v_k \in P_i, \forall s_{p_j} \in S_p, \min_j \leq d(v_k, s_{p_j}) \leq \max_j$ . For vertex  $v, v.SR = \{[d(v, s_{p_j}) - v.d_t, d(v, s_{p_j}) + v.d_t] \mid s_{p_j} \in S_p\}$ . If  $MBB(P_i) \cap v.SR = \emptyset$ , then  $\forall s_{p_j} \in S_p, d(v, s_{p_j}) - v.d_t > \max_j$  or  $d(v, s_{p_j}) + v.d_t < \min_j$ . Therefore,  $d(v_k, s_{p_j}) \leq \max_j < d(v, s_{p_j}) - v.d_t$ , or  $d(v, s_{p_j}) + v.d_t < \min_j \leq d(v_k, s_{p_j})$ , i.e.,  $|d(v, s_{p_j}) - d(v_k, s_{p_j})| > v.d_t$ . Based on the triangle inequality,  $\forall v_k \in P_i, d(v, v_k) \geq |d(v, s_{p_j}) - d(v_k, s_{p_j})| > v.d_t$ , i.e.,  $v_k \notin v.tNN$ . The proof completes.  $\square$

### 5.1.4 The third MapReduce job

After the second MapReduce job,  $v \in P_i$  has multiple  $tNN$  results, produced by evaluating vertices in different partitions. In order to guarantee that  $v.tNN$  eventually maintains the real  $t$  nearest neighbors of  $v$ , we launch the third MapReduce job to merge the multiple  $tNN$  results corresponding to the vertex  $v$  and to generate the final  $tNN$  result.

### 5.1.5 The summary of KDC

Based on one partitioning step and three MapReduce steps, we present KDC algorithm with its pseudo-code shown in Algorithm 1. Initially, KDC maps original vertices to the vector space, and adopts KD-tree technique for partitioning. (line 1). Then, for every partition  $P_i$ , it computes each vertex  $v$ 's local  $t$  nearest neighbors ( $v.tNN$ ),  $v.d_t$ ,  $v.SR$ , and  $P_i.SR$  (lines 2-5). Next, KDC distributes qualified  $v$  to partitions based on (3) (lines 6-12), and Lemmas 1 and 2 using Assign\_KDC function (lines 23-27). Thereafter, KDC updates  $v.tNN$  using  $v_j.SR$  for pruning (lines 13-21). Finally, the  $tNN$  graph w.r.t.  $V$  is returned (line 22).

---

**Algorithm 1** KDC Algorithm.

---

**Input:** a vertex set  $V$ , a metric distance function  $d$ , an integer  $t$ , and an integer  $m$   
**Output:** the  $t$ NN graph w.r.t.  $V$

```
1:  $P = \{P_i \mid 1 \leq i \leq m\} \leftarrow$  perform pivot mapping and partition on  $V$ 
2: foreach partition  $P_i$  do
3:   foreach vertex  $v \in P_i$  do
4:      $\lfloor$  compute  $v.tNN$ ,  $v.d_t$ , and  $v.SR$ 
5:    $\lfloor$  compute  $P_i.SR = \cup_{v \in P_i} v.SR \cup MBB(P_i)$ 
6: foreach vertex  $v \in P_i$  do                                     // allocated by (3)
7:   if  $i \leq \lceil m/2 \rceil$  then
8:     foreach  $i < s < \lceil m/2 \rceil + i$  do
9:        $\lfloor$  Assign_KDC( $v, P_s$ )
10:  else
11:    foreach  $1 \leq s \leq i - \lceil m/2 \rceil$  or  $i < s \leq m$  do
12:       $\lfloor$  Assign_KDC( $v, P_s$ )
13: foreach partition  $P_i$  do
14:   foreach vertex  $v \in P_i$  do
15:     foreach allocated vertex  $v_j$  do //  $v_j$  is allocated to  $P_i$  by (3)
16:       and function Assign_KDC
17:       if  $v \in v_j.SR$  then // using  $v_j.SR$  for pruning
18:         if  $d(v, v_j) \leq v.d_t$  then
19:            $\lfloor$  update  $v.tNN$ 
20:       if  $v_j \in v.SR$  then // using  $v.SR$  for pruning
21:         if  $d(v, v_j) \leq v.d_t$  then
22:            $\lfloor$  update  $v_j.tNN$ 
23: return the final  $tNN$  graph w.r.t.  $V$ 
24: Function: Assign_KDC( $v, P_s$ )
25: if  $\phi(v) \in P_s.SR$  then // pruned by Lemma 1
26:    $\lfloor$  assign  $v$  to  $P_s$ 
27: else if  $MBB(P_s) \cap v.SR \neq \emptyset$  then // pruned by Lemma 2
28:    $\lfloor$  assign  $v$  to  $P_s$ 
```

---

## 5.2 Dynamic update

As graphs might be changed dynamically,  $tNN$  graph construction algorithms should be able to support dynamic update. Since the update of edges has *zero* impact on the  $tNN$  graph, we only discuss the update of vertices (including vertex insertion, vertex deletion, and the update of vertex attributes).

**Vertex insertion** When a new vertex  $w$  is inserted into a vertex set  $V$ , we need to compute  $w.tNN$ , and judge whether  $w$  may update  $tNN$  result of any existing vertex  $v$  in  $V$ . The process of updating can be done by two MapReduce jobs. In the first MapReduce job, to update  $v.tNN$  and to form  $w$ 's local  $tNN$  results, we assign  $w$  to every partition and then derive the metric distances between  $w$  and all the vertices  $v$  in each partition in parallel. Note that, lemmas presented previously can be employed to avoid unnecessary evaluation. In the second MapReduce job, local  $tNN$  results of  $w$  are aggregated into a reducer to form the final global  $tNN$  result of  $w$ .

**Vertex deletion** Given a vertex  $u \in V$  to be removed from  $V$ , we have to delete  $u$  and  $u$ 's outgoing edges from the  $tNN$  graph. In addition, each incoming edge of  $u$ , denoted as  $(v, u)$ , indicates that  $u$  is one of  $v$ 's  $tNN$  result. Hence, we need to replace  $(v, u)$  with another edge  $(v, v')$ , i.e., vertex  $v' \in (V - v.tNN - u)$  is promoted to be one of  $v$ 's  $tNN$  result after  $u$  is removed. Specifically, for each  $u$ 's incoming edge  $(v, u)$ ,  $u$ 's replacement for a specified vertex  $v$  can be identified via the two MapReduce jobs explained earlier, i.e., taking  $v$  as a newly inserted vertex with  $t = 1$  and  $V = V - v.tNN$ .

**The update of vertex attributes** When the attributes of a vertex  $u$  are changed, we need to update  $u.tNN$  and judge whether  $u$  may update  $tNN$  result of any existing vertex. The process can be done by two operations. First, we delete  $u$  with old attributes from  $V$ , and then, the  $tNN$  result of each vertex in  $(V - u)$  is computed. Second, we insert  $u$  with new attributes, and then, the new  $tNN$  result of every vertex in  $V$  are computed or updated.

### 5.3 Complexity analysis

In this subsection, we analyze the complexity of KDC in terms of total computation cost and communication cost.

**Lemma 3** *The total computation cost of KDC is  $O(|S| \times (l + \log |S| \times \log m) + \max_{1 \leq i \leq m} |P_i|^2 + \max_{1 \leq i \leq m} (|P_i| \times |A_i|))$ , where  $l$  is the number of pivots,  $|S|$  is the size of the sample set,  $m$  is the number of partitions,  $|P_i|$  is the cardinality of the vertex subset in partition  $P_i$ , and  $|A_i|$  is the number of the qualified vertices that are mapped to the corresponding partition  $P_i$ .*

*Proof* Let  $l$  be the number of pivots,  $|S|$  be the size of sample set, and  $m$  be the number of partitions. For KD-tree partitioning, it needs  $O(|S| \times l)$  to perform the pivot mapping, and  $O(|S| \times \log |S| \times \log m)$  to split the whole space into  $m$  equal parts. Therefore, the time complexity of KD-tree partitioning is  $O(|S| \times (l + \log |S| \times \log m))$ .

For the first MapReduce job, each reducer computes the similarity between every pair of vertices in the same partition  $P_i$ . Let  $|P_i|$  be the cardinality of the vertex subset in partition  $P_i$ , the time complexity of the first MapReduce job is  $O(\max_{1 \leq i \leq m} |P_i|^2)$ .

For the second MapReduce job, each reducer performs  $tNN$  computation across partitions. Thus, the time complexity of the second MapReduce job is  $O(\max_{1 \leq i \leq m} (|P_i| \times |A_i|))$ ,

where  $|A_i|$  is the number of the qualified vertices that are mapped to the corresponding partition  $P_i$ .

The third MapReduce job is to merge the multiple  $t$ NN sets corresponding to the vertex  $v$ , there is almost no computation.

To sum up, the total computation cost of KDC is  $O(|S| \times (l + \log |S| \times \log m) + \max_{1 \leq i \leq m} |P_i|^2 + \max_{1 \leq i \leq m} (|P_i| \times |A_i|))$ . The proof completes.  $\square$

**Lemma 4** *The total communication cost of KDC is  $O(\sum_{1 \leq i \leq m} (|A_i||R| + |A_i|))$ , in which  $|R|$  is the size of the one record of  $v$  including  $v$ 's id, the attribute of  $v$ ,  $\phi(v)$ ,  $v.tNN$ , and  $v.SR$ .*

*Proof* For the first MapReduce job, KDC has no communication cost. For the second MapReduce job, the mapper needs to distribute qualified vertices to each partition. Let  $|A_i|$  be the number of the qualified vertices that are mapped to the corresponding partition  $P_i$  and  $|R|$  be the size of the one record of  $v$  including  $v$ 's id, the attribute of  $v$ ,  $d(v, c_i)$ ,  $\phi(v)$ ,  $v.tNN$ , and  $v.SR$ . The communication cost of the second MapReduce job is  $O(\sum_{1 \leq i \leq m} |A_i||R|)$ . The third MapReduce job is to merge the multiple  $tNN$  sets corresponding to the vertex  $v$ , and thus, the communication cost of the third MapReduce job is  $O(\sum_{1 \leq i \leq m} |A_i|)$ . Hence, the total communication cost of KDC is  $O(\sum_{1 \leq i \leq m} (|A_i||R| + |A_i|))$ . The proof completes.  $\square$

## 6 Node similarity search

In this section, we propose *Pregel based kNN query* (PNN) algorithm, to support  $kNN$  queries in Pregel, based on RWR. Note that, to make a trade-off between the partitioning cost and the querying overhead, we adopt system-provided hash partitioning in the query stage. The KD-tree partitioning is used for  $tNN$  graph construction. Both KD-tree partitioning and hash partitioning are balanced partitioning methods. Balanced partitioning is crucial for efficiently constructing  $tNN$  graph and querying because it can ensure load balancing, which evenly distributes the vertices across all nodes in the cluster to avoid overloading or completely no work in others.

### 6.1 PNN algorithm

A  $kNN$  query on a hybrid graph finds  $k$  vertices with the highest similarity scores  $s[u]$  w.r.t. a query node  $q$ , sorted in descending order of  $s[u]$ . Here,  $s[u]$  is the RWR score of a vertex  $u \in V$  w.r.t.  $q$ . The recursive RWR iteration keeps computing  $\vec{s}$  until convergence, with the score computed by (4) [22].

$$\vec{s}^{(i)} = \begin{cases} (1 - c)\mathbf{M}\vec{s}^{(i-1)} + c\vec{q} & i > 0 \\ c\vec{q} & i = 0 \end{cases} \quad (4)$$

where superscript  $i$  represents the number of iterations,  $\mathbf{M}$  is a column normalized adjacent matrix, and  $\vec{q}$  denotes a query vertex vector. If  $u$  is not a query node,  $q[u] = 0$ .

Otherwise,  $q[u] = 1$ . To improve the efficiency of the above computation, we utilize  $p_i[u]$  [13], the random walk probability of length  $i$  that starts at a query node and ends at a node  $u$ . For a specified hybrid graph  $G_h$ ,  $p_i[u]$  is computed as:

$$p_i[u] = \begin{cases} q[u] & i = 0 \\ \sum_{v \in N_{in}[u]} m[u, v] p_{i-1}[v] & i \neq 0 \end{cases} \quad (5)$$

In (5),  $N_{in}[u]$  is the set of  $u$ 's in-neighbors. Based on  $p_i[u]$ , we derive tight lower and upper bounds of  $s[u]$ , as presented in (6) and (7), respectively. In (7),  $M_{\max}[u]$  is the maximum probability incident to node  $u$  in the matrix  $\mathbf{M}$ , i.e.,  $M_{\max}[u] = \max\{m[u, v] | v \in V\}$ .

$$s_i[u] = \begin{cases} c p_i[u] & i = 0 \\ \underline{s}_{i-1}[u] + c(1-c)^i p_i[u] & i \neq 0 \end{cases} \quad (6)$$

$$\bar{s}_i[u] = \underline{s}_i[u] + c(1-c)^{i+1} \sum_{v \in N_{in}(u)} m[u, v] p_i[v] + (1-c)^{i+2} M_{\max}[u] \quad (7)$$

As a  $k$ NN query is performed via iterations, symbol  $\theta_i^k$  denotes the  $k^{th}$  highest lower bound  $\underline{s}_i[u]$  among all nodes in the  $i^{th}$  iteration. Hence, Lemmas 5 and 6 are developed to enable pruning based on upper and lower bounds of similarity scores.

**Lemma 5**  $\forall u \in V$ ,  $\bar{s}_i[u] \geq s[u] \geq s_i[u]$  holds in all iterations.

*Proof*  $s[u] \geq \underline{s}_i[u]$  can be proved similarly as Lemma 1 in [13], hence we omit it due to space limitation. Next, we prove that  $\forall u \in V$ ,  $\bar{s}_i[u] \geq s[u]$ .

$$\vec{s} = (1-c)\mathbf{M}\vec{s} + c\vec{q} = c(I - (1-c)\mathbf{M})^{-1}\vec{q}$$

As shown in (6) defined in [13],

$$\vec{s} = c \sum_{j=0}^{\infty} ((1-c)\mathbf{M})^j \vec{q}$$

Next, in the  $i^{th}$  iteration, it has

$$\begin{aligned} s[u] &= \underline{s}_i[u] + (c(1-c)^{i+1} \mathbf{M}^{i+1} \vec{q})[u] + c \sum_{j=i+2}^{\infty} (((1-c)\mathbf{M})^j \vec{q})[u] \\ &\leq \underline{s}_i[u] + (c(1-c)^{i+1} \mathbf{M}^{i+1} \vec{q})[u] + c \sum_{j=i+2}^{\infty} (1-c)^j M_{\max}[u] (\|\mathbf{M}\|_1^{j-1} \|\vec{q}\|_1) \end{aligned}$$



Here,  $\mathbf{M}$  is a column normalized matrix, then  $\|\mathbf{M}\|_1 = 1$ ; and as  $\|\vec{q}\|_1 = 1$ ,

$$s[u] \leq \underline{s}_i[u] + (c(1-c))^{i+1} \mathbf{M}^{i+1} \vec{q}[u] + c \sum_{j=i+2}^{\infty} (1-c)^j M_{\max}[u]$$

Note that,  $c \sum_{j=i+2}^{\infty} (1-c)^j M_{\max}[u]$  is an infinite geometric series, in which the first term is given by  $a_1 = c(1-c)^{i+2} M_{\max}[u]$ , and the common ratio is  $(1-c) = 0.5$ . Since the common ratio has value between  $-1$  and  $1$ , the series will converge to  $\frac{a_1}{1-(1-c)}$ , i.e.,  $c \sum_{j=i+2}^{\infty} (1-c)^j M_{\max}[u] = (1-c)^{i+2} M_{\max}[u]$ , and thus,

$$s[u] \leq \underline{s}_i[u] + c(1-c)^{i+1} \sum_{v \in N_{in}(u)} m[u, v] p_i[v] + (1-c)^{i+2} M_{\max}[u]$$

Hence,  $s[u] \leq \bar{s}_i[u]$ . The proof completes.  $\square$

**Lemma 6** In the  $i^{th}$  iteration, if  $\bar{s}_i[u] \leq \theta_i^k$ ,  $u$  can be pruned.

*Proof* According to Lemma 5,  $s[u] \leq \bar{s}_i[u]$ . If  $\bar{s}_i[u] \leq \theta_i^k$ ,  $s[u] \leq \theta_i^k$  holds, i.e. the exact similarity score of vertex  $u$  cannot be more than the  $k^{th}$  highest lower similarity bound, and thus,  $u$  can be pruned safely. The proof completes.  $\square$

To ensure that (i) lower and upper bounds have monotonic increasing and decreasing property, respectively; and (ii) lower and upper bounds could converge to the exact similarities for all vertices, Lemma 7 and Lemma 8 are proposed.

**Lemma 7**  $\forall u \in V$ ,  $\underline{s}_i[u] \geq \underline{s}_{i-1}[u]$  and  $\bar{s}_i[u] \leq \bar{s}_{i-1}[u]$  hold in the  $i^{th}$  iteration.

*Proof*  $\forall u \in V$ ,  $\underline{s}_i[u] \geq \underline{s}_{i-1}[u]$  is obviously true by (6). Here, we prove that  $\bar{s}_i[u] \leq \bar{s}_{i-1}[u]$  holds in the  $i^{th}$  iteration.

$$\begin{aligned} & \bar{s}_i[u] - \bar{s}_{i-1}[u] \\ &= \underline{s}_i[u] + c(1-c)^{i+1} \sum_{v \in N_{in}(u)} m[u, v] p_i[v] + (1-c)^{i+2} M_{\max}[u] \\ & \quad - (\underline{s}_{i-1}[u] + c(1-c)^i \sum_{v \in N_{in}(u)} m[u, v] p_{i-1}[v] + (1-c)^{i+1} M_{\max}[u]) \\ &= c(1-c)^{i+1} p_i[u] - (1-c)^{i+1} c M_{\max}[u] \\ &= c(1-c)^{i+1} (p_i[u] - M_{\max}[u]) \end{aligned}$$

According to (5) and (17) defined in [13],  $p_i[u] = \sum_{v \in N_{in}[u]} m[u, v] p_{i-1}[u] \leq M_{\max}[u]$ . Therefore,  $\bar{s}_i[u] \leq \bar{s}_{i-1}[u]$ . The proof completes.  $\square$

**Lemma 8** The lower and upper bounds converge to the exact similarity scores, i.e.,  $\forall u \in V$ ,  $\underline{s}_{\infty}[u] = \bar{s}_{\infty}[u] = s[u]$ .

*Proof* Since  $\underline{s}_{\infty}[u] = s[u]$  can be proved similarly as [13], here we only prove  $\bar{s}_{\infty}[u] = s[u]$ . From (7),  $\bar{s}_{\infty}[u] = \underline{s}_{\infty}[u] + (1-c)^{\infty} M_{\max}[u]$ . As  $(1-c)^{\infty} = 0$  and  $0 \leq M_{\max}[u] \leq 1$ ,  $(1-c)^{\infty} M_{\max}[u] = 0$ . Thus,  $\underline{s}_{\infty}[u] = \bar{s}_{\infty}[u] = s[u]$ . The proof completes.  $\square$

---

**Algorithm 2** PNN Algorithm.

---

**Input:** A hybrid graph  $G_h = (V, E_h, w_h)$ , a query vector  $\vec{q}$ , an integer  $k$

**Output:** the result set  $kNN(q, k)$  of a  $kNN$  query

```
1: foreach vertex  $v \in V$  do
2:   if  $superstep = 0$  then
3:      $\forall s \in N_o[v], m[s, v] \leftarrow \frac{w(v, s)}{\sum_{s \in N_o[v]} w(v, s)}$ 
4:      $\forall s \in N_o[v]$ , send message  $m[s, v]$  to  $s$ 
5:   else if  $superstep = 1$  then
6:      $v.flag \leftarrow 0, i \leftarrow 0$ ; compute and save  $M_{max}[v], \underline{s}_0[v], \bar{s}_0[v], p_0[v]$ ; and
       insert  $\underline{s}_0[v]$  to  $KL_b$ 
7:      $\forall s \in N_o[v]$ , send message  $(m[s, v] \times p_0[v])$  to  $s$ 
8:   else if  $superstep = 2$  then
9:      $p_{i+1}[v] = \sum(message)$ 
10:    if  $\bar{s}_i[v] > \theta_i^k$  then
11:       $num \leftarrow num + 1$ 
12:    else // Pruned by Lemma 6
13:       $v.flag \leftarrow -1$ 
14:       $\forall s \in N_o[v]$ , send message  $(p_{i+1}[v] \times m[s, v])$  to  $s$ 
15:       $i \leftarrow i + 1$ 
16:   else if  $superstep \% 2 = 0$  then
17:      $p_{i+2}[v] = \sum(message)$ 
18:     if  $v.flag = 0$  and  $\bar{s}_i[v] > \theta_i^k$  and  $num \leq k$  then
19:       add  $v, \underline{s}_i[v], \bar{s}_i[v]$  to  $S_e, S_r$ , and  $S_u$ , respectively
20:     else if  $v.flag = 0$  and  $\bar{s}_i[v] \leq \theta_i^k$  then
21:        $v.flag \leftarrow -1; num \leftarrow num - 1$ 
22:      $i \leftarrow i + 1$ 
23:   else if  $superstep \% 2 = 1$  then
24:     if  $superstep = 3$  then
25:        $p_{i+1}[v] = \sum(message)$ 
26:     if  $v.flag = 0$  then
27:       update  $\underline{s}_i[v], \bar{s}_i[v]$ ; and add  $\underline{s}_i[v]$  to  $KL_b$ 
28:      $\forall s \in N_o[v]$ , send message  $(p_{i+1}[v] \times m[s, v])$  to  $s$ 
```

---

---

**Algorithm 3** PNN Master Computation Algorithm.

---

```
/* Aggregate information from workers */
1: if superstep %2 = 1 then
2:   | Compute  $\theta_i^k$  according to  $KL_b$ 
3: if superstep %2 = 0 then
4:   | if  $num \leq k$  and  $S_e \neq \emptyset$  then
5:     |   | foreach  $u \in S_e$  do
6:       |   |   | if  $\forall v \in S_e, v \neq u, \underline{s}_i[u] > \bar{s}_i[v]$  or  $\bar{s}_i[u] < \underline{s}_i[v]$  then
7:         |   |   |   | add  $v$  to  $kNN(q, k)$ ,  $S_e \leftarrow S_e - \{u\}$ 
8:         |   |   |   |  $v.flag \leftarrow 1$  and  $num \leftarrow num - 1$ 
9:     |   | if  $|kNN(q, k)| = k$  then
10:    |   | return the final result of  $kNN(q, k)$ 
```

---

Based on these, we present PNN, Algorithm 2 and Algorithm 3 depict its pseudo-codes. In general, PNN evaluates  $kNN(q, k)$  in two steps as follows.

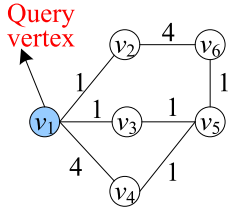
- (i) *Initialization.* First,  $superstep = 0$ . Let  $N_o[v] \subset V$  be the set of  $v$ 's out-neighbors, and  $\cup_{s \in N_o[v]}(v, s)$  be the set of  $v$ 's outgoing edges. For a vertex  $v \in V$ , the edge weight  $w(v, s) (s \in N_o[v])$  is normalized so that  $\sum_{s \in N_o[v]} w(v, s) = 1$ . Thus,  $\mathbf{M}$  becomes a column normalized matrix. Then, PNN sends normalized edge weight to  $v$ 's out-neighbors (lines 3-4 of Algorithm 2). Next,  $superstep = 1$ . PNN initializes parameters  $v.flag$  and  $i$  to 0. Counter  $i$  denotes the number of iterations, and label  $v.flag$  indicates whether  $v$  is a result vertex.  $v.flag$  has three possible values, with 0 indicating  $v$  being a potential result vertex of  $kNN(q, k)$ ,  $-1$  meaning  $v$  not belonging to  $kNN(q, k)$ , and 1 indicating  $v$  belonging to  $kNN(q, k)$ . Then, PNN computes  $M_{\max}[v]$ ,  $\underline{s}_0[v]$ ,  $\bar{s}_0[v]$ ,  $p_0[v]$ , inserts  $\underline{s}_0[v]$  into an aggregator  $KL_b$ , and sends updated  $p[v]$  to all  $v$ 's out-neighbors. Note that,  $KL_b$  is used to aggregate vertices'  $k$  lower bounds which will be reported to the master so that master can compute  $\theta_i^k$  (lines 5-7 of Algorithm 2). Thereafter,  $superstep = 2$ . PNN computes  $p_1[v]$ , initializes potential result vertices by Lemma 6, and sends  $p_1[v]$  to  $v$ 's out-neighbors (lines 8-15 of Algorithm 2). Note that, to avoid excessive aggregation cost, PNN only counts the number of potential result vertices  $num$ .
- (ii) *Iterative computation.* Each iteration of PNN includes two supersteps, one updates bounds if  $superstep \%2 = 1$ , and the other computes potential result vertices if  $superstep \%2 = 0$ . Specifically, if  $superstep \%2 = 0$ , PNN computes  $p_{i+2}[v]$  and updates potential result set  $S_e$  (lines 16-22 of Algorithm 2). Note that, the value of  $num$  could be very big initially, and its value becomes smaller as tighter bounds of vertices' RWR scores are derived iteration by iteration (lines 20-21 of Algorithm 2). Once if  $num \leq k$ ,  $kNN$  result vertices as well as the lower and upper bounds of their RWR scores shall be aggregated to compute rankings on the master. Aggregators  $S_l$  and  $S_u$  preserve lower and upper bounds of  $kNN$  result vertices' RWR scores respectively (lines 18-19 of Algorithm 2). Otherwise (i.e.,  $superstep \%2 = 1$ ), potential result vertex  $v$  updates bounds using (6) and (7) (lines 23-28 of Algorithm 2).

Master performs centralized computation between supersteps, with its pseudo-code shown in Algorithm 3. If  $superstep \%2 = 1$ , master computes  $\theta_i^k$  (lines 1-2 of Algorithm 3). Otherwise, if  $num \leq k$  and  $S_e$  is not empty, master ranks vertices in  $S_e$ . For each vertex

$u \in S_e$ , if  $u$  satisfies  $\forall v (\neq u) \in S_e, \underline{s}_i[u] > \bar{s}_i[v]$  or  $\bar{s}_i[u] < \underline{s}_i[v]$  [13], the ranking of  $u$  can be confirmed. Hence,  $u$  is added to the result set  $kNN(q, k)$  (lines 4-8 of Algorithm 3). If  $|kNN(q, k)|$  reaches  $k$ ,  $kNN$  search has been completed, and then, master terminates computation (lines 9-10 of Algorithm 3). Otherwise, a new iteration is performed (lines 8-28 of Algorithm 2). During the new iteration, the lower and upper bounds of  $s[v]$  are updated, and the value of  $num$ , the content of  $S_e$ , and  $kNN(q, k)$  could be updated accordingly. PNN updates its out-neighbors with new  $p[v]$  to complete this iteration.

**Example 3** Take the hybrid graph with its transition probability matrix shown in Figure 5a as an example. Suppose a query vertex is  $v_1$  and  $k = 4$ .

At superstep 0, for every vertex  $v$ , PNN normalizes the edge weights  $w(v, s) (s \in N_o[v])$ , and obtains the column normalized matrix **M**, which is depicted in Figure 5b.



	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$
$v_1$	0	1	1	4	0	0
$v_2$	1	0	0	0	0	4
$v_3$	1	0	0	0	1	0
$v_4$	4	0	0	0	1	0
$v_5$	0	0	1	1	0	1
$v_6$	0	4	0	0	1	0

	$v_1$	$v_2$	$v_3$	$v_4$	$v_5$	$v_6$
$v_1$	0	1/5	1/2	4/5	0	0
$v_2$	1/6	0	0	0	0	4/5
$v_3$	1/6	0	0	0	1/3	0
$v_4$	2/3	0	0	0	1/3	0
$v_5$	0	0	1/2	1/5	0	1/5
$v_6$	0	4/5	0	0	1/3	0

(a) Graph and matrix

(b) Normalized matrix

$id$	$M_{max}[v_i]$	$\underline{s}_0[v_i]$	$\bar{s}_0[v_i]$	$p_0[v_i]$	$p_1[v_i]$	$v_i, flag$	$S_e$	Message
1	4/5	0.5	0.9	1	0	0	N	{< $v_2, 1/6$ >, < $v_3, 1/6$ >, < $v_4, 2/3$ >}
2	4/5	0	0.4	0	0.17	0	N	{< $v_1, 0$ >, < $v_6, 0$ >}
3	1/3	0	0.17	0	0.17	0	N	{< $v_1, 0$ >, < $v_5, 0$ >}
4	2/3	0	0.33	0	0.67	0	N	{< $v_1, 0$ >, < $v_5, 0$ >}
5	1/2	0	0.25	0	0	0	N	{< $v_3, 0$ >, < $v_4, 0$ >, < $v_6, 0$ >}
6	4/5	0	0.4	0	0	0	N	{< $v_2, 0$ >, < $v_5, 0$ >}

(c) Superstep = 2

$id$	$\underline{s}_1[v_i]$	$\bar{s}_1[v_i]$	$p_1[v_i]$	$v_i, flag$	$S_e$
1	0.5	0.68	0.655	0	Y
2	0.04	0.14	0	0	Y
3	0.04	0.08	0	0	Y
4	0.17	0.25	0	0	Y
5	0	0.09	0.22	0	Y
6	0	0.12	0.13	0	Y

(d) Superstep = 3

$id$	$\underline{s}[v_i]$	$\bar{s}[v_i]$	$v_i, flag$	$kNN(v_1, 4)$
1	0.58	0.62	1	1
2	0.06	0.07	0	3
3	0.05	0.057	1	4
4	0.19	0.20	1	2

(e) Final result

$id$	$\underline{s}_3[v_i]$	$\bar{s}_3[v_i]$	$p_3[v_i]$	$p_4[v_i]$	$v_i, flag$	$S_e$	$kNN(v_1, 4)$
1	0.58	0.62	0	0.54	1	N	1
2	0.055	0.08	0.21	0.06	0	Y	—
3	0.052	0.06	0.18	0.01	0	Y	—
4	0.20	0.22	0.51	0.01	1	N	2
5	0.03	0.05	0.03	0.21	-1	N	N
6	0.02	0.05	0.07	0.18	-1	N	N

(f) Superstep = 8

**Figure 5** Example of a 4NN ( $k = 4$ ) query

At superstep 1, each vertex  $v_i$  receives messages from its in-neighbors. PNN computes  $M_{\max}[v_i]$ ,  $\underline{s}_0[v_i]$ ,  $\bar{s}_0[v_i]$ , and  $p_0[v_i]$ , and sets  $v_i.flag$  as 0. Master computes  $\theta_0^4 = 0$ , which is the 4<sup>th</sup> highest lower bound among all the vertices.

At superstep 2, PNN first computes  $p_1[v]$  for every vertex  $v$ . For instance,  $p_1[v_2] = p_0[v_1] \times m[v_1, v_2] + p_0[v_6] \times m[v_6, v_2] = 0.17$ . Then, according to Lemma 6, PNN computes  $num(= 6)$  since the upper bounds of all vertices' RWR scores are larger than  $\theta_0^4$ . The intermediate result is depicted in Figure 5a.

At superstep 3, PNN first computes  $p_2[v]$ . For example,  $p_2[v_1] = p_1[v_1] \times m[v_1, v_2] + p_1[v_3] \times m[v_1, v_3] + p_1[v_4] \times m[v_1, v_4] = 0.17 \times \frac{1}{5} + 0.17 \times \frac{1}{2} + 0.67 \times \frac{4}{5} = 0.655$ . Then, PNN updates lower and upper bounds of vertices' RWR scores as plotted in Figure 5b. For instance,  $\underline{s}_1[v_1] = \underline{s}_0[v_1] + c \times (1 - c) \times p_1[v_1] = 0.5$ , and  $\bar{s}_1[v_1] = \underline{s}_1[v_1] + c(1 - c)^2 p_2[v_1] + (1 - c)^3 M_{\max}[v_1] = 0.5 + 0.5^3 \times 0.655 + 0.5^3 \times 0.8 = 0.68$ . Master computes  $\theta_1^4 = 0.04$ . The algorithm repeats the iterations until  $k(= 4)$  ranked result vertices are found.

At superstep 8,  $\theta_3^4 = 0.052$ ,  $\bar{s}_3[v_5] < \theta_3^4$ , and  $\bar{s}_3[v_6] < \theta_3^4$ , then  $v_5$  and  $v_6$  are pruned, and  $S_e = \{v_1, v_2, v_3, v_4\}$ . Master performs sorting, with  $v_1$  ranked the first and  $v_4$  ranked the second. Next, we have  $S_e = \{v_2, v_3\}$  and  $kNN(v_1, 4) = \{v_1, v_4\}$ , as shown in Figure 5f. The process proceeds until  $|kNN(v_1, 4)| \in 4$ , with the final result  $kNN(v_1, 4) = \{v_1, v_4, v_2, v_3\}$  depicted in Figure 5e.

## 6.2 Complexity analysis

In this subsection, we analyze the complexity of PNN.

**Communication cost** Let  $\sharp supersteps$  be the number of supersteps, and  $|E_h|$  be the edge number of  $G_h$ , the total communication cost of PNN is  $O(\frac{|E_h| \times (\sharp supersteps + 3)}{2})$ , because when superstep  $\leq 2$ , there is  $3|E_h|$  communication cost. When superstep  $> 2$ , two cases need to be considered: (i) Superstep  $\%2 = 1$ , PNN updates bounds of vertices and sends messages, resulting in  $\frac{|E_h| \times (\sharp supersteps - 3)}{2}$  communication cost. (ii) Superstep  $\%2 = 0$ , PNN updates  $S_e$  or result set, without pumping messages.

**Computation cost** The total computation cost for PNN is  $O(|V| \times \sharp supersteps)$ , which is the inevitable cost due to the Pregel model.

## 7 Experimental evaluation

In this section, we conduct extensive performance studies to evaluate the efficiency, scalability, and effectiveness of our proposed hybrid graph based node similarity search approaches.

Our experiments employ three real datasets, viz., *Flickr*, *DBLP*, and *Check-in*. Table 2 summarizes the statistics of the datasets used, where  $|E_h|$  is the edge number of the cor-

**Table 2** Statistics of the datasets used

Graph	$ V $	$ E $	Dim.	$ E_h $	$d$
<i>Flickr</i>	20M	376,708,131	12	1,153,324,626	$L_2$ -norm
<i>DBLP</i>	10M	9,047,001	3 ~ 70	209,044,633	Jaccard distance
<i>Check-in</i>	3,680,126	561,465,540	2	1,187,956,508	$L_1$ -norm

responding hybrid graph. (i) *Flickr*, in which, for each image, we extract Color Layout [2] information associated with 12-dimensional visual features. If two images are tagged by the same user, we add an edge between them.  $L_2$ -norm is used to compare image features. (ii) *DBLP* is a citation network, which provides a list of research papers. Two papers are connected if one references another. Jaccard distance is employed to measure similarity. (iii) *Check-in* [32] is collected from Foursquare, where venues are extracted as nodes and two venues are linked via an edge if they are checked in by the same user.  $L_1$ -norm is utilized for *Check-in*.

We investigate the efficiency of  $t$ NN graph construction and the performance of node similarity search algorithms under various parameters listed in Table 3, where  $s^+$  is the maximal similarity score between any two vertices, and bold values denote the defaults. In every experiment, we vary one parameter, and fix others to their defaults. As reported in [13],  $c = 0.5$  is recommended. All  $t$ NN graph construction algorithms are implemented in MapReduce, and all query algorithms are implemented in Pregel. For every dataset, we generate 50 queries by selecting query node randomly, and the average query time is reported. Our experiments are conducted on a 18-node Dell cluster, in which one serves as the master node and others serve as the worker nodes. Each node has two processors with 12 cores, 128GB RAM, and 3TB disk.

## 7.1 $t$ NN graph construction and update costs

The first set of experiments verifies the performance of KDC.

We report the time required to construct the  $t$ NN graph, the number of distance computations (*Compdists* for short), and distance pruning rate defined as  $(1 - \frac{Compdists}{Totaldists})$  where  $Totaldists = |V| \times (|V| - 1)/2$ . We focus on exact  $t$ NN graph construction algorithm, and thus, we implemented the representative exact method DKNNG [23] (as described at the beginning of Section 5) based on MapReduce as a comparison, and also implemented ADKNNG, which integrates the allocation strategy presented in Section 5.1.3 with DKNNG to further improve performance, as another comparison.

We follow existing studies to set  $t$  as 20 because (i) it is pointed out in [18] that a small value of  $t$  performs well in practice. (ii) The accuracy of query algorithms under  $t = 20$  is reasonably high. (iii) When  $t \geq 20$ ,  $t$ NN graph construction cost increases significantly and the hybrid graph becomes bigger, resulting in more expensive query cost. Nonetheless, the accuracy has no clear improvement. These can be confirmed in the following experiments.

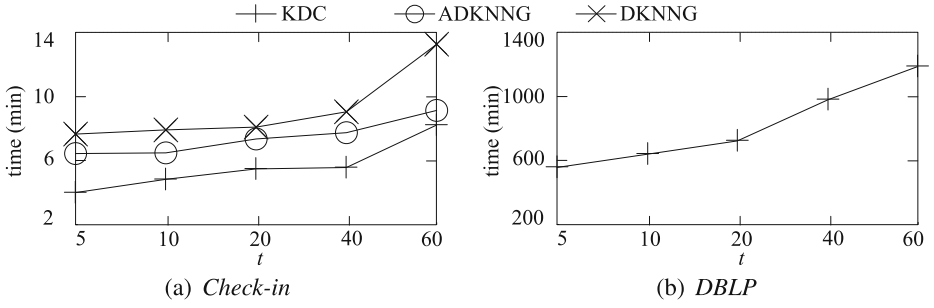
We report the result corresponding to the 20NN ( $t = 20$ ) graph, which is listed in Table 4. Note that, we follow existing work [6] to set the partitioning number  $m = 500$  and sampling size  $S = 4000$ , and “—” denotes that the actual *time* is more than 7 days. It is observed that, KDC outperforms DKNNG and ADKNNG significantly. The reason is that, KDC enables equal-size partition, and its pruning power is stronger (as can be seen from the column entitled *Pruning* of Table 4), contributed by Lemma 1 and Lemma 2. As an example, for

**Table 3** Parameter settings

Parameter	Range
The number $k$ of objects requested	10, 20, 30, 40, <b>50</b> , 70, 100
Balance factor $\alpha$	0, 0.1, 0.3, <b>0.5</b> , 0.7, 0.9, 1
$t$	5, 10, <b>20</b> , 40, 60
The number of cluster nodes	3, 6, 9, 12, 15, <b>18</b>

**Table 4** 20NN ( $t = 20$ ) graph construction cost (time: minutes)

Algorithms	Flickr			DBLP			Check-in		
	Time	Compdists	Pruning	Time	Compdists	Pruning	Time	Compdists	Pruning
DKING	—	—	—	—	—	—	8.083	73,465,234,317	98.915%
ADKING	4350.9	183,902,840,061,290	8.05%	—	—	—	7.367	39,624,032,563	99.415%
<b>KDC</b>	<b>1187.32</b>	<b>12,431,925,821,535</b>	<b>93.78%</b>	<b>722.43</b>	<b>49,999,787,885,159</b>	<b>0.00042%</b>	<b>5.467</b>	<b>487,339,918</b>	<b>99.993%</b>



**Figure 6**  $t$ NN graph construction cost vs.  $t$

*Flickr*, algorithm DKNNG cannot run within 7 days, ADKNN requires 4350.9 minutes, while KDC only requires 1187.32 minutes to complete 12 trillion of distance computation.

The results under different  $t$  values on *Check-in* and *DBLP* are plotted in Figure 6. It is observed that, as  $t$  grows, the number of the edges in the  $t$ NN graph increases with higher construction cost. Again, KDC still exceeds others. In addition to the construction cost of the  $t$ NN graph, we also evaluate update performance when new vertices are inserted into the graph and old vertices are deleted from the graph. Table 5 lists the average update costs of inserting and deleting 10 random vertices. As observed, the update is effective because it can complete within a few minutes.

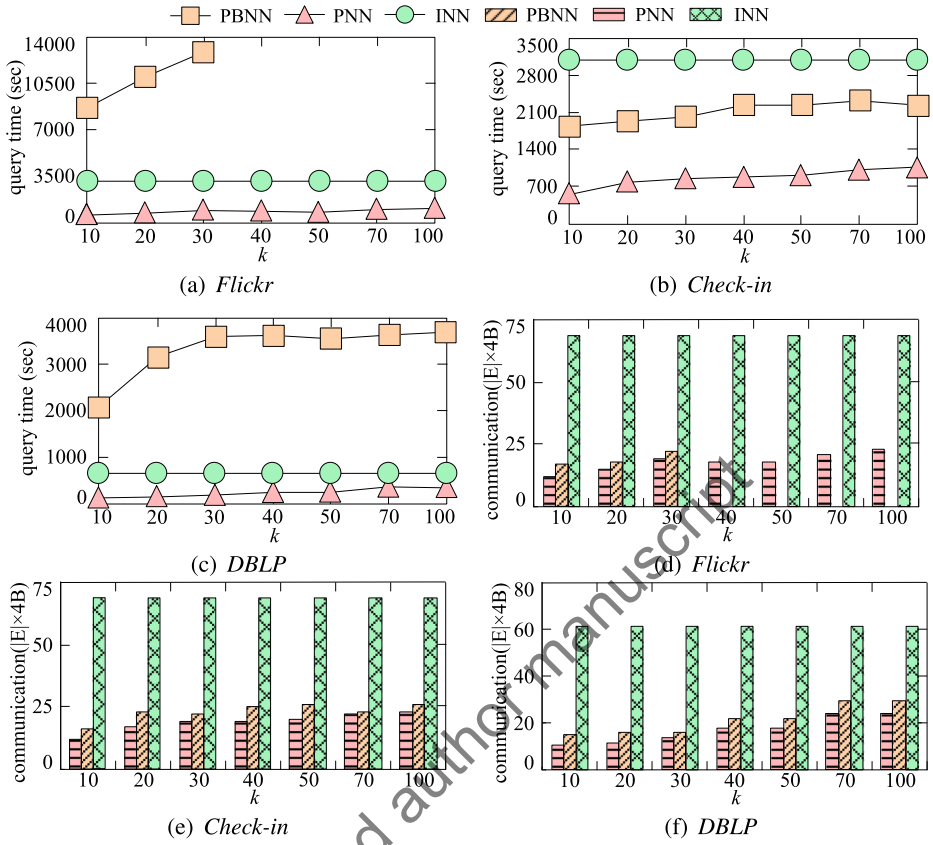
## 7.2 Search performance

**Evaluating PNN** The second set of experiments verifies the performance of PNN for supporting  $k$ NN queries, compared with PBNN and INN, which are implemented based on [16] and initiative iterative approach, respectively (as stated in RWR Computation of Section 2). The results under different  $k$  values is depicted in Figure 7. Note that, we exclude those results with the running time beyond 14,000 seconds. The first observation is that, for PNN and PBNN, as  $k$  increases, more vertices require evaluation, incurring longer search time and more communication cost. INN is not sensitive to  $k$  since the similarities of all vertices must be computed until convergence in the initiative iterative approach. The second observation is that, PNN exceeds PBNN and INN by 10 times and 4 times respectively on average, and INN performs much better than PBNN on *Flickr* and *DBLP*, contributed by two main reasons below. First, even though the communication cost of PBNN is less than INN, PBNN needs to update candidate vertices at every iteration, which requires using persistent aggregators. The overhead for maintaining the aggregators is high since candidate vertices keep changing. This is extremely costly when an input graph is huge. Second, our derived bound is much tighter than the bound used in [16].

**Table 5** Update cost of KDC

	Check-in	DBLP	Flickr
Insertion (seconds)	131.5	153	156.4
Deletion (seconds)	183	212	332.8

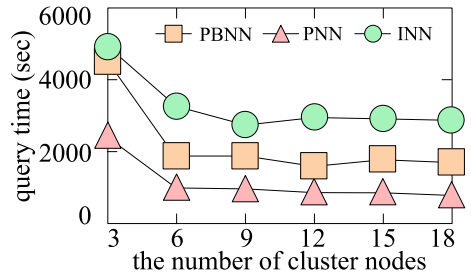




**Figure 7**  $k$ NN query performance vs.  $k$

**Effect of cluster nodes** The third set of experiments aims to explore the impact of cluster nodes on search performance. We vary the number of cluster nodes from 3 to 18, and illustrate the results on *Check-in* in Figure 8. As expected, the query cost first drops and then stays stable or ascends when the number of cluster nodes grows. This is because, both computational power and communication cost increases with more cluster nodes. Again, PNN exceeds PBNN and INN.

**Figure 8** Query time vs. the number of cluster nodes



**Effect of  $\alpha$**  The fourth set of experiments studies the impact of parameter  $\alpha$  on the performance of PNN. Figure 9 plots the results. It is observed that, varying  $\alpha$  from 0 to 1 causes slight fluctuations in running time. This is because, edge weight changes with the growth of  $\alpha$ , which affects computation. In particular, when  $\alpha = 0$  or 1, input graph has much fewer edges, which helps to reduce computation cost.

**Effect of  $t$  on search efficiency** Next, we investigate the impact of  $t$  on search performance, with the results depicted in Figure 10. It is observed that, query cost becomes more expensive as  $t$  grows. This is because,  $t$  decides the number of the edges maintained by each vertex in the  $t$ NN graph and the  $|E_h|$  value of the hybrid graph accordingly. When the hybrid graph contains more edges, it needs longer time in completing a search.

**Effect of  $t$  on search accuracy** Parameter  $t$  affects not only search performance but also the accuracy of the returned results. Note that  $t$  determines the number of the nearest neighbors maintained by each node in the  $t$ NN graph, which represents the knowledge of attribute similarity between vertices. Although RWR performs random walk that enables the transition of nodes' similarity scores, we believe the selection of  $t$  still affects the accuracy of returned results. We take the query results returned by the naïve search method presented at the beginning of Section 3 as the accurate results, and report the percentage of accurate results returned by PNN as *accuracy* in Table 6. It is observed that, for  $k$ NN query, given a value of  $t$ , the accuracy first goes up and then drops with  $k$  increases. This is because, there is nearly no similar paper to a given one when  $k > 50$  by our manual checking, hence either naïve search method or PNN returns  $k^{th}$  ( $k > 50$ ) paper randomly, incurring result difference. Given the improvement on search efficiency brought by PNN, we claim that the accuracy of PNN under  $t = 20$  is reasonably high. It also justifies that using the hybrid graph to support node similarity search is feasible. It guarantees search performance even when the input graph is massive, and meanwhile, it is able to achieve relatively high accuracy. (e.g., near 80% in most of the cases).

### 7.3 Case study

To verify the effectiveness of our proposed methods, we conduct a case study on *DBLP*. We perform 20NN ( $k = 20$ ) search via RWR and PNN respectively under different values of  $\alpha$ . We present the top-5 papers that are most related to paper  $p_0$  in Table 8 but ignore the remaining 15 papers because it is enough for us to verify the effectiveness of our approaches. The original paper titles are listed in Table 7. It is observed that RWR returns  $\{p_i \mid (1 \leq i \leq 5)\}$  as the result. However, papers  $p_1$ ,  $p_4$ , and  $p_5$  do not share any common keyword with  $p_0$ . As expected, our approach is much more flexible. Users can obtain preferred results by tuning parameter  $\alpha$ . As an example, when  $\alpha = 0.7$ , the result is  $\{p_2, p_1, p_6, p_7, p_8\}$ , which

**Figure 9** Query time vs.  $\alpha$

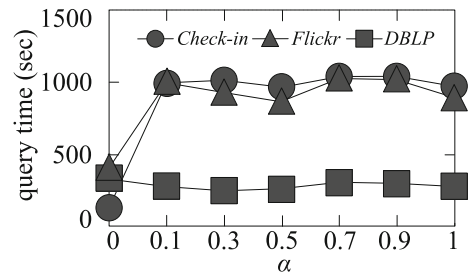


Figure 10 Search performance vs.  $t$

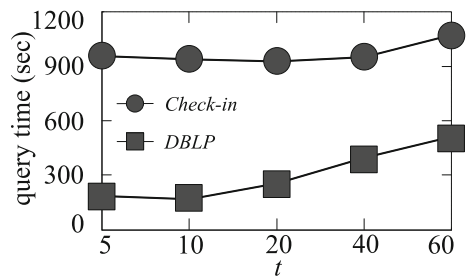


Table 6  $k$ NN query accuracy on DBLP

$k \backslash t$	10	20	30	40	50	70	100
5	0.55	0.7	0.761	0.633	0.532	0.423	0.322
10	0.47	0.64	0.773	0.718	0.634	0.511	0.39
20	0.36	0.47	0.718	0.805	0.768	0.676	0.555
40	0.3	0.29	0.479	0.663	0.792	0.835	0.685
60	0.28	0.25	0.392	0.58	0.684	0.838	0.789

Table 7 Information of original papers

Paper	Title
$p_0$	an efficient $k$ means clustering algorithm
$p_1$	maximum certainty data partitioning
$p_2$	bayesian ying yang machine clustering and number of clusters
$p_3$	parametric and non parametric unsupervised cluster analysis
$p_4$	comparative analysis of statistical pattern recognition methods
$p_5$	lecture notes in computer science
$p_6$	an efficient $k$ means clustering algorithm based on influence factors
$p_7$	an efficient pso based clustering algorithm
$p_8$	an efficient clustering algorithm based on local optimality of $k$ means
$p_9$	an efficient line symmetry based $k$ means algorithm
$p_{10}$	the global $k$ means clustering algorithm

**Table 8** Case study on *DBLP*

Query	RWR	PNN				
		$(\alpha = 1)$	$(\alpha = 0.7)$	$(\alpha = 0.5)$	$(\alpha = 0.3)$	$(\alpha = 0)$
$p_0$	$p_1$	$p_1$	$p_2$	$p_6$	$p_6$	$p_6$
	$p_2$	$p_2$	$p_1$	$p_7$	$p_7$	$p_9$
	$p_3$	$p_3$	$p_6$	$p_8$	$p_8$	$p_8$
	$p_4$	$p_4$	$p_7$	$p_9$	$p_9$	$p_7$
	$p_5$	$p_5$	$p_8$	$p_{10}$	$p_{10}$	$p_{10}$

explicitly considers both graph structure relevance and node attribute similarity. When  $\alpha = 1$ , PNN returns the same result as RWR since it only considers the graph structure. When  $\alpha = 0$ , PNN returns the top- $k$  papers with the highest attribute similarity scores (as  $p_0$ ). This case study confirms that considering both graph structure relevance and vertex attribute correlation in node similarity search is significant, and offers users more flexible search options (Table 8).

## 8 Conclusions

In this paper, we propose an efficiently distributed framework to support node similarity search on massive graphs, which consider both graph structure correlation and node attribute similarity in metric spaces. The framework consists of preprocessing stage and query stage. In the preprocessing stage, a new parallel algorithm KDC is presented for forming a hybrid graph, using efficient metric pruning techniques and allocation strategies to avoid unnecessary communication and computation costs. In the query stage, based on the formed hybrid graph, we present PNN using RWR for answering  $k$ NN queries. Tight similarity bounds are derived to rapidly shrink the search region. Extensive experimental evaluation on three real data sets demonstrates the effectiveness, scalability, and efficiency of our proposed approaches. In the future, we intend to explore how to further reduce the number of supersteps in order to improve the efficiency of node similarity search on massive graphs.

**Acknowledgments** This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, the NSFC under Grants No. 61972338 and 61802344, the NSFC-Zhejiang Joint Fund under Grant No. U1609217, and the ZJU-Hikvision Joint Project. Yunjun Gao is the corresponding author of the work.

## References

1. Batarfi, O., Shawi, R.E., Fayoumi, A.G., Nouri, R., Beheshti, S., Barnawi, A., Sakr, S.: Large scale graph processing systems: Survey and an experimental evaluation. *Clust. Comput.* **18**(3), 1189–1213 (2015)
2. Batko, M., Kohoutková, P., Novak, D.: Cophir image collection under the microscope. In: *SISAP*, pp. 47–54 (2009)
3. Boutet, A., Kermarrec, A., Mittal, N., Taïani, F.: Being prepared in a sparse world: The case of  $k$ NN graph construction. In: *ICDE*, pp. 241–252 (2016)
4. Chen, L., Gao, Y., Li, X., Jensen, C.S., Chen, G.: Efficient metric indexing for similarity search. In: *ICDE*, pp. 591–602 (2015)

5. Chen, L., Gao, Y., Chen, G., Zhang, H.: Metric all- $k$ -nearest-neighbor search. *IEEE Trans. Knowl. Data Eng.* **28**(1), 98–112 (2016)
6. Chen, G., Yang, K., Chen, L., Gao, Y., Zheng, B., Chen, C.: Metric similarity joins using mapreduce. *IEEE Trans. Knowl. Data Eng.* **29**(3), 656–669 (2017)
7. Cheng, H., Zhou, Y., Yu, J.X.: Clustering large attributed graphs: A balance between structural and attribute similarities. *TKDD* **5**(2), 12:1–12:33 (2011)
8. Cohen, S., Kimelfeld, B., Koutrika, G.: A survey on proximity measures for social networks. In: *Search Computing - Broadening Web Search*, pp. 191–206 (2012)
9. Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
10. Dong, W., Charikar, M., Li, K.: Efficient  $k$ -nearest neighbor graph construction for generic similarity measures. In: *WWW*, pp. 577–586 (2011)
11. Dong, Y., Zhang, J., Tang, J., Chawla, N.V., Wang, B.: Coupledldp: Link prediction in coupled networks. In: *SIGKDD*, pp. 199–208 (2015)
12. Fujiwara, Y., Nakatsuji, M., Onizuka, M., Kitsuregawa, M.: Fast and exact top- $k$  search for random walk with restart. *PVLDB* **5**(5), 442–453 (2012)
13. Fujiwara, Y., Nakatsuji, M., Shiokawa, H., Mishima, T., Onizuka, M.: Efficient ad-hoc search for personalized pagerank. In: *SIGMOD*, pp. 445–456 (2013)
14. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: GraphX: Graph processing in a distributed dataflow framework. In: *OSDI*, pp. 599–613 (2014)
15. Jeh, G., Widom, J.: Simrank: A measure of structural-context similarity. In: *SIGKDD*, pp. 538–543 (2002)
16. Khemmarat, S., Gao, L.: Fast top- $k$  path-based relevance query on massive graphs. *IEEE Trans. Knowl. Data Eng.* **28**(5), 1189–1202 (2016)
17. Low, Y., Gonzalez, J., Kyrola, A., Bickson, D., Guestrin, C., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning in the cloud. *PVLDB* **5**(8), 716–727 (2012)
18. Ma, H., Zhu, J., Lyu, M.R., King, I.: Bridging the semantic gap between image contents and tags. *IEEE Trans. Multimedia* **12**(5), 462–473 (2010)
19. Maehara, T., Akiba, T., Iwata, Y., Kawarabayashi, K.: Computing personalized pagerank quickly by exploiting graph structures. *PVLDB* **7**(12), 1023–1034 (2014)
20. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., Czajkowski, G.: Pregel: A system for large-scale graph processing. In: *SIGMOD*, pp. 135–146 (2010)
21. Meng, F., Rui, X., Wang, Z., Xing, Y., Cao, L.: Coupled node similarity learning for community detection in attributed networks. *Entropy* **20**(6), 471 (2018)
22. Pan, J., Yang, H., Faloutsos, C., Doytshu, P.: Automatic multimedia cross-modal correlation discovery. In: *SIGKDD*, pp. 653–658 (2004)
23. Plaku, E., Kavradi, L.E.: Distributed computation of the  $k$ nn graph for large high-dimensional point sets. *J. Parallel Distrib. Comput.* **67**(3), 346–359 (2007)
24. Sarkar, P., Moore, A.W.: Fast nearest-neighbor search in disk-resident graphs. In: *SIGKDD*, pp. 513–522 (2010)
25. Sarkar, P., Moore, A.W.: A tractable approach to finding closest truncated-commute-time neighbors in large graphs. *arXiv:1206.5259* (2012)
26. Shao, B., Wang, H., Li, Y.: Trinity: A distributed graph engine on a memory cloud. In: *SIGMOD*, pp. 505–516 (2013)
27. Shin, K., Jung, J., Sael, L., Kang, U.: Bear: Block elimination approach for random walk with restart on large graphs. In: *SIGMOD*, pp. 1571–1585 (2015)
28. Tian, Y., Balmin, A., Corsten, S.A., Tatikonda, S., McPherson, J.: From “think like a vertex” to “think like a graph”. *PVLDB* **7**(3), 193–204 (2013)
29. Trad, M.R., Joly, A., Boujemaa, N.: Distributed  $k$ NN-graph approximation via hashing. In: *ICMR*, p. 43 (2012)
30. Wu, Y., Jin, R., Zhang, X.: Fast and unified local search for random walk based  $k$ -nearest-neighbor query in large graphs. In: *SIGMOD*, pp. 1139–1150 (2014)
31. Xu, G., Fu, B., Gu, Y.: Point-of-interest recommendations via a supervised random walk algorithm. *IEEE Intell. Syst.* **31**(1), 15–23 (2016)
32. Yang, D., Zhang, D., Qu, B.: Participatory cultural mapping based on collective behavior data in location-based social networks. *ACM TIST* **7**(3), 30 (2016)
33. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: *NSDI*, pp. 15–28 (2012)

34. Zhang, C., Shou, L., Chen, K., Chen, G., Bei, Y.: Evaluating geo-social influence in location-based social networks. In: CIKM, pp. 1442–1451 (2012)
35. Zhang, Q., Li, M., Deng, Y., Mahadevan, S.: Measure the similarity of nodes in the complex networks. arXiv:[1502.00780](#) (2015)
36. Zhang, Y., Huang, K., Geng, G., Liu, C.: Fast  $k$ NN graph construction with locality sensitive hashing. In: PKDD, pp. 660–674 (2013)
37. Zhou, Y., Cheng, H., Yu, J.X.: Graph clustering based on structural/attribute similarities. PVLDB **2**(1), 718–729 (2009)

Accepted author manuscript