Walking in the Cloud: Parallel SimRank at Scale

Zhenguo Li[1], Yixiang Fang[2], Qin Liu[3], Jiefeng Cheng[1], Reynold Cheng[2], John C.S. Lui[3]
[1]Huawei Noah's Ark Lab, Hong Kong
[2]Department of Computer Science, The University of Hong Kong, Hong Kong
[3]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[1]{li.zhenguo, cheng.jiefeng}@huawei.com,
[2]{yxfang, ckcheng}@cs.hku.hk, [3]{qliu, cslui}@cse.cuhk.edu.hk

ABSTRACT

Despite its popularity, SimRank is computationally costly, in both time and space. In particular, its recursive nature poses a great challenge in using modern distributed computing power, and also prevents querying similarities individually. Existing solutions suffer greatly from these practical issues. In this paper, we break such dependency for maximum efficiency possible. Our method consists of offline and online phases. In offline phase, a length-n indexing vector is derived by solving a linear system in parallel. At online query time, the similarities are computed instantly from the index vector. Throughout, the Monte Carlo method is used to maximally reduce time and space. Our algorithm, called CloudWalker, is highly parallelizable, with only linear time and space. Remarkably, it responses to both single-pair and single-source queries in constant time. CloudWalker is orders of magnitude more efficient and scalable than existing solutions for large-scale problems. Implemented on Spark with 10 machines and tested on the web-scale clue-web graph with 1 billion nodes and 43 billion edges, it takes 110 hours for offline indexing, 64 seconds for a single-pair query, and 188 seconds for a single-source query. To the best of our knowledge, our work is the first to report results on clue-web, which is 10x larger than the largest graph ever reported for SimRank computation.

1. INTRODUCTION

Graph data arises from various domains such as telecommunication, the Internet, e-commerce, social networks, and the Internet of things. Usually the scales of the graphs are very large and they continuously grow in rapid speed. While big graph data is of great value, the huge scale poses non-trivial challenge in graph mining.

One fundamental task underpinning many graph mining problems such as recommender systems [14] and information retrieval [8] is the computation of similarity between objects. Among various ways of evaluating object similarity on graph [26, 27, 11], SimRank [15] is probably one of the most popular [10, 22, 21, 19, 29, 16, 23, 28, 25].

SimRank is derived from an intuitive notion of similarity – two objects are similar if they are referenced by similar objects [15]. This is similar to PageRank [24], where a webpage is important if the webpages pointing to it are important. Like PageRank, SimRank is also governed by a random surfer model [15]. Studies show that it captures human perception of similarity and outperforms other related measures such as co-citation which measures the similarity of two objects by counting their common neighbors [15, 10]. In a sense, SimRank is a recursive refinement of co-citation.

One major obstacle in applying SimRank to big graph data lies in its computation, which is notoriously intensive. It may take up to $O(n^3)$ time and $O(n^2)$ space for computing the similarity of two nodes even on a sparse graph, which is clearly far from acceptable for large problems. Indeed, since it emerged [15], its scalability has been a critical issue of study [10, 22, 21, 19, 29, 16, 23, 28]. Despite significant progress, there is still a large gap to a practically scalable solution. The major difficulty arises from the recursive dependency in the computation of SimRank, which incurs high complexity, limits parallelism, and prevents querying individual similarities. This paper addresses all these issues.

Large-scale computation is non-trivial. One of the most effective ways is to distribute it to a large cluster of machines. While a single machine may support large problems [5], the degree of parallelism is still constrained by its limited memory and concurrency in particular. This vision is well shared with the prevalence of distributed computing platforms such as Spark [1]. However, to parallelize an algorithm may not be as easy as it appears. First, if the complexity of the algorithm is high, say super-linear, then the required computation may easily outpace the increased power by more machines. Second, if the algorithm allows only limited parallelism due to computational dependency, then there is no much space for distributed computing – the synchronization and communication cost could be prohibitive. The above arguments suggest that to scale up SimRank using a cluster of machines, it is equally important to reduce its complexity and its computational dependency. Such a perspective guides us in the process of developing a scalable approach to large-scale SimRank computation.

In this paper, we develop a computational model that meets all the following criteria: 1) its complexity is no more than linear, in time and space; 2) it is highly parallelizable; and 3) it can compute individual similarities directly without computing the rest. Our main contributions are

Proceedings of the VLDB Endowment, Vol. 9, No. 1
Copyright 2015 VLDB Endowment 2150-8097/15/09.

summarized as follows.

- We propose a novel distributed approach, called CloudWalker, for SimRank computation, which supports single-pair, single-source, and all-pair queries.
- Our approach can be decomposed into offline preprocessing and online querying. The complexity of offline preprocessing is linear to the graph size, in time and space, while the online querying takes only constant time for both single-pair and single-source queries.
- We implement our approach, CloudWalker, on a general-purpose in-memory dataflow system, Spark [1], and conduct extensive evaluation. In contrast, most existing methods [23, 13] are designed for a single machine, which means that the entire graph must fit in the main memory and the parallelism is limited.
- We test on the web-scale graph, clue-web, with 1 billion nodes and 43 billion edges, which is 10x larger than the largest graph reported in the literature of SimRank computation. It takes 110 hours for preprocessing, 64 seconds for a single-pair query, and 188 seconds for a single-source query.

The rest of the paper is organized as follows. We review related work in Section 2 and give preliminaries in Section 3. In Section 4, we propose a parallel algorithm for SimRank computation with only linear time and space. We describe our implementation in Section 5, and present experimental results in Section 6. Section 7 concludes the paper.

2. RELATED WORK

Many approaches have been proposed to scale up SimRank. One straightforward idea uses iterative matrix multiplication, which is guaranteed to converge to the solution [15]. It can be accelerated by pruning [15], partial calculation memorization [22], or fast matrix multiplication [29]. Most methods are designed for a single machine, and are therefore limited by its restricted power. Recently, a few efforts attempt to distribute SimRank computation on a cluster.

Cao et al. [4] proposed a MapReduce method. Upon every node-pair (c, d) encountered by a mapper job, it attaches the SimRank score $s(c, d)$ of nodes c and d to all neighboring node-pairs (a, b) and outputs a key-value record $(key = (a, b), value = s(c, d))$. The subsequent reducer jobs then use all values under the same key (a, b) to update $s(a, b)$. The amount of data transferred from mapper to reducer in one iteration is $O(p^2 n^2)$, which can be reduced to $O(p^2 M)$ by computing the similarities incrementally and by transferring only non-zero increments, where p denotes the average in-degree, n the number of nodes in the graph, and M the number of non-zero similarities [4]. A similar idea is used by Li et al. [20]. He et al. [13] employed GPU to speedup SimRank computation, which however is a single machine solution and requires the graph to fit in the main memory. As pointed out by [23], [13] only computes a variant of SimRank, like [19, 16].

While the approach of iterative matrix multiplication is easy to implement, it comes at a great cost – in each iteration, it needs to perform expensive matrix multiplication and maintain all n^2 similarities simultaneously, which is clearly not practical for large problems. In addition to its high complexity, another consequence of this approach is that it does not allow querying individual node-pairs or source nodes without querying the rest. In contrast, our method has linear complexity in time and space, and also supports the SimRank query of a single node-pair or source node, while still being highly parallelizable.

Another approach takes advantage of the random surfer model of SimRank [10, 21]. It has been shown that the SimRank score of a single node-pair is solely determined by the first expected meeting time of random walkers starting from two nodes respectively, following the graph backwardly [15]. The appeal of this random surfer view, as opposed to the above iterative matrix multiplication view, is that it can compute individual similarities directly. Two methods exist for evaluating meeting times on a graph. The First Meeting Time (FMT) method [10] simulates random walks for each node and stores the paths, called fingerprints, in external memory; and at query time, estimates the meeting time from the fingerprints. This method is stochastic in nature and usually requires a large number of samples of random paths for reasonable accuracy [23]. In contrast, the other method [21] is deterministic, which needs to evaluate the meeting probability exactly in k-th step for every k. Consequently, it can be time-consuming, compared to the FMT method. The two methods pre-compute and store the paths of random surfers, and both need external storage for large graphs. In that regard, they trade space for scalability. Our approach also consists of offline indexing and online querying, but our index data is just a length-n vector.

The most related work to ours is [23], where the bottleneck is in estimating a diagonal correction matrix that accounts for the boundary conditions of SimRank, i.e., 1) each object is maximally similar to itself, and 2) the self-similarity is defined to be one (Section 4.1). The key difference lies in the way that the correction matrix is estimated, which leads to a dramatically different consequence. In [23], the matrix is initialized with an (inaccurate) correction matrix. Then, its each diagonal entry is updated iteratively, in a greedy manner, with each update dependent on all previous updates. This leaves little room for parallelization. In contrast, we impose linear constraints on all entries in such a way that each entry can be updated independently. Consequently, our algorithm can be easily distributed across different machines, which enables the use of modern distributed computing platforms such as Spark [1] as we show in this paper.

3. PRELIMINARIES

Let $G = \{V, E\}$ be a directed graph with node set V and edge set E ($n{=}|V|$, $m{=}|E|$). An ordered node-pair $(i, j) \in E$ denotes an edge from node i to node j. Here, i is called an in-neighbor of j, and j an out-neighbor of i. The sets of in-neighbors and out-neighbors of node j are respectively denoted by $\mathrm{I}(j) \coloneqq \{i : (i, j) \in E\}$ and $\mathrm{O}(j) \coloneqq \{i : (j, i) \in E\}$. The SimRank score (similarity) between nodes i and j is denoted by $s(i, j)$, and the similarity matrix of SimRank is denoted by S with $S_{ij} = s(i, j)$.

3.1 SimRank

There exists various ways to evaluate node similarity on graph. In SimRank, the similarity of two nodes i and j is the average of the similarities of their in-neighbors, decayed by a factor; and the similarity of a node and itself is maximal and defined to be one. Mathematically, the similarity of nodes i

and j is defined as follows:

$$s(i,j) = \begin{cases} 1, & i = j; \\ \dfrac{c}{|\mathrm{I}(i)||\mathrm{I}(j)|} \displaystyle\sum_{i' \in \mathrm{I}(i), j' \in \mathrm{I}(j)} s(i', j'), & i \neq j. \end{cases} \quad (1)$$

Here, c is a decay factor, $c \in [0, 1]$. Its role is two-fold. It weights down the influence of in-neighbors, and it makes the problem well-conditioned. By definition, $s(i, j) = 0$ if $I(i) = \emptyset$ or $I(j) = \emptyset$. A unique solution to Eq. (1) is guaranteed to exist [15].

A solution to Eq. (1) can be computed in an iterative fashion. Suppose $R^{(k)}$ is an $n \times n$ matrix, and $R_{ij}^{(k)}$ gives the SimRank score between i and j on iteration k. Then we set $R^{(0)}$ to the lower bound of the actual SimRank scores:

$$R_{ij}^{(0)} = \begin{cases} 1, & i = j; \\ 0, & i \neq j. \end{cases} \quad (2)$$

To compute $R^{(k+1)}$, we use Eq. (1) to get

$$R_{ij}^{(k+1)} = \begin{cases} 1, & i = j; \\ \dfrac{c}{|\mathrm{I}(i)||\mathrm{I}(j)|} \displaystyle\sum_{i' \in \mathrm{I}(i), j' \in \mathrm{I}(j)} R_{i',j'}^{(k)}, & i \neq j. \end{cases} \quad (3)$$

It has been shown that $\lim_{k\to\infty} R_{ij}^{(k)} = s(i, j)$ [15].

3.2 Spark

Our approach is distributed in nature. Hence, in principle, it can be implemented on any distributed data processing platforms. We choose Spark [1] because: 1) it is a popular general-purpose distributed dataflow framework for large-scale data processing, and facilitates in-memory cluster computing, which is essential for iterative algorithms including ours; 2) it provides easy-to-use operations for building distributed and fault-tolerant applications; and 3) it enables the development of concise programs: it is developed in Scala, which supports both object-oriented and functional programming. It would also be interesting to study how to deploy our algorithm on other platforms.

Spark provides an in-memory storage abstraction known as Resilient Distributed Datasets (RDDs) [30] that allows applications to keep data in the shared memory of multiple machines. RDDs are fault-tolerant since Spark can automatically recover lost data. An RDD can be seen as a collection of records, where two types of operations over RDDs are available: *transformations* which create new RDDs based on existing ones, and *actions* which return some local variables to the master. The processing in Spark is in multiple stages and the lineage of operations to construct an RDD is automatically logged, which enables fault-tolerance with negligible runtime overhead. Transformations used in this paper include `map`, `flatMap`, `reduceByKey`, and `leftOuterJoin`. In detail, `map` constructs an one-to-one mapping of the input RDD, and `flatMap` constructs an one-to-many of the input (similar to the map operation in MapReduce); `reduceByKey` only works on RDDs of key-value pairs and it generates a new RDD of key-value pairs where the values for each key are aggregated using the given reduce function (similar to the reduce operation in MapReduce); and `leftOuterJoin` can perform left join operation over two key-value RDDs. Note that `reduceByKey` and `leftOuterJoin` need to shuffle data among machines with some network communication cost. In this paper, we only use one action, `collect`, and it returns all elements of an RDD. In addition to these operators, the user can call `persist` to indicate which RDD to be reused in future and Spark will cache persistent RDDs in memory. Spark also allows the user to broadcast a local variable from the master node to every slave node in the cluster.

4. EVALUATING SIMRANK QUERIES

In this section, we propose a novel approach to SimRank computation. We focus on several practical issues: 1) low complexity, in time and space, which is critical for large problems; 2) high degree of parallelism, which allows to harness the power of a distributed cluster; 3) online querying, which needs real-time response and is important for interactive applications; and 4) reasonably high accuracy, which guarantees the correctness of the computation. For the SimRank problem, there are generally three types of queries: 1) single-pair query, which returns the similarity of a single node pair; 2) single-source query, which returns the n similarities of a given source node and every node; and 3) all-pair query, which returns all n^2 pair-wise similarities. We consider all these queries[1].

Towards the above goals, in what follows, we will show 1) how the recursive dependency of SimRank can be eliminated; 2) how offline indexing enables online querying; 3) how offline indexing (and online querying) can be done in parallel; and 4) how the complexity can be reduced by exploiting the problem structures.

4.1 Problem Statement

SimRank, as defined in Eq. (1), can be written in a matrix form [29, 16]:

$$S = (cP^\top SP) \vee I, \quad (4)$$

where $P^\top$ denotes the transpose matrix of P, and P is the transition matrix of the transpose graph[2] $G^\top$,

$$P_{ij} := \begin{cases} 1/|\mathrm{I}(j)|, & (i,j) \in E \\ 0, & (i,j) \notin E \end{cases} \quad (5)$$

and $\vee$ denotes element-wise maximum, i.e., $(A \vee B)_{ij} = \max\{A_{ij}, B_{ij}\}$.

Diagonal Factorization. By Eq. (4), $cP^\top SP$ and S differ only in the diagonal, and so the following *diagonal factorization* holds

$$S = cP^\top SP + D, \quad (6)$$

for some unknown diagonal matrix D [16]. We call D the *diagonal correction matrix*, as it makes up the diagonal of $cP^\top SP$ so that the diagonal of S are 1's.

The Roles of D. The diagonal factorization essentially breaks the original recursive dependence of SimRank. In fact, S can be factorized in terms of D, as follows:

$$S = \sum_{t=0}^{\infty} c^t P^{\top t} D P^t. \quad (7)$$

The significance here is that the computation of the similarity of a node pair does not rely on the similarity of any

[1] Our method can be readily extended to support the top-k query, say, by returning just the top-k nodes obtained from the single-source query, or more efficiently by using the recent technique in [16].

[2] The direction of each edge is reversed.

other node pairs. This allows querying individual similarity independently which enables parallelism. Moreover, as D is independent of any query, it can be computed *offline*. In this regard, D can be seen as the *index* data of the graph for SimRank computation. Below, we first review related work for computing D, and then present our proposed approach.

Related Work on Computing D. In [23], an iterative procedure is proposed to compute D. The key observation is that the operator $S^L(\Theta) := cP^\top S^L(\Theta)P + \Theta$ is linear in Θ, and the key idea is to find a diagonal matrix Θ such that the diagonal of $S^L(\Theta)$ are all 1's, which implies that $S^L(\Theta)$ is the desired similarity matrix [23]. Starting with an initial diagonal matrix D (say $D = I$), the procedure updates its diagonal entries, one at a time. To update D_{kk}, it imposes $S^L(D + \delta E^{(k,k)})_{kk} = 1$, which, by the linearity, yields $\delta = (1 - S^L(D)_{kk})/S^L(E^{(k,k)})_{kk}$, where $E^{(k,k)}$ is a zero matrix except its (k,k)-th entry being 1.

The major limitation of this algorithm is that it needs to evaluate $S^L(D)_{kk}$ for a large number of times. Moreover, the update of D_{kk} relies on previous updates of other diagonal entries. Consequently, it cannot be parallelized. In contrast, our approach is easily parallelizable, as shown in the following section.

4.2 Offline SimRank Indexing

We cast the problem of estimating D as a linear system. We will show how to derive the linear system, and how to solve it in parallel.

4.2.1 Linear System

In light of Eq. (7) and noting that c^i exponentially decreases with i, we can make the following approximation:

$$S \approx S^{(T)} = \sum_{t=0}^{T} c^t P^{\top t} D P^t, \tag{8}$$

where T is a small integer (we set $T = 10$ in this paper). The approximation error is bounded as follows.

THEOREM 4.1. $(S^{(T)})_{ij} \le S_{ij} \le (S^{(T)})_{ij} + \frac{c^{T+1}}{1-c}$.

PROOF. First, we show that $1 - c \le D_{ii} \le 1$. By Eq. (6), $D_{ii} = 1 - c\mathbf{p}_i^\top S\mathbf{p}_i$. Here $\mathbf{p}_i$ denotes the i-th column of P, and so $\mathbf{1}^\top \mathbf{p}_i = 1$ or 0[3]. By definition, $0 \le S \le 1$, and so $0 \le \mathbf{p}_i^\top S\mathbf{p}_i \le 1$, which implies $1 - c \le D_{ii} \le 1$. Based on this result, we can see that $0 \le (P^{\top t}DP^t)_{ij} \le 1$ for $t = 0, 1, \ldots, \infty$.

Now, since $S = S^{(T)} + \sum_{t=T+1}^{+\infty} c^t P^{\top t} D P^t$, $S_{ij} = (S^{(T)})_{ij} + \sum_{t=T+1}^{+\infty} c^t (P^{\top t}DP^t)_{ij}$, implying that $(S^{(T)})_{ij} \le S_{ij} \le (S^{(T)})_{ij} + \sum_{t=T+1}^{+\infty} c^t$. Since $\sum_{t=T+1}^{+\infty} c^t = \frac{c^{T+1}}{1-c}$, our proof is complete. □

Recall that by definition, the diagonal of S are 1's. Combined with the above arguments, we consider the following n linear constraints:

$$\mathbf{e}_i^\top S^{(T)} \mathbf{e}_i = 1, \text{for } i = 1, \ldots, n, \tag{9}$$

where $\mathbf{e}_i$ is the i-th unit vector whose i-th coordinate is 1, and other coordinates are zeros.

Plugging Eq. (8) into Eq. (9) and with some mathematical operations, the constraints can be made explicitly on the unknown diagonal of D, $\mathbf{x} := (D_{11}, \ldots, D_{nn})^\top$, as follows:

$$\mathbf{a}_i^\top \mathbf{x} = 1, i = 1, \ldots, n, \tag{10}$$

where

$$\mathbf{a}_i = \sum_{t=0}^{T} c^t (P^t \mathbf{e}_i) \circ (P^t \mathbf{e}_i), i = 1, \ldots, n. \tag{11}$$

Here $\circ$ denotes element-wise product, i.e., $(\mathbf{a} \circ \mathbf{b})_i = (\mathbf{a})_i (\mathbf{b})_i$.

By Eq. (11), to compute one $\mathbf{a}_i$ takes $O(Tm)$ time, where m denotes the number of edges in the graph, and it takes $O(Tmn)$ time for all $\mathbf{a}_i$'s, which is not practical for large graphs. It turns out that this cost can be greatly reduced using the problem structure. Before we show that, we first show how the linear system can be solved in parallel, assuming $\mathbf{a}_i$'s are available.

4.2.2 The Jacobi Method

Suppose we have a square system of n linear equations:

$$A\mathbf{x} = \mathbf{1}, \tag{12}$$

where $A = (\mathbf{a}_1, \ldots, \mathbf{a}_n)^\top$. By the Jacobi method [12], the following iteration

$$x_i^{(k+1)} = \frac{1}{A_{ii}}(1 - \sum_{j \ne i} A_{ij} x_j^{(k)}), i = 1, \ldots, n, \tag{13}$$

will converge to the true solution under some mild conditions[4]. Note that $A_{ij} = (\mathbf{a}_i)_j$.

From Eq. (13), we can observe the high potential of parallelism of the Jacobi method: since in the $(k+1)$-th iteration, the solution $\mathbf{x}^{(k+1)}$ is calculated only from the solution of the previous iteration $\mathbf{x}^{(k)}$, we can update $x_i^{(k+1)}$ for $i = 1, \ldots, n$ in parallel assuming A is available.

4.2.3 The Monte Carlo Method

Now we show how a linear constraint $\mathbf{a}_i$ can be computed efficiently. According to Eq. (11), in order to compute $\mathbf{a}_i$ we need to compute $P^t\mathbf{e}_i$ for $t = 1, \ldots, T$ which equals to the t-th step distribution of random walks starting from node i following in-edges on the input graph [9]. Let $\mathbf{e}_i^{(t)}$ be the n-dimensional state random variable of the walker after t steps, i.e., $(\mathbf{e}_i^{(t)})_\ell = 1$ if the walker is in state ℓ (node ℓ), and 0 otherwise. Then

$$\mathrm{E}(\mathbf{e}_i^{(t)}) = P^t \mathbf{e}_i. \tag{14}$$

To estimate $P^t\mathbf{e}_i$, the expectation of the random vector $\mathbf{e}_i^{(t)}$, we can use sampling. Particularly, we simulate R independent length-t random walks, and denote by R_ℓ the number of times the walker is in state ℓ. Then

$$(P^t \mathbf{e}_i)_\ell \approx \frac{R_\ell}{R}. \tag{15}$$

It can be shown that to achieve accuracy within error ϵ, we need $R = O((\log n)/\epsilon^2)$ samples [23]. Precisely, it holds

$$\mathbf{P}\left\{\|P^t\mathbf{e}_i - \mathbf{p}_i^{(t)}\| > \epsilon\right\} \le 2n \cdot \exp\left(-\frac{(1-c)R\epsilon^2}{2}\right), \tag{16}$$

where $\mathbf{P}$ denotes probability and $\mathbf{p}_i^{(t)} = (\frac{R_1}{R}, \cdots, \frac{R_n}{R})^\top$.

[3] By definition, $\mathbf{p}_i = \mathbf{0}$ if $\mathrm{I}(i) = \emptyset$.

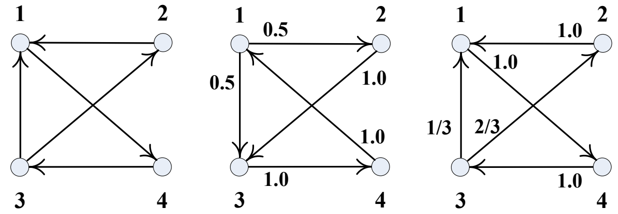
[4] In our experiments, we never encounter convergence issues.

Next we use one example to illustrate the Monte Carlo estimation method.

EXAMPLE 4.1. *Let G be a graph in Figure 1(a). Its transpose graph $G^\top$ is shown in Figure 1(b). The transition matrix of $G^\top$ is*

$$P = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0.5 & 0 & 0 & 0 \\ 0.5 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}. \tag{17}$$

Let $\mathbf{e}_1 = (1, 0, 0, 0)^\top$. To compute $P\mathbf{e}_1$ using Monte Carlo, we can place R walkers on node 1 of $G^\top$ and push them to walk one step randomly, following the out-edges of $G^\top$ (i.e. the in-edges of G). Ideally, there will be $\frac{1}{2}R$ walkers on nodes 2 and 3, respectively, and the estimated distribution of walkers on the nodes will be $(0, 0.5, 0.5, 0)^\top$, which is exactly $P\mathbf{e}_1$.



(a) G (b) $G^\top$ (c) $\hat{G}$

Figure 1: Monte Carlo simulation

4.2.4 Complexity of Computing D

Previously, when we use the Jacobi method to solve $A\mathbf{x} = 1$, we assume that A is available beforehand. Here we discuss the complexity of computing A using the Monte Carlo method. In order to approximate each $\mathbf{a}_i$, we push R walkers to move T steps. Hence, the time complexity to compute A is $O(nTR)$. For each i, $\mathbf{a}_i$ contains at most $O(TR)$ non-zero elements, which means the space used to store A is also $O(nTR)$.

However, for large-scale graphs such as the clue-web of 1 billion nodes and 43 billion edges (Table 1) considered in this paper, it is generally impractical to store the entire A in the distributed memory of a cluster of machines. To solve this problem, we observe that, in the Jacobi method, to compute $x_i^{(k+1)}$, we only need $\mathbf{a}_i$ and also $x_i^{(k)}$. So we can split $1, 2, \ldots, n$ into disjoint intervals of length b, where b is the number of $\mathbf{a}_i$'s we can compute in parallel. In iteration $k+1$ of the Jacobi method, we first compute $\mathbf{a}_i$'s, $i = 1, \ldots, b$, in parallel, and then update $x_i^{(k+1)}$'s, $i = 1, \ldots, b$, also in parallel. After that, we can compute $\mathbf{a}_i$ and update $x_i^{(k+1)}$ for $i = b+1, \ldots, 2b$ in parallel, and we can compute the rest of $\mathbf{a}_i$ and $x_i^{(k+1)}$ in a similar way. By recomputing A, we only need $O(bTR)$ space to store A. Suppose the Jacobi method takes L iterations to converge, then the time complexity of computing D is $O(nLTR)$ and the space complexity is $O(bTR + n + m)$. To reduce the computation time, we parallelize the computation of D on the distributed platform, Spark, as detailed in Section 5.

4.3 Online SimRank Queries

As mentioned above, there are three fundamental problems on SimRank computation, the single-pair, single-source, and all-pair SimRank computation. In this section, we propose algorithms for the three problems based on Monte Carlo simulation, which are called MCSP, MCSS, and MCAP respectively.

4.3.1 The MCSP Algorithm

Given a pair of nodes i and j, we want to return the SimRank score $s(i,j) = \mathbf{e}_i^\top S \mathbf{e}_j$. By Theorme 4.1, it can be done approximately as:

$$s(i,j) \approx \sum_{t=0}^{T} c^t (P^t \mathbf{e}_i)^\top D (P^t \mathbf{e}_j). \tag{18}$$

We could simply evaluate it with matrix-by-vector product, but it would take $O(Tm)$ time, and it would be costly to answer many single-pair queries especially for large graphs.

A more efficient way is to use the Monte Carlo method. Consider one of its items,

$$c^t (P^t \mathbf{e}_i)^\top D (P^t \mathbf{e}_j). \tag{19}$$

Since D has been computed in the preprocessing stage, we only focus on computing $P^t \mathbf{e}_i$ using Monte Carlo. Initially, we assign R' walkers on both nodes i and j. We then obtain the values of $P\mathbf{e}_i, \cdots, P^T\mathbf{e}_i, P\mathbf{e}_j, \cdots, P^T\mathbf{e}_j$ by pushing these walkers to walk T steps following the in-edges of G. Finally we obtain $s(i,j)$ according to Eq. (18).

Complexity. For any node i, since all the $P\mathbf{e}_i$, $P^2\mathbf{e}_i$, $\cdots, P^T\mathbf{e}_i$ can be computed by pushing R' walkers to walk T steps on the graph, the time complexity of Monte Carlo simulation is $O(TR')$. Thus, the total time complexity of MCSP is $O(TR')$. The space complexity is also $O(TR')$, since there are at most R' non-zero entries for each $P^t\mathbf{e}_i$.

4.3.2 The MCSS Algorithm

Given a node i, we want to compute its similarities to every node in the graph, i.e., the i-th column of S given by $S\mathbf{e}_i$. By Theorme 4.1, it can be approximated as follows:

$$S\mathbf{e}_i \approx \sum_{t=0}^{T} c^t P^{\top t} D (P^t \mathbf{e}_i). \tag{20}$$

Computing Eq. (20) by using the matrix-by-vector product would take $O(T^2 m)$ time, which is not efficient enough for large graphs. Below we show that Monte Carlo simulation can again be applied to the single-source problem, leading to an efficient algorithm.

Let us consider one of its items:

$$c^t P^{\top t} D P^t \mathbf{e}_i. \tag{21}$$

As before, we can compute $P^t\mathbf{e}_i$ using Monte Carlo. Then, we compute $DP^t\mathbf{e}_i$ and denote the result as $\mathbf{v} = DP^t\mathbf{e}_i$. Since $1 - c \le D_{ii} \le 1$, as shown in the proof of Theorem 4.1, we can conclude that $\mathbf{v} \ge 0$.

Now the task becomes computing $P^{\top t}\mathbf{v}$. For simplicity, let us focus on $P^\top \mathbf{v}$ first ($P^{\top t}\mathbf{v}$ can be computed simply by repeating this procedure). Note that $P^\top$ is usually not column-normalized as P, and thus $P^\top \mathbf{v}$ is not an one-step distribution of random walks. Fortunately, $P^\top$ can be turned into such a matrix, $\hat{P}$, as follows:

$$P^\top = \hat{P}F, \tag{22}$$

where F is a diagonal matrix with $F_{ii} = \sum_{j=0}^{n} P_{ij} \geq 0$, and $\hat{P}$ is an $n \times n$ matrix with $\hat{P}_{ij} = \frac{P_{ji}}{F_{jj}}$.

Now $P^\top \mathbf{v} = \hat{P}F\mathbf{v}$. Noting that $F\mathbf{v} \geq 0$, we can write

$$F\mathbf{v} = \mathbf{u} \cdot w, \tag{23}$$

such that $\mathbf{1}^\top \mathbf{u} = 1$ and $w = \mathbf{1}^\top (F\mathbf{v})$, $\mathbf{u} \geq 0$, $w \geq 0$.

Now $P^\top \mathbf{v} = w\hat{P}\mathbf{u}$. Let us consider $\hat{P}\mathbf{u}$. Given a matrix $\hat{P}$, we can construct a new directed weighted graph $\hat{G}$ with n nodes so that $\hat{P}$ is its transition probability matrix. $\hat{G}$ has the same graph structure as G, except that it is weighted (Figure 1(c)). Given the start distribution $\mathbf{u}$ on $\hat{G}$, we can simulate $\hat{P}\mathbf{u}$ using Monte Carlo, as follows.

1. For each node i, if $u_i > 0$, we place

$$R_i = \begin{cases} \lceil u_i R' \rceil, u_i R' - \lfloor u_i R' \rfloor \geq 0.5 \\ \lfloor u_i R' \rfloor, u_i R' - \lfloor u_i R' \rfloor < 0.5 \end{cases} \tag{24}$$

   walkers on this node, where R' is the initial total number of walkers.

2. Each walker randomly chooses one out-neighbor according to $\hat{P}$ and walks to it.

3. Count the number of walkers on each node. Denote by c_i the number for node i.

4. Return $(\hat{P}\mathbf{u})_i \approx c_i / \sum_{j=1}^{n} R_j$, $i = 1, \ldots, n$.

Example 4.2. *Consider the graph G in Figure 1(a). The corresponding $\hat{P}$ and F are as follows:*

$$\hat{P} = \begin{bmatrix} 0 & 1 & 1/3 & 0 \\ 0 & 0 & 2/3 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \; F = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 1.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{25}$$

The graph $\hat{G}$ with $\hat{P}$ as transition matrix is shown in Figure 1(c). The values on the edges denote the transition probabilities. Let $\mathbf{u}$=$(0, 0, 1, 0)^\top$. To compute $\hat{P}\mathbf{u}$ using Monte Carlo, we can place R' walkers on node 3 of $\hat{G}$ and push them to walk one step according to the probabilities randomly. Ideally, there will be $\frac{1}{3}R'$, $\frac{2}{3}R'$ walkers on nodes 1 and 2, respectively. Hence, the distribution of walkers on the nodes is $(\frac{1}{3}, \frac{2}{3}, 0, 0)^\top$, which is exactly $\hat{P}\mathbf{u}$.

After $\hat{P}\mathbf{u}$ is simulated, we can compute $P^\top \mathbf{v}$ easily, Then, $P^{\top t}\mathbf{v} = P^{\top (t-1)}(P^\top \mathbf{v})$ can be computed by repeating the same procedure. Finally, we obtain the single-source SimRank scores $S\mathbf{e}_i$. It should be mentioned that the matrices F and $\hat{P}$ need to be computed only once, and can be done offline in the preprocessing stage.

Complexity. The time complexity for computing F is $O(m)$, since for each diagonal entry, we only need to go through all the non-zero entries in one column of $P^\top$. The time complexity for computing $\hat{P}$ is also $O(m)$, since each entry can be computed in $O(1)$. The space complexities for computing F and $\hat{P}$ are $O(n)$ and $O(m)$ respectively.

In the online MCSS computation, all the $P\mathbf{e}_i, \cdots, P^T\mathbf{e}_i$ can be computed by pushing R' walkers to walk T steps on $G^\top$, and thus its time complexity is $O(TR')$. In computing $\hat{P}\mathbf{u}$, placing walkers can be done in $O(R')$ time. To move a step for each walker, we need to select an out-neighbor in $\hat{G}$ randomly according to $\hat{P}$, which can be done in $O(\log d)$ time by binary search [7], where d is the out-degree of the current node (in the original graph G). So the time complexity of moving a step for all the walkers is $O(R' \log d)$ (for an abuse of notion, d here denotes the average out-degree in G). The counting takes $O(R')$ time. So the overall time complexity of computing $\hat{P}\mathbf{u}$ is $O(R' \log d)$.

For each item $c^t P^{\top t} D P^t \mathbf{e}_i$, we need to repeat the random walk procedure $t \leq T$ times and thus the time complexity of computing $c^t P^{\top t} D P^t \mathbf{e}_i$ is $O(TR' \log d)$. Since there are T such items, the overall time complexity of computing MCSS is $O(T^2 R' \log d)$. The space complexity is $O(T^2 R' + m)$.

Compared with the naive algorithm with time complexity $O(T^2 m)$, our algorithm is much faster since $R' \log d \ll m$. Also, since R' is constant and independent of the graph size, our algorithm is scalable for large graphs.

4.3.3 The MCAP Algorithm

The all-pair SimRank computation can be done by calling MCSS algorithm for each node (We call it MCAP). Thus, the overall time complexity is $O(nT^2 R' \log d)$, and the space complexity is the same as in MCSS, which is $O(T^2 R' + m)$.

Our algorithms for MCSP, MCSS, and MCAP are based on random walks. In the distributed environment, the random walk simulation can be executed in parallel. Thus, all our SimRank computation algorithms can effectively harness the power of the distributed cluster for large graphs.

5. IMPLEMENTATION

In this section, we describe the implementation of our algorithms on top of Spark [1]. As explained in Section 4, in the computation of the diagonal correction matrix D and the SimRank scores, we mainly use the Monte Carlo method which needs to simulate random walks frequently. To sample a random walk on the underlying graph, a walker repeatedly walks from one node to its neighbors. To achieve the best possible performance, we consider two different cases when implementing our approach on Spark.

- **Case 1:** The input graph G is small enough to be kept in the main memory of a single machine.

- **Case 2:** G is large and must be stored as an RDD.

In **Case 1**, the random walk sampling can be done on a single machine efficiently, since all the nodes can be accessed instantly from main memory. However, the scale of input graphs is limited by the memory of a single machine. In **Case 2**, since G is stored in an RDD distributively, we can handle much larger graphs. This, however, incurs much network communication overhead, since the involved random walk sampling requires to access the neighbors of each vertex repeatedly, which can be resided on multiple machines. To reduce the communication overhead, we use transformations in Spark to simulate thousands or millions of random walks simultaneously. Next, we introduce the implementation details of both cases.

5.1 Case 1: graph stored in each machine

When the input graph can fit in the memory of a single machine, the implementation of our approach is straightforward. Here we elaborate on how to parallelize our approach using Spark so as to accelerate the computation of D.

Computing D. As shown in Algorithm 1, suppose we already load the input graph and store G as an adjacent list in the memory of the driver program. We first initialize $\mathbf{x}$ with $\mathbf{1}$ (line 2). Note that $\mathbf{x}$ represents the elements on the main diagonal of D. Next, we broadcast G to the memory of each machine (line 3).

After broadcasting, we invoke the iterative processing of the Jacobi algorithm (lines 4-13). In each iteration, we first broadcast the latest values in $\mathbf{x}$ to each machine. Here, $\mathbf{x}$ is of length n and we assume it can also be cached in the memory of each machine. Then we split the set of nodes from 1 to n into intervals of length b and compute $\mathbf{a}_i$ and $\mathbf{x}_i$ in each interval as discussed in Section 4.2.4. For each interval $[p, q]$, we construct an RDD, namely *idxRDD*, which represents the indices from p to q. After that, we use a `map` function to compute a batch of $\mathbf{a}_i$ values in parallel (line 10). Here, "$i \Rightarrow \mathbf{a}_i$" is an anonymous function, in which given i, it computes the corresponding $\mathbf{a}_i$. The procedure of computing $\mathbf{a}_i$ using random walks is executed in a single machine and the details are already described in Section 4.2.3. We store a group of $\mathbf{a}_i$ values in *aiRDD*. According to the Jacobi algorithm, we update the i-th entry of $\mathbf{x}$ as $(1 - \sum_{j \neq i} x_i(\mathbf{a}_i)_j)/(\mathbf{a}_i)_i$ and store it in *xRDD* (line 11). At last, we collect all the updated entries of $\mathbf{x}$ (line 12).

MCSP and MCSS. When G can be stored on a single machine, the implementation of MCSP and MCSS is straightforward and we omit the details here. Note that we can also compute the SimRank scores of multiple node pairs or sources in parallel where each machine executes an instance of MCSP or MCSS algorithm simultaneously.

Algorithm 1 Case 1: Computing D

```
 1: procedure COMPUTINGD($G$)
 2:     $\mathbf{x} \leftarrow (1, 1, \cdots, 1)$;
 3:     broadcast $G$;
 4:     for $l = 1, 2, \ldots, L$ do
 5:         broadcast $\mathbf{x}$;
 6:         $p \leftarrow 1$;
 7:         for $p \leq n$ do
 8:             $q \leftarrow \min(p + b - 1, n)$;
 9:             $idxRDD \leftarrow p, p + 1, \ldots, q$;
10:             $aiRDD \leftarrow idxRDD$.map($i \Rightarrow \mathbf{a}_i$);
11:             $xRDD \leftarrow aiRDD$.map($\mathbf{a}_i \Rightarrow \mathbf{x}_i$);
12:             update $\mathbf{x}$ using $xRDD$.collect();
13:             $p \leftarrow p + b$;
14:     return $\mathbf{x}$.
```

5.2 Case 2: graph stored in an RDD

When the input graph must be stored in an RDD, simulating each random walk independently becomes too costly. Instead, we simulate multiple random walks by utilizing high-level operators provided by Spark.

Computing D. Algorithm 2 shows the details of our procedure for computing D when G is stored as an RDD. Similar to Case 1, we first initialize $\mathbf{x}$ (line 2). Then we use the input graph G to construct an RDD, namely *inedgesRDD* (line 3). Each record in *inedgesRDD* is a pair of a node ID and an array of all in-neighbors of that node. We call `persist` on *inedgesRDD* since it will be reused during the random walk sampling (line 4).

In each iteration of the Jacobi algorithm (lines 5-16), we first broadcast the latest $\mathbf{x}$ values to each machine. Then, similar to Case 1, we split all the n nodes into intervals of length b. For each interval $[p, q]$, we construct *walkerRDD* with R walkers for each node. Each record in *walkerRDD* is a pair of a node ID and an array of all walkers on that node. Each walker is represented as a pair of the current node it resides and the sequence of visited nodes so far during the random walk sampling. All walkers will walk T steps in total. Each step is done with two transformation operations, `leftOuterJoin` and `map` (line 12). After the final step, we compute all $\mathbf{a}_i$ values by counting the landing positions of all walkers (line 13) and update *xRDD* which includes $\mathbf{x}_i$ for $i = p, \ldots, q$ accordingly (line 14). At last, we update $\mathbf{x}$ using the values in *xRDD* (line 15).

Choice of b. In the computation of D of both cases (see Algorithm 1 and Algorithm 2), the parameter b represents the number of $\mathbf{a}_i$ that we store in *aiRDD*. To save the network communication cost used to compute *aiRDD* each time, we usually choose the largest b while *aiRDD* can still be cached in the distributed memory. A special case is when b equals to n. In this case, we only need to compute A once, and then we can cache *aiRDD* during the whole computation of D.

Algorithm 2 Case 2: Computing D

```
 1: procedure COMPUTINGD($G$)
 2:     $\mathbf{x} \leftarrow (1, 1, \cdots, 1)$;
 3:     init $inedgesRDD$ with $G$;
 4:     $inedgesRDD$.persist();
 5:     for $l = 1, 2, \ldots, L$ do
 6:         broadcast $\mathbf{x}$;
 7:         $p \leftarrow 1$;
 8:         for $p \leq n$ do
 9:             $q \leftarrow \min(p + b - 1, n)$;
10:             init $walkerRDD$ using interval $[p, q]$;
11:             for $t = 1, 2, \ldots, T$ do
12:                 $walkerRDD \leftarrow walkerRDD$.
                        leftOuterJoin($inedgesRDD$).map();
13:             $aiRDD \leftarrow walkerRDD$.map().reduceByKey();
14:             $xRDD \leftarrow aiRDD$.map($\mathbf{a}_i \Rightarrow \mathbf{x}_i$);
15:             update $\mathbf{x}$ using $xRDD$.collect();
16:             $p \leftarrow p + b$;
17:     return $\mathbf{x}$.
```

MCSP. The computation of the single-pair SimRank score between nodes i and j using MCSP is shown in Algorithm 3. The idea is to approximate $P^t\mathbf{e}_i$ and $P^t\mathbf{e}_j$ for $t = 0, 1, \ldots, T$ using the Monte Carlo method, where the simulation is performed by Procedure WALKIN. In WALKIN, we start R' random walks from node k to approximate $P^t\mathbf{e}_k$. Here, PE is an $n \times (T + 1)$ matrix and the t-th column $PE(t)$ represents $P^t\mathbf{e}_k$. Since $PE(t)$ is a sparse vector which includes R nonzero elements at most, we assume PE can be stored in the memory of the driver program. *walkerMap* is a map in the driver program which stores the current positions of the R' walkers. For each key-value pair in *walkerMap*, the key is a node ID and the value is the number of walkers on that node. So *walkerMap* is initialized with a map that represents all R' walkers initially at node k (line 9). Then, each walker moves T steps by selecting a random in-neighbor at each step. We use a transformation, `flatMap`, to simulate each step, and then store the new landing nodes in *landRDD* (line 12). Finally, we use all those landing positions to update *walkerMap* and $PE(t)$ (lines 13-14).

Algorithm 3 Case 2: Computing MCSP

```
 1: procedure computingMCSP(inedgesRDD, D, i, j)
 2:     init $s \leftarrow 0$;
 3:     $X \leftarrow$walkIn(inedgesRDD, i);
 4:     $Y \leftarrow$walkIn(inedgesRDD, j);
 5:     for $t = 0, 1, \ldots, T$ do
 6:         $s \leftarrow s + c^t X(t)^T D Y(t)$;
 7:     return $s$.
 8: procedure walkIn(inedgesRDD, k)
 9:     init $PE \leftarrow \mathbf{0}$, $walkerMap \leftarrow \{(k, R')\}$;
10:     for $t = 0, 1, \ldots, T$ do
11:         broadcast walkerMap;
12:         landRDD ← inedgesRDD.flatMap(walkerMap);
13:         walkerMap ← landRDD.collect();
14:         compute $PE(t)$ using walkerMap;
15:     return $PE$.
```

Algorithm 4 Case 2: Computing MCSS

```
 1: procedure computingMCSS(outedgesRDD, D, F, i)
 2:     $\boldsymbol{\gamma} = \mathbf{0}$;
 3:     $X \leftarrow$walkIn(inedgesRDD, i);
 4:     for $t = 0, 1, \ldots, T$ do
 5:         $\mathbf{v} \leftarrow D \cdot X(t)$                    ▷ $\mathbf{v} = DP^t\mathbf{e}_i$.
 6:         for $s = 1, 2, \ldots, t$ do
 7:             $\mathbf{u} \leftarrow F\mathbf{v}$;
 8:             $w \leftarrow \sum_{i=1}^{n} u(i)$;
 9:             $\mathbf{u} \leftarrow w^{-1} \cdot \mathbf{u}$;              ▷ Normalize u.
10:             $\boldsymbol{\phi} \leftarrow$ walkOut(outedgesRDD, u);
11:             $\mathbf{v} \leftarrow w \cdot \boldsymbol{\phi}$;
12:         $\boldsymbol{\gamma} \leftarrow \boldsymbol{\gamma} + c^t\mathbf{v}$;
13:     return $\boldsymbol{\gamma}$.
14: procedure walkOut(outedgesRDD, u)
15:     init walkerMap with u;
16:     broadcast walkerMap;
17:     landRDD ← outedgesRDD.flatMap(walkerMap);
18:     walkerMap ← landRDD.collect();
19:     compute $\boldsymbol{\phi}$ using walkerMap;
20:     return $\boldsymbol{\phi}$.
```

MCSS. The computation of the single-source SimRank scores for node i using MCSS is shown in Algorithm 4. The input parameter F is a diagonal matrix of $n \times n$, so we assume F can be pre-computed and stored in the driver program. Another input parameter *outedgesRDD* is an RDD that represents the matrix $\hat{P}$. The meanings of F and $\hat{P}$ have already described in Section 4.3.2. To represent $\hat{P}$ in an RDD, we define each record in *outedgesRDD* as a pair of node ID i and a list of pairs, where each pair equals to $(j, \frac{P_{ji}}{F_{jj}})$ which corresponds to an out-neighbor j of i. Then, we call Procedure walkIn in Algorithm 3 to compute $P^t\mathbf{e}_i$ for $t = 0, 1, \ldots, T$ (line 3). In line 4, we enter the for-loop to compute $c^t P^{\top t} D P^t \mathbf{e}_i$ for $t = 1, 2, \ldots, T$ as described in Section 4.3.2. Note that at the end of the for-loop, $\mathbf{v}$ equals $P^{\top t} D P^t \mathbf{e}_i$ (line 12). Most computation in the for-loop is done in the driver program, except for line 10, where we call Procedure walkOut to compute $\hat{P}\mathbf{u}$ using the Monte Carlo method. The implementation of Procedure walkOut is similar to Procedure walkIn. First, we initialize the positions of walkers, *walkerMap*, using the distribution $\mathbf{u}$ (line 15). Then, each walker moves one step to an out-neighbor according to the probabilities stored in *outedgesRDD* randomly (line 17). Finally, we obtain $\boldsymbol{\phi} = \hat{P}\mathbf{u}$ from the new landing positions (line 19).

As described above, our approach is highly parallelizable, where the key component is in simulating a large number of random walks on a distributed computing platform ("cloud") in parallel. In this sense, we called our approach CloudWalker.

6. EXPERIMENTAL RESULTS

In this section, we evaluate CloudWalker experimentally. We first describe the experimental setup in Section 6.1. Then we report the results in terms of effectiveness and efficiency in Section 6.2 and Section 6.3, respectively. We show the speedup of CloudWalker in Section 6.4. The comparison with existing SimRank algorithms is presented in Section 6.5.

6.1 Setup

Cluster. We perform all experiments on a cluster of 10 machines, each with two eight-core Intel Xeon E5-2650 2.0 GHz processors, 377 GB RAM, and 20 TB hard disk, running Ubuntu 14.04. All machines are connected via a Gigabit network. We deploy Spark of version 1.2.0 on each machine in the cluster, and configure one machine as the master node and the other nine machines as slaves. We also install HDFS with Hadoop of version 2.2.0 in the same cluster, which is only used to the input graphs. The other details of the configuration are as follows: 1) Spark consumes 360 GB RAM at most on each node; 2) the total number of cores used by Spark is 160; and 3) the data replication factor of HDFS is 3. All our algorithms are implemented in the Scala programming language.

Datasets. We tested on 6 benchmark datasets of various scales[5]. The description of each dataset, including its size in disk, is shown in Table 1. In each dataset file, each line contains a node ID and the IDs of its in-neighbors. All datasets are uploaded into the HDFS before computation.

Table 1: Datasets used in our experiments

Dataset	Nodes	Edges	Size
wiki-vote [18]	7,115	103,689	476.76KB
ca-hepth [18]	9,877	25,998	287.18KB
wiki-talk [18]	2,394,385	5,021,410	45.62MB
twitter-2010 [17]	41,652,230	1,468,365,182	11.43GB
uk-union [3, 2]	131,814,559	5,507,679,822	48.31GB
clue-web [3, 2]	978,408,098	42,574,107,469	401.12GB

Parameters. The meanings and default values of the parameters of our algorithms are listed in Table 2, where c and T are set following the previous study [23].

Accuracy. The accuracy is evaluated by comparison to the *exact* SimRank scores (Section 3). Specifically, we compute the following *mean error*:

$$\mathrm{ME} = \frac{1}{n^2} \sum_{i=1}^{n} \sum_{j=1}^{n} \left| s(i,j) - s'(i,j) \right|, \qquad (26)$$

[5] wiki-vote, ca-hepth, and wiki-talk are downloaded from `http://snap.standford.edu/data/index.html`, and twitter-2010, uk-union, and clue-web are obtained from `http://law.di.unimit.it/datasets.php`.

Table 2: Default values of parameters

Parameter	Value	Meaning
c	0.6	decay factor of SimRank
T	10	# of walk steps
L	3	# of iterations in Jacobi algorithm
R	100	# of walkers in simulating $\mathbf{a}_i$
R'	10,000	# of walkers in MCSP and MCSS

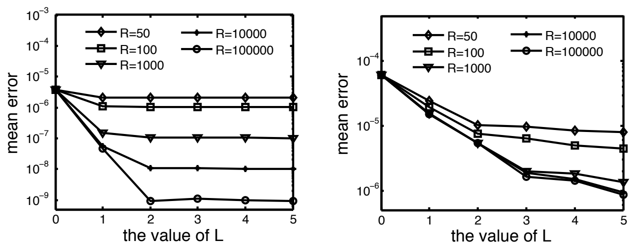
where $s(i,j)$ is the exact SimRank score and $s'(i,j)$ is the one from CloudWalker. Apparently, the lower the mean error, the higher the accuracy.

6.2 Effectiveness

We evaluate the effectiveness of CloudWalker by varying the values of R, L, and R'. Because computing and storing the exact SimRank similarity matrix is impractical for large graphs, we use wiki-vote and ca-hepth for this test.

6.2.1 Effect of R and L on Computing D

In this subsection, we show how to choose appropriate values for R and L when computing D. In the experiment, we first use our proposed approach to compute D, and then compute the estimated SimRank values using Eq. (8). Note that in this experiment, given D, we do not use the Monte Carlo method in computing the SimRank scores and thus do not need to consider R' for now. We vary the values of R and L, and report the mean errors in Figure 2.



(a) wiki-vote

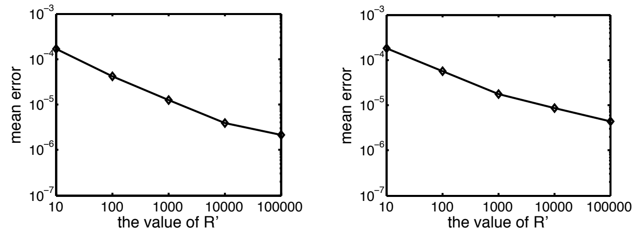
(b) ca-hepth

Figure 2: Effect of L and R

From Figures 2(a) and 2(b), we can observe that, when the value of R increases, the mean error decreases. To achieve ME around $10^{-5} \sim 10^{-6}$, which is the same level as previous work [23, 29], it suffices to set $R = 100$ in simulating the matrix A using Monte Carlo. Besides, it is interesting to notice that our algorithm converges very fast in just 3 to 4 Jacobi iterations.

6.2.2 Effect of R' on Computing MCSP and MCSS

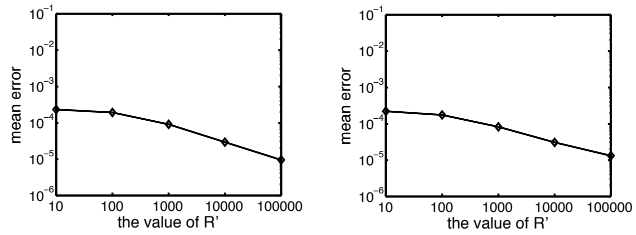
In this experiment, we compute the SimRank scores using our proposed algorithms, MCSP and MCSS. We report the mean error for each algorithm. D is still computed using our proposed approach.



(a) wiki-vote

(b) ca-hepth

Figure 3: Effect of R' on computing MCSP



(a) wiki-vote

(b) ca-hepth

Figure 4: Effect of R' on computing MCSS

The results are shown in Figures 3 and 4. We can observe that the accuracy improves as the number of random walkers R' increases for both MCSP and MCSS. To achieve ME around $10^{-5} \sim 10^{-6}$, MCSP needs R'=1,000 while MCSS needs R'=100,000. The reasons are as follows: 1) in MCSP, T walks can finish all $P\mathbf{e}_i, \ldots, P^T\mathbf{e}_i$, thanks to the nested structure of the problems; 2) in MCSS, it takes $O(T^2)$ walks for estimating $P^{\top t}\mathbf{v}_t$, $t = 1, \ldots, T$, as the start distribution $\mathbf{v}_t := DP^t\mathbf{e}_i$ (upon normalization) changes with t. Note that MCSS computes n scores while MCSP just computes one.

6.3 Efficiency

To measure the efficiency of CloudWalker, we report the running time and the network communication cost, measured by the amount of data transmitted via the network during the operations of broadcasting and shuffling on Spark.

6.3.1 Case 1: graph stored in each machine

All the datasets, except clue-web, can be cached in the memory of a single machine in our cluster, and thus can be handled by the algorithms designed for case 1 (Section 5.1). The clue-web dataset will be considered in case 2.

Let us first focus on the running time of computing D as shown in Table 3. As discussed in Section 5.2, if the matrix A can be kept in the distributed memory, then we only need to compute A for once, and we use "D : $A(1)$" to denote this scenario in the following evaluation. To evaluate our approach under limited memory resource, we also consider the scenario when b is less than n and we have to compute A repeatedly in each iteration of the Jacobi method. Since A is computed for $L = 3$ times, we use "D : $A(3)$" to denote this scenario. In either scenario, as it can be observed, the overall time of computing D on the first two small datasets does not change much. The reason is that, when the datasets are small, the overhead of Spark in cluster management and scheduling tends to be significantly larger than the computation cost. The effect of this overhead becomes smaller as the data size becomes larger. This can be seen from the results on the rest 3 datasets, where the running time of "D : $A(3)$" is about twice of that of "D : $A(1)$".

Although having to compute A for 3 time, the overall running time of "D : $A(3)$" is less than 3 times of "D : $A(1)$". The reason is that since matrix P is broadcast to each machine before computing A, the random walk simulation and updating elements of D can be done in each machine locally,

Table 3: Results of Case 1

Dataset	D:A(1)	D:A(3)	R' of MCSP			R' of MCSS		
			1,000	10,000	100,000	1,000	10,000	100,000
wiki-vote	7s	8s	0.0008s	0.0040s	0.0541s	0.0056s	0.0420s	0.2997s
ca-hepth	8s	9s	0.0042s	0.0300s	0.2749s	0.0161s	0.1225s	0.8396s
wiki-talk	59s	116s	0.0053s	0.0461s	0.4302s	0.0267s	0.1799s	1.977s
twitter-2010	975s	1847s	0.0059s	0.0490s	0.5769s	0.1116s	0.2810s	3.253s
uk-union	3323s	6034s	0.0036s	0.0247s	0.2730s	0.2755s	0.2914s	2.664s

and therefore there is no shuffling cost. So the network communication cost for broadcasting P and D dominates the total running time (Algorithm 1).

As P can be kept in the memory of one machine, MCSP and MCSS can be computed in one machine directly with a given D. The experimental results of MCSP and MCSS on a single machine with $R' = 10^3$, 10^4, and 10^5 are also reported in Table 3. The results are the average of 200 trials. We can draw two observations: 1) for a given R', the running time of MCSS is often 10+ times more than that of MCSP, which should not be surprising as the time complexity of MCSS is higher than that of MCSP, in T; and 2) in MCSP and MCSS, when the value of R' increases 10 times, the running time also increases 10 times, which is expected as the time complexity of each algorithm is linear in R'.

A final note is that we can easily run multiple MCSP and MCSS instances on multiple machines in parallel.

Remark. From the above experimental results, we can conclude that, if each machine can hold G and the remaining memory of the cluster can hold A, we can compute A only once for the best efficiency.

6.3.2 Case 2: graph stored in an RDD

In case 2, the graph is stored as an RDD in the cluster-wide memory, where each machine only keeps one portion of the graph. In our experiment, the clue-web, with size over 400GB, is larger than the memory of each machine and is required to be stored as an RDD. The experimental results of all datasets on case 2 are reported in Table 4, where "network" denotes the total amount of data shuffled via the network.

Let us first examine the results of computing D. For each dataset, we can observe that the running time of computing D is usually 2∼4 times as that in case 1. This is mainly due to the much more frequent network communications in case 2 than in case 1. Recall that we use random walk sampling to approximate $\mathbf{a}_i$. As each machine only keeps a small portion of the graph, a walker can easily walk to a node stored in another machine which incurs network communication. In contrast, in case 1, each machine holds a copy of the graph, and so a walker can access its in-neighbors locally. Like case 1, we also compute D in two different scenarios: "$D : A(1)$" and "$D : A(3)$". Here, the running time of "$D : A(3)$" is around 3 times as that of "$D : A(1)$", because the time used to simulate random walks on a distributed graph dominates the total running time due to more network communication.

The running time of MCSP and MCSS in case 2 is reported in the right-hand side of Table 4. The results are the average of 200 trials. We can observe that, in all cases, the running time grows as the size of dataset increases. However, it does not grow linearly with the size of dataset, but much slower. For example, for a single-source SimRank query (R'=10,000), it takes around 188 seconds on clue-web and 12 seconds on wiki-talk, even though the clue-web (43B edges) is almost 10,000 times larger than wiki-talk (5M edges). This is thanks to the fact that, in principle, the time complexity, in either MCSP or MCSS, is independent of the graph size (Section 4.3).

For a specific dataset, the running time for a single-pair or single-source SimRank computation does not change linearly with the value of R', the number of walkers. Let us take the results of MCSP on clue-web as an example. When R'=1,000, the running time is 62.85 seconds. After increasing R' 10 times to 10,000, the running time increases less than 2 seconds. This can be explained as follows. In Algorithm 3 and Algorithm 4, we use a hash table, *walkerMap*, to maintain all walkers, in which each element is a key-value pair, where the key is a node ID and the value is the number of walkers on that node. In each step of the random walk sampling, we need to broadcast *walkerMap* which incurs a lot network communication and dominates the computation time. However, in power-law graphs, the landing positions of walkers tend to concentrate on a few popular nodes. So when we increase R' 10 times, the size of *walkerMap* will not increase much. This result suggests that, to achieve high accuracy, we can increase the value of R' without sacrificing much efficiency.

We also observe that the speed of MCSP and MCSS computation in case 2 is slower than that in case 1. This is because more network cost is needed in case 2; and as the graph is organized as an RDD, we cannot access a specific node randomly as in case 1, even with built-in index.

Remark. From the above experimental results, we can conclude that, if the memory of each machine is enough to hold the graphs G and $\hat{G}$, the algorithms for case 1 are more efficient than those of case 2, but otherwise the latter are a nice choice due to its great scalability.

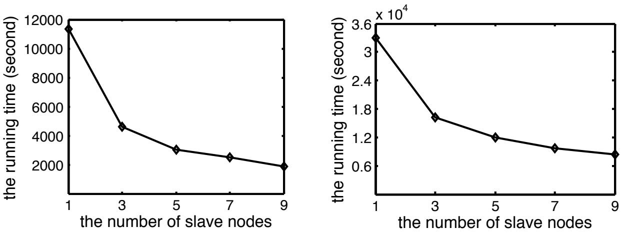
6.4 Speedup

In this experiment, we test our algorithms with different numbers of slave nodes for computing D. We look into both cases 1 and 2, where the matrix A is repeatedly computed for three times ("$D : A(3)$"). The experimental results on the twitter-2010 are depicted in Figure 5.

The overall running time decreases as the number of slave nodes increases, but the speedup is not linear. For example, in case 1, when there is only 1 slave node, it takes 11,325 seconds, which is only 6.13 times larger than that (1,847 seconds) of the case when the number of slave nodes is 9. Similar results can be observed from case 2. This is mainly because the network communication cost increases with more slave nodes. With more slave nodes, the running time on managing the cluster (e.g., task scheduling, heart-beating detection, etc.) also increases.

Table 4: Results of Case 2

Dataset	D:A(1)		D:A(3)		R' of MCSP			R' of MCSS		
	time	network	time	network	1,000	10,000	100,000	1,000	10,000	100,000
wiki-vote	12s	371MB	50s	554MB	2.67s	2.69s	2.84s	2.50s	2.99s	3.73s
ca-hepth	13s	670.2MB	59s	1901MB	6.43s	6.61s	7.27s	11.68s	11.98s	13.09s
wiki-talk	236s	108.7GB	620s	345.8GB	7.85s	8.44s	13.74s	12.97s	13.85s	22.43s
twitter-2010	2967s	2.1TB	8424s	6.1TB	11.2s	11.75s	20.63s	21.15s	22.32s	64.1s
uk-union	–	–	6.4h	16.1TB	12.94s	13.07s	14.21s	25.79s	27.19s	35.31s
clue-web	–	–	110.2h	139.2TB	62.85s	64.01s	66.29s	187.7s	188.1s	191.9s



(a) Case 1

(b) Case 2

Figure 5: Speedup of computing D

6.5 Comparison with Other Algorithms

In this section, we compare CloudWalker with other state-of-the-art algorithms, including **IteMat**, **IteMat-ϵ**, **FMT**, **Fast-IteMat**, and **LIN**, which are described below.

- IteMat is a parallel algorithm for SimRank, implemented by us on Spark, following the iterative matrix multiplication procedure (Section 3). In IteMat, P is broadcast to each machine, and S is constructed as an RDD and initialized as $S = I$.
- IteMat-ϵ is a variant of IteMat, except that the entries below the threshold ϵ in S are pruned during each iteration ($\epsilon = 1.0^{-4}$).
- Fast-IteMat [29] uses the fast multiplication technique, the Coppersmith-Winograd algorithm [6], to accelerate the iterations in IteMat.
- FMT [10] follows the random surfer model of SimRank. It first simulates and stores the random paths for each node, and at query time computes the similarities from the random paths.
- LIN [23] is a recent state-of-the-art algorithm for SimRank computation. Like CloudWalker, it first estimates the diagonal correction matrix D, and at query time computes the similarities based on D. One key limitation of LIN is that its parallelism is quite limited due to its interwinding dependency in computation (Section 4.1).

The results of FMT [10], LIN [23], and Fast-IteMat [29] are obtained from [23]. "N/A" means the experiment was not conducted and "–" means the experiment failed. Since they did not evaluate on uk-union in [23], we use their results on the uk-2007-05 dataset instead. Note that uk-2007-05 is a subgraph of the uk-union dataset with 106 million nodes and 3.7 billion edges. All of their experiments were implemented in C++ and were conducted on a single machine with an Intel Xeon E5-2690 2.90GHz CPU and 256GB memory running Ubuntu 12.04. For CloudWalker, we report case 1 on the first 5 datasets and case 2 on clue-web. We first compute D and then compute the single-pair, single-source, and all-pair SimRank scores, using MCSP, MCSS, and MCAP, respectively ($R' = 10,000$).

The results on preprocessing, single-pair and single-source computation are reported in Table 5. The preprocessing time cost of LIN is the lowest on the two small datasets, wiki-vote and ca-hepth. However, for the preprocessing on large datasets like twitter-2010, CloudWalker is around 15 times faster than LIN. This is mainly because CloudWalker can harness the power of the parallel computation. For single-pair and single-source computation, the running time of FMT and LIN grows as the size of dataset increases, while the time cost of CloudWalker increases only slightly thanks to the fact that the time complexity of either MCSP or MCSS in CloudWalker is independent of the graph size. It is worth mentioning that no results on clue-web can be reported for FMT and LIN. In contrast, CloudWalker completes the preprocessing, the single-pair and the single-source computation in 110 hours, 64 seconds, and 188 seconds, respectively. This shows that CloudWalker is much more scalable than FMT and LIN.

The results on all-pair computation are reported in Table 6. We can see that while all algorithms can easily deal with the two small datasets, only CloudWalker can finish the computation on wiki-talk and twitter-2010 in reasonable time. This again confirms the superior scalability of CloudWalker. While in principle CloudWalker can be applied to compute all-pair SimRank scores for the clue-web dataset, it would take around 5800 years.

7. CONCLUSIONS

We have proposed and implemented a highly parallelizable approach, CloudWalker, for the SimRank computation. It consists of offline indexing and online querying. The indexing takes linear time and results in just a length-n vector. The online phase responds to the fundamental single-pair query and single-source query in constant time, and all-pair query in linear time. The space complexity of our solution is linear in the number of edges. Extensive experimental results show that it is orders of magnitude more efficient and scalable than existing solutions for large-scale problems.

Acknowledgments

Reynold Cheng and Yixiang Fang were supported by RGC (Project HKU 17205115). We thank the reviewers for their comments.

Table 5: Comparison on preprocessing, single-pair and single-source SimRank computation

Dataset	FMT			LIN			CloudWalker		
	Prep.	**SinglePair**	**SingleSrc.**	**Prep.**	**SinglePair**	**SingleSrc.**	**Prep.**	**SinglePair**	**SingleSrc.**
wiki-vote	43.4s	30.4ms	42.5ms	187ms	0.613ms	5.26ms	7s	4ms	42ms
ca-hepth	205s	61.2ms	262ms	698ms	0.493ms	3.24ms	8s	30ms	122.5ms
wiki-talk	N/A	N/A	N/A	N/A	N/A	N/A	59s	46ms	179.9ms
twitter-2010	–	–	–	14376s	3.17s	11.9s	975s	49ms	281ms
uk-union	–	–	–	8291s	9.42s	21.7s	3323s	24.7ms	291.4ms
clue-web	–	–	–	–	–	–	110.2h	64.01s	188.1s

Table 6: Comparison on all-pair SimRank computation

Dataset	IteMat	IteMat-ϵ	Fast-IteMat	LIN	CloudWalker
wiki-vote	95s	79s	8.74s	37.4s	22s
ca-hepth	784s	216s	23.3s	39s	93s
wiki-talk	–	–	N/A	N/A	7.03h
twitter-2010	–	–	–	–	146.9h

8. REFERENCES

[1] Apache Spark - Lightning-Fast Cluster Computing. `http://spark.apache.org/`.

[2] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW*, pages 1–13, 2011.

[3] P. Boldi and S. Vigna. The WebGraph Framework I : Compression Techniques. In *WWW*, 2004.

[4] L. Cao, B. Cho, H. D. Kim, Z. Li, M.-H. Tsai, and I. Gupta. Delta-SimRank computing on MapReduce. In *BigMine*, pages 28–35, 2012.

[5] J. Cheng, Q. Liu, Z. Li, W. Fan, J. C. Lui, and C. He. VENUS: Vertex-Centric Streamlined Graph Computation on a Single PC. In *ICDE*, 2015.

[6] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC*, 1987.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[8] J. Dean and M. R. Henzinger. Finding related pages in the world wide web. *Computer networks*, 31(11):1467–1479, 1999.

[9] P. G. Doyle and J. L. Snell. Random walks and electric networks. *AMC*, 10:12, 1984.

[10] D. Fogaras and B. Rácz. Scaling link-based similarity search. In *WWW*, pages 641–650, 2005.

[11] F. Fouss, A. Pirotte, J.-M. Renders, and M. Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *TKDE*, 19(3):355–369, 2007.

[12] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[13] G. He, H. Feng, C. Li, and H. Chen. Parallel SimRank computation on large graphs with iterative aggregation. In *KDD*, 2010.

[14] M. Jamali and M. Ester. Trustwalker: a random walk model for combining trust-based and item-based recommendation. In *KDD*, 2009.

[15] G. Jeh and J. Widom. SimRank: a measure of structural-context similarity. In *KDD*, 2002.

[16] M. Kusumoto, T. Maehara, and K. Kawarabayashi. Scalable similarity search for SimRank. In *SIGMOD*, 2014.

[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter , a Social Network or a News Media? In *WWW*, 2010.

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

[19] C. Li, J. Han, G. He, X. Jin, Y. Sun, Y. Yu, and T. Wu. Fast computation of SimRank for static and dynamic information networks. In *EDBT*, 2010.

[20] L. Li, C. Li, H. Chen, and X. Du. Mapreduce-based SimRank computation and its application in social recommender system. In *BigData Congress*, 2013.

[21] P. Li, H. Liu, J. X. Yu, J. He, and X. Du. Fast single-pair SimRank computation. In *SDM*, 2010.

[22] D. Lizorkin, P. Velikhov, M. Grinev, and D. Turdakov. Accuracy estimate and optimization techniques for SimRank computation. *VLDBJ*, 19(1):45–66, 2010.

[23] T. Maehara, M. Kusumoto, and K. Kawarabayashi. Efficient SimRank computation via linearization. *CoRR*, abs/1411.7228, 2014.

[24] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, 1999.

[25] L. Sun, R. Cheng, X. Li, D. Cheung, and J. Han. On link-based similarity join. *VLDB*, 4(11):714–725, 2011.

[26] X.-M. Wu, Z. Li, and S.-F. Chang. New insights into laplacian similarity search. In *CVPR*, 2012.

[27] X.-M. Wu, Z. Li, A. M. So, J. Wright, and S.-F. Chang. Learning with partially absorbing random walks. In *NIPS*, 2012.

[28] W. Yu and J. A. McCann. Efficient partial-pairs SimRank search on large networks. *VLDB*, 8(5), 2015.

[29] W. Yu, W. Zhang, X. Lin, Q. Zhang, and J. Le. A space and time efficient algorithm for SimRank computation. *WWW*, 15(3):327–353, 2012.

[30] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.