Manuscript Details

Manuscript number	JPDC_2020_158
Title	VSIM: Efficient Distributed Vertex Similarity Calculation on Big Graphs
Article type	Research Paper

Abstract

In many graph analytical applications, finding all vertex pairs whose similarity scores are above a given threshold is a fundamental operation. As the threshold decreases from 1 to 0, the number of similar vertex pairs increases drastically from millions to billions. However, existing distributed methods for the problem are only efficient under a narrow range of thresholds. To overcome the drawback, we propose a new distributed vertex similarity calculation framework VSIM that is efficient under a broad range of thresholds. VSIM generates and executes similarity calculation tasks for all vertices in parallel. Each task finds vertices similar to a given center vertex. VSIM offers two task execution modes that optimize for high and low thresholds, respectively. Each task picks the suitable mode adaptively. The experimental evaluation shows that VSIM outperforms state-of-the-art distributed methods by up to 72.0x speedup. VSIM can achieve near-linear node scalability in low-threshold and small cache scenarios.

Keywords	distributed graph processing; graph data engineering; task-parallel algorithm; vertex similarity calculation; local structural similarity score
Taxonomy	distributed processing, Parallel implementations, Parallel algorithm, Cluster computing
Manuscript category	Algorithms
Corresponding Author	Rong Gu
Corresponding Author's Institution	Nanjing University
Order of Authors	Zhaokang Wang, Shen Wang, Junhong Li, Chunfeng Yuan, Rong Gu, Yihua Huang
Suggested reviewers	Yinglong Xia, Wenguang Chen

Submission Files Included in this PDF

File Name [File Type]

cover_letter.pdf [Cover Letter]

Highlights.doc [Highlights]

vsim.pdf [Manuscript File]

author-biography.docx [Author Biography]

declaration-of-competing-interests.pdf [Conflict of Interest]

Submission Files Not Included in this PDF

File Name [File Type]

vsim-submission-20200305.zip [LaTeX Source File]

To view all the submission files, including those not included in the PDF, click on the manuscript title on your EVISE Homepage, then click 'Download zip file'.

March 5, 2020

Dear Editor,

We are pleased to submit our manuscript entitled *VSIM: Efficient Distributed Vertex Similarity Calculation on Big Graphs* for your consideration. Our manuscript is in the scope of distributed graph processing. We believe readers of *Journal of Parallel and Distributed Computing* will be very interested in our work.

The vertex similarity calculation is a fundamental operation in many graph analysis applications like social network recommendation and bioinformatics network analysis. It finds out similar vertex pairs whose similarity scores are above a given threshold. There are some distributed methods for the problem, but they optimize only for high or low thresholds. However, different analysis applications use different ranges of thresholds. None of the existing methods can cover a broad range of thresholds.

In this study, we propose a novel distributed framework *VSIM* for the problem. VSIM divides the problem on the whole graph into a group of tasks that can be executed in parallel. To execute a task, VSIM supports two execution modes. They are optimized for high and low thresholds respectively. We propose a threshold-adaptive technique to choose the proper execution mode for each task. Experimental results show that VSIM is efficient under a broad range of thresholds. It outperforms the state-of-the-art distributed methods by up to 72.0x speedup. Our research will be beneficial to real-world graph analysis applications that use vertex similarity calculation in their analysis pipelines. It may also be interesting to researchers in the field of distributed data engineering.

We confirm that this work is original. It has not been published nor has it been submitted simultaneously elsewhere. All authors have checked the manuscript and have agreed on the submission.

Thank you very much for your attention and consideration. We are looking forward to your reply.

Sincerely yours,

Rong Gu, Ph.D., Associate Researcher

Yihua Huang, Ph.D., Professor

Department of Computer Science and Technology, Nanjing University

No. 163 Xianlin Avenue, Nanjing, 210023, Jiangsu Province, China

E-mail: Rong Gu (gurong@nju.edu.cn), Yihua Huang (yhuang@nju.edu.cn)

- A task-parallel framework for the vertex similarity calculation problem
- A threshold-adaptive technique to execute a similarity calculation task
- A Spark-based implementation with three optimization techniques
- Comprehensive performance evaluation of the vertex similarity calculation strategies

VSIM: Efficient Distributed Vertex Similarity Calculation on Big Graphs

Zhaokang Wang, Shen Wang, Junhong Li, Chunfeng Yuan, Rong Gu*, Yihua Huang*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China

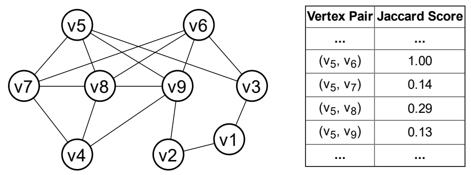
Abstract

In many graph analytical applications, finding all vertex pairs whose similarity scores are above a given threshold is a fundamental operation. As the threshold decreases from 1 to 0, the number of similar vertex pairs increases drastically from millions to billions. However, existing distributed methods for the problem are only efficient under a narrow range of thresholds. To overcome the drawback, we propose a new distributed vertex similarity calculation framework VSIM that is efficient under a broad range of thresholds. VSIM generates and executes similarity calculation tasks for all vertices in parallel. Each task finds vertices similar to a given center vertex. VSIM offers two task execution modes that optimize for high and low thresholds, respectively. Each task picks the suitable mode adaptively. The experimental evaluation shows that VSIM outperforms state-of-the-art distributed methods by up to 72.0x speedup. VSIM can achieve near-linear node scalability in low-threshold and small cache scenarios.

Keywords: distributed graph processing, graph data engineering, task-parallel algorithm, vertex similarity calculation, local structural similarity score

*Corresponding author

Email addresses: `wangzhaokang@smail.nju.edu.cn` (Zhaokang Wang), `cfyuan@nju.edu.cn` (Chunfeng Yuan), `gurong@nju.edu.cn` (Rong Gu), `yhuang@nju.edu.cn` (Yihua Huang)



Vertex Pair	Jaccard Score
...	...
(v_5, v_6)	1.00
(v_5, v_7)	0.14
(v_5, v_8)	0.29
(v_5, v_9)	0.13
...	...

Figure 1: Toy graph and its similar vertex pairs

1. Introduction

Given a graph G with the vertex set V, the problem of finding all the vertex pairs $(v_i, v_j) \in V^2$ whose similarity scores $sim(v_i, v_j)$ are above a given threshold τ is called the *vertex similarity calculation* (VSC) problem. Taking Fig. 1 as an example, if we use the Jaccard index of the adjacency sets as the similarity score and $\tau = 0.05$, the similar vertex pairs are listed on the right side.

The vertex similarity calculation is a fundamental operation in many graph analysis applications. Some social network recommender systems [1, 2] use the number of mutual friends as the similarity score to recommend potential acquaintances to users. Structural link prediction algorithms use the Jaccard similarity score to find vertex pairs that tend to establish links in the future [3]. The bioinformatic network analysis [4] uses the topological overlap score to discover modules in the network. The vertex similarity calculation is also used in many other applications such as community detection [5], graph visualization [6], graph sparsification [7], and graph clustering [8].

In these applications, the local structural vertex similarity scores [3] in Table 1 are very popular. They are defined on the adjacency sets of the two vertices. In this work, we focus on solving the VSC problem with the local structural similarity scores.

The vertex similarity calculation problem seems straightforward to solve, but it is hard to be solved efficiently, especially on big graphs. One of the challenges is that the computation complexity of the problem varies drastically under different ranges of the threshold τ. Fig. 2 shows the numbers of similar vertex pairs under different thresholds in five representative real-world graphs in Table 2. For the `bn-human` dataset, the number of similar vertex pairs is 5.1B under $\tau = 10^{-6}$ while the number is only 12.1K under $\tau = 1.0$. The computation complexity is very different for low and high

Table 1: Local structural vertex similarity scores

Similarity Score	**Definition**
Common Neighbor	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert$
Jaccard	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert / \lvert\Gamma(x) \cup \Gamma(y)\rvert$
Salton	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert / \sqrt{d_x d_y}$
Sorensen	$sim(x,y) = 2\lvert\Gamma(x) \cap \Gamma(y)\rvert / (d_x + d_y)$
Hub Promoted	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert / \min\{d_x, d_y\}$
Hub Depressed	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert / \max\{d_x, d_y\}$
Leicht-Holme-Newman Index	$sim(x,y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert / (d_x d_y)$

* $\Gamma(x)$ and d_x represent the adjacency set and the degree of the vertex x respectively.

Table 2: Statistics of typical real-world graphs

Graph G	$\lvert V(G)\rvert$	$\lvert E(G)\rvert$	Average Degree	Type
com-orkut [9]	3.1M	117.2M	76.3	social network
uk-2002 [10]	18.5M	261.8M	28.3	web
bn-human [11]	0.8M	267.8M	683	bioinformatic network
pp-miner [12]	8.3M	923.6M	223.8	bioinformatic network
friendster [9]	65.6M	1.8B	55.1	social network

thresholds.

The varying complexity demands conflicting optimization directions. For high thresholds, few vertex pairs are similar enough. An efficient method should use filtering techniques to prune the search space. More effective filtering techniques are usually more complicated and bring more overheads. Under low thresholds, many vertex pairs are similar enough. An efficient method should make overheads as low as possible. It conflicts with the optimization direction for high thresholds.

As graphs in real-world applications grow bigger and bigger, the serial methods for

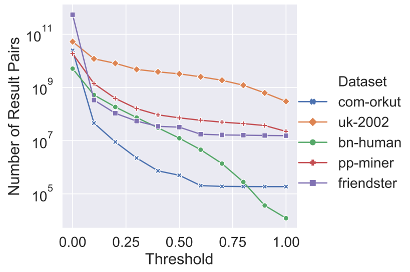


Figure 2: Number of similar vertex pairs under different thresholds. Similarity score: Jaccard.

the VSC problem cannot meet performance requirements. Many distributed methods have been proposed. We classify them as the all-pair and the filtering-based methods, based on the sensitivity of their performance to the threshold.

The performance of the all-pair methods [13, 14, 15, 16, 17] is *insensitive* to the threshold. They first find out all vertex pairs with non-zero similarity scores by counting co-occurrences of vertices in adjacency sets via MapReduce or Pregel. Then they filter out the pairs whose scores are below the threshold. Since they do not use the threshold to reduce computation costs, they are inefficient under high thresholds.

Contrarily, the performance of the filtering-based methods [18, 19, 20, 21, 22, 23] is *sensitive* to the threshold. They generate signatures for vertices based on their adjacency sets and group vertices with the same signature together via MapReduce to get candidate vertex pairs. They calculate exact similarity scores for candidate pairs and filter out dissimilar ones. The threshold is used to guide the signature generation. Higher thresholds generate fewer signatures and produce less data replication during shuffling. However, the number of generated signatures increases dramatically as the threshold decreases, causing load balance issues [24] and making the filtering-based methods inefficient under low thresholds.

In this work, we propose a new distributed framework *VSIM* for the *V*ertex *SIM*ilarity calculation problem. Different from the existing methods that are based on either the MapReduce or the Pregel model, VSIM builds upon the task-parallel model, as illustrated in Fig. 3. VSIM stores the input graph $G = (V, E)$ into a distributed key-value store. VSIM generates a task t_i for each vertex $v_i \in V$. The task t_i finds all vertices v_j similar to v_i with $sim(v_i, v_j) \geq \tau$. VSIM executes the tasks in parallel in a distributed computing platform like Apache Spark. During the task execution, the adjacency sets are queried from the key-value store on demand.

The existing methods are inefficient under either high or low thresholds. To overcome the drawback, we propose a threshold-adaptive technique to execute tasks in VSIM. The technique supports two execution modes: the *verification* mode and the *counting* mode. The verification mode finds similar vertices by using filtering techniques, suitable for high thresholds. The counting mode finds similar vertices by counting co-occurrences of vertices, suitable for low thresholds. VSIM adopts a cost-based

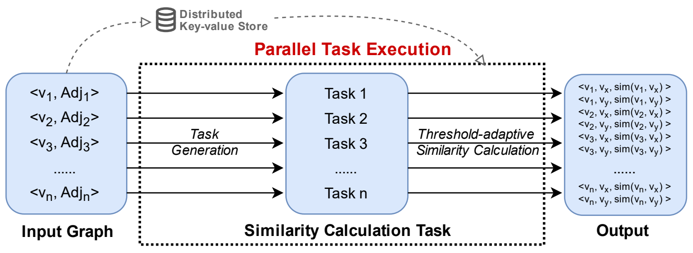


Figure 3: Workflow of VSIM

method to select the best mode for each task adaptively. In this way, VSIM takes advantage of both modes, efficient on the broad range of thresholds.

Contributions. Specifically, in this work, we make the following contributions:

1. We propose a new distributed framework VSIM for the VSC problem based on the task-parallel model and a distributed key-value store.
2. We design a threshold-adaptive technique for VSIM that is efficient under a broad range of thresholds.
3. We propose an efficient implementation of VSIM based on Spark with three optimization techniques.
4. We conduct extensive experiments on real-world graphs to evaluate the performance of VSIM. VSIM achieves up to 28.4x/72.0x speedups compared to the state-of-the-art all-pair/filtering-based method. In the experiments, VSIM is the only method that runs smoothly on all datasets and all thresholds without crash or timeout. VSIM achieves near-linear node scalability in low thresholds and small cache scenarios.

The rest of the paper is organized as follows. Section 2 defines the problem and introduces serial techniques. Section 3 surveys the related work. Section 4 introduces our VSIM framework. An efficient implementation of VSIM is given out in Section 5. Section 6 evaluates the performance of VSIM. We conclude our work in Section 7.

2. Preliminaries

We formally define the vertex similarity calculation problem in Section 2.1. Then we introduce two representative serial techniques for the problem in Section 2.2. Our proposed framework VSIM uses the two techniques in its design.

2.1. Problem Definition

In this paper, we focus on solving the VSC problem defined in Definition 1 in the distributed environment. The input of the problem is an undirected simple graph G with the vertex set V and the edge set E. Each vertex in V has a unique vertex ID. A total order $\prec$ is defined by the vertex IDs. For two vertices v_i and v_j, $v_i \prec v_j$ is and only if the ID of v_i is smaller than v_j. We say v_i and v_j are similar enough if $sim(v_i, v_j) \geq \tau$. Since the similarity scores in Table 1 are symmetric, we only consider (v_i, v_j) with $v_i \prec v_j$ in results. The frequently-used symbols are summarized in Table 3.

Definition 1. *Given an undirected simple graph $G = (V, E)$ with a total order $\prec$ defined on vertex IDs, a local structural similarity score $sim(x, y)$ and a similarity threshold τ, the vertex similarity calculation (VSC) is to find all the vertex pairs $(v_i, v_j) \in V^2$ and their similarity scores $sim(v_i, v_j)$ such that $i \prec j$ and $sim(v_i, v_j) \geq \tau$.*

Table 3: Symbols

Symbol	Definition
G	An undirected simple graph $G = (V, E)$.
$\prec$	The total order defined on vertex IDs.
v_i	The vertex with the ID i.
d_i	The Degree of the vertex v_i. $d_i = \lvert\Gamma(v_i)\rvert$.
$\boldsymbol{v}_i$	The composite ID of the vertex v_i. $\boldsymbol{v}_i = d_i \vert\vert i$. $\vert\vert$ is the concatenation operator.
$\Gamma(v)$	The adjacency set of the vertex v. $\Gamma(v) = \{u \vert (u, v) \in E\}$.
$L(v)$	The adjacency list of the vertex v. The vertices in $L(v)$ are equal to $\Gamma(v)$ but sorted in the ascending order of their IDs.
$L(\boldsymbol{v})$	Same as $L(v)$ but the vertices are sorted by their composite IDs.
τ	Similarity threshold.
$o(x, y)$	The neighborhood overlap of x and y. $o(x, y) = \lvert\Gamma(x) \cap \Gamma(y)\rvert$.
$T(x, y)$	Overlap threshold of x and y. If $sim(x, y) \geq \tau$, $o(x, y) \geq T(x, y)$.

The core of the similarity scores in Table 1 is the neighborhood overlap $o(v_i, v_j)$ of the vertex pair, where $o(v_i, v_j) = |\Gamma(v_i) \cap \Gamma(v_j)|$. It measures the number of common neighbors of v_i and v_j. $sim(v_i, v_j) \geq \tau$ if and only if $o(v_i, v_j) \geq T(v_i, v_j)$ where

$T(v_i, v_j)$ is the corresponding overlap threshold. The definition of $T(v_i, v_j)$ depends on the similarity score. For example, the overlap threshold for the Salton score is $T(v_i, v_j) = \lceil \tau \sqrt{d_i d_j} \rceil$. Finding similar vertex pairs is equivalent to finding vertex pairs with large enough neighborhood overlaps.

2.2. Representative Serial Techniques

There are two representative serial techniques to solve the VSC problem. The prefix filter [25] and the probe-count technique [26] are representative filtering-based and all-pair techniques respectively. They are optimized for high and low thresholds respectively.

2.2.1. Prefix Filter

Given a graph G, the prefix filter first reassigns IDs of vertices in the graph according to their degrees. Vertices with smaller degrees have smaller IDs after the reassignment. In other words, if $v_i \prec v_j$, $d_i \leq d_j$. The prefix filter then converts every adjacency set into an adjacency list by sorting the vertices in the set in the ascending order defined by $\prec$.

The prefix filter technique splits each adjacency list $L(v_i)$ into two parts: the prefix part and the suffix part. The split point is decided by the threshold τ. Take the Jaccard score as the example. The last $\lceil \frac{2\tau d_i}{1+\tau} \rceil - 1$ vertices in $L(v_i)$ form the *suffix* part and the other vertices form the *prefix* part. Fig. 4 illustrates the split point of $L(v_5)$ in the toy graph (Fig. 1) with $\tau = 0.6$. Since $\lceil \frac{2\tau d_5}{1+\tau} \rceil - 1 = 2$, the last two vertices in $L(v_5)$ form the suffix part and the first two form the prefix part.

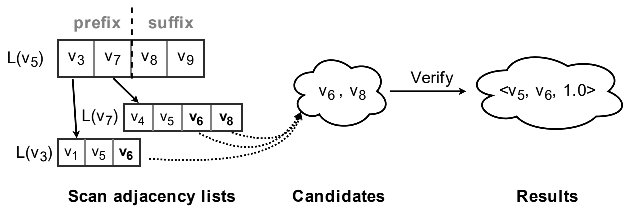


Figure 4: Process v_5 in the toy graph with the prefix filter technique. Similarity score: Jaccard, $\tau = 0.6$

For any two vertices v_i and v_j with $v_i \prec v_j$, if $sim(v_i, v_j) \geq \tau$, $L(v_j)$ must have at least one common vertex with the prefix part of $L(v_i)$. The Jaccard score can be rewritten as $sim(v_j, v_j) = \frac{o(v_i,v_j)}{d_i+d_j-o(v_i,v_j)}$. If $sim(v_i, v_j) \geq \tau$, then $o(v_i, v_j) \geq \frac{\tau(d_i+d_j)}{1+\tau} \geq \frac{2\tau d_i}{1+\tau}$. If $o(v_i, v_j) < \frac{2\tau d_i}{1+\tau}$, v_i and v_j cannot be similar enough. If $L(v_j)$ does not have any common vertices with the prefix part of $L(v_i)$, then $L(v_j)$ can only have common vertices with the suffix part of $L(v_i)$. It means that $o(v_i, v_j) \leq \lceil \frac{2\tau d_i}{1+\tau} \rceil - 1$ and (v_i, v_j) cannot be similar enough. For example, v_5 and v_2 in Fig. 1 cannot be similar because $L(v_2)$ does not have any common vertex with the prefix part of $L(v_5)$ as shown in Fig. 4.

Given a vertex v_i, the candidate vertices that may be similar enough to v_i are the vertices v_j that $v_i \prec v_j$ and $L(v_j)$ has at least one common vertex with the prefix part of $L(v_i)$. Non-candidate vertices cannot be similar enough with v_i. The prefix filter technique gets those candidate vertices by scanning the adjacency lists of the vertices appeared in the prefix part of $L(v_i)$. Taking v_5 in Fig. 4 as the example, $\{v_3, v_7\}$ are in the prefix part of $L(v_5)$. By scanning the adjacency lists of v_3 and v_7, the prefix filter gets $\{v_1, v_4, v_5, v_6, v_8\}$ as vertices that may be similar to v_5. Their adjacency lists share at least one common vertex with the prefix part of $L(v_5)$. Since the prefix filter only considers v_j with $v_5 \prec v_j$ as the candidate vertices of v_5, $\{v_6, v_8\}$ are remained as the candidates in Fig. 4. After getting the candidate vertices, the prefix filter verifies the exact similarity scores $sim(v_i, v_j)$ for the candidate vertices and keeps the vertex pairs whose scores are above the threshold as the results. In the case in Fig. 4, (v_5, v_6) is remained as the result.

The prefix filter technique works efficiently under high thresholds. As the threshold increases, the prefix part becomes shorter, generating less candidate vertices. However, the technique is not suitable for low thresholds. The prefix part becomes longer with lower thresholds. Since adjacency lists are sorted in the ascending order of vertex degrees, a longer prefix part leads to much more false candidates. Verifying a candidate vertex pair is an expensive operation as it conducts set intersection. The technique will spend lots of efforts on verifying dissimilar candidate pairs.

2.2.2. Probe-count Technique

Given a vertex v_i, the probe-count technique scans the adjacency sets of all the neighbors of v_i and counts how many times other vertices v_j with $v_i \prec v_j$ appear in those adjacency sets. If v_j appears k times, v_i and v_j have k common neighbors and their neighborhood overlap $o(v_i, v_j)$ is k. For example, in the case in Fig. 5, v_8 appears in two adjacency sets, $o(v_5, v_8) = 2$. Based on the neighborhood overlaps, the probe-count technique calculates the exact similarity scores and filters out the dissimilar pairs from the results.

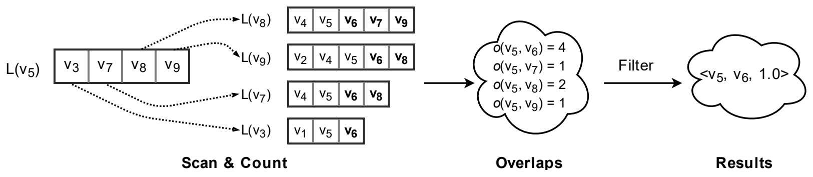


Figure 5: Process v_5 in the toy graph with the probe-count technique. Similarity score: Jaccard, $\tau = 0.6$

The main computation occurs in scanning adjacency sets and counting appearances. Since its cost is independent from the threshold, the performance of the probe-count technique is insensitive to the threshold. The technique is simple and all computation contributes to similarity score calculation. The overhead of the technique is little, making it suitable for low thresholds. However, the technique is not suitable for high thresholds as it will waste lots of efforts on counting appearances of dissimilar vertex pairs.

3. Related Work

In this section, we survey the methods that directly solve the VSC problem. Besides them, we also survey the set similarity join methods because they can solve the VSC problem indirectly. The set similarity join [25] is to find out all set pairs with similarity scores above a given threshold from a large collection of sets. According to the definition of the local structural similarity scores, finding similar vertex pairs is equivalent to finding similar adjacency set pairs. To unify the terminology, we introduce all the related methods from the perspective of the VSC problem. In this paper, we focus on the methods that give out accurate results. Approximate algorithms [27, 28] are inspiring but are not the focus of this paper.

3.1. Serial Methods

Based on the sensitivity of the performance to the threshold, we classify the existing serial methods into two groups: all-pair methods and filtering-based methods.

All-pair Methods. The performance of the all-pair methods Probe-Count and Pair-Count [26] is insensitive to the threshold. They avoid blindly enumerating $O(|V|^2)$ vertex pairs by only enumerating vertex pairs that have at least one common neighbor, since only those pairs can have non-zero similarity scores. Probe-Count enumerates such vertex pairs by scanning adjacency sets of all neighbors of each vertex. Pair-Count enumerates them by counting co-occurrences of vertices in all adjacency sets of the graph. They calculate accurate similarity scores for enumerated pairs and filter out dissimilar ones according to the threshold.

Filtering-based Methods. The performance of filtering-based methods is sensitive to the threshold. Most of the methods follow the filtering-verification framework. They first use filtering techniques to find candidate vertex pairs with potentially high similarity scores. The filtering techniques take advantage of the threshold information to reduce the number of candidate pairs, higher thresholds leading to less candidates. Then they verify accurate similarity scores of candidate pairs and output similar pairs above the threshold.

Different methods vary in filtering techniques. Chaudhuri et al. [25] propose the prefix filter introduced in Section 2.2.1. Arasu et al. [29] proposes the length filter that prunes dissimilar candidate pairs by lengths of adjacency lists. AllPairs [30] combines the prefix filter and length filter together. Xiao et al. [31] proposes the positional filter that uses the position information of the common neighbors of a vertex pair to filter out dissimilar pairs. MPJoin [32] enhances the positional filter by removing useless vertices in inverted indices of adjacency lists. AdaptJoin [33] extends the prefix filter with adaptive prefix lengths. GroupJoin [34] groups vertices with identical prefixes together to reduce computation costs. Mann et al. [35] propose the positional-enhanced length filter. Deng et al. [36] treat large and small vertices separately and propose efficient techniques to handle small vertices. However, their techniques only work with the common neighbor score. McCauley et al. [37] propose a novel index structure to handle skewed adjacency sets and analyze its theoretical bounds. Wang et al. [38] leverage the overlaps between adjacency sets to reduce redundant computation. Mann et al. [39] conduct a comprehensive evaluation of the filtering techniques on the same testbed. They find that AllPairs [30] is the quickest in most cases. Other more complex filtering techniques bring more overheads than benefits.

As the input graph grows bigger, the memory and computing power of a single machine limit the performance of the serial algorithms. Solving the VSC problem on big graphs calls for distributed methods.

3.2. Distributed Methods

Similar to the serial methods, the distributed methods can also be classified as the all-pair methods and filtering-based methods.

All-pair Methods. On the MapReduce platform, FullFilteringJoin [13] implements the Pair-Count method by counting co-occurrences by MapReduce. V-SMART-Join [14] follows a similar framework as FullFilteringJoin but optimizes the load balance. SBM [15] can be regarded as a vectorized version of FullFilteringJoin. It groups the key-value pairs with the same key together to reduce the number of key-value pairs emitted in the map phase. On the BSP/Pregel platform, Mohan et al. propose PCLP [17] to calculate the Adamic–Adar score in a scalable way. Its framework is similar to Probe-Count [26] and can process other similarity scores with slight modification. To avoid the out of memory problem in the VSC calculation on the BSP/Pregel platform, Ching et al. [16] propose the superstep splitting technique to limit the amount of messages exchanged in each superstep by introducing more small sub-supersteps.

Filtering-based Methods. The filtering-based methods focus on implementing the filtering techniques with the MapReduce programming model. Many of them [18, 19, 20, 22, 23] follow the signature-based framework. They generate signatures for each vertex based on the filtering techniques in the map phase. If two vertices have no common signature, the similarity score cannot exceed the threshold. They group vertices and their adjacency lists with the same signature together via shuffling. In the reduce phase, the serial set similarity join is conducted on each group of vertices with the same signature to find similar vertex pairs.

Different methods mainly differ in signature generation and data replication. VernicaJoin [18] uses vertices in prefix parts as signatures and adopts PPJoin+ [31] as the serial similarity join method. SSJ-2R [20] splits the adjacency lists into two parts: the prefix and the residual. It only replicates prefix parts during shuffling and joins residual parts to candidate vertex pairs in another job. MGJoin [19] extends VernicaJoin by introducing an extra load balancing job and using multiple total orders to filter candidate vertex pairs. MassJoin [22] and MRGroupJoin [23] use the partition-based signatures. FS-Join [40] splits adjacency lists vertically and conducts similarity join in each vertical fragment independently in parallel. Fier et al. [24] comprehensively compare the previous methods in the same testbed. They find that VernicaJoin [18] is the fastest and robustest method in term of execution time. However, its scalability is limited. The skewed degree distribution of vertices makes workloads of reduce tasks highly skewed. The reduce task that processes the signature with most vertices hits the memory limitation of a single machine, limiting the scalability of VernicaJoin.

Besides the pure MapReduce methods, RF-SetJoin [21] combines MapReduce with NoSQL database to avoid replicating adjacency lists during shuffling. It adopts the extended prefix filter [33] to generate signatures.

State-of-the-art Methods. According to the performance reported in the literature, we regard the following methods as the state-of-the-art distributed VSC methods on each platform: V-SMART-Join [14] (all-pair MapReduce), PCLP [17] with the superstep splitting technique (all-pair BSP), VernicaJoin [18] (filtering-based MapReduce) and RF-SetJoin [21] (filtering-based MapReduce + distributed key-value store).

The state-of-the-art methods are only optimized either for high thresholds (filtering-based methods) or low thresholds (all-pair methods). However, different graph analysis applications use different ranges of thresholds. Real-world requirements call for a method that is efficient under a broad range of thresholds.

4. VSIM Framework

To overcome the drawbacks of the existing methods, we propose a novel distributed framework *VSIM* for the VSC problem. We elaborate on the framework design in Section 4.1. To make the framework efficient under the broad range of thresholds, an threshold-adaptive technique is introduced in Section 4.2.

4.1. Framework Overview

The vertex similarity calculation naturally shows a task-parallel structure. For each vertex v_i, we can define a task for v_i as finding all vertices v_j that $v_i \prec v_j$ and $sim(v_i, v_j) \geq \tau$. We can solve the VSC problem by generating and executing tasks for all vertices in the graph. The execution of each task is independent from other tasks. Hence, we can execute the tasks in parallel in a distributed computing platform that supports the task-parallel model.

Based on the idea, we propose a novel distributed framework *VSIM* for *V*ertex *SIM*ilarity calculation. Algorithm 1 shows the workflow of VSIM. VSIM receives an undirected simple graph G as input with the similarity score *sim* and the threshold τ as parameters. VSIM outputs all similar enough vertex pairs with *exact* similarity scores. VSIM uses on a distributed key-value store to store adjacency lists of G. VSIM is made up of two procedures.

Algorithm 1 VSIM framework

```
Input: Graph $G = (V, E)$ with $\prec$ defined on $V$
Parameter: Similarity score $sim$, threshold $\tau$
Output: Similar vertex pairs and their scores $\{< v_i, v_j, sim(v_i, v_j) > | v_i, v_j \in V, sim(v_i, v_j) \geq \tau\}$
Environment: Distributed key-value store KVStore
 1: procedure PREPROCESS($G$)                                  ▷ Conducted once for each $G$
 2:     Convert vertex IDs $v_i$ of $G$ into composite IDs $\boldsymbol{v}_i$;
 3:     Convert adjacency sets $\Gamma(v_i)$ to adjacency lists $L(\boldsymbol{v}_i)$;
 4:     Store $<\boldsymbol{v}_i, L(\boldsymbol{v}_i)>$ for every vertex $\boldsymbol{v}_i \in V$ to KVStore;
 5: end procedure
 6:
 7: procedure SIMILARITYCALCULATION($G$, $sim$, $\tau$)
 8:     for all $\boldsymbol{v}_i \in V$ do in parallel
 9:         results ← SCTASK($\boldsymbol{v}_i$, $L(\boldsymbol{v}_i)$, $sim$, $\tau$, KVStore) ;
10:         output results;
11:     end for
12: end procedure
```

The `Preprocess` procedure prepares G for efficient similarity calculation. For every input graph, it only needs to be conducted once. The procedure converts original vertex IDs of G into composite IDs. For a vertex v_i with the original ID i, we get its composite ID $\boldsymbol{v}_i$ by concatenating its degree d_i and its original ID, i.e. $\boldsymbol{v}_i = d_i||i$. The composite ID allows VSIM to get the degree of a vertex from its ID directly. VSIM redefines the total order $\prec$ on composite IDs as $\boldsymbol{v}_i \prec \boldsymbol{v}_j$ if and only if $d_i < d_j$ or $d_i = d_j$ with $i < j$. With the new total order, the `Preprocess` procedure converts all adjacency sets $\Gamma(v_i)$ of G into adjacency lists $L(\boldsymbol{v_i})$ by replacing original vertex IDs in $\Gamma(v_i)$ with composite IDs and sort the composite IDs in the ascending order of $\prec$. The adjacency lists are stored into the distributed key-value store with composite IDs $\boldsymbol{v}_i$ as keys and adjacency lists $L(\boldsymbol{v}_i)$ as values.

The `SimilarityCalculation` procedure conducts the similarity calculation for a given similarity score and a threshold. It generates a *SC task* for every vertex $\boldsymbol{v}_i$. $\boldsymbol{v}_i$ is called the *center* vertex of the corresponding SC task. Each task finds vertices $\boldsymbol{v}_j$ similar enough to $\boldsymbol{v}_i$ and outputs their similarity scores. VSIM executes tasks with the `SCTask` function. We discuss the algorithms of the function later. All the tasks are executed in parallel by a distributed computing platform like Hadoop and Spark. During execution, the tasks query the distributed key-value store for needed adjacency lists. Combining results of all tasks together, VSIM gets the output for the whole graph G.

The real-world graphs often follow the power-law degree distribution, causing skewed workloads of SC tasks. In VSIM, we take advantage of the skewed degree distribution to balance computation workloads among SC tasks. For a SC task with the center vertex $\boldsymbol{v}_i$, VSIM only finds similar vertices $\boldsymbol{v}_j$ for it with $\boldsymbol{v}_i \prec \boldsymbol{v}_j$. As $\boldsymbol{v}_i \prec \boldsymbol{v}_j$ indicates that $d_i \leq d_j$, the SC task of $\boldsymbol{v}_i$ only finds similar vertices with equal and higher degrees. The vertices with higher degrees are rarer than $\boldsymbol{v}_i$ itself. By this way, VSIM distributes the computation workloads of high-degree vertices among the SC tasks with low-degree center vertices. Since there are much more low-degree vertices than high-degree vertices, the workloads are balanced on the task level.

4.2. Threshold-adaptive Similarity Calculation

The `SCTask` function is the core of the VSIM framework. Its execution cost dominates the total execution cost of the whole framework. Using existing serial techniques to implement the `SCTask` function has an inherent drawback that none of the techniques can run efficiently under the broad range of thresholds. To overcome the drawback, we propose a threshold-adaptive technique to implement the `SCTask` function.

4.2.1. Analysis of Existing Techniques

One way to implement the `SCTask` function is to use the serial techniques introduced in Section 2.2, i.e. the prefix filter and the probe-count technique. We can run the serial techniques in the distributed environment by fetching adjacency lists from the distributed key-value store before using them. However, the serial techniques are only efficient in a narrow range of thresholds as discussed in Section 2.2.

We notice that the prefix filter and the probe-count technique share some common computation. To find vertices similar to a given vertex v_i, the prefix filter scans the prefix part of $L(v_i)$ as shown in Fig. 4. The probe-count technique scans the whole $L(v_i)$ as shown in Fig. 5. Scanning the prefix part of $L(v_i)$ is the common operation of the two techniques.

The two techniques differ in how to process the suffix part of $L(v_i)$. The prefix filter just ignores the suffix part. Instead, it calculates the similarity scores of the candidate vertices as shown in Fig. 4. The probe-count technique continues to process the suffix part. As shown in Fig. 5, it scans the adjacency lists of the vertices in the suffix part and counts appearances of other vertices.

The difference makes the two techniques efficient in different ranges of thresholds. The computation costs of the two techniques are determined by the numbers of adjacency lists that they scan. If a technique scans less adjacency lists, its cost is less. Since the two techniques scan the same adjacency lists when they process the prefix part of $L(v_i)$, the numbers of scanned adjacency lists during processing the suffix part are critical to their costs. For the prefix filter, it needs to scan the adjacency lists of the candidate vertices to calculate the similarity scores. The number of scanned adjacency lists is equal to the number of candidate vertices. For the probe-count technique, the number of scanned adjacency lists is equal to the length of the suffix part. When the threshold is high, the prefix part of $L(v_i)$ is short and the suffix part is long. The number of candidate vertices generated by the prefix filter tends to be smaller than the number of vertices in the suffix part. In this scenario, the prefix filter runs more efficiently than the probe-count technique. When the threshold is low, the situation becomes the opposite.

The observation inspires us proposing a threshold-adaptive technique that combines the advantages of the both techniques together.

4.2.2. Threshold-adaptive Technique

The core idea of the threshold-adaptive technique is to process the prefix part first and decide how to process the suffix part later based on the costs of the two techniques. The pseudo-code of the technique is shown in Algorithm 2.

Given a SC task, the technique splits the adjacency list L into the prefix part and the suffix part guided by the prefix filter. The technique fetches and scans the adjacency lists of the vertices in the prefix part (line 4 to 9). During scanning, the technique counts the appearances of the vertices in those adjacency lists and stores the counting values in the *overlaps* map. After scanning, the key set of the *overlaps* map contains all the vertices whose adjacency lists share at least one common vertex with the prefix part of L. Those vertices are the candidate vertices for $\boldsymbol{v}_i$ in the prefix filter. We can further filter the candidate vertices with other filtering techniques introduced in Section 3.1 (line 10).

After processing the prefix part, the technique supports two modes to process the suffix part of L: the *verification* mode and the *counting* mode. The technique estimates the execution costs of the two modes and uses the mode with the smaller cost to process the suffix part (line 13). We will elaborate on how to estimate the execution costs later.

The verification mode works just like the prefix filter. It ignores the suffix part. Instead, it fetches the adjacency lists of the candidate vertices $\boldsymbol{v}_c$ from the distributed key-value store (line 15) and calculates the similarity scores (line 16). In the score calculation, the set intersection is conducted on the two adjacency lists to get the neighborhood overlap of the two vertices. Based on the overlap, the similarity score is calculated

Algorithm 2 Threshold-adaptive technique

```
function SCTask($\boldsymbol{v}_i$, $L$, $sim$, $\tau$, KVStore)
    $L_{\text{prefix}}, L_{\text{suffix}}$ ← SplitAdjacencyList($L$, $\tau$, $sim$);        ▷ $L$ is the adjacency list of $\boldsymbol{v}_i$
    $overlaps$ ← new HashMap[Vid, Int]();
    for $\boldsymbol{v}_j \in L_{\text{prefix}}$ do        ▷ Process the prefix part of $L(\boldsymbol{v}_i)$
        $L_j$ ← KVStore.get($\boldsymbol{v}_j$);
        for $\boldsymbol{v}_{nn} \in L_j$ that $\boldsymbol{v}_i \prec \boldsymbol{v}_{nn}$ do
            $overlaps[\boldsymbol{v}_{nn}] \leftarrow overlaps[\boldsymbol{v}_{nn}] + 1$;        ▷ $overlaps[\boldsymbol{v}_{nn}]$ is 0 for unseen $\boldsymbol{v}_{nn}$
        end for
    end for
    Filter candidate vertices in $overlaps$ with other filtering techniques;
    $cost_{\text{Verification}}$ ← EstimateVerificationCost($overlaps$.keys(), $\boldsymbol{v}_i$);
    $cost_{\text{Counting}}$ ← EstimateCountingCost($L_{\text{suffix}}$);
    if $cost_{\text{Verification}} < cost_{\text{Counting}}$ then
        for $\boldsymbol{v}_c$ ← $overlaps$.keys() do        ▷ Use the verification mode
            $L_c$ ← KVStore.get($\boldsymbol{v}_c$);
            $score \leftarrow sim(L, L_c)$;
            output $< v_i, v_c, score >$ if $score \geq \tau$;
        end for
    else        ▷ Use the counting mode
        for $\boldsymbol{v}_j \in L_{\text{suffix}}$ do
            $L_j$ ← KVStore.get($\boldsymbol{v}_j$);
            for $\boldsymbol{v}_{nn} \in L_j$ that $\boldsymbol{v}_i \prec \boldsymbol{v}_{nn}$ do
                $overlaps[\boldsymbol{v}_{nn}] \leftarrow overlaps[\boldsymbol{v}_{nn}] + 1$;
            end for
        end for
        for $\boldsymbol{v}_c$ ← $overlaps$.keys() do
            $d_c$ ← RetrieveDegree($\boldsymbol{v}_c$); $d_i$ ← RetrieveDegree($\boldsymbol{v}_i$);
            $score \leftarrow sim(overlaps[\boldsymbol{v}_c], d_c, d_i)$;        ▷ $overlaps[\boldsymbol{v}_c] = |\Gamma(v_i) \cap \Gamma(v_c)|$
            output $< v_i, v_c, score >$ if $score \geq \tau$;
        end for
    end if
end function
```

according to its definition. Since the vertices in the adjacency lists are already sorted, the set intersection can be conducted by one-pass merging. The similar enough vertex pairs are outputted. The prefix filter guarantees the correctness of the mode.

The counting mode works just like the probe-count technique. It fetches the adjacency lists of the vertices in the suffix part of L and counts the appearances of other vertices in those adjacency lists (line 20 to 25). After scanning all the adjacency lists, the *overlaps* map contains the neighborhood overlaps of $\boldsymbol{v}_i$ and other vertices $\boldsymbol{v}_c$. The counting mode calculates the similarity scores based on the neighborhood overlaps and outputs the similar enough vertex pairs. The probe-count technique guarantees the correctness of the mode.

The threshold-adaptive technique can run efficiently under the broad range of thresholds. The technique chooses the proper execution mode according to the estimated

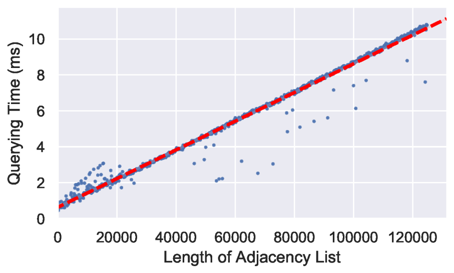


Figure 6: Querying time of adjacency lists with different lengths

costs. It prefers the verification mode for high thresholds and prefers the counting mode for low thresholds. It takes advantages of both the prefix filter and the probe-count technique. Moreover, different SC tasks can choose different modes adaptively. The total execution cost on the whole graph can be less than the total cost of using a single execution mode.

4.2.3. Execution Cost Estimation

The two execution modes have different workflows. To compare their execution costs fairly, we use the execution time as the measurement of execution costs. Thus, the core problem in cost estimation is to estimate execution time. The two execution modes both use two basic operations: *fetching* an adjacency list from the distributed key-value store and *scanning* an adjacency list.

As for the fetching operation, its execution time consists of two parts: the round-trip latency of the key-value store query and the transmission time of the query payload. The round-trip latency is fixed for every query. The transmission time is proportional to the length of the queried adjacency list. We estimate its execution time with a linear model $QTime(l) = \alpha l + \beta$. l is the length of the queried adjacency list. α and β are the transmission and latency coefficients. To get the values of α and β, we measure the querying time of adjacency lists with different lengths in our experimental environment. The results are shown in Fig. 6. By fitting the data with the linear model, we estimate $\alpha = 8.0 \times 10^{-5}$ms per vertex and $\beta = 0.63$ms per query.

As for the scanning operation, we estimate its execution time in the two modes separately because the two modes have different computation logic. The verification

Algorithm 3 Execution cost estimation

```
1:  function EstimateVerificationCost($candidates$, $\boldsymbol{v}_i$)
2:      $cost \leftarrow 0$; $d_i \leftarrow$ RetrieveDegree($\boldsymbol{v}_i$);
3:      for $\boldsymbol{v}_c \in candidates$, $d_c \leftarrow$ RetrieveDegree($\boldsymbol{v}_c$) do
4:          $cost \leftarrow cost + QTime(d_c) + CTime(d_i, d_c)$;
5:      end for
6:      return $cost$;
7:  end function
8:  function EstimateCountingCost($L_{\text{suffix}}$)
9:      $cost \leftarrow 0$;
10:     for $\boldsymbol{v}_c \in L_{\text{suffix}}$, $d_c \leftarrow$ RetrieveDegree($\boldsymbol{v}_c$) do
11:         $cost \leftarrow cost + QTime(d_c) + CTime'(d_c)$;
12:     end for
13:     return $cost$;
14: end function
```

mode scans two adjacency lists in a merge-sort way to calculate the neighborhood overlap of the adjacency lists. Based on the neighborhood overlap, the similarity score can be calculated with $O(1)$ cost (line 16 of Algorithm 2). Thus, its execution time is linear to the lengths of the two adjacency lists. We estimate the time as $CTime(l_1, l_2) = \gamma(l_1 + l_2)$ where l_1 and l_2 are the lengths of the adjacency lists and γ is a system-related coefficient. The counting mode scans adjacency lists to update the *overlaps* map (line 23 of Algorithm 2). Its cost is linear to the length of scanned adjacency lists. Assuming the length is l, we estimate its execution time as $CTime'(l) = \gamma' l$ where γ' is also a system-related coefficient. In our environment, $\gamma = 3.3 \times 10^{-6}$ms per vertex and $\gamma' = 0.0023$ms per vertex. γ' is much larger than γ because updating a hash map involves random access of memory.

With the help of the estimation models, we can estimate the execution costs of the two modes with Algorithm 3.

5. Implementation and Optimization

In this section, we give out an implementation of VSIM with the existing distributed computing platforms. We further propose optimization techniques to make it more efficient.

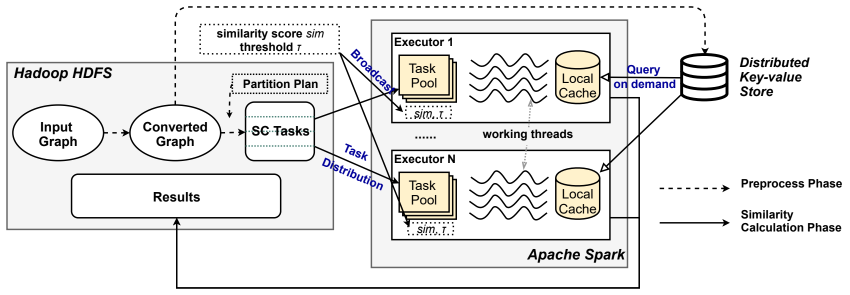


Figure 7: Implementation architecture of VSIM with HDFS and Spark

5.1. Implementation Architecture

Fig. 7 shows the architecture of our demo implementation of VSIM based on Hadoop HDFS and Apache Spark.

In the preprocessing phase, the input graph G is originally stored as a file on HDFS. The implementation composites vertex IDs of G and converts adjacency sets into adjacency lists via a Spark job. The adjacency lists of the converted graph are stored into a distributed key-value store in parallel via another Spark job. The implementation generates several SC task files from the converted graph file with the help of a graph partition plan. The graph partition plan splits the vertex set of G into several partitions. Each SC task file contains vertices and their adjacency lists from a graph partition.

In the similarity calculation phase, the implementation starts a Spark job to run the SC tasks in parallel. The similarity score and the threshold τ are broadcasted to every executor, i.e. worker node, in the Spark platform. The Spark job loads the SC task files on HDFS into a task RDD. Since the SC task files have been partitioned in the preprocessing phase, each file forms a RDD partition. The RDD partitions are distributed among executors. The Spark job conducts a *flatMap* operator on the task RDD to run the tasks in parallel. The results of the *flatMap* operator are outputted to HDFS.

In each Spark executor, we use several working threads to run the SC tasks in a RDD partition concurrently. During task execution, the key-value store querying operations frequently block the tasks, wasting CPU resources. To increase CPU utilization, we create working threads multiple times of available CPU cores in a node.

It is noteworthy that the implementation of VSIM is not restricted to HDFS and Spark. VSIM can also be implemented with other computing platforms that support the task-parallel framework. For example, we can use Hadoop MapReduce as the underlying computing platform. We can conduct the preprocessing with pure MapReduce jobs. We can use a map-only MapReduce job to execute SC tasks in parallel. Each mapper loads a partition of SC tasks from HDFS and executes tasks concurrently with a thread pool. By a similar way, VSIM can also be implemented with Apache Flink.

5.2. Local Cache Technique

To reduce execution costs of key-value store querying operations, we set up an in-memory local cache in each Spark executor. The cache stores the adjacency lists fetched from the distributed key-value store. Only when the cache misses, a query is actually conducted by an executor. The cache is shared among all working threads to capture repeating queries from different SC tasks. Though the cache trades memory space for reduction in execution time, it is worthwhile. According to the coefficient β in Section 4.2.3, querying operations are very expensive. Even a small number of cache hits will reduce execution time obviously.

5.3. Load Balance Technique

The workload of a SC task is highly related to the degree of its center vertex. However, degree distributions of real-world graphs are often skewed, causing workloads of SC tasks also skewed. In Section 4.1, we try to balance computation workloads on the task level by taking advantage of the total order $\prec$. To further balance workloads, we propose two implementation optimizations that work on the partition level and the machine level respectively.

On the partition level, we balance workloads by partitioning SC tasks with a balanced graph partition plan. The number of partitions is several times of the number of nodes in the cluster which is much smaller than the number of vertices. The workloads on the partition level are much more balanced than the workloads on the task level.

On the machine level, we balance workloads by using the dynamic task scheduling mechanism in Spark. A RDD in Spark consists of several RDD partitions. Spark uses a dynamic task scheduler to schedule operator execution on RDD partitions among executors. As long as an executor becomes idle, the Spark driver will assign an unexecuted Spark task to it to conduct operators on a RDD partition. By this way, Spark makes all executors as busy as possible, balancing workloads among executors dynamically.

5.4. Partition Plan Technique

We can find valid partition plans for SC tasks with the help of edge-cut graph partitioning algorithms. An edge-cut graph partitioning algorithm partitions the vertex set into disjoint partitions. In the SC task partition plan, we assign each SC task to the partition that its center vertex belongs to in the graph partition plan. Hence, a graph partition plan corresponds to a SC task partition plan.

The SC task partition plan significantly affects the total execution cost of querying operations. The partition plan determines what queries are conducted by each partition. It directly affects hit rates of local caches. If the SC tasks in the same partition conduct many repeating queries, the local cache can achieve a high hit rate, significantly reducing the costs spent on querying operations. A good partition plan should make the SC tasks in the same partition conduct as many repeating queries as possible.

We can use advanced edge-cut graph partitioning algorithms like METIS [41] and KaHIP [42] to generate good SC task partition plans. Those algorithms heuristically generate partition plans that try to minimize the number of cut edges while balance the workloads among partitions. SC task partition plans derived from such graph partition plans can lead to high cache hit rates. A small number of cut edges in graph partitions means that the vertices in the same partition are closely connected with each other. For a vertex v_i, the vertices in the adjacency list of v_i tends to be in the same partition as v_i. When VSIM executes the SC task with v_i as the center vertex, the task will likely query the adjacency lists of the vertices in the same partition. Executing all the SC tasks from the same partition will conduct many repeating queries for the adjacency lists in the partition. If the cache can hold those adjacency lists, a high cache hit rate can be achieved. Therefore, such partition plans are superior.

6. Evaluation

We introduce the experimental setup in Section 6.1 and validate the effectiveness of the proposed techniques in Section 6.2. The performance of VSIM is compared with the state-of-the-art distributed methods in Section 6.3. Finally, we evaluate the node scalability of VSIM in Section 6.4.

6.1. Experimental Setup

Environment. All the experiments were conducted in a cluster with 17 nodes (1 master + 16 workers) connected via 1Gbps Ethernet. Each node was equipped with two Xeon E5-2620 CPUs (12 cores in total), 50 Gbytes memory and 2 Tbytes RAID0 HDD storage. Each node ran CentOS 7.0 as the operating system. Hadoop 2.7.2 was adopted to provide HDFS and YARN.

VSIM. We implemented VSIM with Spark 2.2.0 and compiled it with JDK 8 and Scala 2.11.8. We adopted Apache Cassandra 3.11.4 as the distributed key-value store and deployed it to the whole cluster. On each node, Cassandra had 5 Gbytes row cache and 1 Gbytes key cache. Without otherwise mentioned, we ran VSIM with 16 Spark executors. We allocated 10 Gbytes heap to each executor and started 48 working threads in it. The local cache in each executor was off-heap and its capacity was 25 Gbytes. SC tasks were split into 128 partitions with a random partition plan. We used the length filter [29] to further prune the candidate vertices in the threshold-adaptive technique.

Datasets. We used 5 typical real-world graphs listed in Table 2 to evaluate the performance. They covered different types of graphs. We used the Jaccard score as the similarity score in all the experiments.

6.2. Effectiveness of Proposed Techniques

In this section, we evaluated the effectiveness of the proposed techniques in VSIM.

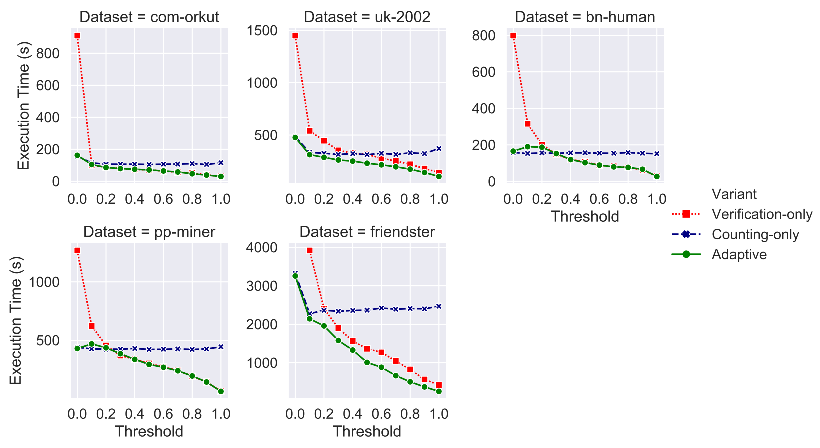


Figure 8: Execution time of different variants of VSIM

6.2.1. *Threshold-adaptive Technique*

To evaluate the threshold-adaptive technique, we implemented three variants of VSIM: the threshold-adaptive, the verification-only, and the counting-only. The later two variants only used a single execution mode.

Fig. 8 shows the execution time of the three variants under different thresholds. The results confirm that the verification mode and the counting mode are only efficient under high and low thresholds respectively. Compared to them, the threshold-adaptive variant is efficient under the whole range of thresholds. When the threshold is high (low), the execution time of the threshold-adaptive variant is very close to the verification-only (counting-only) variant. It indicates that the threshold-adaptive technique is effective and can choose the proper execution mode adaptively.

6.2.2. *Local Cache Technique*

To evaluate the local cache technique, we ran VSIM with different relative cache capacities. The relative cache capacity is defined as the ratio of the cache capacity to the size of the dataset represented with 64-bit integers.

We report the performance of VSIM with different capacities in Fig. 9. As the cache capacity increases, the average hit rate grows quickly and stabilizes. Consequently, the network communication cost and the execution time decrease quickly and stabilize. The local cache can significantly reduce the execution time.

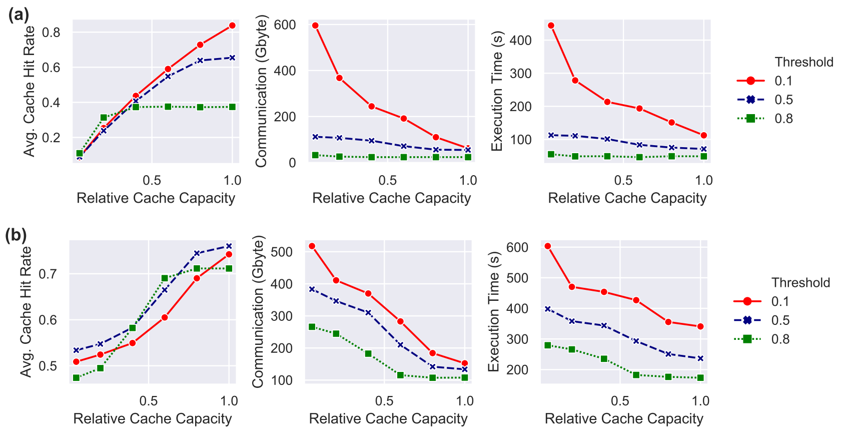


Figure 9: Effects of the cache capacity on performance. Dataset: (a) `com-orkut`, (b) `uk-2002`

Notice the similar trends of communication costs and execution time. It indicates that the communication costs that are caused by the fetching operation dominate the total execution costs of VSIM.

6.2.3. Load Balance Technique

To evaluate the load balance technique, we measured the execution time of SC tasks (task level), partitions (partition level) and Spark executors (machine level) respectively. We used the relative standard deviation (RSD) to measure how skew the distribution of the execution time was on different levels. Given a group of numeric values X, the RSD of X is defined as $RSD(X) = \frac{std(X)}{mean(X)}$. Smaller RSD indicates more balanced workloads on the corresponding level.

Partition Level. The results of the task level and partition level are reported in Table 4. The relative standard deviation on the partition level is smaller than the task level by an order of magnitude. It indicates that workloads are much more balanced on the partition level. We disabled the local cache in the experiments because the local cache interfered with measurement. With the local cache enabled, tasks/partitions executed later were executed with higher cache hit rates. A higher hit rate significantly reduced the execution time, unfair to the tasks/partitions executed earlier.

Table 4: Relative standard deviation of execution time on the task level and partition level

Dataset	bn-human			uk-2002			pp-miner		
Threshold	0.9	0.7	0.5	0.9	0.7	0.5	0.9	0.7	0.5
Task level	0.99	1.07	1.06	2.3	2.28	2.34	1.77	1.43	1.38
Partition level	0.07	0.05	0.05	0.12	0.07	0.07	0.17	0.04	0.04

Table 5: Relative standard deviation of execution time on the machine level

Dataset	Threshold	$p = 32$	$p = 64$	$p = 128$
uk-2002	0.1	0.085	0.051	0.027
uk-2002	0.5	0.072	0.038	0.018
uk-2002	0.8	0.072	0.041	0.018
pp-miner	0.1	0.024	0.017	0.009
pp-miner	0.5	0.014	0.011	0.008
pp-miner	0.8	0.011	0.016	0.009

Machine Level. We ran VSIM with 16 Spark executors and different numbers of SC task partitions p. The results of the machine level are reported in Table 5. The RSDs are less than 0.1 in all test cases, which means the standard deviation of the execution time of all executors is less than 10% of the average execution time. With 128 partitions, the RSDs are even less than 0.03. The dynamic task scheduling mechanism makes workloads well balanced on the machine level.

6.2.4. Partition Plan Technique

To evaluate the effects of the partition plan, we ran VSIM with the random partition plan and the KaHIP [42] partition plan separately. KaHIP is a heuristic graph partitioning algorithm aiming at reduing cut edges.

We compared the performance of VSIM under the two partition plans in Fig. 10. The KaHIP partition plan improves the performance significantly. Compared to the random partition plan, it improves the average cache hit rate by up to 27.2% on `uk-2002` (16% on `bn-human`). The improved hit rates decrease the communication costs by up to 82% on `uk-2002` (69% on `bn-human`) and decrease the execution time by up to 56% on `uk-2002` (31% on `bn-human`). It indicates that VSIM can take advantage of advanced graph partitioning plans to improve its performance.

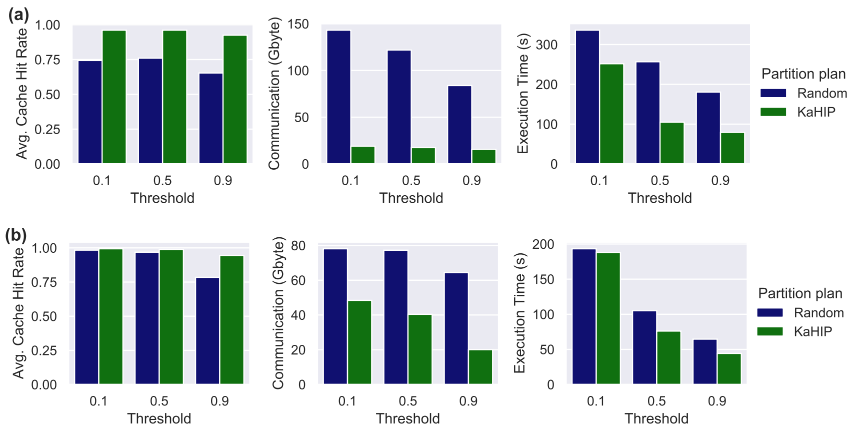


Figure 10: Effects of the partition plan on performance. Dataset: (a) `uk-2002`, (b) `bn-human`

6.3. Comparison with State-of-the-art Distributed Methods

We compared the execution time of VSIM with the state-of-the-art distributed VSC methods surveyed in Section 3.2. We implemented the MapReduce-based methods V-SMART-Join and VernicaJoin with Apache Spark from scratch. We ported the source code of RF-SetJoin provided by the authors [21] from Hadoop MapReduce to Apache Spark and made it use Cassandra as the distributed key-value database. We modified the BSP-based method PCLP to calculate the Jaccard score with Apache Giraph. We applied the superstep splitting technique to PCLP to avoid out of memory exception.

In the experiments, we focused on the execution time of the similarity calculation phase of all the compared algorithms, excluding the threshold-independent preprocessing phase and the output phase. If the execution time exceeded three hours, we regarded it as time out. The results are reported in Fig. 11.

VSIM is the quickest method in 51 out of 55 test cases. The quickest methods in the other 4 cases are RF-SetJoin (`uk-2002` with $\tau = 1.0$, `friendster` with $\tau = 0.9$, `friendster` with $\tau = 0.8$) and VernicaJoin (`friendster` with $\tau = 1.0$). In those 4 cases, the execution time of VSIM is 109%, 180%, 169% and 187% of the quickest method respectively.

Compared with the all-pair methods V-SMART-Join and PCLP, VSIM outperforms them under the whole range of thresholds. The results confirm that the execution time of the all-pair methods is insensitive to thresholds, making them inefficient under high

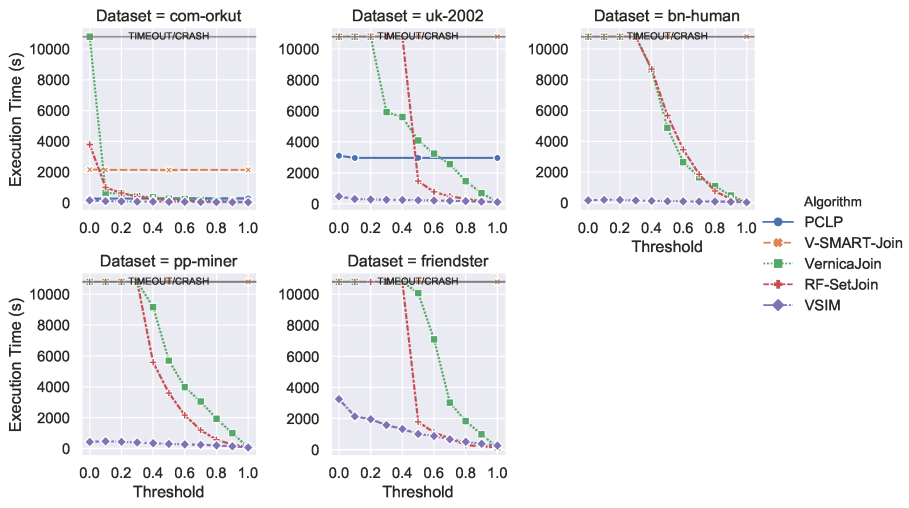


Figure 11: Performance comparison of VSIM with the state-of-the-art distributed methods

thresholds. V-SMART-Join crashed due to out of memory on all the graphs except `com-orkut`. PCLP performed better than V-SMART-Join. It could handle `uk-2002` but it timed out on bigger graphs. Compared to PCLP, VSIM achieves 1.7x to 9.5x speedup on `com-orkut` and 6.5x to 28.4x speedup on `uk-2002`. On the other graphs, V-SMART-Join and PCLP failed while VSIM ran smoothly.

Compared with the filtering-based methods VernicaJoin and RF-SetJoin, the performance of VSIM is comparable to them when $\tau \geq 0.8$. VSIM shows a clear advantage over them when $\tau < 0.8$. Among VernicaJoin and RF-SetJoin, RF-SetJoin performs better, especially on big graphs. Compared to RF-SetJoin, VSIM achieves 1.4x to 1.9x speedups on `com-orkut`, 0.9x to 1.9x on `uk-2002`, 1.3x to 10.0x on `bn-human`, 1.0x to 2.9x on `pp-miner` and 0.56x to 0.59x on `friendster` under $\tau \geq 0.8$; VSIM achieves 1.7x to 23.4x speedups on `com-orkut`, 2.5x to 6.4x on `uk-2002`, 23.6x to 72.0x on `bn-human`, 5.0x to 16.6x on `pp-miner`, 1.1x to 1.8x on `friendster` under $\tau < 0.8$. Under very low thresholds, RF-SetJoin crashed during the shuffle. VernicaJoin timed out on `uk-2002` and crashed due to out of memory on the other graphs.

None of the state-of-the-art could handle all the test cases in the experiments. VSIM was the only one that ran smoothly on all datasets and all thresholds without any crash or timeout. The results indicate that VSIM is scalable to big graphs and is more efficient

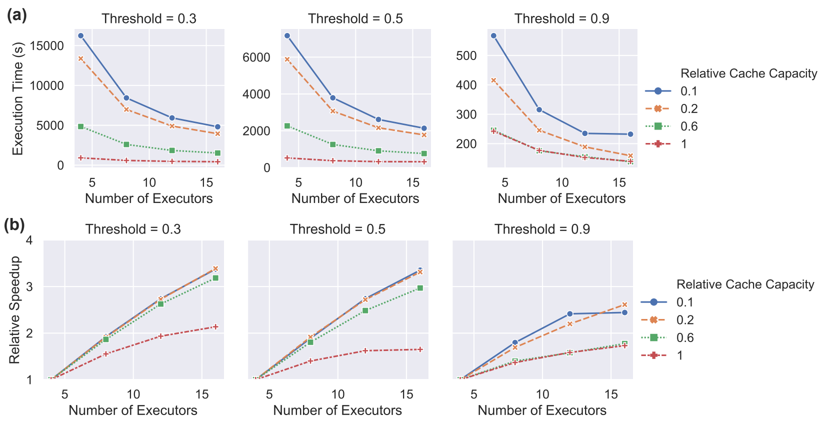


Figure 12: Node scalability of VSIM. (a) Execution time, (b) Relative speedup

than the state-of-the-art under a broad range of thresholds.

6.4. Node Scalability

To evaluate the node scalability of VSIM, we ran VSIM with 4, 8, 12 and 16 Spark executors (i.e. worker nodes) respectively on the `pp-miner` dataset. We choose the `pp-miner` dataset because its size is appropriate. It is large enough to evaluate the node scalability and is also small enough to make the execution time with 4 executors tolerable. We report the execution time of VSIM in Fig. 12(a) and convert it to relative speedups in Fig. 12(b). The relative speedup is defined as dividing the execution time of the current case by the execution time of the case with 4 executors (relative speedup=1). Overall, VSIM achieves near-linear node scalability when the threshold is low and the cache capacity is small. The node scalability is sub-linear when the threshold is high or the relative cache capacity is big.

We observe that the scalability of VSIM is affected by both of thresholds and cache capacities. With a fixed cache capacity, the scalability becomes more linear as the threshold decreases. With a fixed threshold, the scalability becomes more linear as the relative cache capacity decreases. The phenomena indicate that the node scalability has a close connection with the local cache.

Discussion. The computation cost of VSIM is solely determined by the dataset and the threshold. It is irrelevant to the number of nodes. However, the querying cost of VSIM increases with more nodes, which prevents VSIM from achieving ideal scalability.

The increased querying cost mainly comes from the warm-up phase of the local cache. When VSIM fetches the first few adjacency lists in each node, the hit rate of the local cache is very low. As more adjacency lists are fetched, the hit rate increases and stabilizes. The phase that the hit rate climbs is the warm-up phase of the local cache. If a SC task is executed during the warm-up phase, it will have a larger querying cost than being executed after the warm-up phase. Since every node has to undergo the warm-up phase, if we execute the same group of SC tasks with more nodes, more SC tasks will be executed during the warm-up phase. The total querying cost will be larger.

The warm-up phase explains the observed phenomena. When the cache capacity becomes higher, the warm-up phase will be longer. When the threshold becomes higher, VSIM will fetch fewer adjacency lists. In both cases, the warm-up phase will affect more SC tasks. For the experiments in Fig. 12, when $\tau = 0.9$ and the relative cache capacity was 1.0, the average cache hit rates were 83%, 74%, 66% and 61% for 4, 8, 12 and 16 executors respectively. The local cache still underwent the warm-up phase when VSIM finished. On the contrary, when $\tau = 0.3$ and the relative cache capacity was 0.1, the warm-up phase was short and the average cache hit rates stabilized at 14% in all the cases, marginally affecting the scalability.

7. Conclusion and Future Work

The vertex similarity calculation is a fundamental operation in many real-world graph analysis applications. However, it is difficult to get an efficient method for the problem because the computation complexity varies drastically as the similarity threshold changes. The varying complexity requires optimizations in contrary directions. Existing distributed methods for the problem are inefficient under either high or low thresholds. None of them can be efficient under a broad range of thresholds.

To overcome the drawback, we proposed a novel distributed framework *VSIM* for the vertex similarity calculation problem. VSIM built upon the task-parallel model. It stores adjacency lists of a graph in a distributed key-value store. It regards finding vertices similar to a given vertex v_i as a SC task. It conducts the similarity calculation of the whole graph by executing all SC tasks in parallel on a distributed platform. VSIM supports two execution modes to execute a SC task: the *verification* mode and the *counting* mode. The two modes are optimized for high and low thresholds respectively. VSIM uses the threshold-adaptive technique to choose the suitable execution mode for every task independently. In this way, VSIM can run efficiently under the whole range of thresholds.

We also proposed an efficient implementation for VSIM based on HDFS and Spark. To make the implementation more efficient, we further proposed the local cache technique, the load balance technique and the partition plan technique.

Experimental results on real-world graphs validated the effectiveness of the proposed techniques. The threshold-adaptive technique helped VSIM run efficiently under the broad range of thresholds. The local cache technique significantly reduced querying costs. The load balance technique balanced workloads on both of the partition and machine levels. The partition plan technique further reduced querying costs by improving cache hit rates. With the help of those techniques, VSIM outperformed the state-of-the-art distributed methods in 51 out of 55 test cases. None of the state-of-the-art could handle big graphs under a broad range of thresholds. VSIM was the only method that ran smoothly in all test cases. VSIM achieved near-linear scalability in low threshold and small cache scenarios.

For future work, we plan to extend VSIM in three directions. The first is to support higher order similarity scores like SimRank that uses secondary neighborhood information. The second is to support directed graphs. The third is to support dynamic graphs, allowing VSIM to monitor new similar vertex pairs in a continuous way. We also plan to propose a more accurate cost estimation model for VSIM by considering hit rates of the local cache.

Acknowledgements

This work was supported by the National Natural Science Foundation of China [grant numbers 61572250, U1811461, 61702254]; the Jiangsu Province Science and Technology Program [grant number BE2017155]; the National Natural Science Foundation of Jiangsu Province [grant number BK20170651]; the Collaborative Innovation Center of Novel Software Technology and Industrialization; and the program B for Outstanding PhD candidate of Nanjing University.

References

[1] J. Chen, W. Geyer, C. Dugan, M. Muller, I. Guy, Make new friends, but keep the old: Recommending people on social networking sites, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ACM, 2009, pp. 201–210. doi:10.1145/1518701.1518735.

[2] T. Alison, K. S. Sidhu, C. C.-F. Pai, P. H. Martinazzi, F. Ratiu, J. B. F. Inc.), Filtering and ranking recommended users on a social networking system, United States Patent Application 20130103758 (2011).
URL https://patents.google.com/patent/US20130103758A1/en

[3] L. Lü, T. Zhou, Link prediction in complex networks: A survey, Physica A: Statistical Mechanics and its Applications 390 (6) (2011) 1150–1170. doi:10.1016/j.physa.2010.11.027.

[4] E. Ravasz, A. L. Somera, D. A. Mongru, Z. N. Oltvai, A.-L. Barabási, Hierarchical organization of modularity in metabolic networks, Science 297 (5586) (2002) 1551–1555. doi:10.1126/science.1073374.

[5] K. Li, X. Gong, S. Guan, C.-H. Lai, Efficient algorithm based on neighborhood overlap for community identification in complex networks, Physica A: Statistical Mechanics and its Applications 391 (4) (2012) 1788–1796. doi:10.1016/j.physa.2011.09.027.

[6] F. Zhao, A. K. H. Tung, Large scale cohesive subgraphs discovery for social network visual analysis, Proceedings of the VLDB Endowment 6 (2) (2012) 85–96. doi:10.14778/2535568.2448942.

[7] V. Satuluri, S. Parthasarathy, Y. Ruan, Local graph sparsification for scalable clustering, in: Proceedings of the 2011 ACM SIGMOD International Conference

on Management of Data, ACM, 2011, pp. 721–732. doi:10.1145/1989323.1989399.

[8] X. Xu, N. Yuruk, Z. Feng, T. A. J. Schweiger, Scan: a structural clustering algorithm for networks, in: Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2007, pp. 824–833. doi:10.1145/1281192.1281280.

[9] J. Leskovec, A. Krevl, SNAP Datasets: Stanford large network dataset collection, http://snap.stanford.edu/data (2014).

[10] P. Boldi, S. Vigna, The WebGraph framework I: Compression techniques, in: Proceedings of the Thirteenth International World Wide Web Conference, ACM, 2004, pp. 595–601.

[11] R. A. Rossi, N. K. Ahmed, The network data repository with interactive graph analytics and visualization, http://networkrepository.com (2015).

[12] Marinka Zitnik, R. Sosic, S. Maheshwari, J. Leskovec, BioSNAP Datasets: Stanford biomedical network dataset collection, http://snap.stanford.edu/biodata (2018).

[13] T. Elsayed, J. Lin, D. W. Oard, Pairwise document similarity in large collections with mapreduce, in: Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics on Human Language Technologies: Short Papers, ACM, 2008, pp. 265–268.

[14] A. Metwally, C. Faloutsos, V-smart-join: A scalable mapreduce framework for all-pair similarity joins of multisets and vectors, Proceedings of the VLDB Endowment 5 (8) (2012) 704–715. doi:10.14778/2212351.2212353.

[15] S. Schelter, C. Boden, V. Markl, Scalable similarity-based neighborhood methods with mapreduce, in: Proceedings of the Sixth ACM Conference on Recommender Systems, ACM, 2012, pp. 163–170. doi:10.1145/2365952.2365984.

[16] A. Ching, S. Edunov, M. Kabiljo, D. Logothetis, S. Muthukrishnan, One trillion edges: graph processing at facebook-scale, Proceedings of the VLDB Endowment 8 (12) (2015) 1804–1815. doi:10.14778/2824032.2824077.

[17] A. Mohan, R. Venkatesan, K. Pramod, A scalable method for link prediction in large real world networks, Journal of Parallel and Distributed Computing 109 (2017) 89–101. doi:10.1016/j.jpdc.2017.05.009.

[18] R. Vernica, M. J. Carey, C. Li, Efficient parallel set-similarity joins using mapreduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, ACM, New York, NY, USA, 2010, pp. 495–506. doi:10.1145/1807167.1807222.

[19] C. Rong, W. Lu, X. Wang, X. Du, Y. Chen, A. K. H. Tung, Efficient and scalable processing of string similarity join, IEEE Transactions on Knowledge and Data Engineering 25 (10) (2013) 2217–2230. doi:10.1109/TKDE.2012.195.

[20] R. Baraglia, G. D. F. Morales, C. Lucchese, Document similarity self-join with mapreduce, in: The 10th IEEE International Conference on Data Mining, 2010, pp. 731–736. doi:10.1109/ICDM.2010.70.

[21] C. Kim, K. Shim, Supporting set-valued joins in nosql using mapreduce, Information Systems 49 (2015) 52–64. doi:10.1016/j.is.2014.11.005.

[22] D. Deng, G. Li, S. Hao, J. Wang, J. Feng, Massjoin: A mapreduce-based method for scalable string similarity joins, in: IEEE 30th International Conference on Data Engineering, 2014, pp. 340–351. doi:10.1109/ICDE.2014.6816663.

[23] D. Deng, G. Li, H. Wen, J. Feng, An efficient partition based method for exact set similarity joins, Proceedings of the VLDB Endowment 9 (4) (2015) 360–371. doi:10.14778/2856318.2856330.

[24] F. Fier, N. Augsten, P. Bouros, U. Leser, J.-C. Freytag, Set similarity joins on mapreduce: an experimental survey, Proceedings of the VLDB Endowment 11 (10) (2018) 1110–1122. doi:10.14778/3231751.3231760.

[25] S. Chaudhuri, V. Ganti, R. Kaushik, A primitive operator for similarity joins in data cleaning, in: Proceedings of the 22nd International Conference on Data Engineering, 2006, p. 5. doi:10.1109/ICDE.2006.9.

[26] S. Sarawagi, A. Kirpal, Efficient set joins on similarity predicates, in: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, ACM, 2004, pp. 743–754. doi:10.1145/1007568.1007652.

[27] R. B. Zadeh, A. Goel, Dimension independent similarity computation, Journal of Machine Learning Research 14 (1) (2013) 1605–1626.

[28] T. Christiani, R. Pagh, J. Sivertsen, Scalable and robust set similarity join, in: 34th IEEE International Conference on Data Engineering, 2018, pp. 1240–1243. doi:10.1109/ICDE.2018.00120.

[29] A. Arasu, V. Ganti, R. Kaushik, Efficient exact set-similarity joins, in: Proceedings of the 32Nd International Conference on Very Large Data Bases, VLDB Endowment, 2006, pp. 918–929.

[30] R. J. Bayardo, Y. Ma, S. Ramakrishnan, Scaling up all pairs similarity search, in: Proceedings of the 16th International Conference on World Wide Web, ACM, 2007, pp. 131–140. doi:10.1145/1242572.1242591.

[31] C. Xiao, W. Wang, X. Lin, J. X. Yu, Efficient similarity joins for near duplicate detection, in: Proceedings of the 17th International Conference on World Wide Web, ACM, 2008, pp. 131–140. doi:10.1145/1367497.1367516.

[32] L. A. Ribeiro, T. Härder, Generalizing prefix filtering to improve set similarity joins, Information Systems 36 (1) (2011) 62–78. doi:10.1016/j.is.2010.07.003.

[33] J. Wang, G. Li, J. Feng, Can we beat the prefix filtering?: An adaptive framework for similarity join and search, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 85–96. doi:10.1145/2213836.2213847.

[34] P. Bouros, S. Ge, N. Mamoulis, Spatio-textual similarity joins, Proceedings of the VLDB Endowment 6 (1) (2012) 1–12. doi:10.14778/2428536.2428537.

[35] W. Mann, N. Augsten, PEL: position-enhanced length filter for set similarity joins, in: Proceedings of the 26th GI-Workshop Grundlagen von Datenbanken, 2014, pp. 89–94.

[36] D. Deng, Y. Tao, G. Li, Overlap set similarity joins with theoretical guarantees, in: Proceedings of the 2018 International Conference on Management of Data, ACM, 2018, pp. 905–920. doi:10.1145/3183713.3183748.

[37] S. McCauley, J. W. Mikkelsen, R. Pagh, Set similarity search for skewed data, in: Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, ACM, 2018, pp. 63–74. doi:10.1145/3196959.3196985.

[38] X. Wang, L. Qin, X. Lin, Y. Zhang, L. Chang, Leveraging set relations in exact and dynamic set similarity join, The VLDB Journal 28 (2) (2019) 267–292. doi:10.1007/s00778-018-0529-2.

[39] W. Mann, N. Augsten, P. Bouros, An empirical evaluation of set similarity join techniques, Proceedings of the VLDB Endowment 9 (9) (2016) 636–647. doi:10.14778/2947618.2947620.

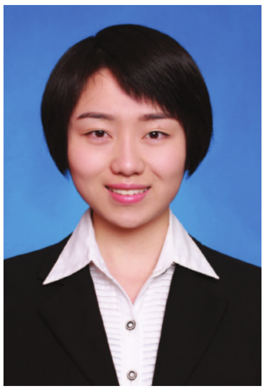
[40] C. Rong, C. Lin, Y. N. Silva, J. Wang, W. Lu, X. Du, Fast and scalable distributed set similarity joins for big data analytics, in: 33rd IEEE International Conference on Data Engineering, 2017, pp. 1059–1070. doi:10.1109/ICDE.2017.151.

[41] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Scientific Computing 20 (1) (1998) 359–392. doi:10.1137/S1064827595287997.

[42] P. Sanders, C. Schulz, Think locally, act globally: Highly balanced graph partitioning, in: V. Bonifaci, C. Demetrescu, A. Marchetti-Spaccamela (Eds.), Proceedings of the 12th International Symposium on Experimental Algorithms,

Vol. 7933 of Lecture Notes in Computer Science, Springer, 2013, pp. 164–175.
doi:10.1007/978-3-642-38527-8_16.

Zhaokang Wang received the BS degree in Nanjing University, China, in 2013. He is currently working towards the Ph.D. degree in Nanjing University. His research interests include large scale graph analysis algorithms and graph processing systems.



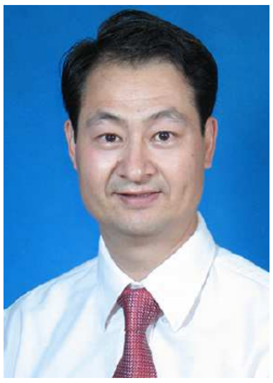
Shen Wang received the BS degree in Nanjing University, China, in 2014. She got the master degree in Nanjing University in 2017. Her research interests include large scale graph analysis algorithms and graph processing systems.

Junhong Li received the BS degree in University of Electronic Science and

Technology of China in 2018. He is currently working towards the Master degree in Nanjing University. His research interests include distributed computing , graph computing systems.

Chunfeng Yuan is currently a professor in the computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. She received her bachelor and master degrees in computer science from Nanjing University. Her main research interests include computer architecture, parallel and distributed computing and information retrieval.

Rong Gu is an assistant researcher at State Key Laboratory for Novel Software Technology, Nanjing University, China. Dr. Gu received the Ph.D. degree in computer science from Nanjing University in December 2016. His research interests include parallel computing, distributed systems and distributed machine learning.



Yihua Huang is currently a professor in computer science department and State Key Laboratory for Novel Software Technology, Nanjing University, China. He received his bachelor, master and Ph.D. degrees in computer science from Nanjing University. His main research interests include parallel and distributed computing, big data parallel processing, distributed machine learning algorithm and system, and Web information mining.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: