Effective Indexing for Dynamic Structural Graph Clustering

Fangyuan Zhang
The Chinese University of Hong Kong
fzhang@se.cuhk.edu.hk

Sibo Wang
The Chinese University of Hong Kong
swang@se.cuhk.edu.hk

ABSTRACT

Graph clustering is a fundamental data mining task that clusters vertices into different groups. The structural graph clustering algorithm (*SCAN*) is a widely used graph clustering algorithm that derives not only clustering results, but also special roles of vertices like hubs and outliers. In this paper, we consider structural graph clustering on dynamic graphs under Jaccard similarity. The state-of-the-art index-based solution focuses on static graphs and incurs prohibitive update costs to maintain indices. Lately, an efficient approximate dynamic structural graph clustering algorithm DynStrClu under Jaccard similarity is proposed. However, their solution needs to fix input parameters while parameter settings of SCAN usually need to be fine-tuned to achieve good clustering results.

Motivated by these limitations, we present a study on devising effective index structures for SCAN algorithm on dynamic graphs. Similar to the state-of-the-art dynamic scheme, our main idea to reduce the time complexity is still by bringing approximation to clustering results. However, our solution does not need to fix the input parameters. To achieve this, our solution includes two key components. The first is to maintain a bottom-k sketch for each vertex so that the similarities of affected vertices can be easily updated. The second key is a bucketing strategy that allows us to update clustering results and roles of vertices efficiently. Our theoretical analysis shows that our proposed algorithm achieves $O(\log n \cdot \log \frac{M+m}{p_f})$ expected update cost and guarantees to return approximate clustering results with probability $1-p_f$ after up to M updates. Extensive experiments show that our solution is up to two orders of magnitude faster than the state-of-the-art index-based solution while still achieving high-quality clustering results.

PVLDB Reference Format:
Fangyuan Zhang and Sibo Wang. Effective Indexing for Dynamic Structural Graph Clustering. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Graph is a fundamental type of data structure to represent objects and their relationships. Much real-world data can be represented as graphs such as online social networks, the Web network, protein interaction networks, and road networks. Graph clustering is one of the most important tasks on graph data analysis, which aims to divide the vertices into different groups where vertices within a group are densely connected while vertices in different groups are sparsely connected. Graph clustering finds many real-world applications, such as finding functional modules in metabolic networks [13], identifying communities on social networks [12], detecting research groups across collaborating networks [39], to name a few.

Structural graph clustering [35] is one of the well-known approaches to graph clustering and Xu et al. [35] present the first algorithm SCAN to solve this problem. The main idea of SCAN is that if two vertices are similar enough in terms of their neighborhood, then the two vertices tend to belong to the same cluster. Unlike other graph clustering algorithms that only find the clustering results, SCAN further finds the special roles of different vertices. For instance, it identifies hub vertices that connect different clusters and outliers that lack strong ties to any cluster. Given an undirected graph $G = \langle V, E \rangle$ with n vertices and m edges and a set similarity measure σ, e.g., the Jaccard similarity, the SCAN algorithm takes two input parameters: a similarity threshold ϵ and an integer μ. If v is a neighbor of vertex u (i.e., there exists an edge between u and v) and they have a similarity score $\sigma(u, v)$ no smaller than ϵ, then v is called an *ϵ-neighbor* of vertex u. If the number of ϵ-neighbors of vertex u is no smaller than μ, u is called a *core vertex*. Then, considering only the edges among core vertices with similarity above the threshold ϵ, we can construct different connected components, resulting in different clusters. For a vertex v that is not a core vertex but an ϵ-neighbor of core vertex(s), it is assigned to the cluster(s) of its ϵ-neighbor(s) that is/are core vertex(s). *Hub vertices* and *outliers* are further defined for unclustered vertices, depending on whether the vertex connects to multiple clusters. Different roles enrich the structural information and help us better understand networks.

It is shown that SCAN can help find meaningful clustering results in biological data [11, 17, 19, 20], web data [16, 22, 23, 26, 28, 29], and social networks [15, 24]. It can also be used for community detection [24], image segmentation [25], and fraud detection on blockchain data [6]. Due to its important applications, it attracts a plethora of research studies, e.g., [5, 30, 33], on improving the efficiency of the SCAN algorithm. As the SCAN algorithm needs to calculate the similarity among all neighbor vertex pairs, some existing research focuses on reducing the costs of similarity computations via pruning unnecessary pairs [5, 7, 30] or exploiting parallelism [7, 8, 18, 31, 32, 37, 38]. In the meantime, it is also pinpointed in existing research studies [4, 14, 34] that input parameters ϵ and μ need to be fine-tuned so as to achieve good clustering results. Thus, another direction is to devise effective indices, e.g., GS-Index [33, 34], so that clustering results and different roles of vertices can be more efficiently derived for different choices of ϵ and μ. Nevertheless, all these research works mainly consider static graphs while real-life graphs, like web graphs and social graphs, are dynamically changing. It is possible to compute the exact GS-Index [34] from scratch with multi-core parallelization [33]. However,

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

the $O(m^{1.5})$ construction cost is still prohibitive on huge graphs, resulting in unnecessarily long waiting time for clustering analysis.

Most recently, Ruan et al. [27] present a dynamic scheme *DynStrClu* for SCAN with the popular Jaccard similarity as the underlying set similarity measure. The expected update cost of DynStrClu is $O\left(\log^2 n + \log n \cdot \log \frac{M+m}{p_f}\right)$, which guarantees that the returned clustering result satisfies the approximation definition with probability $1 - p_f$ after up to M updates. However, the main issue of DynStrClu is that it needs to fix the two input parameters and only maintains the clustering result for such a fixed set of input parameters. It is difficult for DynStrClu to help us find good clustering results when inputting different parameters ϵ and μ. If we simply apply DynStrClu with various settings of ϵ and μ, the space consumption will explode as there are too many such combinations.

Motivated by limitations of existing solutions, we present an effective indexing scheme *BOTBIN (Bottom-k and bucket indexing)* for the SCAN algorithm on dynamic graphs under Jaccard similarity that are insensitive to the input parameters. Following the previous work [27], we consider the case when the updates are edge insertions/deletions as vertex insertion/deletions can be mapped to a series of edge insertions/deletions. Following [27], we consider unweighted graphs and focus on the Jaccard similarity, which is a widely adopted similarity for unweighted sets. As shown in [34], deriving an exact dynamic scheme to SCAN incurs $O(\min\{d_{max}^2, m\} \cdot \log n)$ update cost. To reduce the time complexity, we turn to approximate solutions, which relax the definition of ϵ-neighbors, and show that we can update the index with an expected cost of $O(\log n \cdot \log \frac{M+m}{p_f}))$ for each edge update. The returned clustering result satisfies the approximation definition with probability $1-p_f$ after up to M edge updates. Note that this result is non-trivial as our BOTBIN reduces a term of $\log^2 n$ in update cost compared to the state-of-the-art dynamic scheme DynStrClu, while at the same time it supports different choices of μ and ϵ.

To achieve above goals, our solution includes two key components. The first is a similarity index that can support efficient maintenance of similarity scores on dynamic graphs. Our similarity index includes a bottom-k sketch for each vertex so that we can derive the estimated similarities with approximation guarantees between neighboring vertices by inspecting the sketches of two vertices. We note that using bottom-k sketch to estimate Jaccard similarity is not a new idea. Our key contribution here is to dynamically maintain the bottom-k sketch after graph updates with bounded cost. The key observation is that the larger the number of neighborhood a vertex u has, the smaller the chance is that an edge update to (u, v) will affect the bottom-k sketch of vertex u. To the best of our knowledge, we are the first to make full use of this observation and combine it with the bottom-k sketch to reduce the time complexity of dynamic graph clustering algorithms. It may further inspire other research works to improve the time complexity of graph clustering or dynamic graph algorithms with similar issues. We show that the update cost can be bounded with $O(k^2)$ where $k = O(\log \frac{M+m}{p_f})$ is the size of the bottom-k sketch of each vertex and is independent of the degree of vertex u. Further observing that the update of an edge (u, v) affects at most two entries of the bottom-k sketch of vertex u, we present optimization techniques to reduce the update cost to $O(k \log k)$.

The second key component of our BOTBIN is an effective clustering index that helps return the clustering result given the input parameters ϵ and μ. Notice that even if we can efficiently maintain and update the similarity scores with a cost independent of the degree, it may still incur high update cost if we want to report the clustering results with a cost proportional to the size of the cluster subgraph (Ref. to Definition 6). For example, with the state-of-the-art GS-Index, even if we only change the similarity score of one vertex pair, it still incurs up to $O(d_{max} \cdot \log n)$ update cost (reasons explained in Section 2.2), where d_{max} is the maximum degree of the graph. To tackle this issue, the second key of BOTBIN is a bucketing strategy that maintains different buckets where each bucket is associated with a range and maintains a list of edges whose incident vertex pairs have a similarity score falling inside the associated range. When similarity scores are updated, it only needs to update edges to the correct bucket, which significantly reduces the maintaining overhead. We further present a detailed theoretical analysis and show that our index scheme can achieve an update cost of $O(\log n \cdot \log \frac{M+m}{p_f}))$, independent of d_{max}.

To summarize, our main contributions are as follows.

- We present effective similarity index and efficient update algorithms to dynamically maintain the estimated similarity scores.
- We propose an effective clustering index and efficient update algorithms to return clustering results with a cost proportional to the cluster subgraph given different input parameters μ and ϵ.
- Theoretical analysis shows that our dynamic index scheme achieves a cost of $O(\log n \cdot \log \frac{M+m}{p_f}))$ after each edge update and at the same supports different input parameters of μ and ϵ.
- Extensive experiments on real-world graphs with up to 3.9 billion edges show that our proposed scheme BOTBIN achieves up to two orders of magnitude improvement over the existing state-of-the-art solution without compromising the clustering quality.

2 PRELIMINARIES

2.1 Problem Definition

Notations in graphs. We consider an unweighted and undirected graph $G = (V, E)$, where V is the set of vertices and E is the set of edges in the graph. We use n to indicate the number of vertices and m to indicate the number of edges. An edge e connecting vertex u and v is denoted as an unordered pair of vertices, (u, v) or (v, u), and u and v are called a neighbor of each other or simply neighboring vertices. For a vertex $u \in V$, the *neighborhood* $N[u]$ consists of *(i)* the set of nodes who are neighbors of node u and *(ii)* u itself. The *degree* d_u of vertex u is defined as the number of neighbors of u.

Terminologies in SCAN. Given two neighboring vertices u and v, the similarity $\sigma(u, v)$ between u and v is defined as the set similarity between $N[u]$ and $N[v]$. We focus on the case when the elements in the set are equally weighted. In such a scenario, Jaccard similarity is adopted as the set similarity measure in existing studies, e.g., [5, 27, 33, 36]. The definition of Jaccard similarity is as follows.

Definition 1 (Jaccard Similarity). *Given two sets X and Y, the Jaccard similarity between these two sets is defined as $\frac{|X \cap Y|}{|X \cup Y|}$.*

In other words, given two neighboring vertices u and v, their similarity $\sigma(u, v) = |N[u] \cap N[v]|/|N[u] \cup N[v]|$. If u and v are

not neighboring vertices, their similarity is simply zero. In SCAN, the first input parameter is a threshold ϵ, which identifies the ϵ-neighbors, defined as follows.

Definition 2 (ϵ-neighbor). *Given a threshold $\epsilon \in (0, 1]$ of the similarity, v is called an ϵ-neighbor of u if $v \in N[u]$ and the similarity $\sigma(u, v)$ between u and v is no smaller than ϵ. The ϵ-neighbor set $N_\epsilon[u]$ is defined as the set of ϵ-neighbors of node u, i.e., $N_\epsilon[u] = \{v \in N[u] | \sigma(u, v) \geq \epsilon\}$.*

Given the definition of ϵ-neighbors, the second input parameter is an integer threshold $\mu \geq 2$, which is used to define *core vertices*.

Definition 3 (core vertex). *A vertex $u \in V$ is a core vertex if u has at least a number μ of ϵ-neighbors, i.e., $|N_\epsilon[u]| \geq \mu$.*

Definition 4 (Core graph). *Given the set V_{core} of core vertices, the core graph G_{core} consists of V_{core} and E_{core}, where E_{core} includes an edge (u, v) if and only if* (i) $(u, v) \in E$, (ii) $u, v \in V_{core}$, (iii) *u and v are ϵ-neighbors.*

Definition 5 (cluster). *Each connected component of G_{core} corresponds to a cluster. For a vertex v that is not a core vertex, if it is an ϵ-neighbor of a core vertex u, then it belongs to the cluster containing u. Notice that the non-core vertex v might be the ϵ-neighbor of multiple core vertices, then v will be assigned to multiple clusters.*

Definition 6 (Cluster Subgraph). *Given a clustering result C, the cluster subgraph is a subgraph $G_C = (V_C, E_C)$ of the input graph G where $v \in V_C$ iff* (i) *v is a core vertex or ϵ-neighbor of a core vertex;* (ii) *an edge $(u, v) \in E_C$ iff $u, v \in V_C$ and at least one is a core vertex.*

Definition 7 (Hubs and outliers). *For a vertex v that is not in any clusters, if it has neighbors belonging to multiple clusters, then it is a* hub vertex. *Otherwise, it is an* outlier.

Example 1. Figure 1 shows the clustering result of the input graph G_1 when $\epsilon = 0.5$ and $\mu = 5$. The similarity for every pair of adjacent vertices is shown close to each edge. Vertices v_3 and v_8 are core vertices, which are filled with black colors. The core-graph then consists of only two isolated vertices v_3 and v_8. The clustering result $C = \{C_1, C_2\}$ consists of two clusters where $C_1 = \{v_1, v_2, v_3, v_4, v_5\}$ and $C_2 = \{v_6, v_7, v_8, v_9, v_{10}\}$, as v_1, v_2, v_4, v_5 are the ϵ-neighbors of v_1 and v_6, v_7, v_8, v_9 are the ϵ-neighbors of v_8. The cluster subgraph $G_C = (V_C, E_C)$ then includes v_1 to v_{10}, i.e., $V_C = \{v_1, v_2, \cdots, v_{10}\}$ as they are either core vertices or the ϵ-neighbors of core vertices. The edge set $E_C = \{(v_1, v_3), (v_2, v_3), (v_4, v_3), (v_5, v_3), (v_6, v_8), (v_7, v_8), (v_9, v_8), (v_{10}, v_8))\}$ as at least one of the vertices in an edge must include a core vertex and both vertices of an edge must belong to V_C. There are four un-clustered vertices where v_{11} is a hub vertex as it has neighbors in different clusters, while vertices v_{12}, v_{13}, and v_{14} are outliers according to the definition. □

As mentioned in Section 1, it is important to tune the two input parameters to find good clustering results. Therefore, some existing studies, e.g. [34], present effective index structures which output the clustering results with a cost proportional to the size of result subgraphs. However, such solutions cannot handle index updates efficiently on dynamic graphs. To tackle this issue, we will bring ρ-approximation into the clustering results where ρ is used to balance the trade-off between the update efficiency and clustering quality.

ρ-approximate SCAN. Following previous work [33], we consider ρ-approximate SCAN where we relax the definition of ϵ-neighbor. In particular, we define the (ϵ, ρ)-neighbor as follows.

Definition 8 ((ϵ, ρ)-neighbor). *Given an error parameter ρ and two vertices u and v,*

- *if $\sigma(u, v) > \epsilon + \rho$, u must be an (ϵ, ρ)-neighbor of node v;*
- *if $\sigma(u, v) < \epsilon - \rho$, u must not be an (ϵ, ρ)-neighbor of node v;*
- *otherwise, it does not matter. In other words, u is allowed to be either an (ϵ, ρ)-neighbor of v or not.*

Notice that, the definition here uses the ρ-absolute error. However, it is easy to convert it to ρ'-relative error by setting $\rho = \epsilon \cdot \rho'$. In ρ-approximate SCAN, the core vertices are defined according to the number of (ϵ, ρ)-neighbors. It is worth noting that the clustering results in ρ-approximate SCAN are not unique due to the uncertainty of the "does not matter" cases in the definition of (ϵ, ρ)-neighbors. However, when ρ is small enough, the ϵ-neighbors and (ϵ, ρ)-neighbors of the same node usually do not differ much. Moreover, assume that we have the clustering result $C_{\epsilon+\rho,\mu}$ of SCAN with similarity threshold to be $\epsilon + \rho$ and the clustering result $C_{\epsilon-\rho,\mu}$ with similarity threshold to be $\epsilon - \rho$. Then the ρ-approximate SCAN clustering results, dubbed as $C^\rho_{\epsilon,\mu}$, have the following guarantee.

Theorem 1 (Guarantee of ρ-approximate SCAN). *For any clustering result $C^\rho_{\epsilon,\mu}$ outputted by ρ-approximate SCAN, we have:*

- *For every cluster $C_{\epsilon+\rho} \in C_{\epsilon+\rho,\mu}$, there exists a cluster $C \in C^\rho_{\epsilon,\mu}$ such that $C_{\epsilon+\rho} \subseteq C$;*
- *For every cluster $C \in C^\rho_{\epsilon,\mu}$, there exists a cluster $C_{\epsilon-\rho} \in C_{\epsilon-\rho,\mu}$ such that $C \subseteq C_{\epsilon-\rho}$.*

The proof of the above theorem is omitted as it directly follows from the definitions of $C_{\epsilon+\rho,\mu}$, $C_{\epsilon-\rho,\mu}$, and $C^\rho_{\epsilon,\mu}$.

SCAN on dynamic graphs. We consider dynamic graphs where the input graph is updated with edge insertions/deletions. Note that node insertions/deletions can be converted to a series of edge insertions/deletions. In dynamic graph setting, we aim to maintain an indexing scheme so that we can return the clustering result with a cost proportional to the size of the cluster subgraph. However, even with the state-of-the-art indexing scheme for the exact solution, it still takes $O(\min\{d^2_{max}, m\} \cdot \log n)$ update cost for each edge insertion/deletion. Thus, we focus on maintaining effective index structures for ρ-approximate SCAN, defined as follows.

Problem definition. Given a number $M = O(m)$ of edge updates, the dynamic indexing scheme aims to maintain an effective index with $O(polylog(n))$ cost for each edge update and can output the clustering result (with a cost bounded by the size of the cluster subgraph) satisfying the definition of ρ-approximate SCAN with probability $1 - p_f$, where p_f is used to bound the failure probability.

2.2 Existing Solutions Revisited

SCAN and pSCAN. The problem of structural graph clustering is first presented in [35], where the authors propose the SCAN algorithm to solve this problem. The main idea of SCAN is to calculate the similarity $\sigma(u, v)$ of each edge(u, v) and then scan all vertices in G to obtain the correct clustering results for the given input parameters ϵ and μ. The time complexity of the SCAN algorithm is

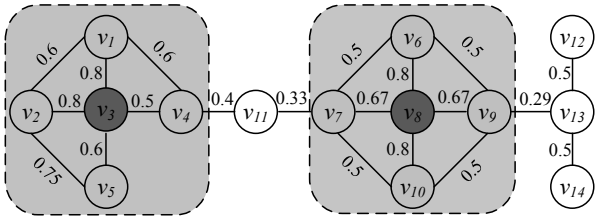


Figure 1: Clustering result with $\epsilon = 0.5, \mu = 5$ for graph G_1

$O(\alpha m)$, where α is the arboricity of graph G, which is also equal to the minimum number of forests that needs to cover all edges of graph G. The worst case running time is $O(m^{1.5})$.

The major cost of SCAN is the computation of set similarities between neighbors. Chang et al. [5] propose *pSCAN* to reduce the cost of calculating the set similarity between neighbors with pruning techniques on identifying ϵ-neighbors and core vertices. With these pruning strategies, pSCAN returns exact results much faster than SCAN, but still has $O(m^{1.5})$ worst-case running time.

GS-Index. The above solutions focus on online calculations for all clusters given the two input parameters ϵ and μ. However, as we mentioned, the two input parameter settings of SCAN usually need to be fine-tuned to achieve good clustering results. Thus, we may need to process SCAN multiple times with different input parameters. It is observed that when we fix one parameter and decrease the other parameter, the new clustering result must be included in the previous clustering result. Based on this observation, Wen et al. [34] propose GS-Index, an index-based method that quickly returns query results by maintaining a *neighbor-order* and a *core-order* index. In the neighbor-order index, each node v maintains a list $L[v]$ of neighbors sorted in decreasing order of the similarity score. Clearly, the neighbor-order index takes $O(m + n)$ space cost. By maintaining the neighbor-order index, given an arbitrary similarity threshold ϵ, each node can quickly identify ϵ-neighbors by scanning $L[v]$ and stop as soon as it reaches a neighboring node with similarity score smaller than ϵ. For the core-order index, it maintains d_{max} ordered set $C[2], \cdots, C[d_{max} + 1]$ where d_{max} is the maximum degree in the input graph. For each vertex v, it appears in $C[2], C[3], \cdots, C[d_v + 1]$. More specifically, a pair (v, ϵ_i) is in ordered set $C[i]$ where $2 \le i \le d_v + 1$ and ϵ_i is the largest threshold such that v has i ϵ-neighbors. This can be derived by computing the similarity scores between v and vertices in its neighborhood $N[v]$ and then taking the i-th largest similarity score. The pairs in the ordered set $C[i]$ are then sorted in descending order of their ϵ_i thresholds. It is easy to verify that the core-order index still takes $O(m + n)$ space. Given the GS-Index and input parameters ϵ and μ, it outputs clusters as follows. Firstly, it goes through the ordered set $C[\mu]$ in the core-order index and returns the set of vertices whose ϵ_μ score is no smaller than ϵ. These vertices are the core vertices. Then, it scans the neighbor-order index of each core vertex, and includes all neighboring vertices with a similarity score no smaller than ϵ. This can be bounded with a running time proportional to the size of the cluster subgraph (Ref. to Definition 6).

GS-Index also supports dynamic updates. Given an edge update to (u, v), it incurs an update cost of $O\left(|E_{2hop}(u)| + |E_{2hop}(v)|\right)$, where $E_{2hop}(x)$ is the set of edges that both ends of vertices in the edge belong to the vertex set $V_{2hop}(x)$, consisting of vertices with distance to x no larger than 2. In the worst case, this incurs $O(\min\{d_{max}^2, m\} \cdot \log n)$ running cost for each update. On one hand, it needs to handle the updates of the similarity scores due to the insertion/deletion of an edge (u, v), which affects the similarity score of up to $d_u + d_v$ neighboring vertex pairs. This incurs $O(d_{max})$ update cost if it takes $O(1)$ cost to check if u (resp. v) exists in the neighbor of vertex w, where w is a neighbor of v (resp. u). On the other hand, it needs to update the neighbor-order and core-order to support efficient clustering for arbitrary input parameters. When the similarity of u and one of its neighbors is changed due to the insertion of an edge, it may change up to $O(d_u)$ ordered sets in the core-order index. There are d_{max} updates to each ordered set in the core-index in the worst case. To support efficient update to the ordered sets of the core-index, a binary search tree is maintained [34]. This incurs $O(d_{max} \cdot \log n)$ running cost to update the core-index for each affected similarity score. Thus, GS-Index takes $O(\min\{d_{max}^2, m\} \cdot \log n)$ cost to update the index in the worst case.

DynStrClu. The dynamic property of real-world graph data brings new challenges to structural graph clustering. To reduce the update cost, Ruan et al. [27] propose *DynStrClu*, an approximate solution that relax the definition of ϵ-neighbors to (ϵ, ρ)-neighbors (Ref. to Definition 8) and reduce the amortized update cost to $O\left(\log^2 n + \log n \cdot \log\left(M/p_f\right)\right)$ for each update. It guarantees that clustering results fulfill the approximation definition with a probability of $1 - p_f$ after up to M updates. DynStrClu considers the case that two input parameters ϵ and μ are fixed. When an edge (u, v) is updated, we only need to check if *(i)* u and v are (ϵ, ρ)-neighbors, *(ii)* if the (ϵ, ρ)-neighbor relationships between the neighbors of u (resp. v) and u (resp. v) have changed. The impact of an insertion/deletion of one edge on the similarity $\sigma(u, v)$ of other edges can be limited to $1/d_{max}(u, v)$, where $d_{max}(u, v) = max\{d_u, d_v\}$. Combining with the definition of (ϵ, ρ)-neighbors, DynStrClu includes a lazy update mechanism that can afford multiple updates for each edge as long as their (ϵ, ρ)-neighbor relationship does not change. When the (ϵ, ρ)-neighbor relationship of an edge (u, v) is unclear given the bounds, DynStrClu recalculates the similarity $\sigma(u, v)$ with a sampling-based method. Combining both techniques, DynStrClu achieves the improved time complexity.

A major issue with DynStrClu is that it can only maintain clustering results under fixed parameters of ϵ and μ, but cannot return clustering results under new input parameters.

3 OUR SOLUTION

Next, we present our proposed index scheme *BOTBIN* in details. There are two main challenges in maintaining an effective index structure to support dynamic (approximate) structural graph clustering. The first main challenge is the insertion/deletion of an edge (u, v) will result in the change of similarity scores of up to $d_u + d_v$ neighbor pairs. To tackle this issue, we maintain a bottom-k sketch for each vertex. The rough idea is that we generate a random number for each node, and then for each vertex u, we maintain the k min values of the neighbors of vertex u. We then use these bottom-k sketches of two vertices u and v to calculate the approximate similarity score. The advantage of our design is that, an edge insertion/deletion will affect the bottom-k sketch with bounded

probability. We show that our solution can update the affected similarity scores with $O(k \cdot \log k)$ expected running time, where $k = O(\frac{1}{\rho^2} \cdot \log \frac{m+M}{p_f})$, independent of the degree of u and v.

Another challenge is to maintain an effective index structure on dynamic graphs so that it can return the clustering results with a cost proportional to the size of the cluster subgraph. Recap that given an updated similarity score, GS-Index will result in the change of all the ordered sets in the core-index in the worst case. To tackle this issue, we present the bucket index, which is built with a bucketing approach which maintains δ evenly split buckets. The discussion of δ will be elaborated more in Section 3.1. In bucket i ($1 \leq i \leq \delta$), it maintains an ordered set $B[i]$ which consists of a set of pairs $(u, |N_{1-i/\delta,\rho}[u]|)$, where $N_{1-i/\delta,\rho}[u]$ is the set of (ϵ, ρ)-neighbors of u when $\epsilon = 1 - i/\delta$. In each ordered set $B[i]$, it is sorted based on the number of (ϵ, ρ)-neighbors of each node when $\epsilon = 1 - i/\delta$. Then, given an updated similarity score, it will affect at most $O(\delta)$ ordered list, which can be regarded as a constant that is independent of the graph parameter. Moreover, given an edge update (u, v), we can further bound the probability that the similarity score between u and a neighbor of u gets affected. This together reduces the expected update cost of our bucketing index to $O(\log n \cdot \log \frac{m+M}{p_f})$.

Next, we first present our BOTBIN index scheme in Section 3.1. Then, we show how to build the index and handle query processing with BOTBIN in Section 3.2. We will present how to update the index when the graph dynamically changes in Section 4.

3.1 Index scheme

Our BOTBIN includes two indices: the *similarity index* and the *clustering index*. The similarity index is mainly used to derive and maintain the Jaccard similarity of adjacent vertices efficiently by exploiting the bottom-k sketch. The clustering index is mainly used to maintain the core vertices and ϵ-neighbors of core vertices efficiently with the bucketing idea for an arbitrary ϵ and μ.

Similarity index. Deriving the approximate similarity score for adjacent vertex pairs are well studied in the literature. For example, Tseng et al. [33] propose to explore k-min hash to estimate the Jaccard similarity score between adjacent vertex pairs. In particular, for every vertex $v \in G$, it first generate a random number $r_v \in (0, 1)$ as the hash value of v. Then, for vertex v, it maintains the min hash value $r_{min}(v)$ among $N[v]$, the neighborhood of vertex v. Then, the Jaccard similarity $\sigma(u, v)$ between adjacent vertices u and v is:

$$\sigma(u, v) = \Pr[r_{min}(u) = r_{min}(v)].$$

Let X be an indicator random variable where it takes 1 if $r_{min}(u) = r_{min}(v)$ and 0 otherwise. Then, $E[X] = \sigma(u, v)$. With this unbiased estimator, we may derive multiple instances of X by sampling k different hash values for each vertex and generate a variable X_i according to the i-th hash value of each vertex. It is shown in [33] that by setting $k = O(\log n/\rho^2)$, it can provide ρ-absolute error to the similarity score. However, a main deficiency of the k-min hash-based solution is that it takes $O(n \cdot k)$ space while k is usually much larger than the average degree of the input graph. For example, consider a social network Orkut with 3 million nodes and 0.2 billion edges. To provide an absolute error of 0.05 to the estimation of the similarity score, it needs to set $k \approx 6000$ and every vertex needs to maintain k min-hash values. This incurs an additional 90 times space (of the input graph), which is too expensive for large graphs.

To tackle this issue, we adopt the bottom-k sketch [9], rather than the popular k-min hash-based solution, to estimate the Jaccard similarity scores. The advantage of the bottom-k sketch is that, instead of maintaining k values for each vertex v, it only needs to maintain $\min\{k, d_v + 1\}$ values. This bounds the space consumption of the bottom-k sketch to be the same as that of the input graph, saving far more space than the k-min hash-based solution. Next, we elaborate more details on our bottom-k sketch-based similarity index. We first define the bottom-k sketch as follows.

Definition 9 (Bottom-k sketch). *Let U be the universe of the elements that may appear in a set S. Let $h : U \to [1 \cdots |U|]$ be a hash function that is a random permutation of U. The bottom-k sketch of S is the set of k smallest hash values among elements in S. If $k > |S|$, it consists of the set of hash values of all elements in S.*

In our problem, given a vertex v, the set $S = N[v]$ is the neighborhood of node v and the universe U is equal to V, the entire vertex set in the input graph G. The similarity index of a node v thus consists of the bottom-k sketch of its neighborhood $N[v]$. As shown in the following lemma, the bottom-k sketch can be further used as an unbiased estimator of the Jaccard similarity. Notice that here we assume that U is sufficiently large and thus we generate enough random numbers at the beginning for vertices.

Lemma 1 (Bottom-k Estimator of Jaccard similarity). *Given two sets $A \subseteq U$ and $B \subseteq U$, let $S_k(A)$ and $S_k(B)$ denote their bottom-k sketches, respectively. If $|A| \geq k$ or $|B| \geq k$, then,*

$$\hat{J}(A, B) = \frac{S_k(A) \cap S_k(B) \cap S_k(A \cup B)}{k} \tag{1}$$

is an unbiased estimator of the Jaccard similarity $J(A, B)$ of A and B.

Here we restrict that either A or B has a size no smaller than k to use Equation 1 to estimate the Jaccard similarity. Otherwise, we can directly compute the Jaccard similarity between A and B with $O(k)$ cost. Although there has been a lot of research work on bottom-k sketch, as far as we know [9, 10], there is no explicit bound for the bottom-k sketch. Dahlgaard et al. [10] focus on the case when the set has no more than k elements and show how to add more entries until there are k elements in the sketch to provide approximation guarantee. In such a scenario, they will face a similar issue as that in k-min hash. For self-completeness, we present the approximation bound with bottom-k sketch and give a formal proof. In k-min hash sketch, it can be derived by sampling with replacement and thus different hash values are independent of each other. Therefore, the estimation can directly apply the Hoeffding inequality to derive a bounded error. However, with the bottom-k sketch, it is derived via sampling without replacement and thus existing concentration bound cannot be directly applied to derive a bounded error. Our proof of Theorem 2 makes a careful connection of the bottom-k sketch problem to the martingale process.

Theorem 2. *Given two sets A and B from a universe U, let $S_k(A)$ and $S_k(B)$ denote their bottom-k sketches, respectively. Given a failure probability p and an error parameter ρ, by setting $k = \frac{1}{2\rho^2} \ln \frac{2}{p}$, we have that $\Pr[|\hat{J}(A, B) - J(A, B)| > \rho] \leq p$.*

We note that our estimation framework can be easily extended to other unweighted set similarity measures, like Dice similarity, another popular similarity measures adopted in existing structural graph clustering algorithms, e.g., [5], for unweighted graphs.

Given Theorem 2, our similarity index is constructed as follows. Firstly, we generate a random permutation h of V. Then, according to h, the similarity index $S_k(v)$ of a vertex v consists of the bottom-k sketch of set $N[v]$. The choice of k is determinate according to Theorem 2 and more discussion will be given. The time complexity to build the similarity index can be bounded with $O(m+n)$. The space cost can be further bounded by $O(\min\{k \cdot n, n+m\})$ as each vertex maintains $\min\{k, 1+d_v\}$ hash values.

Example 2. Still consider the graph G_1 in Figure 1. Assume that the random permutation h is $\{9, 14, 8, 2, 4, 1, 7, 5, 3, 6, 12, 13, 11, 10\}$. In other words, $h(v_1) = 9, h(v_2) = 14, h(v_3) = 8$, etc. Assume that $k = 3$. Then, the similarity index $S_k(v_3)$ of v_3 is $\{2, 4, 8\}$, since $h(v_1), h(v_2), h(v_3), h(v_4), h(v_5)$ are 9, 14, 8, 2, 4, respectively, and 2, 4, and 8 are the 3 smallest hash values. Similarly, the similarity index $S_k(v_2)$ of v_2 is $\{4, 8, 9\}$. Then the estimated similarity $\hat{\sigma}(v_2, v_3)$ between v_2 and v_3 is $2/3$ according to Equation 1.

Extension to Dice similarity. Our estimation scheme for Jaccard similarity can be easily extended to Dice similarity. In particular, for Dice similarity $\sigma_{Dice}(X, Y)$ of two sets X and Y, we have:

$$\sigma_{Dice}(X,Y) = \frac{2|X \cap Y|}{|X| + |Y|} = \frac{2\sigma(X,Y)}{1+\sigma(X,Y)} = 2 - \frac{2}{1+\sigma(X,Y)}, \quad (2)$$

where $\sigma(X, Y)$ denotes the Jaccard similarity between X and Y. Then, if we provide an ρ-absolute error guarantee for the Jaccard similarity with our proposed method, then based on Equation 2, we can derive that the Dice similarity can provide an 2ρ-absolute error guarantee as well. Here, we show how to derive $\sigma_{Dice}(X, Y) - \hat{\sigma_{Dice}}(X, Y) < 2\rho$ and the other side of inequality can be derived in a similar manner. Given the estimation $\hat{\sigma}(X, Y)$ based on Equation 1 and with an estimation error bound of ρ, we use Equation 2 to derive the estimation $\hat{\sigma}_{Dice}(X, Y)$ as:

$$\begin{aligned}\hat{\sigma}_{Dice}(X,Y) &= 2 - \frac{2}{1+\hat{\sigma}(X,Y)} \leq 2 - \frac{2}{1+\sigma(X,Y)+\rho} \\ &\leq 2 - \frac{2}{1+\sigma(X,Y)} + 2\rho = \sigma_{Dice}(X,Y) + 2\rho.\end{aligned}$$

In our experiment, we will further examine our effectiveness to Dice similarity in terms of the quality of the clustering results.

Clustering index. Next, we elaborate on the details of the clustering index of the proposed BOTBIN. The clustering index is mainly used to quickly return the ρ-approximate SCAN (Section 2.1) given the input ϵ and μ. Note that our clustering index is inspired by the GS-Index. However, we tackle the challenging issue with dynamic update on GS-Index as pinpointed at the beginning of Section 3.

The key observation is that given an arbitrary input ϵ, if a vertex u is the (ϵ, ρ)-neighbor (Ref. to Definition 8) of vertex v, given a smaller input $\epsilon' < \epsilon$, then vertex u is still the (ϵ, ρ)-neighbor of vertex v. Thus, to find the (ϵ, ρ)-neighbors of each vertex v efficiently, for each vertex v, BOTBIN maintains a sorted set $NO[v]$, dubbed as the *neighboring index*, of its neighborhood $N[v]$ in the form of pairs $\langle v, \hat{\sigma}(u, v)\rangle$, and the pairs are sorted in descending order of $\hat{\sigma}(u, v)$, where $u \in N[v]$. Note that, the estimated similarity is derived via Equation 1. This obviously takes a space proportional to the graph size $O(n+m)$. Next, we elaborate on the *bucket index* which is used to find the core vertices efficiently.

Algorithm 1: BOTBIN-Index-Construction

Input: Input graph $G = (V, E)$, a random permutation h
Output: BOTBIN-Index of G

```
for vertex v ∈ V in increasing order of their h(v) values do
    update S_k(u) with u ∈ N[v];
for vertex v ∈ V do
    for vertex u ∈ N[v] such that h(u) > h(v) do
        Derive the estimated similarity σ̂(u, v) of u and v
          with S_k(u), S_k(v) via Equation 1;
        Add ⟨u, σ̂(u, v)⟩ into sorted set NO[v];
    for i from δ to 1 do
        cnt ← |{w | w ∈ N[v], σ̂(v, w) ≥ 1 − i/δ}|;
        if cnt == 1 then break;
        Add ⟨v, cnt⟩ into bucket B[i];
return S_k, B, NO;
```

The bucket index takes δ as the number of the buckets and then divide the similarity score into δ evenly partition ranges: $[1-1/\delta, 1], [1-2/\delta, 1-1/\delta), \cdots [1-i/\delta, 1-(i-1)/\delta) \cdots, [0, 1/\delta)$. For the i-th bucket $[1-i/\delta, 1-(i-1)/\delta)$, it maintains a sorted set $B[i]$ of pairs $(u, |N_{1-i/\delta,\rho}[u]|)$, where $N_{1-i/\delta,\rho}[u]$ is the set of (ϵ, ρ)-neighbors of u when $\epsilon = 1 - i/\delta$. For the pairs stored in bucket i, they are sorted in descending order of the number of (ϵ, ρ)-neighbors of each vertex with $\epsilon = 1 - i/\delta$. For the sorted set $B[i]$, we use a binary search tree to maintain the sorted pairs to support efficient updates and searches. Since each vertex appears at most δ times in the bucket index, the space consumption of the bucket index is bounded by $O(\delta \cdot n)$.

Given the bucket index, it can efficiently identify the set of core vertices. In particular, given input parameters ϵ and μ, we first find the bucket i^* whose range covers ϵ, i.e., $\epsilon \in [1-i^*/\delta, 1-(i^*-1)/\delta)$. Then, with bucket $B[i^*]$, it traverses the entries in decreasing order of the $(1-i^*/\delta, \rho)$ -neighbors and identifies the set of nodes with the size of $(1-i^*/\delta, \rho)$-neighbors no smaller than μ as core vertices. This takes a cost proportional to the number of core vertices.

Example 3. Still consider the graph G_1 in Figure 1. Assume that the estimated similarity scores are the same as the exact similarity scores. Given $\delta = 5$, the corresponding bucket index is shown in Figure 2. Then, given $\mu = 5$ and $\epsilon = 0.5$, we use the bucket index to first find the core vertices. In particular, $0.5 \in [0.4, 0.6)$ and thus we use $B[3]$ to find the core vertices. Then, in $B[3]$, the set of vertices that has the $(1-i^*/\delta, \rho)$-neighbor no smaller than $\mu = 5$ consists of $\{v_3, v_8\}$. Then, v_3 and v_8 are returned as the core vertices.

Clearly, with the bucket index, there might exist some loss of the approximation guarantee. We provide the following theorem for the approximation guarantee with our BOTBIN index scheme.

Theorem 3 (Approximation guarantee with bucket index). *Given the bucket size δ and the ρ-absolute approximation guarantee*

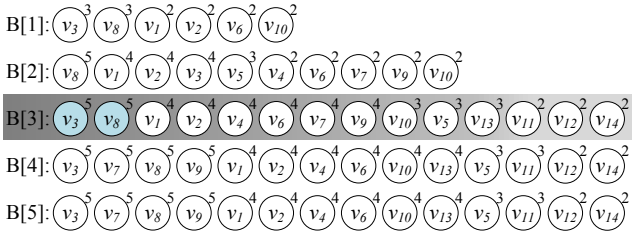


Figure 2: The bucket index with $\delta = 5$

of the estimated similarity score provided by the similarity index, the returned clustering result provides a $(\rho + 1/\delta)$-approximate SCAN.

3.2 Index Construction and Query Processing

Index construction. Algorithm 1 shows the pseudo-code of the BOTBIN index construction algorithm. Given the hash function h generated via a random permutation, BOTBIN builds the similarity index from the vertex with the smallest hash value. Then, the similarity index can be constructed easily with $O(n + m)$ cost (Lines 1-2). After building the similarity index of each vertex, we can calculate approximate similarity of any neighboring node pairs with a running cost of $O(\frac{1}{\rho^2} \log \frac{1}{p_f})$ (Line 5). In total, it takes $O(\frac{m}{\rho^2} log \frac{1}{p})$ cost to compute the similarities. Then it costs $O(\log d_u)$ time to add $\langle v, \hat{\sigma}(u, v)\rangle$ into sorted set $NO[u]$ as shown in Line 6. In total, to build the neighboring index, it takes $O(\sum_{v \in V} d_v \log d_v)$ cost, which can be further bounded by $O(m \log d_{max})$.

After constructing the neighboring index for vertex v, BOTBIN inserts entries about v in the bucket index (Lines 7-9). This can be finished via a linear scan of the neighboring index of v, which takes $O(d_v)$ cost. For the bucket index, notice that each vertex v maintains $O(\delta \cdot \sigma_{max}(v))$ entries among δ buckets, where $\sigma_{max}(v)$ is the maximum similarity score between v and its neighbors. To explain, there is no vertex with similarity larger than $\sigma_{max}(v)$ (except vertex v itself). Thus, for a bucket whose corresponding range is a subset of $(\sigma_{max}(v), 1]$, it will not contain vertex v. Let $\bar{\sigma}_{max}$ denote the average of this maximum similarity score, i.e., $\bar{\sigma}_{max} = \frac{1}{n} \sum_{v \in V} \sigma_{max}(v)$. Then, the bucket index takes $O(\delta \cdot n \cdot \bar{\sigma}_{max})$ space in total. For each entry in the bucket index, it takes $O(\log n)$ cost to do the insertion. In total, the running cost to build the bucket index can be bounded by $O(m + \delta \cdot n \cdot \bar{\sigma}_{max} \log n)$.

According to the above discussions, we have the following theorem about the space cost and running cost of the index construction.

Theorem 4. *Given an error parameter ρ, a failure probability p, BOTBIN takes $O\left(m\left(\frac{1}{\rho^2} \ln \frac{1}{p} + \log d_{max}\right) + \delta \cdot n \cdot \bar{\sigma}_{max} \log n\right)$ running cost to build the index and has a space cost of $O(m + \delta \cdot n \cdot \bar{\sigma}_{max})$.*

For GS-Index, it takes a worst case of $O(m^{1.5})$ time to build the index and its space cost is $O(m + n)$. Clearly, our proposed BOTBIN is more scalable in terms of index construction time complexity. In terms of space cost, both methods has a space cost linear to the input graph. As we will show in our experiment, our BOTBIN may have slightly smaller memory consumption compared to GS-Index (Figure 7) with our default choice of δ. To explain, it is observed that on almost all datasets, $\delta \cdot \bar{\sigma}_{max}$ is actually smaller than the average degree of all the tested graphs on our default choice of δ.

Algorithm 2: BOTBIN-Query

Input: Parameters ϵ and μ, BOTBIN index
Output: the set C of clustering result

1 $i^* \leftarrow \arg_i\{\epsilon \in [1 - i/\delta, 1 - (i-1)/\delta)\}$;
2 $C \leftarrow \emptyset, H \leftarrow$ an empty hash table;
3 **for** $\langle v, cnt\rangle \in B[i^*]$ *in descending order of the cnt values* **do**
4 **if** $cnt < \mu$ **then** break ;
5 Add v to H;
6 Init Q as an empty queue, $visit$ a size n array with all zeros;
7 **for** $v \in H$ **do**
8 **if** $visited[v]$ **then** continue;
9 $C \leftarrow \{v\}$, add v into Q, $visit[v] \leftarrow 1$;
10 **while** Q *is not empty* **do**
11 $u \leftarrow Q.dequeue()$;
12 **for** $\langle w, \hat{\sigma}(w, u)\rangle$ *in sorted set* $NO[u]$ **do**
13 **if** $\hat{\sigma}(w, u) < \epsilon$ **then** break;
14 $C \leftarrow C \cup \{w\}$;
15 **if** $visit[w]$ **then** continue;
16 $visit[w] \leftarrow 1$;
17 **if** $w \in H$ **then** $Q.enqueue(w)$;
18 $C \leftarrow C \cup \{C\}$;
19 return C;

Query processing. Algorithm 2 shows the pseudo-code of the query algorithm. Given the input parameter ϵ and μ, the BOTBIN index, it firstly finds the bucket i^* that covers ϵ (Line 1). Next, it initializes the clustering result C and maintains a hash table H of the core vertices. As described in Section 3.1, it visits bucket $B[i^*]$ in descending order of the size of their $(1 - i^*/\delta, \rho)$-neighbors and stops as soon as the number is smaller than μ. The core vertices are added to the hash table H (Lines 3-5).

Next, it initializes a queue Q and a size n array $visit$ for the cluster generation. The $visit$ array is used to identify the core vertices that have been traversed and added to previous clusters (Line 6). Then, for each core vertex v, it first checks if v has been traversed and added to previous clusters. If the answer is yes, it then continues to the next core vertex. Otherwise, it traverses the graph to generate the clusters corresponding to core vertex v. In particular, it adds v to the cluster C, adds v to Q, and sets $visit[v]$ to be 1. Then, it proceeds a pruned BFS where only neighbors who are core vertices are added to the queue (Lines 11-17). Firstly, it finds the front element u of queue Q. Then, it visits all its neighbors in descending order of the estimated similarity scores via the neighboring index $NO[u]$. If the neighbor w of vertex u has a similarity no smaller than ϵ, w is an (ϵ, ρ)-neighbor of u. Thus, w is added into cluster C (Line 14). If w has been visited before, the pruned BFS continues (Line 15). Otherwise, it sets $visit[w]$ to be 1 (Line 16). Next, w is added to the queue if w is not visited in previous iterations (Line 15) and is a core vertex (Line 17). The iteration stops when the queue is empty. Cluster C is then added to the clustering result C. When all core vertices finish pruned BFSs, it returns C as the final result.

Clearly, the running cost is proportional to the number of vertices and edges visited during the pruned BFS traversals. It is easy to

verify that this cost can be easily bounded by the size of the cluster subgraph (Definition 6) as we will visit each core vertex at most and the (ϵ, ρ)-neighbor of each core vertex at most once. Theorem 5 summarizes the running cost for query processing.

Theorem 5. *The time complexity of Algorithm 2 is bounded by $O(|V_C| + |E_C|)$ where V_C and E_C are the vertex set and edge set, respectively, of the cluster subgraph of the final clustering result C.*

Compared to GS-Index, our BOTBIN achieves the same query time complexity that depends on the size of cluster subgraph, i.e., $O(|V_C| + |E_C|)$. As we will show in the experiment, our BOTBIN achieves an identical query performance as that of GS-Index.

4 DYNAMIC INDEX MAINTENANCE

Next, we present a basic solution to handle dynamic updates in Section 4.1. To further improve the performance and reduce the time complexity, we present our improved scheme in Section 4.2.

4.1 Basic methods

As we mentioned in Section 1, we focus on the case when the updates are edge insertions/deletions as node insertions/deletions can be mapped to a list of edge updates. Next, we discuss how to handle edge insertions and edge deletions one by one.

Edge insertion. Consider inserting an edge (u, v), we first discuss the effects after adding this new edge. Firstly, v (resp. u) is added to the set $N[u]$ (resp $N[v]$) of the neighborhood of vertex u (resp. v). Therefore, the similarity index of vertex u (resp. v) might be changed. Thus, the first step is to update the similarity index of u and v if necessary. Next, if the similarity index of vertex u (resp. v) is not changed, then the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$) before the v (resp. u) is added is also unchanged. No further update is required. On the other hand, if the similarity index gets updated, then further updates are required. We focus on the case when the similarity index $S_k(u)$ of u gets changed as the case for v can be discussed in the same way. When $S_k(u)$ gets changed, for a vertex w in $N[u]$, their similarity score may also get changed. Then, the neighboring index $NO[w]$ of vertex w and the entries related to w in the bucket index might also need to be updated.

Algorithm 3 shows the pseudo-code of BOTBIN index update for insertion. Firstly, it updates the similarity index of u and v respectively if it is needed (Line 1). If the similarity index of vertex u is changed, then it examines all vertices in the neighborhood of u. Let w be a vertex in $N[u]$, it first obtains the similarity score derived before the update (Line 4). Next, given the updated similarity index, it computes the updated similarity score between u and w with $S_k(u)$ and $S_k(w)$ via Equation 1. Then it updates (resp. inserts) $\langle w, \hat{\sigma}(u, w)\rangle$ in $NO[u]$ if w is not v (resp. w is v) (Line 6). Similarly, $NO[w]$ needs to be updated as $\hat{\sigma}(w, u)$ is also changed with vertex u as the key in $NO[w]$ (Line 7). After that, it updates the bucket index for vertex w by invoking the UpdateBucket procedure (Lines 16-23). Given the old similarity score σ_{old} and the updated similarity score σ_{new}, it first identifies the bucket i^* of σ_{old} and bucket j^* of σ_{new}. Then, if i^* is equal to j^*, then the bucket is unchanged. However, notice that, if the updated bucket is related to $\hat{\sigma}(u, v)$, i.e., the edge to be updated (either insertion or deletion),

Algorithm 3: BOTBIN-Update-Insertion

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 **if** $S_k(u)$ *is modified* **then**
3 **for** *vertex* $w \in N[u]$ **do**
4 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
5 Derive $\hat{\sigma}(u, w)$ of u and w with $S_k(u), S_k(w)$;
6 Insert/update $\langle w, \hat{\sigma}(u, w)\rangle$ in $NO[u]$;
7 Insert/update $\langle u, \hat{\sigma}(w, u)\rangle$ in $NO[w]$;
8 UpdateBucket(w, $\hat{\sigma}_{old}$, $\hat{\sigma}(u, w)$, $w = v$);
9 Update all records of B associated with u;
10 **else**
11 Derive $\hat{\sigma}(u, v)$ of u and v with $S_k(u), S_k(v)$;
12 Insert $\langle v, \hat{\sigma}(u, v)\rangle$ to $NO[u]$;
13 UpdateBucket(u, 0, $\hat{\sigma}(u, v)$, 1);
14 Repeat Lines 2-13 for vertex v;
15 **return** S_k, B, NO;
16 **procedure** *UpdateBucket(w, σ_{old}, σ_{new}, is_updated_edge)*:
17 $i^* \leftarrow \arg_i\{\sigma_{old} \in [1 - i/\delta, 1 - (i-1)/\delta)\}$;
18 $j^* \leftarrow \arg_i\{\sigma_{new} \in [1 - i/\delta, 1 - (i-1)/\delta)\}$;
19 **if** $i^* > j^*$ **then** swap(i^*, j^*);
20 **if** !*is_updated_edge* **then** $j^* \leftarrow j^* - 1$;
21 **for** $i \in [i^*, j^*]$ **do**
22 The record of w in $B[i]$ increases by 1 if $\sigma_{old} < \sigma_{new}$ otherwise decreases by 1;
23 Insert/update the entry of w in $B[i]$;

Algorithm 4: BOTBIN-Update-Deletion

Input: Deleted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 **if** $S_k(u)$ *is modified* **then**
3 Do the same operations of Lines 3-9 in Algorithm 3;
4 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
5 Delete the entry of v from $NO[u]$;
6 UpdateBucket(u, $\hat{\sigma}_{old}$, 0, 1);
7 Repeat Lines 2-6 for vertex v;
8 **return** S_k, B, NO;

then the information in bucket i^* also needs to be updated. That is also why it includes an *is_updated_edge* flag. If it is not related to the updated edge, then there is no need to update the bucket information. Otherwise, if the bucket changes from i^* to j^* (assuming $i^* < j^*$ here), then the entries $\langle w, cnt\rangle$ in buckets $B[i^*]$ to $B[j^*]$ all need to be updated from i^* to j^* (resp $j^* - 1$) if the similarity is (resp. is not) for the updated edge (Lines 16-23). The case when $i^* > j^*$ can be handled in the same way. To facilitate

the update, for each vertex w, BOTBIN maintains the count of the approximate neighbors in each bucket. Thus, we can easily use the maintained count information to find the location of each entry $\langle w, cnt \rangle$ in bucket $B[i]$ via a binary search. After updating all the neighborhoods of u, it further updates the bucket information for vertex u via the maintained count information.

If $S_k(u)$ is not changed, then the similarity between u and $w \in N[u] \setminus \{v\}$ is not changed. Thus, only the similarity between u and v must be computed (Line 11). Then, an entry $\langle v, \hat{\sigma}(u,v) \rangle$ is added to $NO[u]$. Lastly, it invokes the UpdateBucket algorithm to update the bucket index related to u with the maintained count information for each bucket. This process is repeated for v, whose steps are identical to that for u. This finishes the update to the BOTBIN index after inserting an edge (u, v).

Edge deletion. Next, we elaborate on how to update the index when there is an edge deletion. With the deletion of an edge (u, v), the first step is still to update the similarity index of vertices u and v. Then, if the similarity index of u is updated, it might affect the similarity between u and all nodes in its neighborhood $N[u]$. Thus, similar to Algorithm 3, it updates the neighboring index of w where $w \in N[u]$, and further updates the bucket index related to w. The steps are the same as that in the insertion by repeating Lines 3-9 in Algorithm 3 (Algorithm 4 Line 3). Next, it handles the deleted edge (u, v) (Lines 4-6). In particular, it deletes the entry of v from the neighboring index $NO[u]$ of vertex u. Then, it further updates entries related to u in the bucket index. Finally, it repeats this process for vertex v (Line 6). After that, it finishes the index update and returns the updated index (Line 8).

Notice that similar to edge insertion, the major cost comes from the recalculation of the similarity scores and the updates to the bucket index. Thus, the time complexity of the BOTBIN index update given an edge deletion is the same as that of edge insertion. We have the following theorem to bound the index update cost.

Theorem 6. *Given a failure probability p, an error parameter ρ, for an edge insertion/deletion, the expected cost to update the BOTBIN index by Algo. 3 /Algo. 4 is $O\left(\frac{1}{\rho^2}\log\frac{1}{p} \cdot (\frac{1}{\rho^2}\log\frac{1}{p} + \delta \log n)\right)$.*

4.2 Improved Solutions

In the proposed solutions in Section 4.1, one key bottleneck is that it takes $O(\frac{1}{\rho^2}\log\frac{1}{p})$ time to recalculate the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$) each time after an edge insertion/deletion to (u, v). This cost dominates the main update cost as we will show in the experiment. A natural question is: can we reduce the cost for the recalculation of the similarity between u (resp. v) and its neighborhood $N[u]$ (resp. $N[v]$)? Intuitively, the need to recalculate the similarity between u and $w \in N[u]$ after updating the sketch of u seems to be redundant as we only update at most one entry in the sketch of u, making the majority of the sketch untouched. This motivates us to devise a more efficient algorithm to re-compute $\hat{\sigma}(u, w)$, reducing the cost to re-compute the similarity score from $O(\frac{1}{\rho^2}\ln\frac{1}{p})$ to $O(\log(\frac{1}{\rho^2}\log\frac{1}{p}))$. Next, we elaborate on the improved solutions to re-compute the similarity scores.

Faster update for insertion. Consider the impact of adding an edge (u, v) to the sketch of vertex u. If $h(v)$ is larger than the k-th smallest hash value in similarity index $S'_k(u)$ before the update,

Algorithm 5: BOTBIN-Update-Insertion-Opt

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

1 Update $S_k(u), S_k(v)$;
2 Let pop_u be the old k-th smallest element in $S_k(u)$;
3 **if** $S_k(u)$ *is modified* **then**
4 **for** *vertex* $w \in N[u]$ **do**
5 $k_{old} \leftarrow kth(u, w)$;
6 **if** $h(v) \le k_{old}$ **then**
7 **if** $h(v) \in S_k(w)$ **then**
8 $cnt(u, w) \leftarrow cnt(u, w) + 1$
9 **else**
10 $kth(u, w) \leftarrow find\_kth(S_k(u), S_k(w), k_{old})$;
11 **if** (k_{old} *in* $S_k(u)$ *or* $pop_u = k_{old}$) *and* k_{old} *in* $S_k(w)$ **then**
12 $cnt(u, w) \leftarrow cnt(u, w) - 1$;
13 $\hat{\sigma}_{old} \leftarrow$ estimated similarity $\hat{\sigma}(u, w)$ before update;
14 Derive $\hat{\sigma}(u, w)$ of u and w as $cnt(u, w)/k$;
15 Do the same operations of Lines 6-8 in Algorithm 3;
16 Update all records of B associated with u;
17 **else**
18 Do the same operations of Lines 11-13 in Algorithm 3;
19 Repeat Lines 2-18 for vertex v;
20 **return** S_k, B, NO;

then the sketch $S_k(u)$ after the insertion remains the same. There is no need to re-compute the similarity between u and w where $w \in N[u] \setminus \{v\}$. If $h(v)$ is among the k smallest hash values in $N[u]$, $h(v)$ will be added into the updated similarity index $S_k(u)$ and the k-th smallest hash value is removed from original similarity index $S'_u(k)$. It is easy to verify that $S'_k(u)$ and $S_k(u)$ differ by at most two elements. The first element that may be different is the k-th smallest element of original similarity index $S'_k(u)$. Let pop_u denote this element and it will be eliminated from $S_k(u)$. The second different element is $h(v)$ because it has just been added to $S_k(u)$.

Given this information, how can we quickly compute the approximate similarity of $S_k(u)$ and $S_k(w)$? To speed up the similarity re-computation, we maintain two values attached to each edge (u, w). The first value is $cnt(u, w)$, which stores the number of elements in the intersection of $S_k(u)$, $S_k(w)$, and $S_k(N[u] \cup N[w])$. Notice that, given this $cnt(u, w)$, we can then directly derive the estimated similarity $\hat{\sigma}(u, w)$ between u and w as $cnt(u, w)/k$ according to Equation 1. The second element that we maintain for edge (u, w) is $kth(u, w)$, which stores the k-th element of $S_k(N[u] \cup N[w])$. All these values can be initialized during the index construction without affecting the time complexity of the index construction algorithm. Since this incurs $O(m)$ cost, it also does not affect the space complexity of the BOTBIN index. Algorithm 5 shows the pseudo-code of the optimized insertion algorithm.

Recap that pop_u is the k-th element in $S'_k(u)$ before the update (Line 2). If $S_k(u)$ is modified, then it affects the similarity between

u and $w \in N[u]$ and re-computation is required (Line 3). For each vertex w in the neighborhood of u, denote k_{old} as the k-th element of $S_k(N[u] \cup N[w])$ before the insertion, i.e., being equal to $kth(u, w)$ before the edge insertion (Line 5). We focus on the case when $h(v) \leq k_{old}$ (Line 6) as otherwise, it will not affect $cnt(u, w)$ or $kth(u, w)$ and thus the estimated similarity between u and w is unchanged. In this case, i.e., $h(v) \leq k_{old}$, if $h(v)$ is also in $S_k(w)$, then we can know that $cnt(u, w)$ will be incremented by one (Lines 7-8). Notice that for the removed pop_u, it will be larger than $h(v)$, thus is further larger than k_{old}. Thus, it is not in the intersection of $S_k[u] \cap S_k[w] \cap S_k[S_k[u] \cup S_k[w]]$ and will not affect $cnt(u, w)$. If $h(v)$ is not in $S_k(w)$, then we first update $kth(u, w)$ according to $S_k(u)$, $S_k(w)$, and k_{old}. In particular, as we only modify the k-th smallest by one position, we can first find the largest element in $S_k(u)$ that is smaller than k_{old} and the largest element in $S_k(w)$ that is smaller than k_{old} then take the larger one of these two. This will be the k-th smallest element in $S_k(N[u] \cup N[w])$ in the new similarity index (Line 10). After that, we check if k_{old} that is kicked out from $S_k(N[u] \cup N[w])$ is in both $S_k(u)$ and $S_k(w)$. If this is the case, we decrease $cnt(u, w)$ by 1 (Lines 11-12). Notice that we might not be able to find k_{old} in $S_k(u)$ as it has been removed from $S_k(u)$, i.e., pop_u. Thus, if $k_{old} = pop_u$, it also indicates that k_{old} was in $S_k(u)$ (Line 11).

After updating $cnt(u, w)$ and $kth(u, w)$, we are able to derive the updated similarity between u and w via $cnt(u, w)/k$ according to Equation 1. For the remaining parts to update th neighboring index and bucket index, it is the same as that in Algorithm 3 (Lines 15-18). Finally, it repeats this process for v and returns the updated index.

Example 4. *Given the input graph G_1 as shown in Figure 1, assume that the has function h is generated by a random permutation $\{13, 5, 3, 2, 4, 6, 8, 11, 1, 7, 12, 9, 14, 10\}$, namely, $h(1) = 13$, $h(4) = 2$, etc. Further assume that $k = 3$ for the similarity index. Then, before the update, the similarity index of v_1 and v_2 are $S'_k(v_1) = \{2, 3, 5\}$ and $S'_k(v_2) = \{3, 4, 5\}$, respectively. Besides, for each edge (u, v), we maintain a count $cnt(u, v)$ of common hash values in $S_k(u)$ and $S_k(v)$ and the k-th smallest hash value $kth(u, v)$ of $S_k(u) \cup S_k(v)$. For instance, for edge (v_1, v_2), $cnt(v_1, v_2) = 2$ and $kth(v_1, v_2) = 4$.*

Now suppose that an edge (v_1, v_5) is added to graph G_1. Since $h(v_5) = 4$, the similarity index $S_k(v_1)$ of v_1 needs to be updated to $\{2, 3, 4\}$. Clearly, $k_{old}(v_1, v_2) = 4$, which is the value of $kth(v_1, v_2)$ before the edge insertion. Comparing to the added new hash value 4 in $S_k(v_1)$, it is no larger than $k_{old}(v_1, v_2)$. Thus, the $kth(v_1, v_2)$ is not changed and is still 4. In a similar way, we update $cnt(v_1, v_3)$ from 2 to 3 and keep $kth(v_1, v_3)$ unchanged; keep $cnt(v_1, v_4)$ unchanged and update $kth(v_1, v_4)$ from 5 to 4. In addition, since the edge (v_1, v_5) is newly added, we directly calculate the similarity score $\hat{\sigma}(v_1, v_5)$ through $S_k(v_1)$ and $S_k(v_5)$ and record $cnt(v_1, v_5) = 2$ and $kth(v_1, v_5) = 4$. At the meantime, if the cnt score is changed, then the similarity score is updated and its entries in the clustering index is updated accordingly. After processing v_1 and its incident edges, we find that the insertion of edge (v_1, v_5) does not change the similarity index of v_5. Thus, it does not affect similarity scores between v_5 and its neighbors. That means the cnt and kth information on the incident edges of v_5 is also unchanged. Then, we only insert the new information of (v_5, v_1) into the clustering index.

Algorithm 6: BOTBIN-Update-Deletion-Opt

Input: Inserted edge (u, v), the updated graph G, and BOTBIN index for the original graph G'
Output: Updated BOTBIN index for graph G

```
1  Update S_k(u), S_k(v);
2  if S_k(u) is modified then
3      for vertex w ∈ N[u] do
4          k_old ← kth(u, w);
5          if h(v) ≤ k_old then
6              if h(v) in S_k(w) then
7                  cnt(u, w) ← cnt(u, w) − 1
8              else
9                  kth(u, w) ← find_kth(S_k(u), S_k(w), k_old);
10                 if kth(u, w) ∈ S_k(u) and kth(u, w) ∈ S_k(w)
                    then
11                     cnt(u, w) ← cnt(u, w) + 1;
12         Do the same operations of Lines 13-15 of
            Algorithm 5
13     Update all records of B associated with u;
14 Do the same operations of Lines 4-6 of Algorithm 4
15 Repeat Lines 2-14 for vertex v;
16 return S_k, B, NO;
```

Faster update for deletion. The improved update algorithm for edge deletion shares the similar idea as that for insertion, i.e., maintaining $cnt(u, w)$ and $kth(u, w)$ for each pair of edges and dynamically tracking the changes of these two. Algorithm 6 shows the pseudo-code of the index update algorithm for edge deletion. Firstly, the similarity between u and nodes in the neighborhood of u are not changed if $S_k(u)$ is not changed. When $S_k(u)$ is updated, then $kth(u, w)$ is the k-th smallest hash value for set $N[u] \cup N[w]$ before the graph update. Denote this as k_{old} (Line 4). Next, we only need to handle the case when $h(v) \leq k_{old}$, otherwise the deletion of $h(v)$ will not affect $cnt(u, w)$ and $kth(u, w)$. In such a case, i.e., $h(v) \leq k_{old}$, if $h(v)$ is in $S_k(w)$, then $cnt(u, w)$ decreases by 1. However, $kth(u, w)$ remains unchanged as $h(v)$ remains in $S_k(N[u] \cup N[v])$ and thus $S_k(N[u] \cup N[v])$ is not changed. If $h(v)$ is not in $S_k(w)$, we first derive the updated $kth(u, w)$ by computing the smallest element in $S_k(u)$ that is larger than k_{old} and the smallest element in $S_k(w)$ that is larger than k_{old} and taking the smaller one. Then, if $kth(u, w)$ appears in both $S_k(u)$ and $S_k(w)$, then $cnt(u, w)$ increases by 1. This finishes the update to $kth(u, w)$ and $cnt(u, w)$. Next, the remaining steps to update the bucket index and neighboring index are similar to that of Algorithm 5 (Lines 12-13). Next, it removes v from the neighboring index of u, updates the bucket index of u based on the estimated similarity score $\hat{\sigma}(u, w)$ before the update (Line 14), which is the same as Algorithm 4 Lines 4-6. Finally, it repeats this process for vertex v (Line 15) and returns the updated index. We have the following theorem to bound the index update cost for the improved algorithms.

Example 5. *Given the input graph G_1 as shown in Figure 1, still use the same hash function h in Example 4. Now, consider an edge*

(v_1, v_2) is deleted. For v_2, the removal of edge (v_1, v_2) does not affect its similarity index. Thus, there is no need to update to v_2 or the maintained cnt and kth information on its incident edges. For v_1, the removal of edge (v_1, v_2) makes $S_k(v_1)$ change from $\{2, 3, 5\}$ to $\{2, 3, 13\}$. Then, it may further affect the cnt and kth information on its incident edges. Consider the incident edge (v_1, v_4). Since $S_k(v_4) = \{3, 4, 13\}$. Then we know $cnt(v_1, v_4) = 1$ and $kth(v_1, v_4) = 4$ before the deletion. After the deletion of edge (v_1, v_2), $kth(v_1, v_4)$ remains unchanged and $cnt(v_1, v_4)$ is updated to 2. Then, their similarity score is changed and thus the clustering index is updated according. Similarly, we can update the cnt and kth information for edge (v_1, v_3). Finally, since (v_1, v_2) is removed, the record related to this edge in the clustering index is removed accordingly.

Theorem 7. *Given a failure probability p, an error parameter ρ, for an edge insertion/deletion, the expected cost to update the BOTBIN index by Algo. 5/Algo. 6 is bounded by $O\left(\frac{\delta}{\rho^2} \cdot \log \frac{1}{p} \cdot \log n\right)$.*

Choice of p. Finally, we discuss how to set p so that we can provide an approximation guarantee with $1-p_f$ probability after a sequence of updates. Suppose that there are m edges in the initial graph. Regardless of whether they modify the similarity index of u, v or not, each update of edge (u, v) can be seen as a re-sampling of the neighbors of u and v. After adding an edge, this is equivalent to $d_u + d_v + 1$ times of re-sampling. For deleting an edge, it affects the similarity score of $d_u + d_v - 2$ edges. Assuming that there are M random updates, on average, it affects $M \cdot d_E$ edges, where d_E represents the mean of the degrees of the two endpoints of each edge in the edge set E. So if we want to handle M updates, we should set $p = \frac{p_f}{M \cdot d_E}$. A total of m edges need to be initialized during the construction of the BOTBIN, and this part also needs to be considered in the error probability. By setting $p = p_f/(M \cdot d_E + m)$, we can guarantee that we have $1-p_f$ success probability to produce an $(\epsilon, \rho + 1/\delta)$-approximate SCAN. We have the following theorem on the choice of k according to the above discussion.

Theorem 8. *Given an approximate parameter ρ and a probability parameter p_f, let $k = \frac{1}{\rho^2} \log \frac{2(M \cdot d_E + m)}{p_f}$, then BOTBIN can support M random updates and return $(\epsilon, \rho + 1/\delta)$-approximate SCAN with probability $1 - p_f$ for input graph G.*

Notice that $\log \frac{2(M \cdot d_E + m)}{p_f} = O(\log \frac{m+M}{p_f})$. As δ and ρ can be treated as small constants, the final time complexity is $O(\log n \cdot \log \frac{M}{p_f})$. With edge insertions/deletions, the number of affected edges may exceed $M \cdot d_E$. We set a reconstruction threshold $\zeta = M \cdot d_E$ with $M = m$. When the total number of affected edges reaches ζ, we rebuild our index and then double ζ. By setting $p = \frac{p_f}{M \cdot d_E}$ to Theorem 4, the index construction time is then bounded by $O\left(m\left(\frac{1}{\rho^2} \ln \frac{M+m}{p_f} + \log d_{max}\right) + \delta n \log n\right)$. The amortized update time of BOTBIN remains the same.

5 THEORETICAL ANALYSIS

Proof of Lemma 1. Given two sets A and B, assume that there are a elements in their intersection set and b elements in their union set. The bottom-k sketch of set A or B can be regarded as the sampling without replacement according to the permutation of h (assume that $k \leq b$). It is easy to verify that $S_k(A) \cap S_k(B) \cap S_k(A \cup B)$ indicates a subset of the bottom-k sketch of $A \cup B$ that also appears in the bottom-k sketch of set $A \cap B$. Let X_i be the indicator random variable such that the i-th smallest hash value in $S_k(A \cup B)$ also appears in the bottom-k sketch of $A \cap B$. Then, the probability of $X_i = 1$ is $\frac{a(b-1)!}{(b)!}$. Let $X = \sum_{i=1}^{k} X_i$. Then, we know that $X = S_k(A) \cap S_k(B) \cap S_k(A \cup B)$. By the linearity of expectation, we have $E[X] = k \cdot a/b$. So $S_k(A) \cap S_k(B) \cap S_k(A \cup B)/k$ is an unbiased estimator of the Jaccard similarity of A and B. □

Proof of Theorem 2. We use the following theorem for martingales to prove the guarantees in Theorem 2.

Theorem 9 (Azuma-Hoeffding Inequality[21]). *Let $M_0, M_1, \cdots, M_t$ be a martingale such that:*

$$B_k \leq M_k - M_{k-1} \leq B_k + d_k$$

for some constants d_k and for some random variables B_k that may be functions of $M_0, M_1, \ldots, M_{k-1}$. Then, for all $t \geq 0$ and any $\lambda > 0$,

$$Pr[|M_t - M_0| \geq \lambda] \leq 2exp(-2\lambda^2/(\sum_{k=1}^{t} d_k^2))$$

Given above theorem, we will define the martingale step by step. Define X as the same in Lemma 1. We consider the concentration of X. Define $f(X_1, \ldots, X_k) = \sum_{i=1}^{k} X_i$. Let $M_0 = \mathbb{E}[f(X_1, \ldots, X_k)]$, $M_i = \mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_i]$ and $M_k = \mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_k]$. Then, $M_0, M_1, \ldots, M_k$ forms a martingale, i.e., $\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] = M_i$. To prove that $\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] = M_i$, we have the following:

$\mathbb{E}[M_{i+1}|X_1, \ldots, X_i] =$
$\mathbb{E}[\mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_{i+1}]|X_1, \ldots, X_i]$ (Definition of M_{i+1})
$= \mathbb{E}[\mathbb{E}[f(X_1, \ldots, X_k)|X_1, \ldots, X_i]$ (The law of total expectation)
$= M_i$ (Definition of M_{i+1})).

In the second equation, we use the law of total expectation, i.e., $\mathbb{E}[X|Y] = \mathbb{E}[\mathbb{E}[X|U, Y]|Y]$. Let $X = f(X_1, \ldots, X_k), U = X_{i+1}$ and $Y = X_1, \ldots, X_i$. Therefore, $\{M_i\}_{i=0}^{k}$ is a martingale. Then we aim to get the gap of $M_i - M_{i-1}$ so as to apply Theorem 9. Let R_i be the number of elements in the intersection of A and B among the first i elements, i.e., $R_i = \sum_{j=1}^{i} X_j$. Clearly,

$$M_i = R_i + (k-i)\frac{a - R_i}{b - i}, M_{i-1} = R_{i-1} + (k - i + 1)\frac{a - R_{i-1}}{b - i + 1}.$$

Then, using the fact that $R_i - R_{i-1} = X_i$, we can derive that:

$$M_i - M_{i-1} = \frac{b-k}{b-i}(X_i + \frac{R_{i-1} - a}{b - (i-1)}).$$

Fix $\frac{b-k}{b-i} \cdot \frac{R_{i-1}-a}{b-(i-1)}$ as B_i. Since X_i is 0 or 1. We have that:

$$B_i \leq M_i - M_{i-1} \leq B_i + \frac{b-k}{b-i}$$

Thus, we can set $b_i = 1$ as $\frac{b-k}{b-1} < 1$. Then we divide by k for every $M_i, 0 \leq i \leq k$. Clearly, $\frac{M_k}{k} = \hat{J}(A, B)$, and $\frac{M_0}{k} = \frac{a}{b} = J(A, B)$. Then, $\frac{d_i}{k}$ will be also less than $\frac{1}{k}$. Then, $\sum_{i=0}^{k}(\frac{d_i}{k})^2 < \frac{1}{k}$. Applying Theorem 9, we have:

$$\Pr[|\hat{J}(A, B) - J(A, B)| > \rho] \leq 2\exp(-2k\rho^2)$$

When $k = \frac{1}{2\rho^2} \ln \frac{2}{p_f}$, then we have:

$$\Pr[|\hat{J}(A,B) - J(A,B)| > \rho] \leq p_f$$

This finishes the proof of Theorem 2. □

Proof of Theorem 3. Consider the cluster $C_{\epsilon+\rho+1/\delta} \in \mathcal{C}_{\epsilon+\rho+1/\delta,\mu}$. Because the similarity score σ of the edge in $C_{\epsilon+\rho+1/\delta}$ is no smaller than $\epsilon + \rho + 1/\delta$. With the ρ-absolute approximation guarantee of the estimated similarity score, we know $\hat{\sigma} > \sigma - \rho$ and $\hat{\sigma} > \epsilon + \frac{1}{\delta}$. So this edge must be included in the bucket where ϵ is located. Since all edges in cluster $C_{\epsilon+\rho+1/\delta}$ will be handled in this bucket. There exists a cluster $C \in \mathcal{C}_{\epsilon,\mu}^{\rho+1/\delta}$, such that $C_{\epsilon+\rho+1/\delta} \subseteq C$. So by definition, we can know the first condition is satisfied. For every cluster $C \in \mathcal{C}_{\epsilon,\mu}^{\rho+1/\delta}$, there exists a cluster $C_{\epsilon-\rho-1/\delta} \in \mathcal{C}_{\epsilon-\rho-1/\delta,\mu}$, such that $C \subseteq C_{\epsilon-\rho-1/\delta}$. Consider an edge whose similarity $\hat{\sigma}$ is in this bucket. Since $\epsilon - \hat{\sigma} < \frac{1}{\delta}$ and $\sigma > \hat{\sigma} - \rho$, we get $\sigma > \epsilon - \rho - \frac{1}{\delta}$. Then, $C_{\epsilon-\rho-1/\delta}$ would contain this approximate cluster. □

Proof of Theorem 6. We analyze the expected update cost of updates step by step. We focus on insertion as the running cost of deletion can be similarly analyzed. First, consider the probability of adding $h(v)$ to the similarity index of u, which should be k/d_u. After adding $h(v)$ to the similarity index of u, we need to scan all the neighborhood of u and recompute the approximate similarity $\hat{\sigma}(u,w)$ between u and w. This takes a cost of $O(d_u \cdot k)$ cost in total. When the similarity index of u is not changed, then there is no need to re-calculate the similarity between u and the neighborhood of u.

After updating the similarity index and the approximate similarity scores, we further update the neighboring index of w for $w \in N[u]$ if $S_k(u)$ is changed. This can be finished with $O(\log d_w)$ time. To update the bucket index, for each w in $N[u]$, we update the bucket index for each w at most δ times and each time it takes at most $O(\log n)$ cost as the size of bucket-index is $O(n)$ and we use a binary search tree to maintain each bucket index. So the total time to update the bucket index is bounded by $O(d_u \cdot \delta \cdot \log n)$. To summarize, with k/d_u probability, it incurs $d_u \cdot (k + \delta \cdot \log n)$ running cost. When $S_k(u)$ is not modified, we only need to estimate the similarity between u and v, insert $\langle v, \hat{\sigma}(u,v)\rangle$ into $NO[u]$, and modify the bucket index related to u. The time complexity of this part can be bounded by $O(k + \delta \log n)$. Combining these together, we have the expected update cost to be:

$$\frac{k}{d_u} \cdot d_u(k + \delta \cdot \log n) + (1 - \frac{k}{d_u})(k + \delta \cdot \log n) = O\left(k \cdot (k + \delta \cdot \log n)\right).$$

The running cost to update for vertex v will be the same, and thus will not change the time complexity. Taking $k = \frac{1}{2\rho^2} \ln \frac{2}{p_f}$, we derive the final result in Theorem 6. □

Proof of Theorem 7. As analyzed in Section 4.2, we reduce the cost to re-compute the similarity from $O(k)$ to $O(\log k)$. We assume that $k \ll n$ and the expected time complexity is:

$$\frac{k}{d_u} \cdot d_u(\log k + \delta \cdot \log n) + (1 - \frac{k}{d_u})(\log k + \delta \cdot \log n) = O(k \cdot \delta \cdot \log n).$$

Taking $k = \frac{1}{2\rho^2} \ln \frac{2}{p_f}$, we derive the result in Theorem 7. □

Table 1: Summary of datasets ($K = 10^3, M = 10^6, B = 10^9$)

Dataset	n	m	$\bar{d}$	$\bar{c}$	$\bar{\sigma}_{max}$
Skitter	1.7M	22.2M	13.1	0.258	0.245
Pokec	1.6M	44.6M	27.3	0.109	0.171
Topcats	1.8M	50.9M	28.4	0.274	0.202
Livejournal (LJ)	4.8M	85.7M	17.7	0.117	0.297
Orkut	3.1M	234.4M	76.3	0.166	0.202
Brain	784.3K	535.7M	683.1	0.486	0.724
PP-Miner (PP)	8.3M	1.8B	223.8	0.341	0.455
Twitter	41.7M	2.4B	57.7	0.073	0.091
Friendster (FS)	65.6M	3.6B	55.1	0.134	0.130
Web	90.3M	3.9B	42.9	0.194	0.178

6 EXPERIMENT

We experimentally evaluate our BOTBIN against the state-of-the-art index-based method GS-Index in terms of efficiency and effectiveness. All experiments are conducted on a Linux machine with an Intel Xeon(R) CPU clocked at 2.30GHz with 768GB memory.

6.1 Experimental Settings

Datasets. We use 10 large real datasets in the experiments. All these datasets are publicly available at SNAP[2], Konect[1] and Network Repository[3]. Among them, Pokec, Orkut, Twitter, Livejournal, and Friendster are social networks; Skitter, Topcats, and Web are web networks; Brain and PP are biological networks where Brain is a gene-gene interaction network and PP is a protein-protein association network. Following previous works [27, 34], directed graphs, like Twitter, are converted to an undirected graph by treating directed edges as undirected ones. The statistics of datasets are shown in Table 1, where $\bar{d}$ is the average degree, $\bar{c}$ is the average clustering coefficient, and $\bar{\sigma}_{max}$ is the average of the maximum similarity between each vertex to its neighbors (Ref. to Section 3.2).

Main Competitor. We mainly compare our BOTBIN against the state-of-the-art index-based method GS-Index [34], as discussed in Section 2.2. The implementation of GS-Index is provided by their inventors. We do not compare against DynStrClu [27] as it cannot support different input parameters of μ and ϵ as we discussed in Section 2.2. To examine the update performance, we further include the solution in Section 4.1 as a baseline, dubbed as BOTBIN-Basic. Our full-fledged solution in Section 4.2 is dubbed as BOTBIN.

Parameter settings. Recall that all our algorithms include a failure probability parameter p_f. We set the failure probability as $\frac{1}{1000}$. We set $M = m$ and d_E based on each dataset (Ref. to Theorem 8 for their definition), and then we can calculate k for the bottom-k sketch. In addition, BOTBIN includes an error parameter ρ to control the trade-off between the update efficiency and clustering quality. We tune the impact of ρ in Exp 5. The experimental results suggest that when $\rho = 0.1$, it achieves the best trade-off between the update efficiency and clustering quality. For the parameter of bucket size δ (Refer to Section 3 for its definition), we tune δ and examine its impact in Exp 5. We observe that when $\delta = 100$, it achieves the best trade-off between the efficiency and accuracy and thus we choose $\delta = 100$ as the default setting in the remaining experiments.

Table 2: Clustering Quality: Core Labelled Rate (Precision, Recall), Overall Clustering Quality (ARI) (%) and the average number of core-vertices ($K = 10^3, M = 10^6$)

Dataset	ARI	Precision	Recall	Number of cores
Skitter	99.98	99.94	99.47	11.9K
Pokec	99.43	99.96	99.78	4.0K
Topcats	99.90	100.00	99.73	11.8K
LJ	99.97	100.00	99.90	212.5K
Orkut	99.46	100.00	99.92	64.5K
Brain	99.00	99.60	99.61	524.8K
PP	98.88	99.91	99.92	1.4M
Twitter	98.54	99.89	99.75	86.0K
FS	99.41	99.37	99.43	104.2K
Web	99.16	99.46	99.95	2.6M

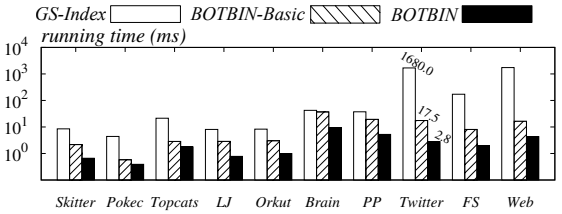


Figure 3: Index update time with edge insertions.

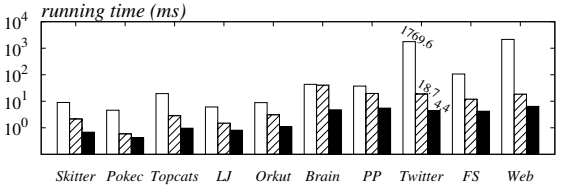


Figure 4: Index update time with edge deletions.

6.2 Experimental Comparisons

Exp 1: Update efficiency. In the first set of experiments, we examine the update performance of our BOTBIN scheme. Firstly, we examine the performance of our index update algorithms for edge insertions. Figure 3 shows the results of the average index update cost for 10,000 random edge insertions. Notice that the y-axis is log-scale. As we can observe, our BOTBIN achieves superb performance and can update the index with milliseconds even on billion edge graphs like PP, Twitter, FS, and Web. Compared to GS-Index, our BOTBIN achieves up to 2 order of magnitude speed-up on the Twitter dataset and an order of magnitude speedup on most datasets. Our BOTBIN further achieves significant speed-up over BOTBIN-Basic as it reduces the cost to re-compute the similarity scores from $O(k)$ to $O(\log k)$. BOTBIN-Basic is still more efficient than GS-Index on all datasets and achieves up to an order of magnitude speed-up over GS-Index on the Twitter dataset.

Next, we examine the performance of the index update algorithms for edge deletions. Figure 4 shows results of average index update costs for 10,000 random edge deletions. The performance gain shows a similar trend as that on edge insertions. We further examine the performance of our index scheme under a mixing workload of insertions and deletions. We find that all algorithms show a stable performance under mixing workloads and is insensitive to the change of workloads.

Table 3: Dice similarity clustering Quality: Core Labelled Rate (Precision, Recall), Overall Clustering Quality (ARI) (%) and the average number of core-vertices ($K = 10^3, M = 10^6$)

Dataset	ARI	Precision	Recall	number of cores
Skitter	99.87	99.97	99.95	43.1K
Pokec	99.99	99.99	99.99	35.9K
Topcats	99.98	99.99	99.99	44.3K
LJ	99.92	100.00	99.99	458.9K
Orkut	99.97	99.99	99.99	305.1K
Brain	99.32	99.94	99.94	652.0K
PP	99.41	99.95	99.80	2.6M
Twitter	98.94	99.88	99.92	306.7K
FS	99.96	99.98	99.99	633.8K
Web	99.24	99.98	99.97	3.9M

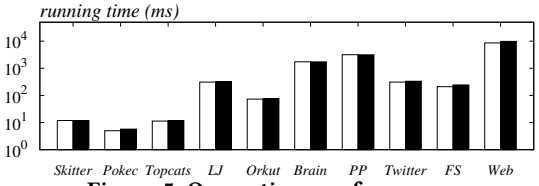


Figure 5: Query time performance.

Exp 2: Query efficiency. Finally, we randomly choose 100 queries with different ϵ and μ parameters by varying ϵ in $\{0.2, 0.21, \ldots, 0.80\}$ and μ in $\{2, 3, 4, \ldots, 15\}$, which follows the previous work [34]. Figure 5 shows the average query time. We can see that GS-Index and BOTBIN show identical query performance as both GS-Index and BOTBIN has a query cost linearly depending on the size of the cluster subgraph. The results show that BOTBIN gains a good trade-off among index update cost, query cost, and query accuracy, and is the preferred choice on dynamic graphs.

Exp 3: Clustering quality. We evaluate the clustering quality of our approximate algorithm and the baseline algorithm on all datasets. In the first set of experiment, the queries are generated following the settings in Exp 2. Following previous work [27], our evaluation uses three metrics: *precision*, *recall*, and *adjusted rand index* (ARI). The precision and recall here are defined based on core-vertex. We use the baseline algorithm to get the ground truth of the core vertex, and then we test the recall and precision of the core vertices found by the approximate algorithm. For ARI, it is a popular metric to evaluate the clustering quality of outputted clustering results under the ground-truth of known clustering results and the higher (up to 1) the better the clustering results are. Table 2 reports the precision, recall, and ARI of the clustering result returned by BOTBIN. We further report the average number of core vertices per query as a reference. As we can observe, BOTBIN achieves close to 100% accuracy, recall, and ARI on all datasets. As we show in Section 3.1, our solution can be directly extended to Dice similarity. Table 3 shows the clustering quality of our solution on Dice similarity. We can observe that BOTBIN still achieves highly accurate clustering results. The update time and indexing cost for Dice similarity are almost identical to that of BOTBIN on Jaccard similarity. Thus, the results are omitted for the interest of space.

In the second set of experiment, we examine the impact of ϵ on the quality of the clustering results returned by BOTBIN by varying

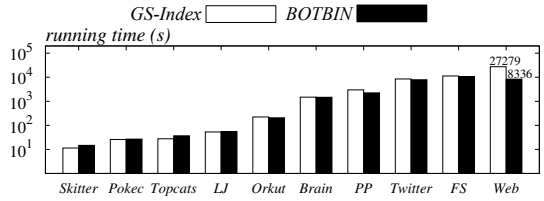


Figure 6: Index construction time.

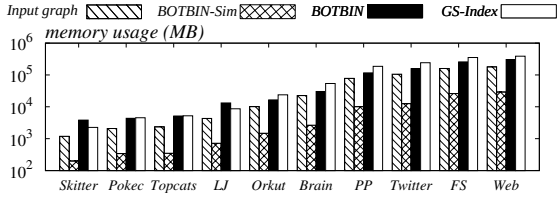
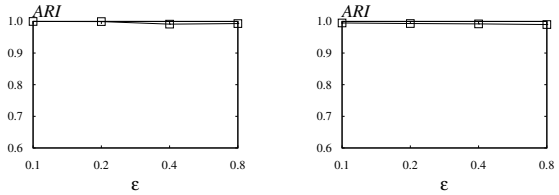


Figure 7: Memory usage.



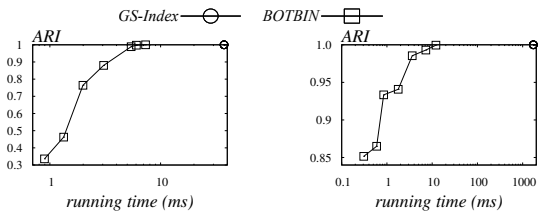
(a) PP (b) Twitter

Figure 8: Impact of ϵ: ARI of clustering results by BOTBIN

ϵ in $\{0.1, 0.2, 0.4, 0.8\}$. To examine clustering results, we report the average ARI score with different choice of μ from 2 to 15. Figure 8 reports ARI scores when we vary ϵ on two large datasets PP and Twitter. As we can observe, our BOTBIN still provides almost identical clustering result as that of the exact SCAN algorithm. This shows the effectiveness of our approximate solution.

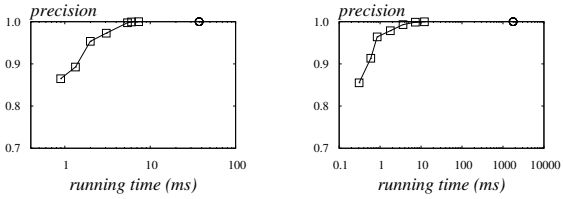
Exp 4: Indexing Costs. Next, we compare the indexing cost of BOTBIN against GS-Index. Figure 6 reports the index construction time of both methods. It can be observed that our BOTBIN can be up to 3.3x faster than GS-Index on index construction. To explain, our BOTBIN has an improved time complexity over GS-Index on index construction. Thus, on huge graphs, BOTBIN tends to be more scalable than GS-Index. In addition, the index construction cost is almost 10^6 times as that of the index update. This further demonstrates the effectiveness of our proposed index update scheme. Figure 7 reports the memory usage of the built indices. We have reported the size of the similarity index, dubbed as BOTBIN-Sim, and the size with both similarity index and clustering index, dubbed as BOTBIN. We further include the memory cost of the input graph as a reference. Note that to maintain the input graph, we need to use dynamic data structures such as BST or hash table (we used hash table) to store the neighboring vertices of each node. Thus, the memory consumption is generally higher than that of the compressed sparse array design on static graphs. The results show that the major space cost of BOTBIN comes from the clustering index. The results also show that BOTBIN takes less space cost compared to GS-Index on large graphs since $\delta \cdot \bar{\sigma}_{max}$ is generally smaller than the average degree of the input graph when $\delta = 100$.

Exp 5: Tuning parameters. Finally, we examine the impact of ρ (error parameter) and δ (bucket number). We show results on the four largest datasets. To examine the impact of ρ, we vary ρ with $\{0.05, 0.075, 0.1, 0.2, 0.25, 0.4, 0.5\}$ and see the trade-off between index update time and the query accuracy in terms of ARI as shown in Figure 9. Notice that the smaller ρ it is, the higher the update cost it is while the more accurate the clustering results are. We observe that when $\rho = 0.1$, it achieves a good trade-off between the update cost and clustering quality. We further examine the impact of ρ to precision (resp. recall) as shown in Figure 10 (resp. Figure 11). It shows a similar trend and when $\rho = 0.1$, it strikes a good balance between update time and clustering quality. Thus, we set $\rho = 0.1$ as the default settings. Next, we examine the impact of δ. Figure 12 shows the results when we vary δ with $\{50, 100, 200, 400\}$. Notice that the larger δ is, the more accurate the clustering results are while the higher update cost it incurs. The results show that when when we set $\delta = 100$, it achieves almost 100% ARI score. The results with precision/recall shares a similar trend and thus is omitted. According to the experimental results, we set $\delta = 100$ as the default setting as it strikes a good balance between update efficiency and clustering quality.



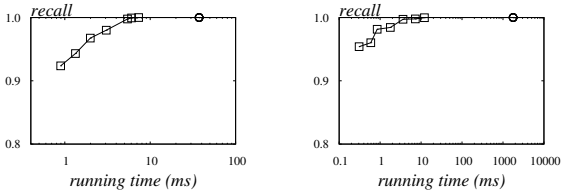
(a) PP (b) Twitter

Figure 9: Impact of ρ: ARI and update time by BOTBIN



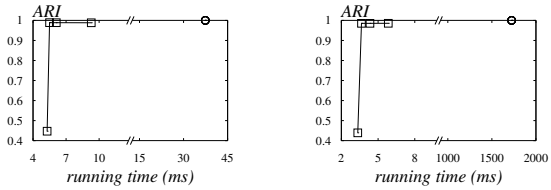
(a) PP (b) Twitter

Figure 10: Impact of ρ: precision and update time by BOTBIN



(a) PP (b) Twitter

Figure 11: Impact of ρ: recall and update time by BOTBIN



(a) PP (b) Twitter

Figure 12: Impact of δ: ARI and update time by BOTBIN

7 CONCLUSION

In this paper, we present an efficient and effective index scheme for dynamic structural graph clustering, verified with extensive experiments. For further work, we plan to investigate how to devise efficient estimation schemes for weighted similarity measures like Cosine and weighted Jaccard similarities on weighted graphs so as to support efficient index updates.

ACKNOWLEDGMENTS

This research is supported by the Hong Kong RGC ECS (Grant No. 24203419), Hong Kong RGC CRF (Grant No. C4158-20G), Hong Kong ITC ITF (Grant No. MRP/071/20X), CUHK Direct Grant (Grant No. 4055181), National Natural Science Foundation of China (Grant No. U1936205), and a gift grant from Huawei.

REFERENCES

[1] 2013. KONECT. http://konect.cc/networks/.
[2] 2014. SNAP Datasets. http://snap.stanford.edu/data.
[3] 2015. Network Repository. https://networkrepository.com.
[4] Dustin Bortner and Jiawei Han. 2010. Progressive clustering of networks using Structure-Connected Order of Traversal. In *ICDE*. 653–656.
[5] Lijun Chang, Wei Li, Lu Qin, Wenjie Zhang, and Shiyu Yang. 2017. pSCAN: Fast and Exact Structural Graph Clustering. *IEEE Trans. Knowl. Data Eng.* 29, 2 (2017), 387–401.
[6] Sudarshan S Chawathe. 2019. Clustering blockchain data. In *Clustering Methods for Big Data Analytics*. 43–72.
[7] Yulin Che, Shixuan Sun, and Qiong Luo. 2018. Parallelizing Pruning-based Graph Structural Clustering. In *ICPP*. 77:1–77:10.
[8] Jiajun Chen, Jimeng Chen, Jie Liu, and Va-Lou Huang. 2013. PSCAN: A Parallel Structural Clustering Algorithm for networks. In *ICMLC*. 839–844.
[9] Edith Cohen and Haim Kaplan. 2007. Summarizing data using bottom-k sketches. In *PODC*. 225–234.
[10] Søren Dahlgaard, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. 2017. Fast Similarity Sketching. In *FOCS*. 663–671.
[11] Yijun Ding, Minjun Chen, Zhichao Liu, Yanbin Ye, Min Zhang, Reagan Kelly, Li Guo, Zhenqiang Su, Stephen Harris, Feng Qian, Weigong Ge, Hong Fang, Xiaowei Xu, and Weida Tong. 2012. atBioNet– an integrated network analysis tool for genomics and biomarker discovery. *BMC genomics* 13 (2012), 325.
[12] Santo Fortunato. 2010. Community detection in graphs. *Physics reports* 486, 3-5 (2010), 75–174.
[13] Roger Guimera and Luis A Nunes Amaral. 2005. Functional cartography of complex metabolic networks. *Nature* 433, 7028 (2005), 895–900.
[14] Jianbin Huang, Heli Sun, Qinbao Song, Hongbo Deng, and Jiawei Han. 2013. Revealing Density-Based Clustering Structure from the Core-Connected Tree of a Network. *IEEE Trans. Knowl. Data Eng.* 25, 8 (2013), 1876–1889.
[15] Sungsu Lim, Seungwoo Ryu, Sejeong Kwon, Kyomin Jung, and Jae-Gil Lee. 2014. LinkSCAN*: Overlapping community detection using the link-space transformation. In *ICDE*. 292–303.
[16] Cindy Xide Lin, Yintao Yu, Jiawei Han, and Bing Liu. 2010. Hierarchical Web-Page Clustering via In-Page and Cross-Page Link Structures. In *PAKDD*. 222–229.
[17] Zhichao Liu, Qiang Shi, Reagan Kelly, Hong Fang, and Weida Tong. 2011. Translating Clinical Findings into Knowledge in Drug Safety Evaluation - Drug Induced Liver Injury Prediction System (DILIps). *PLOS computational biology* 7, 12 (2011).
[18] Son T. Mai, Sihem Amer-Yahia, Ira Assent, Mathias Skovgaard Birk, Martin Storgaard Dieu, Jon Jacobsen, and Jesper Kristensen. 2019. Scalable Interactive Dynamic Graph Clustering on Multicore CPUs. *IEEE Trans. Knowl. Data Eng.* 31, 7 (2019), 1239–1252.
[19] Venkata-Swamy Martha, Zhichao Liu, Li Guo, Zhenqiang Su, Yanbin Ye, Hong Fang, Don Ding, Weida Tong, and Xiaowei Xu. 2011. Constructing a robust protein-protein interaction network by integrating multiple public databases. In *BMC bioinformatics*, Vol. 12. 1–10.
[20] Mutlu Mete, Fusheng Tang, Xiaowei Xu, and Nurcan Yuruk. 2008. A structural approach for finding functional modules from large biological networks. In *BMC Bioinformatics*, Vol. 9. 1–14.
[21] Michael Mitzenmacher and Eli Upfal. 2005. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*.
[22] Symeon Papadopoulos, Yiannis Kompatsiaris, and Athena Vakali. 2009. Leveraging collective intelligence through community detection in tag networks. In *CKCaR*.
[23] Symeon Papadopoulos, Yiannis Kompatsiaris, and Athena Vakali. 2010. A Graph-Based Clustering Scheme for Identifying Related Tags in Folksonomies. In *DaWaK*. 65–76.
[24] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in Social Media - Performance and application considerations. *Data Min. Knowl. Discov.* 24, 3 (2012), 515–554.
[25] Symeon Papadopoulos, Christos Zigkolis, Yiannis Kompatsiaris, and Athena Vakali. 2011. Cluster-Based Landmark and Event Detection for Tagged Photo Collections. *IEEE MultiMedia* 18, 1 (2011), 52–63.
[26] Symeon Papadopoulos, Christos Zigkolis, Giorgos Tolias, Yannis Kalantidis, Phivos Mylonas, Yiannis Kompatsiaris, and Athena Vakali. 2010. Image clustering through community detection on hybrid image similarity graphs. In *ICIP*. 2353–2356.
[27] Boyu Ruan, Junhao Gan, Hao Wu, and Anthony Wirth. 2021. Dynamic Structural Clustering on Graphs. In *SIGMOD*. 1491–1503.
[28] Manos Schinas, Symeon Papadopoulos, Yiannis Kompatsiaris, and Pericles A. Mitkas. 2015. Visual Event Summarization on Social Media using Topic Modelling and Graph-based Ranking Algorithms. In *ICMR*. 203–210.
[29] Manos Schinas, Symeon Papadopoulos, Georgios Petkos, Yiannis Kompatsiaris, and Pericles A. Mitkas. 2015. Multimodal Graph-based Event Detection and Summarization in Social Media Streams. In *ACM Multimedia*. 189–192.
[30] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-scale Graphs. *PVLDB* 8, 11 (2015), 1178–1189.
[31] Thomas Ryan Stovall, Sinan Kockara, and Recep Avci. 2015. GPUSCAN: GPU-Based Parallel Structural Clustering Algorithm for Networks. *IEEE Trans. Parallel Distributed Syst.* 26, 12 (2015), 3381–3393.
[32] Tomokatsu Takahashi, Hiroaki Shiokawa, and Hiroyuki Kitagawa. 2017. SCAN-XP: Parallel Structural Graph Clustering Algorithm on Intel Xeon Phi Coprocessors. In *NDA@SIGMOD*. 6:1–6:7.
[33] Tom Tseng, Laxman Dhulipala, and Julian Shun. 2021. Parallel Index-Based Structural Graph Clustering and Its Approximation. In *SIGMOD*. 1851–1864.
[34] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2019. Efficient structural graph clustering: an index-based approach. *VLDB J.* 28, 3 (2019), 377–399.
[35] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas AJ Schweiger. 2007. Scan: a structural clustering algorithm for networks. In *SIGKDD*. 824–833.
[36] Nurcan Yuruk, Xiaowei Xu, and Thomas A. J. Schweiger. 2008. On Structural Analysis of Large Networks. In *HICSS*. 143.
[37] Weizhong Zhao, Venkata Swamy Martha, and Xiaowei Xu. 2013. PSCAN: A Parallel Structural Clustering Algorithm for Big Networks in MapReduce. In *AINA*. 862–869.
[38] Qijun Zhou and Jingbin Wang. 2016. SparkSCAN: A Structure Similarity Clustering Algorithm on Spark. In *BDTA*. 163–177.
[39] Yang Zhou, Hong Cheng, and Jeffrey Xu Yu. 2009. Graph Clustering Based on Structural/Attribute Similarities. *PVLDB* 2, 1 (2009), 718–729.