DISK: A Distributed Framework for Single-Source SimRank with Accuracy Guarantee

Yue Wang*
Shenzhen Institute of Computing Sciences, Shenzhen University
yuewang@sics.ac.cn

Ruiqi Xu*
University of Edinburgh
ruiqi.xu@ed.ac.uk

Zonghao Feng
Yulin Che
HKUST
zfengah,yche@cse.ust.hk

Lei Chen
HKUST
leichen@cse.ust.hk

Qiong Luo
HKUST
luo@cse.ust.hk

Rui Mao†
Shenzhen Institute of Computing Sciences, Shenzhen University
mao@sics.ac.cn

ABSTRACT

Measuring similarities among different nodes is important in graph analysis. SimRank is one of the most popular similarity measures. Given a graph $G(V, E)$ and a source node u, a single-source SimRank query returns the similarities between u and each node $v \in V$. This type of query is often used in link prediction, personalized recommendation and spam detection. While dealing with a large graph is beyond the ability of a single machine due to its limited memory and computational power, it is necessary to process single-source SimRank queries in a distributed environment, where the graph is partitioned and distributed across multiple machines. However, most current solutions are based on shared-memory model, where the whole graph is loaded into a shared memory and all processors can access the graph randomly. It is difficult to deploy such algorithms on shared-nothing model. In this paper, we present DISK, a distributed framework for processing single-source SimRank queries. DISK follows the linearized formulation of SimRank, and consists of offline and online phases. In the offline phase, a tree-based method is used to estimate the diagonal correction matrix of SimRank accurately, and in the online phase, single-source similarities are computed iteratively. Under this framework, we propose different optimization techniques to boost the indexing and queries. DISK guarantees both accuracy and parallel scalability, which distinguishes itself from existing solutions. Its accuracy, efficiency, parallel scalability and scalability are also verified by extensive experimental studies. The experiments show that DISK scales up to graphs of billions of nodes and edges, and answers online queries within seconds, while ensuring the accuracy bounds.

PVLDB Reference Format:
Yue Wang, Ruiqi Xu, Zonghao Feng, Yulin Che, Lei Chen, Qiong Luo, and Rui Mao. DISK: A Distributed Framework for Single-Source SimRank with Accuracy Guarantee. PVLDB, 14(3): 351-363, 2021.
doi:10.14778/3430915.3430925

1 INTRODUCTION

Nowadays, many real world information systems, *e.g.,* social media and online shopping platforms, use graphs to model different data objects and their relationships: nodes represent data objects, and edges represent the relationships among them. Measuring similarity among data objects plays a key role in data analysis and mining. Several link-based similarity measures have been proposed, including Personalized PageRank(PPR)[12], P-Rank [46], Random Walk with Restart(RWR)[30], and so on. Among them, SimRank [11] is one of the most promising, and has a comparable impact to PageRank for link-based ranking [17, 20].

The intuition behind SimRank is two-fold: *a)* two nodes are *similar* if they are linked by similar nodes; *b)* two identical nodes have the similarity of 1. SimRank is defined recursively, thus it can aggregate similarities of the multi-hop neighbors of the original pair of nodes, and produce high-quality results. Consequently, SimRank has received a lot of research attention since it was first introduced [6–9, 14–17, 19, 21, 23, 25, 33–37, 40–43, 45]. Given an unweighted directed graph $G(V, E)$, the SimRank score of two nodes $u, v \in V$ is formulated as follows:

$$s(u,v) = \begin{cases} 1 & (u = v), \\ \displaystyle\sum_{u' \in I_u, v' \in I_v} \frac{c \times s(u', v')}{|I_u||I_v|} & (u \neq v), \end{cases} \tag{1}$$

where I_u denotes the set of in-neighbors of node u, and $c \in (0, 1)$ is the *decay factor* which is usually set to 0.6 [20] or 0.8 [11].

Processing and analysing large-scale graphs becomes necessary due to the large amount of data generated in today's world. For example, as of fall 2011, Twitter had over 100 million users worldwide [2], and there were over 721 million active users on Facebook [31]. Single-source SimRank queries on such social media are useful for recommending friends, tweets, groups and so on. While efficiently dealing with large graph problems is beyond the ability of a single machine due to the limited memory and concurrency, it is necessary to develop efficient distributed algorithms for graphs which are too large to fit into the memory of a single machine.

While most current work propose efficient SimRank algorithms on a single machine [14, 19, 25, 26, 38], only a few aim to design efficient and effective algorithms on a shared-nothing model, where a large-scale graph is partitioned and distributed among multiple machines. [3] firstly introduces a MapReduce procedure, called

*Co-first authors.
†Corresponding author.

Proceedings of the VLDB Endowment, Vol. 14, No. 3 ISSN 2150-8097.
doi:10.14778/3430915.3430925

Delta-SimRank, which uses the node-pair graph structure to iteratively compute SimRank. It provides techniques to reduce the data amount transformed in each map/reduce phase from $O(I_{avg}^2 n^2)$ to $O(I_{avg}^2 M)$, where M is the number of non-zero similarities. However, it is an all-pairs solution and needs to maintains $O(V^2)$ node pairs simultaneously, which would lead to high computational cost and space usage, though MapReduce can dump the data to disk. In [3], the largest graph processed has only 269 037 edges, it cannot efficiently process single-source queries over large graphs. Meanwhile, [28] presents UniWalk, which parallelizes random walk sampling over the vertex-centric model. It costs $O(RL^2)$ time for a single-source query, where R is the sample size and L is the path length. However, UniWalk is only applicable to undirected graph and does not provide theoretical accuracy guarantee (due to truncated random walks). Recently, [18] introduces CloudWalker, which utilizes Spark to parallelize SimRank computation, however, it does not have an accuracy guarantee *w.r.t.* query results. Particularly, the Jacobi method used in [18] for offline index does not guarantee the convergence (details in Section 3). Besides, the Monte Carlo method for online queries does not provide any error bound of final SimRank scores. As shown in [36], the maximum error of CloudWalker can be larger than 1.

Moreover, none of the existing distributed solutions provide any theoretical guarantee of the parallel scalability, *i.e.,* the cost decreases with more machines being added. When we turn to multiple machines, we always want to make use of the additional resources to speed up the execution of our algorithms. However, parallel scalability is not always grounded [5]. It is still unknown whether single-source SimRank queries can be processed *accurately* and *more efficiently* when more machines are available, either from the theoretical aspect or in experimental results.

We present DISK, a Distributed Framework for processing single-source SimRank queries. The contributions are as follows.

Accurate. DISK follows the linearized formulation of SimRank, but it is different to current solutions. To our knowledge, this is the first *distributed* solution for single-source SimRank, which is proved to be *accurate*. The accuracy is guaranteed by our theoretical result of combining the errors of the estimated diagonal correction matrix $\hat{D}$, and the truncated series of the recursive SimRank equation.

Parallel Scalable. DISK consists of an offline indexing phase and an online query phase. In the offline phase, we propose a tree-based method to estimate $\hat{D}$. The tree-based method differs from current ones in that the sampled forest is discarded at the end of each round, and thus we do not need to store them. In the online phase, the single-source SimRank is processed iteratively across different workers. Our cost analysis indicates that both the offline phase and the online phase are parallel scalable.

Optimizations. We propose effective optimizations for DISK. For the offline indexing, we propose a sample reduction method which reduces the number of sampled forests. We also boost each round by adopting the pointer-jumping technique, which is usually used in a shared-memory model. For online queries, we propose an early stop technique, which uses the values computed in real time to reduce the number of iterations. We also present a method of refining underlying partitions, which mitigates the skewness of the workload for online queries. The optimization techniques speed up the computation by 4.2 and 2.4 times on average, in the offline and online phases, respectively. Besides, they do not undermine the parallel scalability and accuracy.

Experimental Study. We conduct extensive experiments using real-life and synthetic graphs. We experimentally verify the accuracy bounds for both the offline indexing and online queries. We use graphs with up to 1 billion nodes and 42 billion edges, to test the efficiency of DISK, along with its optimization techniques, and then compare DISK with other methods. Results show that the accuracy and the parallel scalability of DISK are in accord with our theoretical analysis. In additional, its scalability outperforms the state-of-the-art competitors while achieving comparable efficiency.

This paper is organized as follows. Preliminaries are in Section 2. We give an overview of DISK and show its accuracy results in Section 3. The offline phase and online phase are presented in Sections 4 and 5, respectively. Optimization techniques for offline indexing and online query are presented in Sections 6 and 7. Experimental results are shown in Section 8. We present related works in Section 9 and conclude the paper in Section 10.

2 PRELIMINARIES

In this section, we first give the problem definition, present the distributed environment for the graph computation, and then discuss the challenges of designing a distributed solution for SimRank. Notations and symbols are summarized in Table 1.

2.1 SimRank

Given an unweighted directed graph $G(V, E)$ ($|V| = n, |E| = m$), there exists a unique SimRank matrix S for G. According to [45], the matrix form of Eq. (1) is:

$$S = cP^{\top}SP \vee I, \tag{2}$$

where $\vee$ denotes the element-wise maximum operation, I is an identity matrix, P is the column normalized adjacency matrix of G, and $P^{\top}$ denotes the transpose of P. The difficulty of computing SimRank lies in that Eq. (2) is recursive and nonlinear [15]. We study the following approximate single-source queries for SimRank.

Definition 2.1 (Approximate Single-source Queries). *Given a node $u \in G$, an error bound ϵ and a failure probability δ, an approximate single-source SimRank query returns an estimated value $\hat{s}(u, v)$ for each $v \in G$, such that*

$$|\hat{s}(u, v) - s(u, v)| \leq \epsilon, \tag{3}$$

holds for any v with at least $1 - \delta$ probability.

Since we can always return $s(u, v) = 1$ when $u = v$, in this paper, we only consider the accuracy of $\hat{s}(u, v)$ when $u \neq v$.

2.2 Distributed Graph Computation

We adopt a coordinator-based shared-nothing environment, which consists of a master (coordinator) W_0 and a set of p workers $\mathcal{W} = \{W_1, \cdots, W_p\}$. The workers (including W_0) are pairwise connected by bi-directional communication channels. Meanwhile, G is partitioned and distributed across these workers, and fragmented into $\mathcal{F} = (F_1, \cdots, F_p)$, where each fragment $F_i = (V_i, E_i)$ is a subgraph of G, such that $V = \cup_{i\in[1,p]}V_i$ and $E = \cup_{i\in[1,p]}E_i$. Each worker W_i hosts and works on a fragment F_i of G. Let $|G| = |V| + |E|$. We also assume G is evenly partitioned, *i.e.,* $\max_{i\in[1,p]} |F_i| = O(\frac{|G|}{p})$,

Table 1: Notations and Symbols

Notation	Description
p	The number of workers
$\mathcal{W}$	A set of workers $\{W_1, \cdots, W_p\}$
c	The decay factor of SimRank
N	The number of samples in the offline phase
T	The number of truncated terms of linearized SimRank
I_v	The set of in-neighbors of v
O_v	The set of out-neighbors of v
$\mathbf{e}_i$	A unit vector whose i-th position is 1
P	The column-normalized transition matrix of G
$P^\top$	The transpose of P
S	The SimRank matrix
$S_{u,*}$	The u-th row of matrix S
$S_{*,u}$	The u-th column of matrix S
D	The diagonal correction matrix of SimRank
F_i	A fragment which is the subgraph (V_i, E_i) of G
$\mathbf{q}_u^{(t)}$	the distribution of t-th reverse random walk from u

where $|F_i| = |V_i| + |E_i|$. We also assume $p \ll |G|$, since in practice the number of machines is much less than the size of a graph.

To simplify the algorithm design and analysis, we assume an edge-cut partitioning strategy. Specifically, if a node v is assigned to F_i, then its incident edges (incoming and outgoing edges) are assigned to F_i. Instead of designing a new partitioning strategy from scratch, we utilize existing ones, such as XtraPuLP[27]. How to design a partitioner is orthogonal to the design of our algorithms, and is beyond the discussion of this paper.

We follow the Bulk Synchronous Parallel (BSP) model for computing. Under BSP, computation and communication are performed in supersteps: at each superstep, each worker W_i first reads messages (sent in the last superstep) from other workers, and then performs the local computation, and finally sends messages (to be received in the next superstep) to other workers. The barrier synchronization of each superstep is coordinated by W_0. One desired property of a distributed graph algorithm ρ is parallel scalability, *i.e.,* T_ρ (time taken by ρ) decreases with an increasing p, indicating the more resources added, the more efficient ρ is. To ensure parallel scalability, it is crucial to make the workload of ρ balanced across different workers. There are specific models for distributed graph processing, such as the vertex-centric model [24] and its variant [39]. The reason we choose the BSP model is as follows. Vertex-centric model simplifies the design of distributed graph algorithms by "thinking like a vertex" and modeling each vertex as a worker. However, in our application, each worker holds a fragment of G instead of a single vertex and communication is among these workers instead of adjacent vertices. Besides, vertex-centric model follows BSP and our solution can easily be adopted to it.

2.3 Challenges

The challenges of designing a distributed solution for SimRank queries, lie in that it is hard to maintain the parallel scalability and the accuracy, simultaneously.

Accurate but Not Parallel Scalable. Most recent algorithms, which have an accuracy guarantee [19, 29, 32, 38], rely on sampling a bunch of random walks, either for index or query. The accuracy is based on the fact that the SimRank score is the probability of two random walks meeting. It is easy to parallelizing random walk sampling over a shared-memory model, since samples are independent. However, when a graph is partitioned over multiple machines, running multiple random walks simultaneously leads to congestion (details in Section 4.2), *i.e.,* some nodes may receive or send large numbers of messages, which undermines the parallel scalability.

Parallel Scalable but Not Accurate. A promising solution is to use the linearized technique [15, 18, 23] to remove the non-linearity of Eq.(2), where retrieving single-source SimRank scores becomes computing a column of S. It can be further transformed to matrix-vector multiplications, and simulated by the structure of G. This method has good parallel scalability. However, none of the current linearized based methods guarantee the accuracy of the final similarities, due to the multiple sources of errors (details in Section 3). Besides that, the linearized equation relies on estimating a diagonal correction matrix D beforehand, which still challenges parallel scalability and accuracy.

3 BASIC IDEA OF DISK

We follow the linearized formula to estimate SimRank scores, which is firstly proposed in [15] to overcome the non-linearity of Eq.(2). According to [15], S satisfies:

$$S = cP^\top SP + D = \sum_{t=0}^{\infty} c^t P^{\top t} D P^t, \tag{4}$$

where D is called the *diagonal correction matrix*, making up the diagonal of $cP^\top SP$ to 1's. There are two issues when utilizing Eq.(4) for SimRank computation: (1) D is unknown in advance, and; (2) Eq.(4) is an infinite series, making it impractical to compute all terms. Therefore, we make two approximations: we compute an estimated diagonal matrix $\hat{D}$, and only compute the truncated version of Eq.(4). Specifically, let the diagonal matrix $\hat{D}$ be an approximation of D, we use the following equation to approximate S:

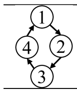
$$\hat{S}^{(T)} = \sum_{t=0}^{T} c^t P^{\top t} \hat{D} P^t. \tag{5}$$

As a result, the error of $\hat{S}^{(T)}$ has two factors: one is from the error of $\hat{D}$, the other is from the truncated series. To get the final error of $\hat{S}^{(T)}$, we present the following result.

Theorem 1. *If $\|\hat{D} - D\|_{max} < \epsilon_d$, then for any $v, u \in V$ and $v \neq u$, $|\hat{S}^{(T)}_{v,u} - S_{v,u}| \le \frac{c(1-c^T)\epsilon_d}{1-c} + c^{T+1}$.* □

From above, we can see that as $\epsilon_d \to 0$ and $T \to \infty$, $\hat{S}^{(T)} \to S$. In light of Theorem 1, our method consists of offline and online phases: in the offline phase, we estimate $\hat{D}$ with ϵ_d error bound; in the online query phase, given a node $u \in V$, we compute its single-source SimRank by retrieving the u-th column of $\hat{S}^{(T)}$. Note that this is not the first attempt to approximate SimRank scores using Eq.(5) [15, 18, 23]. However, we are the first to present the error bound ϵ *w.r.t.* both ϵ_d and T, which ensures the final accuracy of our distributed solution. The linearization technique is also used in [15, 18, 23], but they all suffer from the accuracy issue. Eq.(4) is firstly proposed in [15], but they approximate D as $(1-c)I$, which would lead to the wrong SimRank scores, as pointed out in [36, 44]. [23] formulates D as the solution to a linear system, and solves an approximate linear system to get $\hat{D}$. However, as analysed in [29], [23] does not provide any formal error analysis of $\hat{D}$ and its effects on accuracy of S, and their Gauss-Seidel based method does not guarantee the convergence. Recently, [18] uses the

Table 2: Example: the Jacobi method for computing D [18] diverges ($T = 10$, $c = 0.6$)

G	k	1	2	3	4	5	$\cdots$	10	Truth
	$\hat{\mathbf{d}}$	$\begin{bmatrix}1\\1\\1\\1\end{bmatrix}$	$\begin{bmatrix}-0.49\\-0.49\\-0.49\\-0.49\end{bmatrix}$	$\begin{bmatrix}1.73\\1.73\\1.73\\1.73\end{bmatrix}$	$\begin{bmatrix}-1.58\\-1.58\\-1.58\\-1.58\end{bmatrix}$	$\begin{bmatrix}3.36\\3.36\\3.36\\3.36\end{bmatrix}$	$\cdots$	$\begin{bmatrix}-21.39\\-21.39\\-21.39\\-21.39\end{bmatrix}$	$\begin{bmatrix}0.4\\0.4\\0.4\\0.4\end{bmatrix}$

Jacobi method to compute $\hat{D}$, using an approximate linear system based on Eq.(5). Specifically, [18] considers n linear constraints, *i.e.*, $\forall i = 1, \cdots, n$, $\mathbf{e}_i^\top \hat{S}^{(T)} \mathbf{e}_i = 1$, which can be compressed into the following linear system:

$$(I + cP \circ P + c^2 P^2 \circ P^2 + \cdots + c^T P^T \circ P^T)^\top \hat{\mathbf{d}} = \mathbf{1}, \tag{6}$$

then a Jacobi method is used to compute $\hat{\mathbf{d}}$ iteratively. However, the convergence of $\hat{\mathbf{d}}$ is not guaranteed either. Consider a simple graph in Table 2, the results of the first 10 iterations of the Jacobi method indicate the divergence. Besides, Eq.(6) itself is constructed approximately by Monte Carlo sampling. Hence, [18] has no accuracy bound on either $\hat{D}$ or $\hat{S}^{(T)}$. In short, none of the above linearized methods guarantee the ϵ error of each SimRank score. Another linearized method is [44]. They present a "varied-D" SimRank model, which computes a series $\{D_0, \cdots, D_T\}$, and they show the estimated $S^{(T)} = D_T + cP^\top D_{T-1} P + \cdots + c^k P^\top D_0 P$ has an error bound c^{T+1}. However, it requires $O(Tmn)$ time to compute $\{D_0, \cdots, D_T\}$, which is costly for large-scale graphs. Our method directly estimates $\hat{D}$, and is more efficient and parallel scalable (shown later).

4 DISTRIBUTED $\hat{D}$ ESTIMATION

In this section, we first review the physical meaning of D, then present our distributed method for estimating D with an ϵ_d error bound, and finally analyse its cost.

4.1 Physical Meaning of D

We rely on the physical meaning of D and the $\sqrt{c}$-walk interpretation of SimRank, which are firstly introduced in [29].

Definition 4.1. *Given a node $u \in G$, an $\sqrt{c}$-walk is a reverse random walk from u and stops with $1 - \sqrt{c}$ probability at each step.*

According to [29], $s(a, b)$ is equal to the probability of two $\sqrt{c}$-walks from u and v meeting. The following lemma reveals the physical meaning of D.

Lemma 2. *Let d_v be the probability that two $\sqrt{c}$-walks from v that do not meet after the 0-th step, then $d_v = D_{v,v}$.*

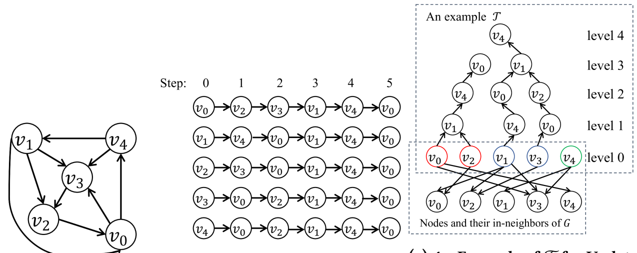
Based on that, [29] presents the following equation for d_v,

$$d_v = 1 - \frac{c}{|I_v|} - c \overbrace{\frac{1}{|I_v|^2} \sum_{\substack{a,b \in I_v \\ a \neq b}} s(a, b)}^{\mu}. \tag{7}$$

Now, consider the following random process for v: (1) Sample a pair of nodes a and b from I_v; (2) If $a \neq b$, perform two $\sqrt{c}$-walks from a and b. Let X be a random variable which indicates whether these two $\sqrt{c}$-walks meet or not, it can be verified that $\mathbb{E}[X] = \mu$. Suppose N samples are generated from v, then d_v is estimated as:

$$\hat{d_v} = 1 - \frac{c}{|I_v|} - c\hat{\mu}, \tag{8}$$

where $\hat{\mu} = \frac{1}{N} \sum_{i=1}^{N} X_i$. Therefore, making $|d_v - \hat{d}_v| < \epsilon_d$ is equal to making $|\mu - \hat{\mu}| < \frac{\epsilon_d}{c}$.


v_1
v_4
v_3
v_2
v_0
Step: 0 1 2 3 4 5
v_0 v_2 v_3 v_1 v_4 v_0
v_1 v_4 v_0 v_1 v_4 v_0
v_2 v_3 v_0 v_1 v_4 v_0
v_3 v_0 v_2 v_1 v_4 v_0
v_4 v_0 v_2 v_1 v_4 v_0
An example $\mathcal{T}$
level 4
level 3
level 2
level 1
level 0
v_4
v_0 v_1
v_4 v_0 v_2
v_1 v_4 v_0
v_0 v_2 v_1 v_3 v_4
v_0 v_2 v_1 v_3 v_4
Nodes and their in-neighbors of G

(a) A Toy Graph G

(b) A Bunch of Random Walks Lead to Congestion

(c) An Example of $\mathcal{T}$ for Updating $\hat{d}_v$ for $\forall v \in V$

Figure 1: Examples of G, Random Walks and $\mathcal{T}$

Lemma 3 (Hoeffding Inequality [10]). *Let $A_1, \cdots, A_N$ be independent random variables where each A_i is strictly bounded by the interval $[a_i, b_i]$, let $\bar{A} = \frac{1}{N}(A_1 + A_2 \cdots + A_N)$, then for any $\epsilon > 0$, $Pr\{|\bar{A} - \mathbb{E}[\bar{A}]| \geq \epsilon\} \leq 2e^{\frac{-2N^2\epsilon^2}{\sum_{i=1}^{N}(b_i - a_i)^2}}$.*

According to Lemma 3, to achieve the ϵ_d bound, we can set $N = \frac{c^2}{2\epsilon_d^2} \ln \frac{2}{\delta_d}$. Even though above sampling process can be used to estimate D [29], it is still challenging for distributed computation *w.r.t.* parallel scalability, as shown in the next section.

4.2 The Design of Distributed Indexing

In this section, we first discuss the difficulty of computing $\hat{D}$ on a shared-nothing model, and then present our solution in detail.

It seems easy to parallelize the computation of $\hat{D}$, since each $\hat{d}_v$ can be estimated simultaneously, and the N pair of samples generated for a specific v are independent. This leads to two parallel solutions. The first one contains N rounds: at each round, each node $v \in V$ samples a pair of $\sqrt{c}$-walks at the same time, and updates $\hat{d}_v$ based on the sampling result. However, this naive method would lead to *congestion*, which undermines the parallel scalability. Particularly, since random walks from different sources are running simultaneously, it is possible that certain nodes are visited by many walks at the same time. These nodes then become the hubs of the walks, and the hubs have to communicate with their neighbors at a high communication cost. The same issue of the congestion in conducting random walks from different sources simultaneously, is also reported in PageRank computation [22].

Example. Consider a toy graph in Figure 1a, suppose each node samples a $\sqrt{c}$-walk simultaneously, Figure 1b shows the first 5 steps of those walks. At step 3, all 5 walks pass through v_1, giving v_4 a high communication cost (5 messages) at step 4, though v_4 has only 2 out-neighbors and 1 in-neighbor. Note that we cannot simply merge random walks when they are visiting the same node, since each walk contains the information of the source node, for updating the meeting results later. Here we only show one walk for each node. In reality, each node needs to generate 2 $\sqrt{c}$-walks at each round, which would lead to more congestion.

Another parallel strategy consists of $|V|$ rounds: at each round, a specified node $v \in V$ generate N pairs of $\sqrt{c}$-walks simultaneously, and $\hat{d}_v$ is computed at this round. However, in this case, the source node v itself becomes a hub, since all walks start from v and pass through I_v. The failure of the above two attempts lies in that those random walks are *memorable*, *i.e.,* each random walk has to carry the information of the source node ID (the first method), or the sample ID (the second method), during walking over G.

Algorithm Overview. We present a parallel scalable tree-based method, which contains N rounds. In each round, a forest $\mathcal{T}$ is built across $\mathcal{W}$ from leaves to roots, level by level , and then we color the leaves such that two leaves are of the same color iff they are in the same tree. Each node $v \in V$ can query the leaves in $\mathcal{T}$ simultaneously to get a sample result, and update the corresponding $\hat{d}_v$. That is, $\mathcal{T}$ is shared by all nodes in V in the round, and is destroyed at the end of this round. In addition, the building, coloring, and querying phases are all parallel scalable. We build $\mathcal{T}$ as follows.

The set of nodes in each level $\mathcal{T}$ are a subset of V. Denote by H_l the set of nodes at level l. Initially, $H_0 = V$, H_l is generated from H_{l-1} as follows: each node $v \in H_{l-1}$: (1) Samples an in-neighbor a with probability $\frac{\sqrt{c}}{|I_v|}$, and we set v's parent to a; (2) Does nothing with the other $(1 - \sqrt{c})$ probability. The forest keeps growing until $H_l = \emptyset$ for some $l > 0$. Since any node v may appear on different levels, we use $v^{(l)}$ to denote v's copy on level $l \in [0, +\infty)$, and the sampling process determines whether $v^{(l)}$ may belong to H_l or not. A key property of $\mathcal{T}$ is that, the path from $v^{(0)}$ to its root can be viewed as a sampled $\sqrt{c}$-walk from v, and any two paths are sampled independently before they merge. Therefore, if we want to check whether two $\sqrt{c}$-walks from a and b meet, we only need to check whether $a^{(0)}$ and $b^{(0)}$ are in the same tree of $\mathcal{T}$. After coloring the leaves with their roots, to update $\hat{d}_v$ according to Eq.(8), we check the colors of two randomly selected nodes in I_v.

Example. Figure 1c shows a sampled $\mathcal{T}$ from the toy graph of Figure 1a, Initially, $H_0 = \{v_1, v_2, v_3, v_4, v_5\}$, then each node samples an in-neighbor randomly with $\sqrt{c}$ probability: v_0, v_2 selects v_1, v_1 selects v_4, v_3 selects v_0, and v_4 does not sample (with $1 - \sqrt{c}$ probability), so $H_1 = \{v_1, v_4, v_0\}$. This process continues until $H_5 = \emptyset$, when v_4 in level 4 chooses to stop sampling. We then label each leaf with its root as its color, by top-down propagation, then v_0, v_2 have the same color (with root $v_0^{(3)}$), v_1, v_3 has color $v_4^{(4)}$, and v_4 has color $v_4^{(0)}$. Leaves of the same color meet when a $\sqrt{c}$-walk is issued from each of them. To get a sample result for v_3, it first samples two of its in-neighbors, *e.g.,* v_1 and v_0, since they are of different colors, $\hat{d}_{v_3}$ is updated correspondingly.

Algorithm Detail. Algorithm 1 has N rounds, in each round, a status variable $v.cnt$ (initialized in line 4), which counts the number of meeting times for v, is updated for each node v (line 39). Finally, each $\hat{d}_v$ is calculated based on Eq.(8) (line 42). For the storage, each node v maintains its multiple occurrences $v^{(l)}$ for a different l by a hash map, and checking whether $v^{(l)}$ appears in H_l can be done in $O(1)$ time. Besides that, $v^{(l)}$ has several fields:

- $v^{(l)}$*.parent* records the parent of $v^{(k)}$, it is *nil* if $v^{(l)}$ is root.
- $v^{(l)}$*.children* is a set storing the children of $v^{(l)}$ in $\mathcal{T}$;
- $v^{(l)}$*.color* stores the root of the tree where $v^{(l)}$ resides.

Each round has three phases: (1) Build a forest $\mathcal{T}$ from leaves to roots (lines 7-24); (2) Label roots for leaves (lines 26-33); (3) Update $v.cnt$ for each node v (lines 36-39). While building $\mathcal{T}$, H_0 contains all nodes in V (line 9), then the forest grows in a bottom-up manner: each node in H_l samples an in-neighbor a with $\sqrt{c}$ probability (lines 17-20), indicating node a appears the $(l+1)$-level of $\mathcal{T}_i$ (line 22). We then set the corresponding parent-child relationship between $v^{(l)}$ and $a^{(l+1)}$ (line 21-23). The building process is continued until no new node is generated for $(l + 1)$-level of $\mathcal{T}_i$ (line 11). Next, we turn to the root finding phase. A top-down approach is used to label roots. Starting from the roots of each tree (line 30-31), we iteratively pass the labels $v.color$ to v's children level by level (line 33), until the leaves are labeled (line 26). We then query the sample results for each $v \in V$ using colors of leaves in $\mathcal{T}$, and update $v.cnt$ if the colors of the sampled neighbors are the same (line 39).

Algorithm 1 Parallel Scalable Computation of $\hat{D}$

```
Input: $\mathcal{F} = \{F_1, \cdots, F_p\}, c, \epsilon_d, \delta_d$
Output: $\hat{d}_v$ for $\forall v \in V$
 1: $N \leftarrow \frac{c^2}{2\epsilon_d^2} \ln \frac{2}{\delta_d}$
 2: for all $F \in \mathcal{F}$ in parallel do
 3:     for all $v \in F$ do
 4:         $v.cnt \leftarrow 0$
 5: for all $i = 1, 2, \cdots, N$ do
 6:     /* Phase 1: build a forest $\mathcal{T}_i$ in parallel */
 7:     for all $F \in \mathcal{F}$ in parallel do
 8:         for all $v \in F$ do
 9:             add $v^{(0)}$ to $H_0$
10:     $l \leftarrow 0$                                        ▷ the level counter
11:     while $H_l \neq \emptyset$ do
12:         for all $F \in \mathcal{F}$ in parallel do
13:             for all $v \in F$ do
14:                 $v^{(l)}.parent \leftarrow nil$
15:                 $v^{(l+1)}.children \leftarrow \emptyset$
16:         for all $F \in \mathcal{F}$ in parallel do
17:             for all $v \in F$ and $v^{(l)} \in H_l$ do
18:                 $\theta \leftarrow$ a random number in $[0, 1]$
19:                 if $\theta < \sqrt{c}$ then
20:                     $a \leftarrow$ a randomly selected node from $I_v$
21:                     $v^{(l)}.parent \leftarrow a^{(l+1)}$
22:                     add $a^{(l+1)}$ to $H_{l+1}$
23:                     $a^{(l+1)}.children.add(v^{(l)})$
24:         $l \leftarrow l + 1$
25:     /* Phase 2: label leaves with roots in parallel */
26:     while $l > 1$ do
27:         $l \leftarrow l - 1$
28:         for all $F \in \mathcal{F}$ in parallel do
29:             for all $v \in F$ do
30:                 if $v^{(l)} \in H_l$ and $v^{(l)}.color$ is unset then
31:                     $v^{(l)}.color \leftarrow v^{(l)}$
32:                 for all $x \in v^{(l)}.children$ do
33:                     $x.color \leftarrow v^{(l)}.color$
34:     /* Phase 3: update $\hat{d}_v$ in parallel */
35:     for all $F \in \mathcal{F}$ in parallel do
36:         for all $v \in F$ do
37:             sample a pair of nodes $(a, b)$ from $I_v$
38:             if $a \neq b$ and $a^{(0)}.color = b^{(0)}.color$ then
39:                 $v.cnt \leftarrow v.cnt + 1$
40: for all $F \in \mathcal{F}$ in parallel do
41:     for all $v \in F$ do
42:         $\hat{d}_v \leftarrow 1 - \frac{c}{|I_v|} - c \cdot \frac{v.cnt}{N}$
```

4.3 Cost Analysis

Number of Supersteps. When building $\mathcal{T}$, since only $\sqrt{c}$ portion of H_l generate the nodes for H_{l+1} on average, the expected height of each $\mathcal{T}$ is $O(\log n)$. Therefore, both the building phase and the labeling phase take $O(\log n)$ supersteps. Besides, updating $\hat{d}_v$ takes 2 supersteps: one superstep is where each node samples two in-neighbors and requests their colors; the other is the sampled in-neighbors response to the request. Hence, in one round, it takes $O(\log n)$ supersteps. The total number of supersteps is $O(N \log n)$.

Computation Cost. We analyse the maximum local computation cost of any fragment in any superstep. We first analyse the local cost in the manner of "think like a vertex". Each node v has $O(1)$ computation cost for lines 4, 9, 14-15, 17-21, 31, 33, 37, 39 and 39. Besides that, v has at most $O(|O_v|)$ children to be added at any superstep of building $\mathcal{T}$ (line 23), and has $O(|O_v|)$ children to propagate the color information at any superstep of labeling colors (line 32), and has $O(|O_v|)$ responses to the the requests of the color information (line 38). Therefore, at any superstep, a node v has at most $O(|O_v|)$ local computation cost, and the local cost for each fragment F_i is $O(\sum_{v \in V_i} |O_v|) = O(|E_i|) = O(\frac{|E|}{p})$.

Communication Cost. We analyse the maximum local computation cost of any fragment in any superstep. Now consider the local communication for each node first. Communication only happens on the instructions labeled in red in Algorithm 1. In line 22, if v and a are not in the same fragment, v will send $O(1)$ message a, and a receives at most $O(|O_a|)$ messages to update the children (line 23). Similarly, there are at most $O(|O_v|)$ messages sent from $v^{(l)}$ to its children (line 32-33), and there are $O(1)$ messages received by each child. In line 38, each node v sends at most 2 messages to request the colors of a and b, and each a receives at most $O(|O_a|)$ messages. Therefore, any node $v \in F_i$ can send and receive at most $O(|O_v|)$ messages at any superstep, and the communication cost for each fragment is: $O(\sum_{v \in F_i} |O_v|) = O(|E_i|) = O(\frac{|E|}{p})$.

Space Cost. At each round, each node v needs to maintain its multiple copies $(v^{(0)}, \cdots, v^{(l)}, \cdots)$ at different levels, which is $O(\log n)$, and for each $v^{(l)}$, it has at most $O(|O_v|)$ children and $O(1)$ parent, so the space cost for v is $O(\log n |O_v|)$. The space cost of F_i is $O(\sum_{v \in F_i} \log n |O_v|) = O(\log n |E_i|) = O(\frac{|E| \log n}{p})$.

Using trees for indexing random walks are a common technique in SimRank computation. [6] first introduces fingerprint trees, by encoding the first meeting time, which can be retrieved by computing the depths of two nodes. However, the fingerprint trees can only compress truncated random walks with no stop probabilities, which affects the precision. [25] presents the one-way graph structure, which compresses the random walks for all nodes, but brings in dependency among samples and thus influences the accuracy. [14] designs SA forests, whose structure is similar to $\mathcal{T}$, but has a truncated height, leading to a loss in accuracy [34]. Our method differs from the above in the following aspects. First, while the above tree-based methods are used to estimate SimRank scores directly, we use $\mathcal{T}$ for updating all $\hat{d}_v$s. A forest in [6, 14, 25] is used once as one sample of a single SimRank query, while one $\mathcal{T}$ in Algorithm 1 is used as one sample for updating the whole $\hat{D}$ once. Second, all previous methods need to store the sample forests for later queries, which leads to large space cost ($O(N|V|)$ in [6, 25] and $O(tN|V|)$ in [14]) when G is big or high precision is required (large sample size). On the contrary, we drop the sampled forest at the end of each round, and our off-line index $\hat{D}$ only costs $O(\frac{|V|}{p})$ for each fragment. Third, our solution is based on the shared-nothing model, which assumes G and $\mathcal{T}$ are partitioned over workers and challenges the parallel scalability *w.r.t.* computation/communication, while previous works are based on the shared-memory model.

5 ONLINE SINGLE-SOURCE QUERY

In this section, we first present our distributed single-source solution in detail, and then analyse its cost. After obtaining $\hat{D}$, the single-sources scores *w.r.t.* u can be retrieved easily, by computing u-th column of $\hat{S}^{(T)}$, *i.e.,*:

$$\hat{S}^{(T)}_{*,u} = \hat{S}^{(T)} \mathbf{e}_u = \sum_{t=0}^{T} c^t P^{\top t} \hat{D} P^t \mathbf{e}_u. \tag{9}$$

Algorithm 2 Parallel Scalable Single-source SimRank

```
Input: $\mathcal{F} = \{F_1, \cdots, F_p\}, c, T, \hat{D}, u$
Output: $\hat{s}(v, u)$ for $\forall v \in V$
 1: $u.q[0] \leftarrow 1$
 2: /* Phase 1: compute $\{\mathbf{q}_u^{(1)}, \cdots, \mathbf{q}_u^{(T)}\}$ */
 3: for all $t = 1, \cdots T$ do
 4:     for all $F \in \mathcal{F}$ in parallel do
 5:         for all $v \in F$ and $v.q[t-1] \neq 0$ do
 6:             for all $w \in I_v$ do
 7:                 $w.q[t] \leftarrow w.q[t] + \frac{v.q[t-1]}{|I_v|}$
 8: /* Phase 2: compute $\{\mathbf{g}_u^{(1)}, \cdots, \mathbf{g}_u^{(T)}\}$ */
 9: for all $t = 1, \cdots T$ do
10:     for all $F \in \mathcal{F}$ in parallel do
11:         for all $v \in F$ and do
12:             $v.g[t] \leftarrow \hat{d}_v \cdot v.q[t]$
13:         if $t = T$ then
14:             $v.sim \leftarrow v.g[t]$        ▷ Initialize estimated SimRank score
15: /* Phase 3: compute $\hat{s}(*, u)$ */
16: for all $t = T, \cdots 1$ do
17:     for all $F \in \mathcal{F}$ in parallel do
18:         for all $v \in F$ and $v.g[t] \neq 0$ do
19:             for all $w \in O_v$ do
20:                 $w.sim \leftarrow w.g[t-1] + c \cdot \frac{v.sim}{|I_w|}$
21: $u.sim \leftarrow 1$
```

Computing $\hat{S}^{(T)}_{*,u}$ directly involves $O(T^2)$ matrix-vector multiplications. Therefore, we use the following alternative, let vector $\mathbf{q}_u^{(t)} = P^t \mathbf{e}_u$ (the distribution of t-th step reverse random walk from u), and vector $\mathbf{g}_u^{(t)} = \hat{D} \mathbf{q}_u^{(t)}$, then:

$$\text{Eq.(9)} = \mathbf{g}_u^{(0)} + cP^{\top}(\mathbf{g}_u^{(1)} + \cdots cP^{\top}(\mathbf{g}_u^{(T-1)} + cP^{\top}\mathbf{g}_u^{(T)}) \cdots), \tag{10}$$

which has only $O(T)$ matrix-vector multiplications. The trade-off is that we need to maintain $\{\mathbf{g}_u^{(0)}, \cdots, \mathbf{g}_u^{(T)}\}$, which costs extra space. However, in a distributed environment, it is always practical to reduce the number of rounds, which can reduce the overheads of synchronization. Besides, each $\mathbf{g}_u^{(t)}$ is also distributed and maintained locally by each node, which can make use of the sparsity of the vector when T is small.

Algorithm Detail. Our solution is in Algorithm 2. The idea is to use the graph topology for simulating sparse matrix-vector multiplication. Each node $v \in V$ maintains the following data locally:

- $v.q$ is a hash table: $v.q[t]$ is the v-th coordinate of $\mathbf{q}_u^{(t)}$;
- $v.g$ is a hash table: $v.g[t]$ is the v-th coordinate of $\mathbf{g}_u^{(t)}$;
- $v.sim$ is the estimated SimRank score $\hat{s}(v, u)$.

The algorithm first computes the series of vectors $\{\mathbf{q}_u^{(1)}, \cdots, \mathbf{q}_u^{(T)}\}$ iteratively (line 1-7). It then computes $\{\mathbf{g}_u^{(1)}, \cdots, \mathbf{g}_u^{(T)}\}$ by multiplying each $P^T \mathbf{e}_u$ by $\hat{D}$ (line 9- 14). Finally, it updates $\hat{S}^{(T)}_{*,u}$ iteratively by Eq.(10) (line 16-20).

Cost Analysis. Each phase in Algorithm 2 has $O(T)$ rounds, and each round has at most $O(1)$ supersteps. Then the total number of supersteps is $O(T)$. At any step, each node v only accesses its neighbors or does local computation, so the computation cost for v is at most $O(|I_v| + |O_v|)$. Therefore, the total cost for each fragment F_i at any superstep is $O(T \sum_{v \in F_i} |I_v| + |O_v|) = O(\frac{|E|}{p})$. Communication only occurs when v and w are not in the same fragment (line 7, 20).

Since each node v only communicates with its in-neighbors/out-neighbors, its communication cost is either $O(|I_v|)$ or $O(|O_v|)$. The communication cost for F_i at any superstep is $O(\frac{|E|}{p})$. Since each node maintains $O(T)$ fields, the space cost for each fragment is $O(\sum_{v \in F_i} T) = O(T|V_i|) = O(\frac{T|V|}{p})$.

6 OPTIMIZING OFFLINE INDEXING

There is redundant computation in the offline indexing. Firstly, the leaf coloring phase (phase 2) in each round of Algorithm 1 actually colors all nodes in the sampled forest, which is unnecessary. Second, currently $\hat{d}_v$ is estimated according to Eq.(7), the information in term μ is not fully utilized, since once we know $a \neq b$, we can further expand it using Eq.(1), and thus redundant samples are dropped. We present two techniques, the first one boosts leaf coloring, the second one is sample reduction.

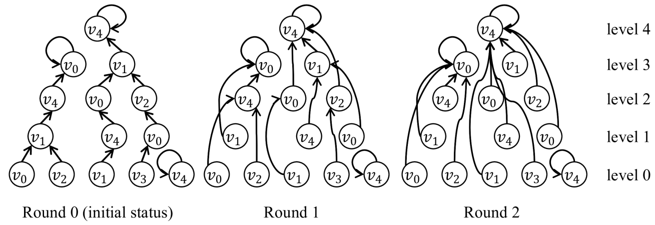
6.1 Boosting Leaf Coloring

In this section, we introduce techniques for optimizing Phase 2 of Algorithm 1, *i.e.,* labeling roots for leaves, which applies a top-down propagation manner and requires $O(\log n)$ rounds. Here we adopt a bottom-up strategy, which uses $O(\log\log n)$ rounds. The strategy is based on the following observations: (1) We only need to color the leaves, and colors of nodes on other levels are not necessary, but the top-down approach (phase 2) in Algorithm 1 labels all nodes, which leads to redundant computation; (2) After building $\mathcal{T}$, each node knows its level information, which is not utilized in top-down propagation. Our idea is to adopt the *pointer jumping* technique [13] for root finding, which is a basic tool for parallel algorithms over shared-memory models, to the shared-nothing environment. Given a tree, initially, each node v has a pointer $p(v)$ to its parent, and the root points to itself. The idea of pointer jumping is: at each iteration, each node v updates $p(v)$ to $p(p(v))$. The process terminates when $p(v) = p(p(v))$ for all nodes. Finally, all nodes point to the root.

Example. Figure 2 shows an example of root finding by pointer jumping, on the sampled forest in Figure 1c. At step 0, the three roots $v_4^{(0)}$, $v_0^{(3)}$ and $v_4^{(4)}$ point to themselves, while other nodes point to their parents. At step 1, each node points to its grandparent, *e.g.,* $v_1^{(0)}$ points to $v_0^{(2)}$ and $v_2^{(2)}$ points to $v_4^{(4)}$. At round 2, all nodes in $\mathcal{T}$ point to their corresponding roots. While the top-down propagation needs 4 steps to finish, the pointer jumping method only has 2 steps.

However, directly applying the pointer jumping technique still has computation and scalability issues. First, we only need to label leaves, while throughout pointer jumping, every node is updated, which leads to redundant computation and communication. Considering the example in Figure 2, the updates for $v_1^{(1)}, v_0^{(1)}, v_4^{(1)}$ are not necessary, since their updates make no contribution towards leaf color. Second, the pointer jumping may violate the scalability. In previous analysis, the communication scalability is guaranteed by the fact that *each node only communicates with its neighbors at each round*. However, this does not hold for pointer jumping, since the structures of trees evolve over time. Consider the case where all descendants of node v ask v for $p(v)$ simultaneously, the fragment which holds v would receive a large volume of messages.

Algorithm Detail. In Algorithm 3. initially, we set the color of each node v in $\mathcal{T}$ to its parent if the $v.parent$ exists, otherwise, it is set to v itself (line 3-10). Next, at each iteration r, we only update nodes at level $k \cdot 2^r$ for some $k \geq 0$ (line 15-18), instead of updating all nodes as pointer jumping. The reason is that we are only concerned about the leaf color, and want to make sure the colors of nodes in levels which are multiples of 2^r are up-to-date.



Round 0 (initial status)

Round 1

Round 2

Figure 2: An Example of Pointer Jumping for Root Finding

Cost Analysis. The initialization phase of Algorithm 3 takes 1 superstep, and the expected height of $\mathcal{T}$ is $O(\log(n))$, thus the total number of supersteps is $O(\log\log(n))$. Communication only happens in line 18 of the labeling phase. where node $v^{(l)}$ asks for the parent of its current parent $x = v^{(l)}.color$, and x returns $x.color$ to $v^{(l)}$. At each superstep, each node has made at most $O(1)$ requests to its parent. At the same time, each node may receive multiple requests. However, it can be verified that the number of requests a node receives is at most the size of its leaves, which can be bounded by the following results.

Theorem 4. *The expected size of leaves for any tree in $\mathcal{T}$ is bounded by $\|S\|_1$.*

Proof. Given any specific node $u \in V$, then for any $v \in V$, let Z_v be an indicator random variable, indicating whether $v^{(0)}.color = u^{(0)}.color$. Then event $Z_v = 1$ is when u and v are in the same tree, which indicates two $\sqrt{c}$ walks from u and v meet. Hence, we can conclude $\mathbb{E}[Z_v] = s(u,v)$, and the random variable $\sum_{v \in V} Z_v$ indicates how many leaf nodes share the same color with u. By the linearity of expectation, we get: $\mathbb{E}[\sum_{v \in V} Z_v] = \sum_{v \in V} \mathbb{E}[Z_v] = \sum_{v \in V} s(u,v) \leq \|S\|_1$. □

In practice, $\|S\|_1$ is very small, since a node is similar to only a few nodes, and most node pairs are dissimilar. In other words, $\mathcal{T}$ consists of numbers of small trees, instead of large giant trees. Similar findings are also reported in [25], which shows that sampled *one-way* graphs are highly disconnected, indicating most nodes are dissimilar. Due to each node only needing to maintain its parent information, and not needing to store the information of the children, then the space cost of each worker is reduced to $O(\frac{|V|\log n}{p})$.

6.2 Sample Reduction

In this section, we present a method to reduce the number of sampled forests in Section 4, *i.e., N*. We re-formulate Eq.(7) to the following equation for d_v:

$$d_v = 1 - \frac{c}{|I_v|} - \frac{c}{|I_v|^2} \sum_{\substack{a,b \in I_v \\ a \neq b}} s(a,b) = 1 - \frac{c}{|I_v|} -$$

$$c^2(1 - \frac{1}{|I_v|}) \underbrace{\frac{1}{|I_v|^2 - |I_v|} \sum_{\substack{a,b \in I_v \\ a \neq b}} \frac{1}{|I_a||I_b|} \sum_{x \in I_a, y \in I_b} s(x,y)}_{\eta}. \quad (11)$$

Algorithm 3 Bottom-up Leaf Coloring

```
Input: $\mathcal{F} = \{F_1, \cdots, F_p\}$, a sampled forest $\mathcal{T}$
Output: $v^{(0)}.color$ for $\forall v \in V$
L ← the height of $\mathcal{T}$
/* Initialization */
for all $F \in \mathcal{F}$ in parallel do
    for all $v \in F$ do
        for all $L \in [0, L]$ do
            if $v^{(l)} \in H_l$ then
                if $v^{(l)}.parent \neq nil$ then
                    $v^{(l)}.color \leftarrow v^{(l)}$
                else
                    $v^{(l)}.color \leftarrow v^{(l)}.parent$
/* Bottom-up Labeling */
for all $r = 1, 2, \cdots, \lceil \log_2 L \rceil$ do
    for all $F \in \mathcal{F}$ in parallel do
        for all $v \in F$ do
            for all $l \in [0, L]$ and $l \bmod 2^r = 0$ do
                if $v^{(l)} \in H_l$ and then
                    $x \leftarrow v^{(l)}.color$
                    $v^{(l)}.color \leftarrow x.color$
```

Now, consider the following random process for v: (1) Randomly select a node a from I_v; (2) Randomly select a node b from $I_v \setminus \{a\}$; (3) Randomly select two nodes x and y from I_a and I_b; (4) Sample two $\sqrt{c}$-walks from x and y, respectively. Let Y be a random variable indicating whether the two $\sqrt{c}$-walks generated above meet or not, then it can be easily verified that $\mathbb{E}[Y] = \eta$. Therefore, we can generate N samples from v, and d_v can be estimated as:

$$\hat{d}_v = 1 - \frac{c}{|I_v|} - c^2(1 - \frac{1}{|I_v|})\hat{\eta}, \tag{12}$$

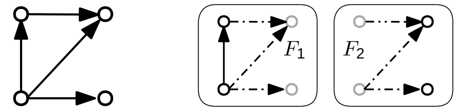
where $\hat{\eta} = \sum_{i=1}^{N} Y_i$. According to Eq.(12), to make $|\hat{d}_v - d_v| < \epsilon_d$, it is sufficient to make $|\hat{\eta} - \eta| < \frac{\epsilon_d}{c^2(1-\frac{1}{|I_v|})}$. Based on Lemma 3, given the error bound ϵ_d, it is sufficient to set the sample size $N = \frac{c^4(1-\frac{1}{\max_{v \in V}|I_v|})^2}{2\epsilon_d^2} \ln \frac{2}{\delta_d}$, due to that $Y_i \in [0, 1]$. Therefore, the sample size is smaller than that of Section 4. The probabilistic interpretation of Eq.(11) is different from that of Eq.(7). Consider step 3 of the above random process, there is no longer a $1 - \sqrt{c}$ stop probability for selecting x and y, thus there is a minor modification in building each $\mathcal{T}$: when generating H_{l+1} from H_l, there is no stop probability when $l = 0$, otherwise, there is a $1 - \sqrt{c}$ (line 19 in Algorithm 1). This modification does not affect the cost, since the expected height of $\mathcal{T}$ is still $O(\log n)$.

7 OPTIMIZING ONLINE QUERY

There is inefficiency in online queries. Firstly, the number of iterations T in Algorithm 2 is determined by Theorem 1, which uses $\forall t \in [1, T]$, $\|\mathbf{q}_u^{(t)}\|_\infty \leq 1$ to bound the results, which is loose. Second, Algorithm 2 has both forward and backward iterations, thus requires both in-edges and out-edges balanced in each fragment to achieve the parallel scalability. However, currently there is no edge-cut partitioner designed for this. In this section, we introduce two optimization techniques. One is *early stop*, which reduces the number of iterations by utilizing computed information at running time. The other is a *partition refiner*, which aims to further balance the workload among workers after the graph is partitioned by some existing partitioners.

7.1 Early Stop

In our distributed query method (Algorithm 2), the number of rounds T is predefined and determined by Theorem 1, which shows a general accuracy bound *w.r.t.* T and ϵ_d. However, during the query phase, as the transition probability vectors, *i.e.*, $\mathbf{q}_u^{(t)}$ $(t \geq 1)$, are generated, we can make the accuracy bound more specific, which can further reduce T in practice. We show the following accuracy bound *w.r.t.* $\{\mathbf{q}_u^{(1)}, \cdots, \mathbf{q}_u^{(T)}\}$.



(a) A toy graph G **(b)** A skewed partition of G

Figure 3: Examples of G, and Skewed Partition

Theorem 5. *Given ϵ_d and $\{\mathbf{q}_u^{(0)}, \cdots, \mathbf{q}_u^{(T)}\}$ for source u, for any node $v \in V$ that $v \neq u$, $|\hat{S}_{v,u}^{(T)} - S_{v,u}| < \epsilon_d \sum_{t=1}^{T} c^t \|\mathbf{q}_u^{(t)}\|_\infty + c^{T+1}$.* □

Since $\|\mathbf{q}_u^{(t)}\|_\infty \leq 1$ for any $t \geq 1$, Theorem 5 is tighter than Theorem 1. Our early stop strategy dynamic decides T during computing $\{\mathbf{q}_u^{(1)}, \cdots\}$ in phase 1 of Algorithm 2. Specifically, the coordinator W_0 keeps an accumulated variable *accu*, holding the value of $(c\|\mathbf{q}_u^{(1)}\|_\infty + \cdots + c^t\|\mathbf{q}_u^{(t)}\|_\infty)$. At the end (line 7) of each round t , each fragment F_i declares a local variable $b_i = \max_{v \in F_i} v.q[t]$, and sends it to W_0. After receiving all b_i for $\forall i \in [1, p]$, W_0 selects $\max_{i \in [1,p]} b_i$ and updates *accu* accordingly. If $\epsilon_d \cdot accu + c^{t+1} \geq \epsilon$, W_0 notifies workers to move to the next round; otherwise, W_0 sets $T = t$ and Algorithm 2 turns directly to phase 2. The early stop but reduces the number of rounds T in practice.

7.2 Graph Partition Refinement

For online queries, to achieve efficiency and parallel scalability, we require each fragment F_i to be approximately equal-sized, and share an equal amount of computation cost in each superstep. Particularly, in Algorithm 2, iterative computation is firstly performed forward (phase 1), and then backwards (phase 3). The cost on a fragment in a superstep for the forward iteration (resp. backward iteration) is $O(\Sigma_{v \in F_i}(|I_v|))$ (resp. $O(\Sigma_{v \in F_i}(|O_v|))$). Therefore, it is essential to keep *both* in-edges and out-edges balanced across different workers. Unfortunately, most existing edge-cut partitioners only ensure balanced vertices for each fragment, and can be heavily-skewed *w.r.t.* $\Sigma_{v \in F_i}(|I_v|)$ or $\Sigma_{v \in F_i}(|O_v|)$. Skewness is particularly evident in graphs with skewed degree distribution, *e.g.*, power-law graphs [1, 4]. To our knowledge, no previous work balances the load of $\Sigma_{v \in F_i}(|I_v|)$ and $\Sigma_{v \in F_i}(|O_v|)$ at the same time, which makes the load balancing optimization for online query non-trivial.

Example. Consider the toy graph illustrated in Figure 3a. The graph is partitioned into two fragments F_1 and F_2, with cut edges shown in dashed lines and mirror vertices colored in gray, shown in Figure 3b. These two partitions are balanced *w.r.t.* to both vertex load and edge load, as there are 2 and 2 non-mirror vertices, and 4 and 3 edges, in F_1 and F_2, respectively. However, its workload for online queries is not balanced. For instance, during backward iteration, the workload of F_2 is proportional to $\Sigma_{v \in F_2}(|O_v|)$, and F_2 becomes idle as it possesses no outgoing edges.

Instead of designing a new partitioner from scratch, we propose a partition refiner, which refines a given edge-cut partition, and makes it more balanced *w.r.t.* both $\Sigma_{v \in F_i}(|I_v|)$ and $\Sigma_{v \in F_i}(|O_v|)$. As seen in Algorithm 4, the algorithm takes as input 1) p fragments,

Algorithm 4 Partition Refinement

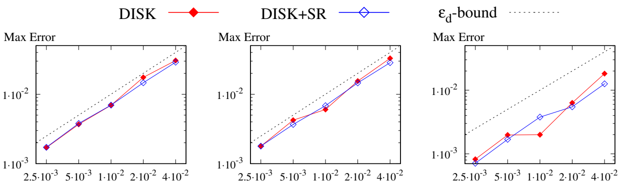
```
Input: $\mathcal{F} = \{F_1, \cdots, F_p\}$, balance factor $\epsilon_p$, threshold $\gamma$
Output: $v$.part for $\forall v \in V$
 1: for all $F_i \in \mathcal{F}$ in parallel do
 2:     for all $v \in F_i$ do
 3:         $v$.part $\leftarrow i$
 4:     $W_I[i] \leftarrow \Sigma_{v \in F_i}(|I_v|)$; $W_O[i] \leftarrow \Sigma_{v \in F_i}(|O_v|)$
 5: $W_{\max} = \frac{1+\epsilon_p}{p}|E|$; $W_{\min} = \frac{1-\epsilon_p}{p}|E|$
 6: for all $F_i \in \mathcal{F}$ in parallel do
 7:     swap$_i \leftarrow 0$
 8:     for all $v \in F_i$ do
 9:         if $W_I[v.\text{part}] > W_{\min} \wedge W_O[v.\text{part}] > W_{\min}$ then
10:             continue
11:         cand $\leftarrow \{j \mid W_I[j] < W_{\max} \wedge W_O[j] < W_{\min}\}$
12:         $v$.part $\leftarrow \arg_{j \in \text{cand}} \max |\{u \mid \wedge u.\text{part} = j\} \cap (I_v \cup O_v)|$
13:         if $v$.part is updated then
14:             update $W_I[\cdot]$, $W_O[\cdot]$ locally; swap$_i$++
15: synchronize $W_I[\cdot]$, $W_O[\cdot]$ globally
16: repeat lines 6, 15
17: until $(W_I[\cdot] < W_{\max} \wedge W_O[\cdot] < W_{\max})$ or $\Sigma$swap$_i < \gamma|V|$
```

partitioned from a graph using any existing vertex partitioning tools; 2) a user defined balance factor ϵ_p, and 3) a user defined threshold $\gamma \in (0, 1)$. The algorithm iteratively reduces the sizes of overloaded fragments until near convergence. The algorithm first initializes the partition label for each vertex, denoted as v.part (lines 1-4). After initialization, the algorithm iteratively swaps vertices out of overloaded partitions (lines 6-15). It terminates when either the balance goal is achieved, or part of less than $\gamma|V|$ vertices are updated in the previous iteration (line 17).

The algorithm declares two load thresholds $W_{\min}$ and $W_{\max}$ according to the input ϵ_p (line 5). Partition-i is 1) overloaded, if both $W_I[i]$ and $W_O[i]$ are larger than $W_{\max}$; or 2) underloaded, if both $W_I[i]$ and $W_O[i]$ are smaller than $W_{\min}$. The algorithm picks a vertex from a partition that is not underloaded (lines 9-10), and swaps it to a partition in its neighbor with the most frequency, which is not overloaded (lines 11-12). At the end of each iteration, the loads $W_I[\cdot]$ and $W_O[\cdot]$ are globally synchronized (line 15). At each iteration, each fragment has the computation and communication cost of $O(\frac{|E|}{p})$. In practice, the number of iterations till termination is usually less than 20 according to our experimental study.

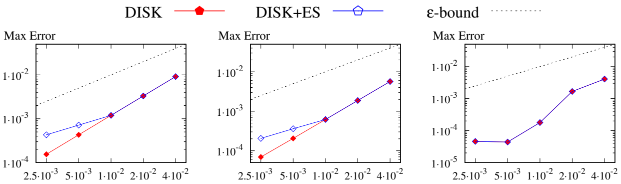
Table 3: Datasets

	Name	$\|V\|$	$\|E\|$	Type
Small	ca-GrQc(GQ)	5242	14 496	undirected
	ca-HepTh(HT)	9877	25 998	undirected
	p2p-Gnutella06(GT)	8717	31 525	directed
Large	com-LiveJournal(LJ)	3 997 962	34 681 189	undirected
	twitter-2010(TW)	41 652 230	1 468 365 182	directed
	com-friendster(FS)	65 608 366	1 806 067 135	undirected
	uk-2007-05(UK)	105 895 555	3 738 733 648	directed
	clueweb12(CW)	978 408 098	42 574 107 469	directed



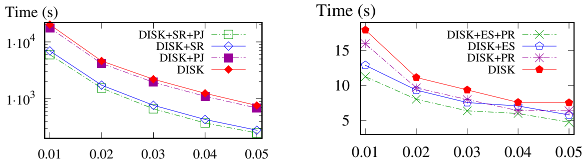
(a) Varying ϵ_d (GQ) **(b) Varying ϵ_d (HT)** **(c) Varying ϵ_d (GT)**

Figure 4: Offline Indexing: Impact of ϵ_d on Accuracy



(a) Varying ϵ (GQ) **(b) Varying ϵ (HT)** **(c) Varying ϵ (GT)**

Figure 5: Online Query: Impact of ϵ on Accuracy



(a) Varying ϵ (Offline) **(b) Varying ϵ_d (Online)**

Figure 6: Impact of Accuracy on Efficiency (TW)

8 EXPERIMENTS

In this section, we conduct experiments with the aim of answering the following research questions:

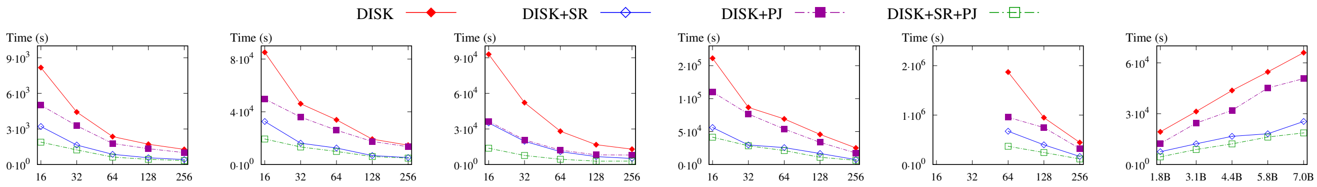
(1) Is the accuracy of our framework bounded?
(2) Do our optimization techniques work for boosting offline indexing and online queries?
(3) Is our solution parallel scalable, *i.e.,* taking less time when more computation resources are used?
(4) Does our solution scale well with large-scale graphs?
(5) Does our method outperform other solutions?

We use 8 real-life datasets, which are commonly used in the literature [14, 34]. The statistical information of these datasets is shown in Table 3. All of them can be downloaded from SNAP [1] or LAW [2]. We equip and compare the basic version of DISK (Sections 4 and 5) with different optimization techniques. For offline indexing, we have Sample Reduction (SR) in Section 6.2 and Pointer Jumping (PJ) in Section 6.1. For online queries, the optimizations are Early Stop (ES) in Section 7.1 and Partition Refiner (PR) in Section 7.2. For the ground truth of SimRank, we compute the exact SimRank matrix S, using the original algorithm of [11] with 100 iterations, which guarantees the error is less than 10^{-22}. Due to its expensive computational and space cost, the ground truth is only computed for the small datasets. Using the exact S, we also obtain the ground truth of D by the equation $D = S - cP^{\top}SP$. By default, we set $c = 0.6$ (as suggested in [20]), $\delta = 0.01$ and $p = 128$. For the partition optimization algorithm in Section 7.2, we refine an edge-cut partition generated by XtraPuLP[27], with $\epsilon_p = 0.1$ and $\gamma = 0.001$. The algorithms are implemented with C++11 and OpenMPI. All experiments are performed on a cluster of 64 linux machines, and each machine has two Intel(R) Xeon(R) CPU E5-2640v4 @2.40GHz processors (each with 10 cores) and 64GB RAM, running Ubuntu18.04. All machines are connected via a Gigabit network. The p processes are randomly scattered across the cluster.

8.1 Accuracy

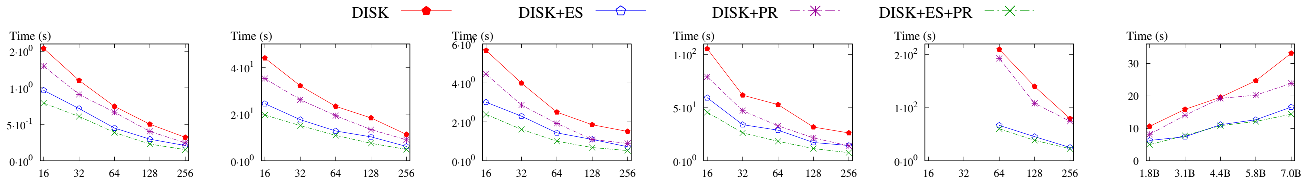
Offline Phase. We test the accuracy of the estimated diagonal correction matrix $\hat{D}$. We vary ϵ_d from 2.5×10^{-3} to 4×10^{-2}, with the scaling factor of 2. The accuracy is evaluated by the *maximum error*, *i.e., ME* $= \max_{v \in V} |\hat{d}_v - d_v|$. The results are shown in Figure 4a-4c.

[1] https://snap.stanford.edu/data/index.html
[2] http://law.di.unimi.it/datasets.php



(a) Varying p (LJ) **(b) Varying p (TW)** **(c) Varying p (UK)** **(d) Varying p (FS)** **(e) Varying p (CW)** **(f) Varying $|G|$**

Figure 7: Offline Indexing: Parallel Scalability and Scalability



(a) Varying p (LJ) **(b) Varying p (TW)** **(c) Varying p (UK)** **(d) Varying p (FS)** **(e) Varying p (CW)** **(f) Varying $|G|$**

Figure 8: Online Query: Parallel Scalability and Scalability

The accuracy of PJ is omitted since PJ boosts the leaf coloring without affecting the accuracy . To make the results more readable, we also draw an ϵ_d-bound line which is a dotted black line. We find that the MEs of both DISK and DISK +SR are at least 13% smaller than a given ϵ_d over all datasets. This verifies the accuracy of our distributed solutions for estimating $\hat{D}$, and is consistent with our analysis in Sections 4 and 6.2.

Online Queries. We test the accuracy of estimated SimRank scores. For each dataset, we randomly select 100 query nodes. We set $\epsilon_d = 1 \times 10^{-3}$, and vary ϵ from 2.5×10^{-3} to 4×10^{-2}, with a scaling factor of 2. For each method, we use the maximum error to evaluate its accuracy, *i.e.,* $ME = \max_{u\in Q, v\in V} |\hat{s}(u,v) - s(u,v)|$, where Q is the set of query nodes, $\hat{s}(u,v)$ and $s(u,v)$ are the approximate and the exact SimRank score, respectively. The results are shown in Figure 5a-5c. The results of PR are omitted since it does not affect the output of the algorithm. Similarly, we also draw an ϵ-bound line as a dotted black line. We find that for all datasets the MEs of both DISK and DISK +ES are at least 77% less than a given ϵ. This justifies the correctness of Theorems 1 and 5. We also observe that sometimes the MEs of DISK +ES are larger than basic DISK, *e.g.,* $\epsilon = 2.5 \times 10^{-2}$ on CG and HT. This is because ES trades off the precision for efficiency, while the final accuracy is still bounded.

8.2 Efficiency

Next, we test the impact of 1) ϵ_d and ϵ, 2) the number of processor cores p, 3) different optimization techniques and 4) the size of graph $|G|$, on the efficiency of DISK, for both offline indexing and online queries. For each setting, the average elapsed time of 100 randomly selected query nodes is reported for the online phase.

8.2.1 Impact of Accuracy on Efficiency. We test the efficiency of the offline phase under different precisions. Using the dataset TW, we vary ϵ_d from 1×10^{-2} to 5×10^{-2}, with the step of 1×10^{-2}. The indexing time of different methods is shown in Figure 6a. We find that the indexing time of all methods increases with a decreasing ϵ_d, as more samples are required. We can see both optimizations work. After being equipped with SR, the indexing time is significantly decreased by 2.8 times. This is consistent with our analysis in Section 6.2, *i.e.,* the number of samples is around a c^2 portion of the naive method. Applying PJ also saves the indexing time, as it reduces the total number of iterations in one sampling round. Besides, DISK +PJ +SR has the smallest indexing time, indicating PJ and SR are orthogonal, and can be used together.

For online queries, using the dataset TW, we set ϵ_d to 1×10^{-3} and then vary ϵ from 1×10^{-2} to 5×10^{-2}, with the step of 1×10^{-2}. The query time of different methods are shown in Figure 6b. We find that both optimizations improve the efficiency. After adding ES and PR techniques, the query time decreases correspondingly, by 54% and 27% on average. We also find when ϵ increases from 0.04 to 0.05, the query time of DISK does not change, due to the iterative synchronization nature of online query. That is, different precisions within small gaps may share the same number of iterations. DISK +ES also shows a similar stair-shape pattern, but its iterations are still less than basic DISK. DISK +ES +PR has the has the smallest query time, showing ES and PR are orthogonal.

8.2.2 Parallel Scalability. Here we test the parallel scalability of our methods, *i.e.,* the change in the cost *w.r.t.* a varying p of workers.

We first test the parallel scalability of the offline phase. We set ϵ_d to 1×10^{-2}, and then vary p from 16 to 256, with a scaling factor of 2. We test different methods on the large datasets. The results are shown in Figures 7a-7e. We can find that over all datasets, the indexing time of all of our methods (including the basic version and the optimized versions) decreases with an increasing p, indicating the more workers which are added, the faster the indexing phase is. For example, on the dataset UK, when the number of workers increases from 16 to 256, the the indexing time of DISK decreases from 9.3×10^4s to 1.3×10^4s, achieving around 7.2X time speed-up. This is consistent with our theoretical cost analysis in previous sections. Besides, we can also find that under optimizations of PJ and SR, the parallel scalability still holds. Using $p = 256$, DISK is able to index CW within 1.1×10^5 seconds. DISK failed to process CW when $p \leq 32$, because the portion of graph data assigned to a single worker grows when p decreases. It then eventually exceeds the memory volume of a machine.

Table 4: Comparison of Different Methods (OOM denotes Out Of Memory).

Dataset	Time (seconds)										
	READS		SLING		ProbeSim	PRSim		SimPush	UniWalk	DISK	
	Preproc.	Query	Preproc.	Query	Query	Preproc.	Query	Query	Query	Preproc.	Query
LJ	6.1×10^2	0.86	1.2×10^3	0.14	1.6	8.5×10^1	0.15	0.12	**0.02**	3.4×10^2	0.15
TW	OOM		OOM		9.0×10^2	5.7×10^3	3.5	**3.4**	-	4.9×10^3	5.0
UK	OOM		OOM		OOM	1.4×10^3	**0.18**	0.27	-	2.9×10^3	0.53
FS	OOM		OOM		OOM	2.7×10^3	1.0	0.25	**0.05**	6.4×10^3	7.9
CW	OOM		OOM		OOM	OOM		OOM	-	1.1×10^5	**23**

We then test the parallel scalability for the online phase. We set $\epsilon_d = 1 \times 10^{-2}$ and $\epsilon = 2 \times 10^{-2}$, and then vary p from 16 to 256, with the scaling factor of 2. We test different methods on the large datasets. The results are shown in Figures 8a-8e. We can find that as p increases, the query time of different methods decreases over all datasets. The average speed-ups when p varies from 16 to 256 are 4.3 for LJ, 4.9 for FS, 3.9 for TW and 4.3 for UK. Another observation is that the online optimizations *i.e.,* ES and PR, always work over a varying p. For example, over the dataset TW, the optimized version (DISK +ES +PR) is always at least 2X faster than the basic version. This reveals that the online optimization techniques does not undermine the parallel scalability, this is also consistent with our analysis in Section 7. As for CW, a graph with 1 billion vertices and 42 billion edges, DISK is able to answer queries when $p \geq 64$, and responds in 23 seconds when $p = 256$. This shows DISK efficiently answers queries over massive datasets, whose volume can not fit in a single machine.

8.2.3 Scalability. We test the scalability of our methods *w.r.t.* graph size $|G|$. We develop a generator to produce synthetic graphs controlled by the number of nodes $|V|$ and edges $|E|$. We vary $|V|$ from 6.5×10^7 to 2.5×10^8 and the corresponding $|E|$ from 1.8×10^9 to 7×10^9. We set $\epsilon_d = 5 \times 10^{-3}$, $\epsilon = 1 \times 10^{-2}$ and $p = 256$. The offline indexing time of different methods is shown in Figure 7f. We can find that the indexing time of all of our proposed methods grows linearly with the graph size $|G|$. That is, it takes 3.1X more time to index, when $|G|$ grows 3.9 times larger.This is consistent with our previous cost analysis in Sections 4 and 6. We can observe after equipping DISK with SR and PJ, the indexing time dramatically decreases, *i.e.,* the speed of growth of indexing time *w.r.t.* $|G|$ is slower than basic DISK, and the scalability is still guaranteed.

Under the same setting, the query time of different methods is shown in Figure 8f. We find that our methods take more time when $|G|$ grows, and can answer an online query in 5.0 seconds when $|G|$ is up to 7 billion. The optimized version of DISK +ES +PR achieves the lowest responding time. This shows our optimization techniques do not undermine the scalability, and all of our methods scale well with the graph size. This is consistent with our previous analysis in Sections 5 and 7.

8.3 Comparison with Other Algorithms

In this section, we compare DISK with other algorithms, including: (1) **READS** [14]: a tree-based single-source solution using sampling; (2) **SLING** [29]: an index-based method which indexes the hitting probability of each node and supports shared-memory parallelism for indexing; (3) **ProbeSim** [19]: an index-free method which finds the potential meeting nodes around the source node; (4) **PRSim** [38]: an efficient single-source solution for power-law graphs; (5) **SimPush** [26]: an index-free statistic based single-source method (6) **UniWalk** [28]: an index-free distributed method for undirected graphs (7) **DISK** equipped with optimization techniques. We did not compare DISK with CloudWalker [18], whose result accuracy is unbounded (pointed out in Section 3 and [29]). As shown in [36], the ME of CloudWalker can be larger than 1, which is no better than randomly returning a real number in [0, 1]. We set $\epsilon = 2 \times 10^{-2}$. Distributed methods UniWalk and DISK are deployed on the cluster, with $p = 256$, while other methods run on a single machine. For parallel SLING, we use all 20 available cores of a machine for its indexing phase. For UniWalk, we only run it on undirected LJ and FS with default parameters in [28], as it can not handle directed graphs. We use the default setting in [28] for UniWalk, and set the sampling number $R = 10000$ and path length $L = 5$. The results are summarized in Table 4. From the results we can see 1) DISK is the most scalable method of all, it successfully builds index and answers queries over all tested datasets. Other methods fail to handle CW due to a lack of memory. READS and SLING even fail to process TW and UK due to their excessive memory consumption. 2) For offline indexing, DISK is comparable with other index-based methods. For instance, it is 1.8 and 3.6 times faster than READS and SLING over LJ, and at most 4 times slower than PRSim. 3) For online query answering, DISK is comparable with other methods on directed graphs. Over TW and UK, it is on average 2.2 and 1.7 times slower than PRSim and SimPush. On undirected LJ, its responding time is 0.17, 0.99 and 0.09 of READS, SLING and ProbeSim. UniWalk performs the best on undirected graphs, and is insensitive to the size of graphs, which is in accord with [28]. This is because UniWalk adopts a sampling method, whose complexity is independent of $|G|$. However, it has no accuracy guarantee. That is, it provides no guideline of how to set R and L for an ϵ, and thus the quality of results is unbounded.

Summary of Findings. We summarize our findings in experiments as follows: (1) Both the offline phase and the online phase of DISK guarantee the accuracy, and the maximum errors are on average 46% and 16% of the theory bound for ϵ_d and ϵ, respectively; (2) DISK scales well *w.r.t.* the number of processors p. When p varies from 16 to 256, it speeds up the algorithm 5.9 and 4.5 times, for offline indexing and online queries, respectively; (3) DISK scales well *w.r.t.* the size of graph $|G|$. It can index graphs up to 42 billion edges, and answers online queries in 23 seconds; (4) The optimization techniques significantly boost the offline indexing and online queries, on average by 4.2 and 2.4 times, respectively; (5) Among tested SimRank methods, DISK is the most scalable one. Compared with other methods, it takes a comparable time for both offline indexing and online query answering over directed graphs.

9 RELATED WORKS

Single-source SimRank Algorithms. In addition to tree-based methods [6, 14, 25] and linearization based methods [15, 23] discussed previously, there are also other algorithms for single-source

SimRank. [19] introduces ProbSim, which is an index-free method for queries over dynamic graphs. ProbSim firstly samples a bunch of $\sqrt{c}$-walks from the source node, and then finds potential meeting nodes using these walks, and the query time is $O(\frac{n}{\epsilon^2}\log n)$. [38] proposes PRSim, which uses an index of size $O(m)$. The complexity of PRSim is related to the reverse PageRank of G, which is the same as ProbSim on worst-case graphs, and is sub-linear on real-world power-law graphs. [26] proposes SimPush, which firstly identifies a small subset of nodes in G that are most relevant to the query, then computes important statistics and performs graph traversal starting from attention nodes only. Recently, [32] proposes ExactSim for probabilistic exact single-source SimRank queries, with $O(\frac{\log n}{\epsilon^2} + m\log\frac{1}{\epsilon})$ cost. It utilizes the linearization formula, and computes D on the fly based on the distribution of l-hop PPR scores. ExactSim is highly parallel on shared-memory model, since it relies on sparse matrix-vector multiplication and independent random walk sampling. While above methods are based on the *shared-memory model*, where the random access to the whole graph is permitted, DISK is a distributed solution on the *shared-nothing model*, which challenges the parallel scalability *w.r.t.* local computation and communication. For example, running multiple random walks simultaneously to achieve parallelism is applicable on shared-memory model [29, 32], but leads to congestion on shared-nothing model (discussed in Section 4.2), undermining parallel scalability.

Parallel SimRank Computation. In addition to the distributed methods discussed in Section 1, there are also other parallel methods for SimRank computation. [8] introduces a parallel algorithm for all-pairs SimRank, which firstly formulates SimRank to a homogeneous first-order Markov chain, and then utilizes iterative aggregation techniques for parallel computation on GPUs. However, there is no accuracy guarantee for this method since it approximates D as $(1-c)I$. Furthermore, this method is not applicable for large graphs since n^2 coordinates are materialized in the memory. Recently, [33] proposes a parallel method for all-pairs SimRank, by formulating exact SimRank as a solution of a linear system, and introducing a local push based method. In practice, the memory usage is much less than $O(n^2)$ due to the asynchronous update. The above methods differ from our work in that (1) they return an all-pairs SimRank matrix (dense or sparse) instead of single-source SimRank queries and (2) they parallelize SimRank computation on a single machine, using GPUs or multi-core CPUs, while our solution is distributed.

10 CONCLUSION

In this paper, we present DISK, a framework for processing single-source SimRank queries on a shared-nothing environment. DISK follows the linearized SimRank formulation, and consists of offline and online phases. In the offline phase, a tree-based method is used to estimate the diagonal correction matrix $\hat{D}$. We propose two optimization techniques for the offline indexing: one is sample reduction, the other is boosting leaf coloring by adopting pointer jumping. In the online phase, single-source similarities are computed iteratively, and we propose two optimization techniques. One is early stop which reduces the number of iterations, the other is a partition refiner, which aims to balances the number of in-edges and out-edges simultaneously in each worker. The accuracy, efficiency, scalability and parallel scalability of DISK are verified theoretically and experimentally. In the future, we plan to design efficient distributed algorithms for other SimRank-based queries, such as single-pair queries and top-k queries.

ACKNOWLEDGMENTS

This work is partially supported by National Key R&D Program of China (Grant No. 2018AAA0101100 and 2018YFB1003201), the Hong Kong RGC GRF Project 16214716, CRF Project C6030-18G, C1031-18G, C5026-18G, AOE Project AoE/E-603/18, Guangdong Basic and Applied Basic Research Foundation (No. 2019A1515110473 and 2019B151530001), China NSFC (No. 62002235, 61729201 and 62072311), Hong Kong ITC ITF grants ITS/044/18FX and ITS/470/18FX, Microsoft Research Asia Collaborative Research Grant, Didi-HKUST joint research lab project, and Wechat and Webank Research Grants.

APPENDIX

Proof of Theorem 1. From the definition of S and $\hat{S}^{(T)}$, we have:

$$S_{v,u} - \hat{S}^{(T)}_{v,u} = \mathbf{e}_v^\top (S - \hat{S}^{(T)})\mathbf{e}_u = \underbrace{\sum_{t=0}^{T} c^t \mathbf{e}_v^\top P^{\top t}(D - \hat{D})P^t \mathbf{e}_u}_{A} + \underbrace{\sum_{t=T+1}^{\infty} \mathbf{e}_v^\top c^t P^{\top t} D P^t \mathbf{e}_u}_{B}. \quad (13)$$

Considering part A of Eq.(13), since $v \neq u$, we can get:

$$A = \sum_{t=1}^{T} c^t \mathbf{e}_v^\top P^{\top t}(D - \hat{D})P^t \mathbf{e}_u = \sum_{t=1}^{T} c^t P^{\top t}_{v,*}(D - \hat{D})P^t_{*,u} \leq \sum_{t=1}^{T} c^t \|\hat{D} - D\|_{max} = \frac{c(1-c^T)\epsilon_d}{1-c}. \quad (14)$$

Considering part B of Eq.(13), we get:

$$B = \sum_{t=T+1}^{\infty} \mathbf{e}_v^\top c^t P^{\top t} D P^t \mathbf{e}_u = \mathbf{e}_v^\top \Big(\sum_{t=T+1}^{\infty} c^t P^{\top t} D P^t \Big) \mathbf{e}_u$$
$$= c^{T+1} \mathbf{e}_v^\top P^{\top T+1} \Big(\sum_{t=0}^{\infty} c^t P^{\top t} D P^t \Big) P^{T+1} \mathbf{e}_u = c^{T+1} \mathbf{e}_v^\top P^{\top T+1} S P^{T+1} \mathbf{e}_u$$
$$= c^{T+1} P^{\top T+1}_{v,*} S P^{T+1}_{*,u} \leq c^{T+1} \|S\|_{max} = c^{T+1}. \quad (15)$$

We finish the proof by combining the two inequalities.

Proof of Theorem 5. We follow the similar proof of Theorem 1. However, instead of using Eq.(14) to bound the part A of Eq.(13), we present the following inequality:

$$A = \sum_{t=1}^{T} c^t P^{\top t}_{v,*}(D - \hat{D})\mathbf{q}_u^{(t)} \leq \epsilon_d \sum_{t=1}^{T} c^t P^{\top t}_{v,*} \mathbf{q}_u^{(t)}$$
$$< \epsilon_d \sum_{t=1}^{T} c^t \|P^t_{*,v}\|_1 \|\mathbf{q}_u^{(t)}\|_\infty < \epsilon_d \sum_{t=1}^{T} c^t \|\mathbf{q}_u^{(t)}\|_\infty, \quad (16)$$

based on that $P^{\top t}$ is a row-normalized matrix, and that $\mathbf{x}^\top \mathbf{y} \leq \|\mathbf{x}\|_1 \|\mathbf{y}\|_\infty$. Our proof is complete by combining Eq.(16) and (15).

REFERENCES

[1] Florian Bourse, Marc Lelarge, and Milan Vojnovic. 2014. Balanced graph edge partition. In *SIGKDD*. 1456–1465.

[2] Michael Busch, Krishna Gade, Brian Larson, Patrick Lok, Samuel Luckenbill, and Jimmy Lin. 2012. Earlybird: Real-time search at twitter. In *ICDE*. 1360–1369.

[3] Liangliang Cao, Brian Cho, Hyun Duk Kim, Zhen Li, Min-Hsuan Tsai, and Indranil Gupta. 2012. Delta-simrank computing on mapreduce. In *BigMine*.

[4] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. PowerLyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *EuroSys*. 1:1–1:15.

[5] Wenfei Fan, Kun He, Qian Li, and Yue Wang. 2020. Graph algorithms: parallelization and scalability. *SCIENCE CHINA Information Sciences* 63, 10 (2020), 203101:1–203101:21. https://doi.org/10.1007/s11432-020-2952-7

[6] Dániel Fogaras and Balázs Rácz. 2005. Scaling link-based similarity search. In *WWW*.

[7] Yasuhiro Fujiwara, Makoto Nakatsuji, Hiroaki Shiokawa, and Makoto Onizuka. 2013. Efficient search algorithm for SimRank. In *ICDE*.

[8] Guoming He, Haijun Feng, Cuiping Li, and Hong Chen. 2010. Parallel SimRank Computation on Large Graphs with Iterative Aggregation. In *KDD* (Washington, DC, USA). 10. https://doi.org/10.1145/1835804.1835874

[9] Jun He, Hongyan Liu, Jeffrey Xu Yu, Pei Li, Wei He, and Xiaoyong Du. 2014. Assessing Single-pair Similarity over Graphs by Aggregating First-meeting Probabilities. *Inf. Syst.* 42 (June 2014), 107–122. https://doi.org/10.1016/j.is.2013.12.008

[10] Wassily Hoeffding. 1963. Probability Inequalities for Sums of Bounded Random Variables. *J. Amer. Statist. Assoc.* 58, 301 (1963), 13–30. http://www.jstor.org/stable/2282952

[11] Glen Jeh and Jennifer Widom. 2002. SimRank: a measure of structural-context similarity. In *KDD*.

[12] Glen Jeh and Jennifer Widom. 2003. Scaling personalized web search. In *WWW*.

[13] Joseph JéJé. 1992. *An introduction to parallel algorithms*. Reading, MA: Addison-Wesley.

[14] Minhao Jiang, Ada Wai-Chee Fu, Raymond Chi-Wing Wong, and Ke Wang. 2017. READS: A Random Walk Approach for Efficient and Accurate Dynamic SimRank. *PVLDB* 10, 9 (2017), 937–948.

[15] Mitsuru Kusumoto, Takanori Maehara, and Ken-ichi Kawarabayashi. 2014. Scalable Similarity Search for SimRank. In *SIGMOD* (Snowbird, Utah, USA). 12. https://doi.org/10.1145/2588555.2610526

[16] Cuiping Li, Jiawei Han, Guoming He, Xin Jin, Yizhou Sun, Yintao Yu, and Tianyi Wu. 2010. Fast Computation of SimRank for Static and Dynamic Information Networks. In *EDBT* (Lausanne, Switzerland). 12. https://doi.org/10.1145/1739041.1739098

[17] Pei Li, Hongyan Liu, Jeffrey Xu Yu, Jun He, and Xiaoyong Du. 2010. Fast single-pair simrank computation. In *SDM*. SIAM, 571–582.

[18] Zhenguo Li, Yixiang Fang, Qin Liu, Jiefeng Cheng, Reynold Cheng, and John C. S. Lui. 2015. Walking in the Cloud: Parallel SimRank at Scale. *PVLDB* 9, 1 (2015), 24–35.

[19] Yu Liu, Bolong Zheng, Xiaodong He, Zhewei Wei, Xiaokui Xiao, Kai Zheng, and Jiaheng Lu. 2017. ProbeSim: Scalable Single-Source and Top-k SimRank Computations on Dynamic Graphs. *PVLDB* 11, 1 (2017), 14–26.

[20] Dmitry Lizorkin, Pavel Velikhov, Maxim Grinev, and Denis Turdakov. 2010. Accuracy estimate and optimization techniques for SimRank computation. *VLDB J* 19, 1 (2010), 45–66.

[21] Juan Lu, Zhiguo Gong, and Xuemin Lin. 2017. A Novel and Fast SimRank Algorithm. *TKDE* 29, 3 (2017), 572–585. https://doi.org/10.1109/TKDE.2016.2626282

[22] Siqiang Luo. 2019. Distributed pagerank computation: An improved theoretical study. In *AAAI*, Vol. 33. 4496–4503.

[23] Takanori Maehara, Mitsuru Kusumoto, and Ken-ichi Kawarabayashi. 2014. Efficient SimRank Computation via Linearization. *CoRR* (2014).

[24] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *SIGMOD*. 135–146. https://doi.org/10.1145/1807167.1807184

[25] Yingxia Shao, Bin Cui, Lei Chen, Mingming Liu, and Xing Xie. 2015. An Efficient Similarity Search Framework for SimRank over Large Dynamic Graphs. *PVLDB* 8, 8 (2015), 838–849.

[26] Jieming Shi, Tianyuan Jin, Renchi Yang, Xiaokui Xiao, and Yin Yang. 2020. Real-time Index-Free Single Source SimRank Processing on Web-Scale Graphs. *PVLDB* 13, 7 (2020), 966–978.

[27] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2017. *PuLP/XtraPuLP: Partitioning Tools for Extreme-Scale Graphs*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

[28] J. Song, X. Luo, J. Gao, C. Zhou, H. Wei, and J. X. Yu. 2018. UniWalk: Unidirectional Random Walk Based Scalable SimRank Computation over Large Graph. *TKDE* 30, 5 (May 2018), 992–1006. https://doi.org/10.1109/TKDE.2017.2779126

[29] Boyu Tian and Xiaokui Xiao. 2016. SLING: A Near-Optimal Index Structure for SimRank. In *SIGMOD* (San Francisco, California, USA). https://doi.org/10.1145/2882903.2915243

[30] Hanghang Tong, Christos Faloutsos, and Jia-Yu Pan. 2006. Fast Random Walk with Restart and Its Applications. In *ICDM*. 613–622. https://doi.org/10.1109/ICDM.2006.70

[31] Johan Ugander, Brian Karrer, Lars Backstrom, and Cameron Marlow. 2011. The anatomy of the facebook social graph. *arXiv preprint arXiv:1111.4503* (2011).

[32] Hanzhi Wang, Zhewei Wei, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2020. Exact Single-Source SimRank Computation on Large Graphs. In *SIGMOD*. ACM, 653–663. https://doi.org/10.1145/3318464.3389781

[33] Yue Wang, Yulin Che, Xiang Lian, Lei Chen, and Qiong Luo. 2020. Fast and Accurate SimRank Computation via Forward Local Push and Its Parallelization. *TKDE* (2020). https://doi.org/10.1109/TKDE.2020.2976988

[34] Yue Wang, Lei Chen, Yulin Che, and Qiong Luo. 2019. Accelerating Pairwise SimRank Estimation over Static and Dynamic Graphs. *VLDB J* 28, 1 (Feb. 2019), 99–122.

[35] Yue Wang, Zonghao Feng, Lei Chen, Zijian Li, Xun Jian, and Qiong Luo. 2019. Efficient Similarity Search for Sets over Graphs. *TKDE* (2019). https://doi.org/10.1109/TKDE.2019.2931901

[36] Yue Wang, Xiang Lian, and Lei Chen. 2018. Efficient SimRank Tracking in Dynamic Graphs. In *ICDE*. 545–556.

[37] Yue Wang, Zhe Wang, Ziyuan Zhao, Zijian Li, Xun Jian, Lei Chen, and Jianchun Song. 2020. HowSim: A General and Effective Similarity Measure on Heterogeneous Information Networks. In *ICDE*. 1954–1957. https://doi.org/10.1109/ICDE48307.2020.00212

[38] Zhewei Wei, Xiaodong He, Xiaokui Xiao, Sibo Wang, Yu Liu, Xiaoyong Du, and Ji-Rong Wen. 2019. PRSim: Sublinear Time SimRank Computation on Large Power-Law Graphs. In *SIGMOD*. 1042–1059.

[39] Da Yan, James Cheng, Yi Lu, and Wilfred Ng. 2014. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB* 7, 14 (2014), 1981–1992.

[40] Weiren Yu, Xuemin Lin, and Wenjie Zhang. 2013. Towards efficient SimRank computation on large networks. In *ICDE*. 601–612. https://doi.org/10.1109/ICDE.2013.6544859

[41] Weiren Yu, Xuemin Lin, and Wenjie Zhang. 2014. Fast incremental simrank on link-evolving graphs. In *ICDE*.

[42] Weiren Yu, Xuemin Lin, Wenjie Zhang, Lijun Chang, and Jian Pei. 2013. More is Simpler: Effectively and Efficiently Assessing Node-Pair Similarities Based on Hyperlinks. *PVLDB* 7, 1 (2013), 13–24.

[43] Weiren Yu and Julie A. McCann. 2015. Efficient Partial-Pairs SimRank Search for Large Networks. *PVLDB* 8, 5 (2015), 569–580.

[44] Weiren Yu and Julie Ann McCann. 2015. High Quality Graph-Based Similarity Search. In *SIGIR* (Santiago, Chile). 10. https://doi.org/10.1145/2766462.2767720

[45] Weiren Yu, Wenjie Zhang, Xuemin Lin, Qing Zhang, and Jiajin Le. 2012. A space and time efficient algorithm for SimRank computation. *WWW* (2012). https://doi.org/10.1007/s11280-010-0100-6

[46] Peixiang Zhao, Jiawei Han, and Yizhou Sun. 2009. P-Rank: a comprehensive structural similarity measure over information networks. In *CIKM*.