Vertex Encoding for Edge Nonexistence Determination With SIMD Acceleration

Hangyu Zheng, Youhuan Li, *Member, IEEE*, Fang Xiong, Xiaosen Li, Lei Zou, Peifan Shi, and Zheng Qin

***Abstract*—We propose to design vertex encoding for determinations of no-result edge queries that should not be executed. Edge query is one of the core operations in mainstream graph databases, which is to retrieve edges connecting two given vertices. Real-world graphs may be too large to be stored in memory and frequently accessing edge data on disk usually incurs much overhead. The average degree of real-world graph tends to be much less than the vertex number, and edges may not exist in most pairs of vertices. Efficiently avoiding no-result edge query executions will certainly improve the performance of graph database. In this article, we propose a new and important problem for determining no-result edge queries: vertex encoding for edge nonexistence determination (VEND, for short). We build a low dimensional vertex encoding for all vertices, and we can efficiently determine most vertex pairs that are connected by no edges just with their corresponding codes. The encoding can be efficiently adjusted when data updates happen. With VEND, we can utilize in-memory efficient operations to filter no-result disk accesses for edge query. We also design SIMD-oriented compression optimizations to further improve performance. Extensive experiments on real-world datasets confirm the effectiveness of our solution.**

***Index Terms*—Edge query, graph database, vertex encoding.**

I. Introduction

EDGE query is one of the fundamental operations in main stream graph databases [1], [2], [3], [4], which is to retrieve edges that connect two given vertices. It is frequently executed in many important graph computations, such as relation retrieval in knowledge graph [5], [6], clustering coefficient [7], triangle counting [8], [9], [10] and subgraph matching [11]. However, most vertex pairs in real-world graphs are connected by no edges. In fact, the average degree of a large real-world graph tends to be much lower than the vertex count [12]. Therefore, for each vertex, there are far fewer vertices adjacent to it than those that are non-adjacent. It is just a waste of time to execute edge queries over vertex pairs that are not adjacent. What's more, graph-structured data proliferated from mobile applications is usually too large to be stored in memory [13], and executing edge queries over them may result in time-consuming disk accesses. Filtering no-result edge queries before they are evaluated over graph storage can certainly improve the system performance of graph databases.

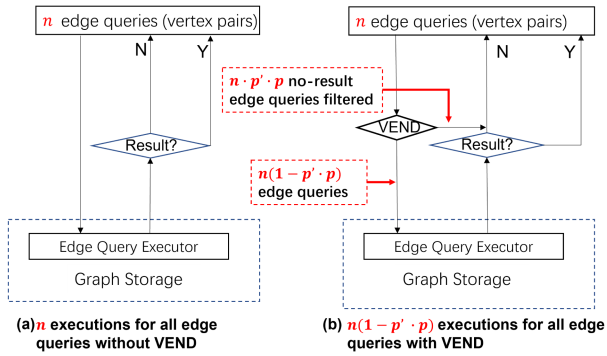

Fig. 1. Application scenario of VEND: assume that $n \cdot p$ edge queries are no-result and $n \cdot p \cdot p'$ $(0 \leq p' \leq 1)$ of them can be detected by VEND.

In this article, we creatively propose a new and important problem: vertex encoding for edge nonexistence determination (VEND, for short) which could be used to filter no-result edge queries. We design a mechanism (called as VEND solution) to encode each vertex into a low-dimension vector, with which we can efficiently detect and filter no-result edge queries as many as possible. We require that both the space cost for a vertex vector and the time cost for an edge nonexistence determination be linear to the dimension number. In this way, an edge nonexistence determination costs only constant time and space, which is much more efficient than executing an edge query over a large graph that is stored on disk. Since vector of each vertex is low-dimension, we can persist all vectors in memory. Essentially, VEND is to utilize in-memory vertex encodings and the corresponding constant time operations to filter no-result disk accesses for edge data.

Fig. 1 demonstrates the application of VEND. We can see that graph databases with VEND could reduce the executions

Manuscript received 3 March 2023; revised 16 September 2023; accepted 23 December 2023. Date of publication 9 January 2024; date of current version 10 June 2024. This work was supported in part by NSFC under Grant 62102142, in part by the Hunan Provincial Natural Science Foundation of China under Grant 2022JJ40093, and in part by the Aid program for Science and Technology Innovative Research Team in Higher Educational Instituions of Hunan Province. Recommended for acceptance by G. Wang PhD. *(Corresponding authors: Youhuan Li; Fang Xiong.)*

Hangyu Zheng, Youhuan Li, Peifan Shi, and Zheng Qin are with the College of Computer Science and Electricity Engineering, Hunan University, Changsha 410082, China (e-mail: zhenghangyu@hnu.edu.cn; liyouhuan@hnu.edu.cn; spf@hnu.edu.cn; zqin@hnu.edu.cn).

Fang Xiong is with Xiangya Hospital, Central South University, Changsha 410008, China (e-mail: Xiongf@csu.edu.cn).

Xiaosen Li is with the The Chinese University of Hong Kong, Hong Kong 999077, China (e-mail: pkuhansen@163.com).

Lei Zou is with Peking University, Beijing 100871, China (e-mail: zoulei@pku.edu.cn).

This article has supplementary downloadable material available at https://doi.org/10.1109/TKDE.2024.3350919, provided by the authors.

Digital Object Identifier 10.1109/TKDE.2024.3350919

Algorithm 1: Edge Iterator based Triangle Counting.

```
   Input: Graph G = (V, E)
1  count = 0
2  for each node i ∈ V do
3  |  Get adj(i) from storage on disk
4  |  for each edge (i, j) where i < j do
5  |  |  if VEND(j, j') = NO_EDGE for each
   |  |    j' ∈ adj(i) ∧ j < j' then
6  |  |  |  Continue;
7  |  |  Get adj(j) from storage disk
8  |  |  let K = {k ∈ adj(j) | j < k}
9  |  |  count = count+ |adj(i) ∩ K|
10 return count
```

Algorithm 2: Disked based Triangle Counting [15].

```
   Input: Graph G = (V, E) on disk, memory size M
1  Merge-sort E on disk into E_disk by source and destinations
2  Group edges into p = ⌈|E|/M⌉ partitions according to their
     destinations w.r.t. p disjoint consecutive intervals
3  Let [L_P, U_P) denote interval range of destinations in P
4  for each i and adj(i) in E_disk do
5  |  for each (i, j, L_P, U_P) where i < j ∈ adj(i) do
6  |  |  Let K = {k ∈ adj(i) | j < k} ∩ [L_P, U_P)
7  |  |  if VEND(j, k) = NO_EDGE for ∀k ∈ K then
8  |  |  |  Continue;
9  |  |  Write triple < i, j, K > into P's companion file
10 count = 0
11 for each partition P loaded in memory do
12 |  for each triple < i, j, K > in P's companion file do
13 |  |  Let J denotes j's neighbors in P // in-memory
14 |  |  count = count+ |J ∩ K|
15 return count
```

of no-result edge queries. We can also see that VEND makes no assumption on how an edge query is evaluated and works independently over the underlying graph storage.

A VEND solution may not be able to detect every no-result edge query. If an edge query can not be determined as no-result, it should still be executed over database since the corresponding edge existence is uncertain. A VEND solution should be updated efficiently in dynamic scenarios, which is a must for database consistency and efficiency. Additionally, it is reasonable that VEND only considers edge nonexistence instead of existence. In fact, if we apply an edge existence determination solution, some vertex pairs connected by edges may not be detectable with low-dimension vertex vectors. Hence, we need still conduct the corresponding edge query execution to ensure the correctness of computation. In this way, no edge queries can be filtered in edge existence determination solutions.

A. Applications

1) Relation Retrieval: Relation retrieval over entities is the most basic application of edge query, such as predicates search over given subject and object in knowledge graph [5], [6].

2) Triangle Counting: We demonstrate how VEND could accelerate state-of-the-art (SOTA) triangle counting methods. In scenarios of VEND, graph is stored on disk and our discussions focus on external-memory algorithms. Another application for accelerating subgraph matching (Graphflow [14]) are presented in Appendix (B).

Edge iterator based method is the SOTA in-memory triangle counting solution utilizing ordered adjacent lists intersections. We extend it into an external-memory version by organizing the adjacent lists with Key-Value store on disk. Algorithm 1 presents the corresponding framework. We can see that when a vertex i is visited, for each edge (i, j) where $j \in adj(i)$ and $i < j$, we conduct VEND tests between j and every other vertex j' ($j < j'$) in $adj(i)$. If j is confirmed to be not adjacent to any such j' in $adj(i)$ (Line 5 in Algorithm 1), then we need no disk access for adjacent list of j (Line 7 in Algorithm 1). In this way, we save one costly disk access ($O(|adj(i)|)$) with in-memory efficient VEND tests ($O(|adj(j)|)$).

Trigon [15] is the SOTA disk-based framework for triangle counting. It divides the range $[0, maxID]$ into several consecutive intervals, where edges with destination id falling in the same interval are grouped together into a partition that can be loaded into limited memory. For each partition P, Trigon builds a companion file storing a series of triples $< i, j, K >$ where $j \in adj(i)$ is a source vertex of at least one edge in P, and K is the set of i's neighbors within the range interval of P. In this way, when P is loaded into memory, for each triple $< i, j, K >$ in the corresponding companion file, j's adjacent edges in P are already organized in memory and conducting intersection between K and j's neighbors in P could enumerates all triangles containing i and j where the third vertex is within the range of interval corresponding to P. VEND could accelerate Trigon by reducing the number of triples in companion files. Specifically, before we write a triple $< i, j, K >$ into a companion file, we can conduct VEND tests between j and vertices in K (Line 7 in Algorithm 2). If j is confirmed to be not adjacent to any vertex in K, this triple can be discarded safely without incurring incorrectness. Since companion files shrunk, Trigon saves the I/O cost for writing/loading discarded triples (Lines 9 and 12 in Algorithm 2), as well as the corresponding intersections over them.

B. Our Contributions

We summarize our contributions as follows:

- We are the first to propose and solve VEND problem to avoid no-result edge query executions.
- We propose an effective VEND solution.
  - We first design a partial solution which can perfectly determine all no-result edge queries related to a subset of vertices. We additionally design range based and hash based baselines over the partial solution (Section IV).
  - We propose a uniform hybrid VEND solution incorporating both range based ideas and hash based methods. We as well design efficient update algorithms over the hybrid version (Section V).
  - We further optimize the hybrid version with SIMD-oriented compression to enhance the encoding performance. We also propose tree search strategy combining SIMD mechanism to significantly improve decoding efficiency (Section VI).

- We conduct various experiments to evaluate our solutions and find that even the baseline could bring performance improvements, which confirms the importance and effectiveness of our VEND solutions.

II. Related Work

To the best of our knowledge, this is the first work that proposes to design vertex encoding for edge nonexistence determination. Let's discuss some works that are semantically similar to the proposed problem.

Graph Embedding: Graph embedding is to map each vertex into a low-dimensional vector, which tries to preserve the connection strength between vertices in the original graph [16], [17], [18], [19], [20], [21], [22], [23], [24]. The key similarity between graph embedding and VEND is that both of them build vertex vectors. However, graph embeddings provide no guarantees on the edge nonexistence, and graph embedding can not be used to solve VEND problem.

Link Prediction: Link prediction focuses on how to predict missing edges or future ones when the set of edges is only partially given [25]. Existing works tend to compute heuristic similarity between two vertices and predict the edge existence with the similarity as likelihood, such as Common Neighbors (CN) [26], [27] and Katz Index (KI) [28]. Although link prediction methods pay close attention to the edge existence over a pair of vertices, their determinations rely heavily on probabilities. It is possible that a link prediction method makes a wrong prediction and cause a false negative, which is not allowed in VEND. Therefore, link prediction methods can not be applied to VEND problems.

Bloom Filter: A possible alternative for reducing no-result edge queries is Bloom filter, which is a space-efficient probabilistic membership query solution with an acceptable false positive rate [29]. We can build bits based Bloom filter (with maximum hash slot) over all edges and conduct the corresponding membership queries for edge existence determinations. However, global Bloom filter is not easy to update since a single deletion of edge would result in a total reconstruction over entire edge set, which introduces huge update overhead. In fact, there are also many variants of Bloom filter designed for efficient element deletions [30], [31], [32], [33]. Counting Bloom filter (CBF) [30] extends the bits based Bloom filter by setting each position as a counter with multiple bits. In this way, adjustment for inserting/deleting an element can be done by increasing/decreasing the corresponding counters by 1. However, with the hash slot of size many times smaller than that of bit based Bloom filter, CBF suffers from much higher false positive rate. Deletable Bloom filter (DBF) [31] only resets collision-free bits for a deletion, and more and more bits would remain to be 1 forever with element deletions happen. Hence, DBF can not be applied to VEND scenarios. Ternary Bloom filter (TBF) [32] improves the DBF by allocating two bits for each counter. However, counters where collisions happen more than twice may lead to false negatives, which are definitely not allowed in VEND. Blocked Bloom filter (BBF) partitions hash slot into multiple blocks, each of which is a small standard Bloom filter. The first hash value of an element is used to select a block, inside which additional hash values are then used to set or test bits as usual. When deletion of an element happens, only the corresponding block need to be reconstructed. However, we need to hash every element in the entire set with the first hash function to determine elements corresponding to the block for reconstruction, which makes deletions quite inefficient.

We can see that existing methods of similar semantic can hardly contribute to VEND and its maintenance. Since we are the first that propose VEND problem and the corresponding solution, our work is highly innovative and important.

III. Preliminaries

In this section, we define the VEND. Before formally introducing the problem, we present some important concepts.

Definition 1 (Data Graph): A data graph $G = (V, E)$, where V denotes the vertex set and E is the edge set. Without loss of generality, G is assumed to be an undirected and unweighted simple graph, namely, there is no loop (edge that connects a vertex to itself) and at most one edge connecting a pair of vertices. We use $N_G(v)$ to denote the neighbor set of v in G. We may use $V(G)$ ($E(G)$, resp.) to denote the vertex (edge, resp.) set of G.

Definition 2 (Vertex Vector & Encoding Function f): Given a graph G and a dimension number k, encoding function f is defined over $V(G)$, where for each vertex v, $f(v)$ is a k-dimension vector of integers and $f(v)[i]$ denotes the i-th dimension.

We define vertex pair that is connected by no edges as NEpair. For convenience, we regard NEpair as an equivalent concept to no-result edge query.

Definition 3 (NEpair & NEneighbor): Given a graph G and $v_1, v_2 \in V(G)$, we say that (v_1, v_2) is an NEpair if $v_1 \neq v_2$ and there is not edge connecting v_1 and v_2. We use $\mathbb{NE}(G)$ to denotes the set of NEpairs in G, namely:

$$\mathbb{NE}(G) = \{(v_1, v_2) \mid v_1 \neq v_2 \wedge (v_1, v_2) \notin E\}$$

Also, we say that v_1 and v_2 are NEneighbors of each other.

VEND is proposed to determine NEpairs as many as possible, and determinations from VEND are required to be made just based on vertex vectors. Let's formally define the determination function.

Definition 4 (NEpair Determination Function F): Given a graph $G = (V, E)$ and k-dimension encoding function f, an NEpair determination function (NDF, for short) F is a boolean function defined over $f(V) \times f(V)$ that satisfies the following conditions: $\forall v_1, v_2 \in V$

- $F(f(v_1), f(v_2)) = 1$ only if (v_1, v_2) is an NEpair, that is, when $F(f(v_1), f(v_2)) = 1$, (v_1, v_2) must be an NEpair. While for the case when $F(f(v_1), f(v_2)) = 0$, there is no guarantee on whether (v_1, v_2) is an NEpair.
- $F(f(v_1), f(v_2))$ can always be computed in $O(k)$ time.

Apparently, the set $\{(v_1, v_2) \mid F(f(v_1), f(v_2)) = 1\}$ must be a subset of $\mathbb{NE}(G)$. Also, if $F(f(v_1), f(v_2)) = 1$, we say that NE pair (v_1, v_2) is detectable by F. We use $F(v_1, v_2)$ to denote $F(f(v_1), f(v_2))$ when the context is clear.

A good NDF can detect NEpairs as many as possible. We define an indicator, VEND score, to evaluate f and F.

Definition 5 (VEND Score): Given a graph $G = (V, E)$, dimension number k, an encoding function f and an NDF F, the VEND score over G is defined as the proportion of NEpairs that can be detected by F. We use $Score_{G,k}(f, F)$ to denote the corresponding VEND score, namely,

$$Score_{G,k}(f, F) = \frac{\Sigma_{v_1 \in V, v_2 \in V}(F(f(v_1), f(v_2)))}{|\mathbb{NE}(G)|}$$

We use $Score(f, F)$ to denote the VEND score when the context is clear. Apparently, $0 \leq Score(f, F) \leq 1$.

With the concepts as above, we formally define our problem.

Definition 6 (Problem Definition): Given a graph G and k, VEND is to design an encoding function f and an NDF F such that $Score(f, F)$ is as high as possible, and meanwhile, f can be updated efficiently when edge updates happen.

Note that we make no assumptions on the form of edge, since we only consider the edge (non)existence of two given vertices in this article. A case study in Appendix E.3 demonstrates that our problem can be easily extended over directed graphs.

Framework: We discuss our method in Sections IV-VI. In Section IV, we propose a partial VEND solution that can optimally encode a part of vertices in data graph such that all NEpairs related to these encoded vertices can be efficiently determined. In addition, we extend the partial VEND into two full versions with range-based and hash-based encoding, respectively. In Section V, we present our hybrid VEND solution incorporating both range-based and hash-based methods, where we also discuss how to maintain encoding when data updates happen. In Section VI, we optimize the hybrid VEND solution with the SIMD-oriented encoding and SIMD-accelerated decoding, which further improve the performance. We evaluate our methods in Section VII and conclude in Section VIII.

IV. Baselines With Partial VEND Solution

In this Section, we introduce a partial VEND solution, denoted as (f^α, F^α), over graph G with dimension number k. (f^α, F^α) can optimally encode a part of vertices in G such that all NEpairs related to these encoded vertices can be efficiently determined. We introduce the encoding function f^α in Section IV-A and NDF F^α in Section IV-B, after which we propose a range-based VEND solution in Section IV-C and a hash-based VEND solution in Section IV-D.

A. Encoding Function f^α

Given a graph $G = (V, E)$, we construct f^α as follows:

- Step 0: We initialize $i = 1$ and build a set of comparative flag τ_i ($\forall i, \tau_i < \tau_{i+1}$) that is distinguished from vertex ID. For example, τ_i could be a negative integer.
- Step 1: For each vertex v of degree less than k, set $f^\alpha(v)[0] = \tau_i$ and use the remaining $k-1$ dimensions of $f^\alpha(v)$ to store all neighbors of v in G_i, i.e.,

$$f^\alpha(v) = [\tau_i, v_1, v_2, \ldots, v_{|N_{G_i}(v)|}] \quad (1)$$

where $v_1, v_2, \ldots, v_{|N_{G_i}(v)|}$ are neighbors in $N_{G_i}(v)$.

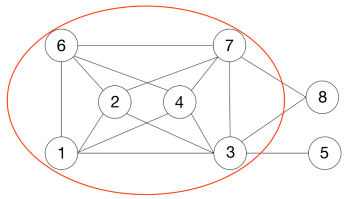

Fig. 2. Data graph example and the core subgraph.

- Step 2: Remove all vertices with a degree less than k and their corresponding adjacent edges from G. Update the degree distribution of G after those removals. If there are still vertices with a degree less than k, let $i = i + 1$ and repeat Steps 1 and 2; otherwise, terminate.

After constructing of $f^\alpha(v)$, the remaining subgraph of G is denoted as C_G^k, where $V(C_G^k)$ and $E(C_G^k)$ represent the corresponding vertex set and edge set, respectively. We refer to C_G^k as the core subgraph of G w.r.t. k, and in fact, the maximal connected component of C_G^k is exactly k-Core of G [34]. We use V_k^α to denote $V \setminus V(C_G^k)$, which exactly contains all vertices encoded in f^α. For example in Fig. 2, let $k = 3$, then $f^\alpha(5) = \{\tau_1, 3\}$ while $f^\alpha(8) = \{\tau_1, 3, 7\}$. The subgraph within the red circle is exactly C_G^3.

B. NDF in Partial Solution

Let's discuss how to design the NDF F^α over f^α. We know that in each iteration of the construction, the remaining neighbors of each vertex v with a degree less than k are fully encoded in $f^\alpha(v)$. Consider $v_1, v_2 \in N_G(v_1)$. If v_1 is in V_k^α, then either v_1 is encoded in $f^\alpha(v_2)$ (when $f^\alpha(v_1)[0] > f^\alpha(v_2)[0]$) or v_2 is encoded in $f^\alpha(v_1)$. Also, if both v_1 and v_2 are not in V_k^α, whether (v_1, v_2) is an NEpair cannot be determined by f^α. For example, consider f^α over the graph in Fig. 2. $8 \in V_3^\alpha$ and $f^\alpha(8) = \{\tau_1, 3, 7\}$, hence, 1, 2, 4, 5, 6 can be determined to be NEneighbors of 8. With these observations, we formally present F^α as follows:

- If both v_1 and v_2 are encoded by f^α, then

$$F^\alpha(v_1, v_2) = \begin{cases} v_2 \notin f^\alpha(v_1) & \text{if } f^\alpha(v_1)[0] \leq f^\alpha(v_2)[0] \\ v_1 \notin f^\alpha(v_2) & \text{if } f^\alpha(v_1)[0] > f^\alpha(v_2)[0] \end{cases} \quad (2)$$

- If only one of v_1 and v_2 are encoded by f^α, assuming that v_1 is encoded, then

$$F^\alpha(v_1, v_2) = (v_2 \notin f^\alpha(v_1))$$

- For any other case, we set $F^\alpha(v_1, v_2) = 0$ which means (v_1, v_2) can not be determined to be NEpair by (f^α, F^α).

Apparently, NEpairs related to vertices in V_k^α (i.e., $V \setminus V(C_G^k)$) can be determined by F^α. Since f^α does not encode any vertex in $V(C_G^k)$, when designing a full VEND solution over a graph G, we can always safely use f^α to encode vertices in V_k^α. We then need only focus on designing VEND solution over C_G^k, namely, designing vertex encoding over $V(C_G^k)$ and the NDF over vertex pairs where the two adjacent vertices are both in $V(C_G^k)$.

C. Range-Based Encoding

Since vertices in $V(C_G^k)$ may have degrees larger than k, we cannot just encode all neighbor IDs into the vector. A straightforward method is to set the encoding vector of each vertex v with a subset of $N_{C_G^k}(v)$, leaving other neighbors not recorded. For example, $\forall\, v$ in $V(C_G^k)$, assuming that $N_{C_G^k}(v) = \{v_1, v_2, \cdots, v_t\}$ where $t \geq k$ and $v_i < v_j$ for $1 \leq i < j \leq t$, we can set encoding vector of v with the smallest k neighbor IDs in $N_{C_G^k}(v)$, i.e., $[v_1, v_2, \cdots, v_k]$. Thus, $\forall\, v' \in V(C_G^k)$ where $v' \leq v_k$, either v' is in the vector of v, or v' is an NEneighbor of v, based on which we can naturally build an NDF. Consider the C_G^3 in red cycle in Fig. 2. The basic range encoding of vertex 6 is $\{1, 2, 4\}$ and vertex 3 can be easily detected as an NEneighbor of vertex 6 since $3 < 4$ while $3 \notin \{1, 2, 4\}$.

Actually, intuition of the basic range encoding is to set each vector with a consecutive block of the corresponding ordered neighbor sequence. While, there are multiple blocks that can be used for building encodings and different selections of block may result in different performance of VEND. We formally define the consecutive block as neighbor block in Definition 7, and then we will discuss how to select and use these blocks for constructing more efficient VEND solution.

Definition 7 (Neighbor Block): Given a graph $G = (V, E)$, a vertex $v \in V(C_G^k)$. Assume that sequence $s = \{-\infty, v_1, v_2, \cdots, v_{|N_G(v)|}, \infty\}$ where $v_1, v_2, \cdots, v_{|N_G(v)|}$ are all neighbors (IDs) of v and $v_i < v_j$ for $1 \leq i < j \leq |N_G(v)|$. Then:

- Each nonempty subsequence of s is called as a neighbor block of v.A neighbor block is usually called as a block for short, and we use B to denote a block.
- The size of B (i.e., $|B|$) is the number of items it contains. There are $|N_G(v)| + 3 - k$ blocks of size k: $\{-\infty, v_1, \ldots, v_{k-1}\}, \cdots, \{v_{|N_G(v)|-k+2}, \cdots, v_{|N_G(v)|}, \infty\}$.
- For a neighbor block B, we use $B.head$ and $B.tail$ to denote the corresponding head and tail items, respectively. And we define interval $[B.head, B.tail]$ as the range of B, denoted as $R(B)$. For example, the range of the block $\{-\infty, v_1, \ldots, v_{k-1}\}$ is the interval $(-\infty, v_{k-1}]$.
- We use $\mathbb{B}_G(v)$ to denote the set of all blocks of v in G.

For a block B of v, the vertex within $R(B)$ is either an item in B or an NEneighbor of v, hence the larger $R(B)$ is, the more NEneighbors of v we tend to determine. We can encode a k-size block B of v into a k-dimension vector to determine all NEneighbors of v within range $R(B)$. We propose to select the block B where the range $R(B)$ covers most NEneighbors of v. For example, consider the block $B = \{-\infty, v_1, \ldots, v_{k-1}\}$ and the corresponding range $R(B) = (-\infty, v_{k-1}]$. There are v_{k-1} vertices within $R(B)$, and $(v_{k-1} - (k-1))$ of them are NEneighbors of v. Fig. 3 shows that this VEND version can detect more NEpairs than that of basic range VEND. We use (f^R, F^R) to denote this VEND version, specifically,

- f^R : for each vertex $v \in V$,
  - if $v \in V_k^\alpha$, $f^R(v) = f^\alpha(v)$
  - if $v \in V(C_G^k)$, $f^R(v) = B \in \mathbb{B}_{C_G^k}(v)$, where B covers most NEneighbors of v.
- $F^R(v_1, v_2)$: for vertex pair (v_1, v_2),

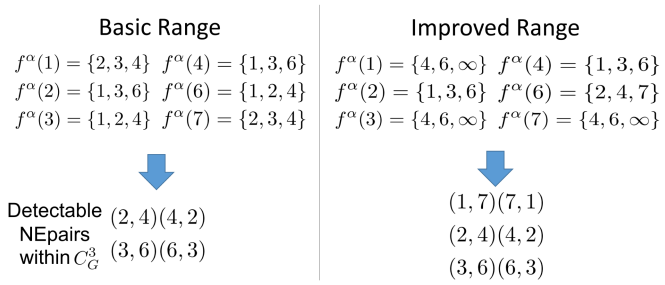

Fig. 3. Basic range VEND V.S. optimized range VEND.

  - if $v_1 \in V_k^\alpha$ or $v_2 \in V_k^\alpha$, $F^R(v_1, v_2) = F^\alpha(v_1, v_2)$
  - if both v_1 and v_2 are in $V(C_G^k)$, then:

$$F^R(v_1, v_2) = (v_1 \in R_2 \wedge v_1 \notin f^R(v_2)) \vee (v_2 \in R_1 \wedge v_2 \notin f^R(v_1))$$

    where R_1 and R_2 are the intervals bounded by head and tail items in $f^R(v_1)$ and $f^R(v_2)$, respectively.

We call (f^R, F^R) as range version of VEND solution.

D. Hash-Based VEND

Another solution for encoding vertex v in $V(C_G^k)$ is to hash neighbor IDs into a k-dimension vector. We incorporate a straightforward hash based VEND solution, denoted as (f^{hash}, F^{hash}), where we hash each neighbor ID into an integer hash value within $\{0, 1, \cdots, k-1\}$ and set $f^{hash}(v)[i] = 1$ $(0 \leq i < k)$ if and only if there exists a neighbor v' (of v) such that $v'\%k = i$; otherwise, $f^{hash}(v)[i] = 0$. Then, vertex pair (v_1, v_2) is an NEpair if both $f^{hash}(v_1)[v_2\%k]$ and $f^{hash}(v_2)[v_1\%k]$ are 0. Formally,

$$F^{hash}(v_1, v_2) = f^{hash}(v_1)[v_2\%k] = 0 \wedge f^{hash}(v_2)[v_1\%k] = 0$$

We call VEND (f^{hash}, F^{hash}) as the hash version. For example, $f^{hash}(6)$ is $\{1, 1, 0\}$ for vertex 6 of C_G^3 in Fig. 2.

For each vertex v, value in each dimension of $f^{hash}(v)$ is binary, namely, the value is either 1 or 0. It is easy to extent (f^{hash}, F^{hash}) into a bitset-based hash version, denoted as (f^{bit}, F^{bit}), where we take k-dimension vector as an entire bitset of size $k \cdot I$. I is the number of bits for each dimension, which is usually 32. We use $b(v)$ to denote the corresponding bitset of v. In this way, $b(v)[i] = 1$ if and only if there exists a neighbor v' (of v) such that $v'\%(k \cdot I) = i$. We call bitset-based hash version as bit-hash version for short.

V. Hyrbid VEND Solution

We now present our hybrid VEND solution, which incorporates range and hash based ideas. In hybrid VEND solution, some dimensions of a vertex vector are used for range based encoding while the remaining ones are taken together as a bitset for hash based method. We present a hybrid VEND example in Section V-A, based on which we discuss some important extensions in Section V-B. We formally introduce our hybrid

VEND solution in Section V-C and discuss the corresponding maintenance in Section V-D.

A. An Example for Hybrid VEND

Let's start with an example hybrid VEND solution (f', F') where we use two dimensions for range based encoding while the remaining $k-2$ for the hash based method.

- f': for each vertex $v \in V$,
  - if $v \in V_k^\alpha$, $f'(v) = f^\alpha(v)$
  - if $v \in V(C_G^k)$ (i.e., $v \in V - V_k^\alpha$), assume that $N_{C_G^k}(v) = \{ v_1, \cdots, v_{|N_{C_G^k}(v)|} \}$. We set the first two dimensions of v's vector as v_1 and v_2, i.e., $f'(v)[0] = v_1$ and $f'(v)[1] = v_2$. We then build a bitset based hash slot on the remaining $k-2$ dimensions and hash neighbors in $N_{C_G^k}(v)/\{v_1, v_2\}$ into the bitset as what we do in hash-based version.
- F': for vertex pair (v_1, v_2),
  - if $v_1 \in V_k^\alpha$ or $v_2 \in V_k^\alpha$, $F'(v_1, v_2) = F^\alpha(v_1, v_2)$
  - if neither v_1 nor v_2 is in V_k^α (i.e., both v_1 and v_2 are in $V(C_G^k)$), let $b(v_1)$ and $b(v_2)$ denote the bitsets over the last $k-2$ dimensions of vectors of v_1 and v_2, respectively, and h is the hash function, then $F'(v_1, v_2) = 0$ if and only if one of the following conditions holds:
    1) $v_1 = f'(v_2)[0]$ or $v_1 = f'(v_2)[1]$;
    2) $v_2 = f'(v_1)[0]$ or $v_2 = f'(v_1)[1]$;
    3) $(v_1 > f'(v_2)[1]) \wedge (v_2 > f'(v_1)[1]) \wedge b(v_2)[h(v_1)] \wedge b(v_1)[h(v_2)]$, where $b(v_1)$ ($b(v_2)$, resp.) is the bitset of v_1 (v_2, resp.).

  Otherwise $F'(v_1, v_2) = 1$.

Apparently, F' can be computed in $O(k)$ time.

It is easy to understand that (f', F') incorporates both range and hash ideas. We call $f'(v)$ as a 2-hybrid encoding for v under parameter k. Formally, given $1 < k' < k$, the vertex encoding with k' dimensions for range based method while the remaining $k-k'$ for hash based method is called as k'-hybrid encoding.

B. Extensions

We introduce a series important optimizations over the example VEND, forming the hybrid VEND solution in Section V-C.

Dynamic selection of block: We can select a uniform k' to encoding vertex in $V(C_G^k)$ by maximizing the VEND score as much as possible. However, a uniform k' may not be the best choice for some vertices. Therefore, we extend the hybrid VEND solution in a finer grained way: independently select k' for each vertex. To achieve this, we can take out $\log_2(k)$ bits from the hash slot to indicate the specific k' for v. Actually, for each vertex v, we can build different vectors with all possible selections of block in $\mathbb{B}_{C_G^k}(v)$ and choose one of them as the target vector. More details on block selection are available in Section V-C3.

Encoding compression: Lots of methods can be used to compress a block and reduce the corresponding overhead [35]. We need to find a compression strategy that will not cause too much decompression overhead compared to NDF computation. In fact, the number of bits for a vertex ID can be set as a tunable parameter I' where $\lceil \log_2(|V|) \rceil \le I' \le I$ (number of bits in each dimension). In this way, there could be more bits in the corresponding hash slot. We may need to adjust I' since the vertex number would change, which will be discussed in Section V-D on the maintenance of VEND.

Distinguish vertices in V_k^α from those in $V(C_G^k)$: Recall the partial VEND solution (f^α, F^α) where we use the first dimension of $f^\alpha(v)$ as a flag to indicate whether v is in V_k^α (See (1) in Section IV-A). In fact, we can just use one bit as a flag to indicating whether a vertex v is in V_k^α or not. In this way, for the vertex $v \in V_k^\alpha$, the maximum number of neighbors that can be encoded changes to $(k \cdot I - 1)/\ I'$ from $(k-1)$. We turn (2) into the following one to remove the dependency on $f^\alpha(v)[0]$:

$$F^\alpha(v_1, v_2) = v_1 \notin f^\alpha(v_2) \wedge v_2 \notin f^\alpha(v_1)$$

Indicating infinite flags with only two bits: For vertex v in $V(C_G^k)$, assume that $N_{C_G^k}(v) = \{v_1, v_2, \cdots, v_x\}$ where $x = |N_{C_G^k}(v)|$. There are $(x+3-k)$ blocks of size k: $\{-\infty, v_1, \ldots, v_{k-1}\}, \cdots, \{v_{x-k+2}, \cdots, v_x, \infty\}$. We can see that there are 3 types of k-size blocks: the leftmost block containing $-\infty$, the rightmost block containing ∞ and the remaining blocks consisting of k neighbor IDs. Once a block containing infinite flag ($-\infty$ or ∞) is selected to be encoded, consuming I' for each flag will be an obvious waste of bits, which should be avoided. Therefore, we take $2 = \lceil \log_2(3) \rceil$ bits to indicate the type of selected block. In this way we can enlarge the first block $\{-\infty, v_1, \ldots, v_{k-1}\}$ and the last block $\{v_{x-k+2}, \cdots, v_x, \infty\}$ into $\{v_1, \cdots, v_k\}$ and $\{v_{x-k+1}, \cdots, v_x\}$, respectively.

C. Formal Hybrid VEND Solution

Let's formally introduce our hybrid VEND solution with those optimizations in Section V-B. We use (f^{hyb}, F^{hyb}) to denote our hybrid version.

1) Encoding Function f^{hyb}: We take each vertex vector as a bitset of size $k \cdot I$ where I is the number of bits for storing an integer in the system ($I = 32$ in the experiments). For simplicity, we use k^* to denote the maximum number of vertices that can be encoded in a vector, i.e., $k^* = (k \cdot I - 1)/I'$. $V_{k^*+1}^\alpha$ and $C_G^{k^*+1}$ can be computed according the construction of partial version in Section IV. Every bit of each $f^{hyb}(v)$ is cleared as 0 before we build the encoding. We use $f_{[i]}^{hyb}(v)$ to denote the i-th bit of $f^{hyb}(v)$.

For each $v \in V_{k^*+1}^\alpha$, we set the first bit of $f^{hyb}(v)$ as 0, namely, $f_{[0]}^{hyb}(v) = 0$. The remaining $k \cdot I - 1$ bits will be used to encode not more than k^* neighbor IDs of v. Since $v \in V_{k^*+1}^\alpha$, at the time when we encode v, the number of remaining neighbors of v must be not more than k^*. We omit details on building $f^{hyb}(v)$ for $v \in V_{k^*+1}^\alpha$ since they are quite similar to those in Section IV.

For each vertex $v \in V(C_G^{k^*+1})$, we set the first bit of $f^{hyb}(v)$ as 1. The next two bits are used to indicate the type of encoded block B (We will discuss block selection in Section V-C3). For example, we can use '00' to indicate the leftmost block, '11' for the rightmost while '01' for those neither leftmost nor rightmost. The further $\lceil \log_2(k^*) \rceil$ bits will store the size of B, i.e., $|B|$. Let $x = 1 + 2 + \lceil \log_2(k^*) \rceil = \lceil \log_2(k^*) \rceil + 3$. The $|B| \cdot I'$ bits

starting from the $(x+1)$-th position to the right are used to store $|B|$ vertex IDs. Finally, the remaining $(k \cdot I - |B| \cdot I' - x)$ bits will be used as hash slot, where we will hash each vertex ID in $N_{C_G^{k^*+1}}(v) \setminus B$ by setting the corresponding bit to 1.

We use $HybEncode(v, V')$ to denote the process of encoding $f^{hyb}(v)$ w.r.t. neighbor set V', where V' is the set of neighbors to be encoded if $v \in V^{\alpha}_{k^*+1}$, and otherwise, $V' = N_{C_G^{k^*+1}}(v)$. We can use the size of V' to indicate whether v is in $V^{\alpha}_{k^*+1}$ or not, since $v \in V^{\alpha}_{k^*+1}$ if and only if $|V'| \leq k^*$.

2) NDF F^{hyb}: Let's discuss the computation of F^{hyb}.

Definition 8 (NE-test): Consider a data graph G, a dimension k, the corresponding $V^{\alpha}_{k^*+1}$ and f^{hyb}. For any two vertices v and v', we say that v' can pass the NE-test of $f^{hyb}(v)$ if and only if one of the following conditions hold:

- If $f^{hyb}_{[0]}(v) = 0$, v' is not one of the IDs in $f^{hyb}(v)$.
- If $f^{hyb}_{[0]}(v) = 1$ (i.e., $v \in V(C_G^{k^*+1})$), assuming that B is the block encoded in $f^{hyb}(v)$, then v' satisfies that either $v' \in R(B) \wedge v' \notin B$, or $v' \notin R(B)$ while v' misses the hash in the corresponding slot of $f^{hyb}(v)$.

We use $v' \overset{\text{NE}}{\mapsto} f^{hyb}(v)$ to denote that v' can pass the NE-test of $f^{hyb}(v)$. Furthermore, the set of vertices that can pass the NE-test of $f^{hyb}(v)$ is denoted as $NT(f^{hyb}(v))$, namely:

$$NT(f^{hyb}(v)) = \{v' \in V \mid v' \overset{\text{NE}}{\mapsto} f^{hyb}(v)\}$$

We define $|NT(f^{hyb}(v))|$ as *NT-size* of vector $f^{hyb}(v)$. Symmetrically, we define a set $NT(v)$ containing such vertex v' that $v \overset{\text{NE}}{\mapsto} f^{hyb}(v')$, namely:

$$NT(v) = \{v' \in V \mid v \overset{\text{NE}}{\mapsto} f^{hyb}(v')\}$$

Apparently, NE-test can be computed in $O(k)$ time.

It is easy to prove that if v' can pass the NE-test of $f^{hyb}(v)$, we can conclude that v' is not a neighbor of v in $C_G^{k^*+1}$. However, v' can still be a neighbor of v in G since (v, v') may be one of the edges in $E \setminus E(C_G^{k^*+1})$ that are removed for computing the core subgraph $C_G^{k^*+1}$. The following Theorem 1 tells us when we can determine an edge as an NEpair.

Theorem 1:

Consider a data graph G, a dimension k, the corresponding $V^{\alpha}_{k^*+1}$ and encoding function f^{hyb}. For any two vertices v_1 and v_2, the following claims hold:

- If $f^{hyb}_{[0]}(v_1) = f^{hyb}_{[0]}(v_2)$, then (v_1, v_2) is NEpair if v_1 and v_2 pass the NE-test of the hybrid encoding of each other:

$$v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2) \wedge v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$$

- If $f^{hyb}_{[0]}(v_1) \neq f^{hyb}_{[0]}(v_2)$, assume that $f^{hyb}_{[0]}(v_1) = 0$ while $f^{hyb}_{[0]}(v_2) = 1$, then (v_1, v_2) is NEpair if $v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$.

Proofs of theorems are presented in Appendix A in the supplementary.

With Theorem 1, we can compute $F^{hyb}(v_1, v_2)$ as follows:

- If $f^{hyb}_{[0]}(v_1) < f^{hyb}_{[0]}(v_2)$, $F^{hyb}(v_1, v_2) = v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$.
- If $f^{hyb}_{[0]}(v_1) > f^{hyb}_{[0]}(v_2)$, $F^{hyb}(v_1, v_2) = v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2)$.
- If $f^{hyb}_{[0]}(v_1) = f^{hyb}_{[0]}(v_2)$, $F^{hyb}(v_1, v_2)$ is equal to the value of $v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2) \wedge v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$.

3) Block Selection: Let's consider the selection of block to be encoded in $f^{hyb}(v)$ for each vertex v in $V(C_G^{k^*+1})$. Intuitively, we always target the block that maximizes $|NT(f^{hyb}(v))|$, i.e., the number of vertices which can pass the NE-test of $f^{hyb}(v)$. For each block $B \in \mathbb{B}_{C_G^{k^*+1}}(v)$ where $N_{C_G^{k^*+1}}(v) = \{v_1, v_2, \cdots, v_x\}$ $(x = |N_{C_G^{k^*+1}}(v)|)$, we first encode the bitset $f^{hyb}(v)$ and then we discuss the computation of $|NT(f^{hyb}(v))|$ as follows:

- If B is empty, then $k \cdot I - 3 - \lceil \log_2(k^*) \rceil$ bits in the vector will be used as hash slot. We can compute $|NT(f^{hyb}(v))|$ by counting the number of vertices missing the hash in $f^{hyb}(v)$.
- If B is a leftmost block, assume that $B = \{v_1, v_2, \cdots, v_{|B|}\}$, according to the definition of NE-test, for $v' \leq v_{|B|}$, $v' \in NT(f^{hyb}(v))$ if and only if $v' \notin B$; while, for $v' > v_{|B|}$, $v' \in NT(f^{hyb}(v))$ if and only if v' misses the hash in $f^{hyb}(v)$. Hence, $|NT(f^{hyb}(v))|$ is equal to $v_{|B|} - |B| + c$ where c is the number of such vertex v' that $v' > v_{|B|}$ and v' misses the hash in $f^{hyb}(v)$.
- If B is a rightmost block, the computation of $|NT(f^{hyb}(v))|$ is symmetrical to that of leftmost block.
- If B is neither a leftmost block nor a rightmost one, the computation of $|NT(f^{hyb}(v))|$ is still similar to that of leftmost/rightmost block. Assuming that $B = \{v_i, v_{i+1}, \cdots, v_j\}$, $|NT(f^{hyb}(v))|$ is equal to $v_j - v_i - (j - i) + c$ where c is the number of such vertex v' that not only $v' < v_i$ or $v' > v_j$, but also v' misses the hash in $f^{hyb}(v)$.

The time cost of building $f^{hyb}(v)$ for each block is $O(|N_{C_G^{k^*+1}}(v)|)$. For computing $|NT(f^{hyb}(v))|$ with $f^{hyb}(v)$, a brute force way is to enumerate every vertex not in $R(B)$ and count the number of vertices that miss the corresponding hash, which cost $O(|V|)$ time. In fact, the modular hash function in our method is periodic and the time for computing $|NT(f^{hyb}(v))|$ with $f^{hyb}(v)$ can be optimized to $O(m)$. Specifically, assume that the slot size is m. For any integer i where $(i+1)m \leq |V|$, it is easy to understand that the number of vertices within interval $[i \cdot m, (i+1)m)$ that miss the hash is exactly the number of bits of value 0 in the slot. Therefore, for a block B where $R(B) = [v_i, v_j]$, we can partition vertices outside $R(B)$ into six parts: $[1, m)$, $[m, t_1 \cdot m)$, $[t_1 \cdot m, v_1)$, $(v_2, t_2 \cdot m)$, $[t_2 \cdot m, t_3 \cdot m)$, $[t_3 \cdot m, |V|]$ where $t_1 = \lfloor \frac{v_1}{m} \rfloor$, $t_2 = \lceil \frac{v_2}{m} \rceil$, $t_3 = \lfloor \frac{|V|}{m} \rfloor$. And the number of vertices outside $R(B)$ that miss the corresponding hash in $f^{hyb}(v)$, denoted as c, can be compute as following:

$$\begin{aligned} c &= Z(m) - Z(1) + (t_1 - 1) \cdot Z(m) + Z(v_1 \% m) \\ &\quad + Z(m) - Z(v_2 \% m) + (t_3 - t_2) \cdot Z(m) + Z(|V| \% m) \\ &= (t_1 + t_3 - t_2 + 1) \cdot Z(m) \\ &\quad + Z(v_1 \% m) + Z(|V| \% m) - Z(v_2 \% m) - Z(1) \end{aligned} \tag{3}$$

where Z is a function such that $Z(i)$ is the number of value 0 in the first i positions of the corresponding hash slot in $f^{hyb}(v)$. Computing $Z(i)$ costs $O(m)$ time and hence, the time for computing $|NT(f^{hyb}(v))|$ with $f^{hyb}(v)$ is optimized to $O(m)$.

Thus, the total time cost for block selection for v is

$$\begin{aligned} &O\left(|\mathbb{B}_{C_G^{k^*+1}}(v)| \cdot (|N_G^{k^*+1}(v)| + m)\right) \\ &= O\left(k^* \cdot |N_G^{k^*+1}(v)| \cdot (|N_G^{k^*+1}(v)| + m)\right) \\ &= O\left(k \cdot |N_G(v)| \cdot (|N_G(v)| + m)\right) \\ &= O\left(k \cdot |N_G(v)| \cdot (|N_G(v)| + k \cdot I)\right) \end{aligned} \tag{4}$$

where I is the number of bits in a dimension and the upper bound of m is $k \cdot I$.

In fact, the key to computing $|NT(f^{hyb}(v))|$ for block B is the function Z according to (3). We propose a sliding-window like optimization for computing Z without generating $f^{hyb}(v)$ for each block, which costs only $O(m)$ time to compute $|NT(f^{hyb}(v))|$ for each block and $O(k^* \cdot |N_G^{k^*+1}(v)| \cdot m)$ time in total for block selection.

Consider all t-size blocks in $\mathbb{B}_{C_G^{k^*+1}}(v)$ over $N_{C_G^{k^*+1}}(v) = \{v_1, v_2, \cdots, v_x\}$ where $x = |N_{C_G^{k^*+1}}(v)|$: $B_1 = \{v_1, \cdots, v_t\}$, $\cdots$, $B_{x-t+1} = \{v_{x-t+1}, \cdots, v_x\}$. We first build an array H_{B_1} of size m where $H_{B_1}[i]$ $(0 \le i < m)$ record the number of such vertex $v' \in N_{C_G^{k^*+1}}(v) \setminus B_1$ that $v' \,\%m = i$. Apparently, for block B_1, $Z(i)$ is exactly the number of value 0 in the first i items in H_{B_1}. We can instantiating $Z(i)$ as an m-size array over H_{B_1}, which costs $O(m)$ time, and hence, computing $|NT(f^{hyb}(v))|$ for block B_1 with Z costs $O(1)$ time ((3)). In addition, we can construct H_{B_2} based on H_{B_1} in $O(1)$ time. Specifically, the difference between $N_{C_G^{k^*+1}}(v) \setminus B_1$ and $N_{C_G^{k^*+1}}(v) \setminus B_2$ is the join of v_{t+1} and the exit of v_1, which is quite similar to a window of size t "slides" from B_1 to B_2 over the sorted neighbor sequence of $N_{C_G^{k^*+1}}(v)$. In this way, H_{B_2} can be constructed by conducting $H_{B_1}[v_1 \% m]$– and $H_{B_1}[v_{t+1} \% m]$++, which costs only $O(1)$ time. Also, with H_{B_2}, it take $O(m)$ time to compute Z for block B_2, which can be used to figure out the corresponding $|NT(f^{hyb}(v))|$ in $O(1)$ time. Similarly, we can construct H_{B_3}, H_{B_4}, $\cdots$, $H_{B_{x-t+1}}$ successively in $O(1)$ time for each. Therefore, the total time cost for computing $|NT(f^{hyb}(v))|$ for all t-size blocks is

$$\begin{aligned} &O\left(|N_{C_G^{k^*+1}}(v)| + (|N_{C_G^{k^*+1}}(v)| - t) \cdot m\right) \\ &\quad = O\left(|N_G(v)| \cdot k \cdot I\right) \end{aligned} \tag{5}$$

where $0 \le t \le k^*$. Thus, the total time cost for block selection is

$$O\left(k^* \cdot |N_G(v)| \cdot k \cdot I\right) = O\left(k^2 \cdot |N_G(v)| \cdot I\right) \tag{6}$$

Thus, the time cost for block selection of v is linear to $|N_G(v)|$. Building $f^{hyb}(v)$ for a block costs $O(N_G(v))$ time, and hence,the time cost for computing $HybEncode(v, V')$ is linear to $|V'|$.

Actually, we can further optimize the computation. When building $H_{B_{i+1}}$ over H_{B_i} $(1 \le i \le x - t + 1)$, if $B_i.head\%m = B_{i+1}.tail\%m$ or $H_{B_i}[B_i.head\%m] > 1 \wedge H_{B_i}[B_{i+1}.tail\%m] > 0$, then the distribution of Z will remain after conducting $H_{B_i}[B_i.head\%m]$– and $H_{B_i}[B_{i+1}.tail\%m]$++, which means we can save the scan over $H_{B_{i+1}}$ for reconstructing Z.

D. Maintenance

Let's discuss the maintenance of VEND solution (f^{hyb}, F^{hyb}). There are four types of graph updates, i.e., vertex/edge insertion/deletion. We use $Ins(v)/Del(v)$ to denote the insertion/deletion of vertex v, and $Ins(v_1, v_2)/Del(v_1, v_2)$ for insertion/deletion of edge (v_1, v_2).

For the sake of presentation, we propose some important concepts that will be used in the illustration of maintenance. For vertex v, if $f^{hyb}_{[0]}(v) = 0$, then we can fully recover the neighbor set encoded in $f^{hyb}(v)$. While, if $f^{hyb}_{[0]}(v) = 1$, some neighbors are hashed in the slot and can not be recovered. Hence, we say that a vector $f^{hyb}(v)$ is *decodable* if $f^{hyb}_{[0]}(v) = 0$, and otherwise, $f^{hyb}(v)$ is *non-decodable*. We say that a decodable vector $f^{hyb}(v)$ is *full* if the number of encoded vertex IDs is k^* (there is not enough unused bits for storing an extra ID), and otherwise, $f^{hyb}(v)$ is *unfilled*.

We discuss the adjustment of f^{hyb} separately for each type of update. We use G^u (G, resp.) to uniformly denote the data graph after (before, resp.) the update. Note that $HybEncode(v, V')$ (See Section V-C) will be frequently used in maintenance discussion. We first discuss the adjustment for edge update, which will be extended for handling vertex update.

1) Insertion of Edge (v_1, v_2): If $F^{hyb}(v_1, v_2) = 0$, (f^{hyb}, F^{hyb}) is still adaptive for G^u and we do not need to update anything since edge query over (v_1, v_2) will not be erroneously filtered. While, if $F^{hyb}(v_1, v_2) = 1$, then:

- If one of $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ is unfilled decodable vector, assuming that it is $f^{hyb}(v_1)$, then we can just conduct encoding v_2 in the extra unused bits of $f^{hyb}(v_1)$ and finish the maintenance. Note that we can easily locate the bits for storing v_2 by decoding $f^{hyb}(v_1)$.
- If both $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ are full decodable vectors, assume that V_1' and V_2' are two sets of vertex IDs decoded from $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$, respectively. We can conduct either $HybEncode(v_1, V_1' \cup \{v_2\})$ or $HybEncode(v_2, V_1' \cup \{v_1\})$ for maintenance. In fact, we always reconstruct the vector that will result in larger NT-size (Definition 8) than that of the other. Specifically, let c_1 and c_2 denote vectors built by $HybEncode(v_1, V_1' \cup \{v_2\})$ and $HybEncode(v_2, V_1' \cup \{v_1\})$, respectively. Assuming that $|NT(c_1)| > |NT(c_2)|$, then we set $f^{hyb}(v_1) = c_1$ and $f^{hyb}(v_2)$ remains, otherwise, $f^{hyb}(v_2) = c_2$ while $f^{hyb}(v_1)$ remains. Since V_1' (V_2', resp.) can be directly recovered by decoding from $f^{hyb}(v_1)$ ($f^{hyb}(v_2)$, resp.), reconstructing the vector need no storage accesses.
- If both $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ are non-decodable vectors, similarly, we always reconstruct the vector that will result in larger NT-size. However, the reconstruction is not easy since we can not recover the set of vertex IDs encoded in a non-decodable vector. A naive method is to retrieve the entire neighbor set $N_G(v_1)$ ($N_G(v_2)$, resp.) and conduct reconstruction w.r.t. $N_G(v_1) \cup \{v_2\}$ ($N_G(v_2) \cup \{v_1\}$, resp.) for building a new vector. In fact, it is easy to understand that, for vertex $v' \in N_G(v_1)$, if v_1 cannot pass the NE-test of $f^{hyb}(v')$, encoding v' into $f^{hyb}(v_1)$ contributes nothing to NEpair determinations. While, if v_1 can pass the

NE-test of $f^{hyb}(v')$, it is a must to encode v' into $f^{hyb}(v_1)$ for the correctness. Hence, for reconstructing $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$, we need only to conduct $Hyb(v_1, V_1')$ and $Hyb(v_2, V_2')$, respectively, where

$$V_1' = \left\{ v \in N_G(v_1) \mid v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v) \right\} \cup \{v_2\}$$

$$V_2' = \left\{ v \in N_G(v_2) \mid v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v) \right\} \cup \{v_1\}$$

Note that computing V_1' and V_2' only costs $O(k * |N_G(v)|)$ time where the dimension number k is a constant.

- If only one of $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ is a full decodable vector while the other is non-decodable, we tend to reconstruct the decodable one to avoid storage accesses.

2) Deletion of Edge (v_1, v_2): Similar to the edge insertion in Section V-D1, we also discuss the adjustment of edge deletion according to the types of vectors $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$.

- If both $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ are decodable vectors, we can just remove v_1 (v_2, resp.) from neighbor set encoded in $f^{hyb}(v_2)$ ($f^{hyb}(v_1)$, resp.) for vector reconstructions.
- If both $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$ are non-decodable vectors, without loss of generality, assume that v can not pass the NE-test of $f^{hyb}(v_2)$. If v_2 can pass the NE-test of $f^{hyb}(v_1)$, then v_2 is not encoded in $f^{hyb}(v_1)$ before, and hence, we need only reconstruct $f^{hyb}(v_2)$ by conducting $HybEncode(v_2, N_G(v_2) \setminus \{v_1\})$. However, if v_2 can not pass the NE-test of $f^{hyb}(v_1)$, then we need to reconstruct both $f^{hyb}(v_1)$ and $f^{hyb}(v_2)$.
- If $f^{hyb}(v_1)$ is decodable while $f^{hyb}(v_2)$ is non-decodable (We omit the discussions for the symmetrical case), ① for $f^{hyb}(v_1)$, if v_2 is encoded in $f^{hyb}(v_1)$, then we can reconstruct $f^{hyb}(v_1)$ by removing v_2; otherwise, we need no update on $f^{hyb}(v_1)$. ② For $f^{hyb}(v_2)$, if v_1 passes the NE-test of $f^{hyb}(v_2)$, then v_1 is not encoded in $f^{hyb}(v_2)$ before and we need no update on $f^{hyb}(v_2)$; while, if v_1 cannot pass the NE-test of $f^{hyb}(v_2)$, we can reconstruct $f^{hyb}(v_2)$ by conducting $HybEncode(v_2, N_G(v_2) \setminus \{v_1\})$.

Note that our adjustment of encoding function for edge deletion provides no guarantee of detecting new NEpair (v_1, v_2). A VEND solution may not be able to detect all NEpairs since the corresponding performance is heavily influenced by the graph distribution and the dimension parameter k.

3) Insertion/Deletion of Vertex V: For insertion of vertex v, we can just allocate a vector $f^{hyb}(v)$ for v where every bit of $f^{hyb}(v)$ is initialized with value 0. An issue we need to consider is that with the growth of vertex number, the bits used for storing a vertex ID (i.e., I') may not be enough and we need to reconstruct all vertex vectors. In fact, this case will happen only when the data graph double in the vertex number, and the amortized cost for each vertex insertion is $O(degr(G))$, where $degr(G)$ is the average degree of G. For deletion of vertex v, we reconstruct all such vector $f^{hyb}(v')$ where $v' \in N_G(v)$ and v can not pass the NE-test of $f^{hyb}(v')$. We also clear every bit in $f^{hyb}(v)$ with value 0. We omit the discussion on the extra bits as the vertex volume shrinks since it is a symmetrical case to that of vertex insertion.

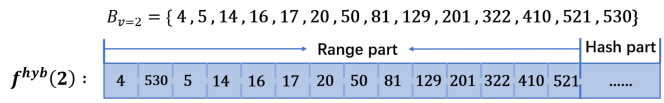

Fig. 4. Example block and its encoding in hybrid version.

4) Analysis: Time cost for vertex update is constant. While, the time for insertion/deletion of edge (v_1, v_2) is equal to that for computing $Hyb(v_1, V_1')$ and $Hyb(v_2, V_2')$, i.e., $O(k^2 \cdot I \cdot (|V_1'| + |V_2'|))$, where V_1' (V_2', resp.)is a subset of $N_G(v_1)$ ($N_G(v_2)$, resp.). Therefore, the time cost for edge update is linear to the neighbor number of the adjacent vertices.

VI. SIMD-Oriented Acceleration

In this section, we further extend the hybrid encoding into a new SIMD-oriented version, denoted as (f^{hyb+}, F^{hyb+}). We propose SIMD-oriented search tree(SS-tree, for short) to transfer neighbor block into a new permutation, where we can conduct neighbor membership query of NE-test in an efficient tree search way, instead of previous sequential scan. We also design the SIMD-oriented encoding and decoding in the new permutation for further performance enhancement. Specifically, we present our SS-tree based VEND framework in Section VI-A, where we illustrate the definition, construction and array implementation of SS-tree, including the array-based tree search. For the sake of presentation, we present the detailed SIMD-oriented encoding and acceleration over array-implemented SS-tree in Section VI-B. Due to space limitation, we discuss neighbor block selection of hyb+ in Appendix C. Note that we omit the discussion of maintenance for hyb+ version, which is quite similar to that of the hybrid version.

A. SIMD-Oriented Search Tree

Let's discuss our SS-tree based acceleration strategy for the neighbor existence search over hybrid version. Consider the example vector $f^{hyb}(2)$ of vertex 2 in Fig. 4, where the encoded block (for the range part) is $B_{v=2} = \{4, 5, 14, 16, \cdots, 201, 322, \cdots, 530\}$. In hybrid version, when we conduct the NE-test of $f^{hyb}(2)$ for vertex 201, we would decode each the encoded vertex in $B_{v=2}$ and then determine the membership of 201 in $B_{v=2}$. We can see that this decoding and search costs $O(|B_{v=2}|)$ time.

Actually, the search is a fine grained and frequently used operation, which should be optimized. We propose an SIMD-oriented search tree for accelerating the neighbor existence search, where we can not only optimize the sequential scan into efficient tree search, but also incorporate SIMD [36] capabilities to achieve performance enhancement.

1) SS-Tree Definition: SIMD enables the processing of multiple data simultaneously with a single instruction. The number of data processed in parallel is called as scalar value, which is usually 4, 8 or 16. Without loss of generality, we use s to

Algorithm 3: Construction of SS-Tree.

Input: Block B = $\{x_0, ..., x_{n-1}\}$, B^- = $B \setminus \{x_0, x_{n-1}\}$
Output: SS-Tree $T(B)$
1 Let $T(B).size$ = $\lceil |B^-|/s \rceil$
2 Construct topology structure of $T(B)$ of size $\lceil |B^-|/s \rceil$
3 Set ID = 1
4 **foreach** *node of* $T(B)$ *from top to bottom and left to right* **do**
5 | $node.ID$ = ID++
6 $SetElements(T(B).root, B^-, pos = 0)$
7 **return** $T(B)$
8 –
9 **Function** $SetElements\ (N_x,\ B^-,\ pos)$ **:**
10 | Let n_i $(1 \le i \le s)$ be node number of i-th subtree of N_x
11 | Compute $\{p_0, p_1, ..., p_s\}$ where $p_i = pos + \sum_{k=1}^{i} (n_k + 1)$
12 | Set $\{B^-[p_1], \cdots, B^-[p_s]\}$ as the s sorted keys in N_x
13 | **for** *i-th child* N_i *of* N_x **do**
14 | | $SetElements(N_i, p_{i-1})$
15 **return**

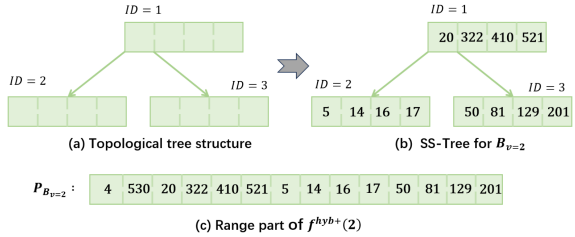

Fig. 5. SS-tree construction and array-implementation.

denote the scalar value. Given s, we formally define SS-tree in Definition 9.

Definition 9 (SIMD-oriented Search Tree (SS-tree)): Consider a scalar value s, neighbor block B= $\{x_0, \ldots, x_{n-1}\}$ and the corresponding $B^- = B \setminus \{x_0, x_{n-1}\}$. An SIMD-oriented search tree (SS-tree, for short) is a complete s-branch search tree built over B^-, which is similar to the complete binary search tree. Specifically, SS-tree satisfies the following conditions:

- Each tree node contains at most s elements (in ascending order) from B^- as sorted keys, except for the leaf nodes.
- Each tree node has at most $s + 1$ children.
- Every level of SS-tree, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.
- SS-tree are of properties of search tree with key of each internal node being greater than all the keys in the respective node's left subtree and less than the ones in its right subtree [37].

We use $T(B)$ to denote the SS-tree constructed over block B.

2) SS-Tree Construction: Consider a block B=$\{x_0, \ldots, x_{n-1}\}$, B^-=$B \setminus \{x_0, x_{n-1}\}$ and a scalar value s. Let's discuss the SS-tree construction with pseudo codes in Algorithm 3. According to Definition VI-A1, $T(B)$ is a complete s-branch search tree built over B^- where each internal level is completely filled while nodes in the last level are all arranged on the left side. In this way, once the number of tree nodes (i.e., $\lceil |B^-|/s \rceil$) is given, we can determine the topology structure of an SS-tree (Line 2 in Algorithm 3). For example, $|B^-_{v=2}| = 12$, and node number of $T(B_{v=2})$ would be 3 with $s = 4$. Fig. 5(a) presents the topology structure of $T(B_{v=2})$, where we also assign ID to each tree node from top to the bottom and left to the right (Line 5 in Algorithm 3).

With the topology structure, we can compute the tree node number of each subtrees from the root, and consequently determine the s elements in root (Lines 10-12 in Algorithm 3). For example, assume that the first element in root (of ID 1) is v, since there is totally 1 tree node (of ID 2) in the first subtree of the root, there would be $1 * 4 = 4$ elements less than v and hence v is the 5-th element in $B^-_{v=2}$, i.e., $v = 20$. Similarly, we can determine the second element in root is the 10-th one of $B^-_{v=2}$, i.e., 322. 410 and 521 would be the last two elements since there are no branches. Apparently, we can recursively determine elements in each descendant of root (Line 14 in Algorithm 3). Fig. 5(b) presents the SS-tree of example block $B_{v=2}$ with $s = 4$.

3) Array-Implemented SS-Tree & Its Search: Similar to a complete binary search tree, an SS-tree can be implemented as an array, where we can also equivalently conduct the tree search. Recall the tree node ID that is assigned from top to the bottom and from left to the right in SS-tree. Actually, an array implementation of a complete s-branch is to store tree nodes in an array sorted by their IDs [37]. We create a new permutation of each B, denoted as P_B, where $P_B[0]$ stores x_0 and $P_B[1]$ stores x_{n-1}. Also, items from $P_B[(i-1) * s + 2]$ to $P_B[(i-1) * s + s + 1]$ store elements in the tree node of ID i $(1 \le i \le \lceil (\rceil |B^-|/s))$. In this way, we can encode each array-implemented SS-tree $T(B)$ into the range part in the vertex vector, so that we can conduct membership query with tree search over $T(B)$ that is more efficient than previous sequential scan over B. Fig. 5(c) presents the $P_{B_{v=2}}$ for $B_{v=2}$, where the first two elements, i.e., 4 and 530, are minimum and maximum elements in B_{v_2}, respectively. And the subsequent four elements $\{20, 322, 410, 521\}$ correspond to those in tree node of ID 1 (see Fig. 5(b)). We can encode P_B in the range part of the vector, where we would conduct the corresponding tree search.

According to the properties of SS-tree, once a node ID is given, we can directly figure out all IDs of its children. Since each tree node ID corresponds to a unique position in P_B, branching operations over SS-tree can be equivalently performed over P_B, and hence we can directly conduct the search for membership tests on P_B. Pseudo codes for NE-test of v_1 over $f^{hyb+}(v_2)$ are available in Algorithm 4 and let's focus on the case when $v_1 \in (P_B[0], P_B[1])$. We would firstly decode the s elements of root: $v_{i_1}, v_{i_2}, \cdots, v_{i_s}$ (Line 10 in Algorithm 4), and then we would test whether v_1 is one of these s elements (Line 11 in Algorithm 4). If not, we then determine which branch we should follow for further recursive search (Line 13 in Algorithm 4). We would not stop the search until we find that v_1 is in some tree node or there is no children node for further search.

There are three main operations in an NE-test, i.e., decoding the s elements (Line 10), testing on whether v_1 exists in the decoded elements (Line 11), and the branching operation for further search (Line 13). We discuss how these operations can be accelerated with SIMD in the following Section VI-B.

Algorithm 4: NE-test of v_1 against $f^{hyb+}(v_2)$ in Hyb+.

Input: Vertex v_1 and encoding $f^{hyb+}(v_2)$
Output: NE-test result of v_1 against $f^{hyb+}(v_2)$

```
1  Decode P_B[0] and P_B[1] from f^{hyb+}(v_2)
2  Return NO_PASS if v_1 ∈ {P_B[0], P_B[1]}
3  if v_1 < P_B[0] ∨ v_1 > P_B[1] then
4      if Hash(v_1) does not hit then
5          return PASS
6      return NO_PASS
7  /* v_1 ∈ (P_B[0], P_B[1]) */
8  Set Node ID N_ID = 1   /* ID of T(B).root */
9  while true do
10     Let {v_{i_1}, .., v_{i_s}} = SIMD-Decode(N_ID, f^{hyb+}(v_1))
11     Let isExist = SIMD-Membership({v_{i_1}, .., v_{i_s}}, v_1)
12     Return NO_PASS if isExist = true
13     Let branch = SIMD-Determine({v_{i_1}, .., v_{i_s}}, v_1)
14     Return PASS if branch = NULL
15     Set PreID = N_ID
16     N_ID = GetNodeID(PreID, branch)
```

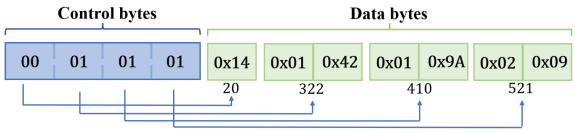

Fig. 6. Encoding example for sorted keys: {20, 322, 410, 521}.

B. SIMD Acceleration Over SS-Tree

Consider a block B, the SS-tree $T(B)$ with scalar value s and the corresponding new permutation P_B. We incorporate an SIMD-oriented encoding over array-implemented SS-tree such that we can not only improve the compression ratio, but also enable SIMD capabilities for accelerating the decoding of NE-tests.

1) SIMD-Oriented Encoding: There are two different parts in P_B, i.e., the first two vertices $(P_B[0], P_B[1])$ and the remaining part of elements in $T(B)$. We use straightforward encoding to compress each of $P_B[0]$ and $P_B[1]$ into $\lceil \log_2(|V|) \rceil$ bits on the range encoding part. Remaining items in P_B are groups of s elements for SS-tree nodes. We apply Stream VBytes [38] to encode these groups of elements. This encoding over a sequence of integers contains two parts: control bytes and data bytes. The former indicates the number of bytes occupied by each integer, while the latter stores integers subsequently. In the mainstream computer architecture, it usually takes at most 4 bytes to record a 32-bit integer. Hence, data bytes part uses a variable-length (4, at most) sequence of bytes to store an integer. We can always use 2 bits to indicate the number of bytes for an integer, and that is to say, one control byte (8 bits) can indicate the variable-lengths of 4 integers, which is well suited for one SIMD instruction, since 4 is usually a divisor of scalar value s. For the sake of presentation only, we assume that $s = 4$ in the following discussion.

Fig. 6 presents an encoding example over integer sequence {20, 322, 410, 521}. We can see that these 4 integers occupy 1, 1, 2, and 2 bytes respectively, thus, the corresponding control byte is 00000101. In this way, we can divide all vertices in neighbor block into groups of (at most) 4 and encode these groups together with the corresponding control bytes according to Stream VByte strategy. This encoding itself improves the compression ratio because of the variable-length based storage for integers. We can also further improve the compression ratio by applying differential coding to the node keys. For example, for a set of node keys $\{x_1, x_2, x_3, x_4\}$, we can encode them as $\{x_1, x_2 - x_1, x_3 - x_2, x_4 - x_3\}$. More importantly, once we read a control byte for decoding, we can accordingly get the corresponding permutation of four variable-lengths and locate the bits for encoded four integers, which can be decoded out simultaneously with the "shuffle" operation of SIMD and then accelerate the decoding [38].

2) SIMD-Accelerated Decoding: Let's discuss our decoding process with SIMD acceleration over the NE-test of v_1 against $f^{hyb+}(v_2)$. Without loss of generality, we focus on the case when $P_B[0] < v_1 < P_B[1]$. Starting from the root node, we traverse the encoded SS-tree until we reach the desired vertex. Suppose we are currently visiting a node with ID i. Since we have divided the encoding into control and data bytes, we need to calculate the encoding offset for the data byte of the current node. This can be easily obtained by accumulating the length of data indicated by the first $i - 1$ control bytes. The i-th control byte records the length of the data bytes for the sorted keys of the current tree node. After obtaining the corresponding data bytes, we need to conduct the decoding with two steps. The first is to convert the variable-length data bytes into a fixed-length integer list, which can be efficiently decoded using the "shuffle" operation of SIMD based on the control byte. The second is to restore the original list of integers. Recall the example of differential coding presented in Section VI-B1. Assuming that we get the list $\{x_1, x_2 - x_1, x_3 - x_2, x_4 - x_3\}$, we can restore the original list by building two additional lists, namely, $\{0, 0, x_2 - x_1, x_3 - x_2\}$ and $\{0, 0, x_1, x_2 - x_1\}$, over the given one. The whole computation can be accelerated by the "shift" and "addition" mechanism of SIMD. These two steps correspond to the "SIMD-Decode" function at Line 10 in Algorithm 4. Once we figure out the original list, we can conduct a membership query of v_1 against the list in parallel using the SIMD "compare" operation (Line 11 in Algorithm 4). If v_1 is not any of these elements, we can determine a branch of current tree node for further search with another SIMD "compare" (Line 13 in Algorithm 4). We would not stop our search until we find the target vertex or reach a leaf node.

VII. Experimental Evaluation

All methods are implemented in C++ and run on a CentOS machine of 128 G memory and two Intel(R) Xeon(R) Silver-4210R 2.40 GHz CPUs.The processor provides 128-bit registers to support the required SIMD instructions.Codes are available on Github [39].

We use six real world datasets in our experiments. **As-Sk** [40] dataset is an Internet topology graph generated from trace-routes. **Wiki** [41] is a graph of Wikipedia hyperlinks. **Uk** [42] is a 2005 crawl of the. UK domain performed by UbiCrawler [43]. **Gsh** is a large snapshot of the web taken in 2015 by BUbiNG [44]. **Orkut** [45] dataset is created from a free online social

TABLE I
SUMMARY OF DATASETS

Datasets	$\|V\|$	$\|E\|$	d	Power -law	Ratio of Encoded Vertices					Ratio of Encoded Edges				
					k=2	k=4	k=8	k=16	k=32	k=2	k=4	k=8	k=16	k=32
As-Sk	1.6M	11.0M	13	✓	13.6%	31.1%	73.3%	94.43%	99.42%	2.0%	7.2%	30.3%	76.96%	91.39%
Wiki	1.7M	25.4M	28	✓	2.8%	7.7%	34.8%	72.9%	97.20%	0.0%	0.3%	8.7%	59.2%	88.85%
Uk	39.4M	783.0M	40	✓	18.1%	22.8%	34.9%	56.7%	81.7%	0.5%	1.2%	4.1%	16.4%	44.7%
Gsh	988.4M	25.6B	52	✓	11.2%	16.1%	33.0%	45.0%	75.9%	0.2%	1.0%	2.1%	11.0%	35.6%
Orkut	3.0M	117.1M	76	✓	3.6%	4.9%	9.9%	27.0%	70.2%	0.0%	0.1%	0.7%	5.4%	22.0%
Cage	1.5M	27.1M	36	×	0.0%	0.0%	0.0%	50.8%	96.4%	0.0%	0.0%	0.0%	47.9%	90.2%

network where users form friendships each other. **Cage** [46] dataset is a collection of CAGE tags representing transcript ends for gene expression analysis. The default storage backend for adjacent list is RocksDB [47] (on disk). Each graph is taken as undirected and the adjacent list of each vertex contains both in and out neighbors. For each dataset, we set five different dimension numbers k: 2, 4, 8, 16, 32.

Table I presents the statistical information about these datasets. Note that the first 5 graphs (i.e., As-Sk, Wiki, Uk, Gsh and Orkut) are indeed of power-law distribution while Cage follows a non-power-law one where most vertices are of degree larger than 10. We can see from Table I that, for graph Cage, only a quite tiny part of vertices and edges are encoded when k is not larger than 10. Note that we omit all results when k is larger than the average degree. Specifically, VEND is proposed to avoid disk accesses for no-result edge queries, and if k is larger than the average degree, we can just load the entire graph into memory avoid any disk access. Due to space limitation, we demonstrate these graph distributions in Fig. 12 in Appendix D, where we also present more discussions for graph distributions.

We also present a series case studies in Appendix. Specifically, we demonstrate VEND acceleration for disk-based triangle counting and edge query operations in graph database Neo4j [1] (Appendix E.1). We also compare our methods against in-memory graph framework Aspen [48]. Additionally, we extend our methods over directed graphs and evaluate our solutions over a directed graph dataset Pokec [49] (Appendix E.3). More extra evaluations are available in Appendix.

A. Comparative Setting

We evaluate our solutions against four Bloom filter (BF) based methods. The first is standard BF (SBF) built over a bitset of size $|V| \cdot k \cdot I$, where we hash each edge with the corresponding two adjacent vertices as input. Note that edge deletion requires reconstruction to guarantee consistency. The second one is Blocked BF (BBF) [33] where the bitset is partitioned into a series of smaller standard BF (blocks) and when a deletion happens, only the affected block need to be reconstructed. We set the block size to 512 according to [33]. The last one is the variant of the BF that is only applied to encode vertices in the core subgraph. Similar to the hybrid version, vertices not in K-core are explicitly encoded. We denote this version as local BF (LBF), which apparently can be efficiently updated without global reconstruction, and hence we only applied the standard BF there in view of its highest false positive rate. Apparently, bit-hash version in Section IV-D is essence a special case of LBF. The optimal number of hash functions in BF can be computed by $(\ln 2 \cdot m)/n$ where m is the average number of items to be hashed and n is the fixed size of hash slot [29]. We also include the range version (Section IV-C) for comparison, which can be maintained in a similar way to that of hybrid. Further more,we implement the hybrid optimized version (Section VI) with SSE SIMD instruction set that can operate 4 scalars in parallel. We first evaluate these VEND solutions from three aspects: VEND score (Section VII-B), edge queries acceleration (Section VII-C) and the maintenance efficiency (Section VII-D).

B. VEND Score

We evaluate the VEND score of each version over given datasets. Note that we did not enumerate all vertex pairs in $V \times V$ to count the precise number of NEpairs, which is more than a thousand billions. In fact, we randomly generate one billion vertex pairs for each dataset as edge queries. We also create another set of edge queries where the corresponding two vertices are close to each other in light of the locality of many edge query related computation, such as clustering coefficient (triangle counting) and subgraph matching. We generate this edge query set by sampling pairs of vertices having at least one common neighbor. The VEND scores over these two query sets are presented in Figs. 7 and 8, respectively. We can see that our methods and SBF score almost equally highest, and hyb+ VEND is even better than hybrid version. Over Cage dataset, both hyb+ and hybrid VEND outperform comparative ones, which confirms that our techniques are applicable not only on graphs of power-law distributions but also those of non-power-law ones. An interesting observation is that performance gaps of all methods in Fig. 7 are significantly smaller that those in Fig. 8. Actually, an equivalent problem of VEND is to determine vertex pairs where the two vertices are of distance larger than 1. Hence, for randomly generated vertex pairs of which the two vertices tend to be far from each other, even simple VEND ideas could identify most NEpairs, that is why performance gaps in Fig. 6 is small. While, for vertex pairs of common neighbors, the distance of those two vertices are at most 2, it is not easy for those naive VEND solutions to identify the corresponding NEpairs. Hence, in the scenario where edge queries mainly happen on vertices from the same local part of graph, our performance advantages would be more significant.

C. Edge Query Acceleration

For edge queries acceleration, we generate two query sets where the first contains one million randomly generated vertex pairs (denoted as $RandPair$) while the second set is built by sampling one million vertex pairs over those of common neighbor (denoted as $CommPair$). We report the total time for answering these edge queries in Fig. 9. We can see from that all VEND solutions exhibits considerable acceleration on edge queries over both power-law and non-power-law graphs. Our hyb+ VEND perform best while hybrid version performs similarly to SBF. All these three methods significantly outperform the others. Also, the Non-VEND version is the slowest than any VEND-accelerated method. In fact, graphs are stored on disk while VEND encoding are persisted in memory, and hence, VEND-accelerated methods would filter lots of disk accesses for

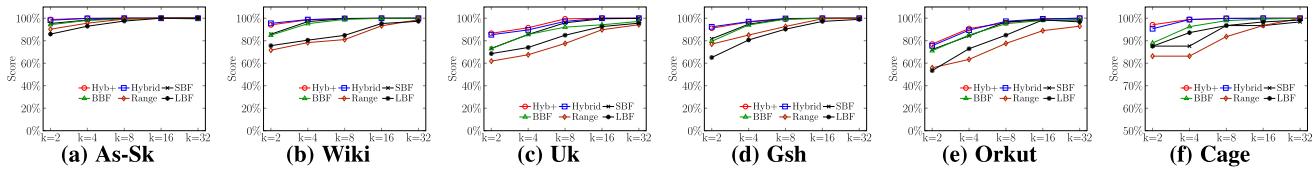

Fig. 7. VEND score over different datasets on randomly generated vertex pairs.

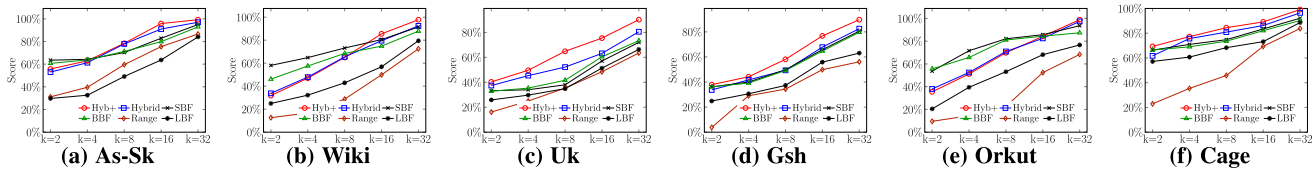

Fig. 8. VEND score over different datasets on vertex pairs of common neighbor.

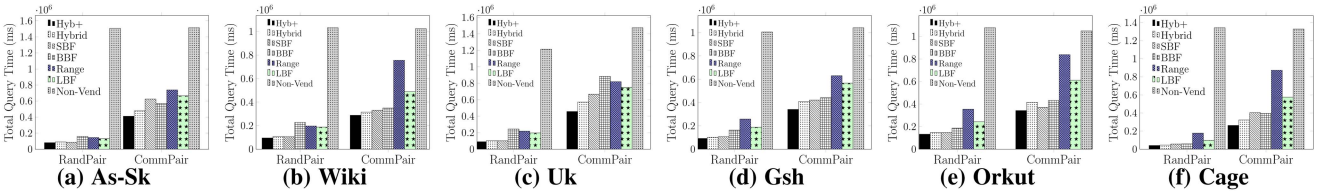

Fig. 9. Total time over different edge query sets ($k = 8$).

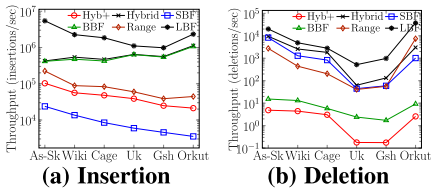

Fig. 10. Throughput for addressing edge updates ($k = 8$).

no-result edge queries with in-memory NDF, while Non-VEND version would always conduct disk accesses for each query.

D. Maintenance Evaluation

We compare our work against comparative ones on maintenance efficiency. We sample 100,000 existing edges for conducting edge deletions and randomly generate another 100,000 new edges for edge insertion. Note that we do not include insertion/deletion of vertices in our evaluation since they are trivial compared to those of edges. Edge insertions and deletions are evaluated independently and we report throughput for addressing these updates in each group. We only consider the time cost for updating vectors and we omit the time for committing updates in storage, which varies a lot on different graph databases.

We can see from Fig. 10 that although comparative works are more efficient than our method on insertion, we can still address nearly tens of thousands of edge insertions per second. An important observation is that performances of SBF and BBF are so terrible that they can hardly be applied in real-world scenarios. In general, our method is apparently the best with high VEND score and efficient maintenance. Also, edge deletion takes much more time that insertion. ① For BF based methods (i.e., SBF, BBF and LBF), they can incrementally update the slot with trivial hash operations for insertion. While, for deletion, related hash slots need to be totally reconstructed when deletions happen. We can also see that deletion maintenance of LBF is much more efficient than that of SBF and BBF since LBF create a relatively small and independent slot for each vertex, and hence the corresponding reconstruction would be more lightweight. ② For our counterparts, as indicated in Section V-D1, there are some cases for edge insertion that we need no encoding construction, while for edge deletion (see Section V-D2), we always need at least one encoding reconstruction. Therefore, the insertion throughput is usually larger than that of deletion.

E. Index Construction and Space Usage

We report the index construction time and space cost of hybrid and hyb+ VEND in Table II. Hyb+ costs a bit (about 10%) more index construction time than that of hybrid version, since encoding of the former is more complicated than that of the latter. Note that space costs of hybrid and hyb+ versions are the same under a given k, since both of their data structure are $|V(G)|$ vectors of length k. Percentage numbers present the corresponding proportion of space saved by VEND. We can see that VEND memory usage is highly linear to k. Some space costs

TABLE II
INDEX CONSTRUCTION AND MEMORY EFFICIENCY

Dataset	$\|G\|$	Index Space $\|f\|\ (1-\|f\|/\|G\|)$					Construction Time (k=8)	
		k=2	k=4	k = 8	k=16	k=32	Hybrid	Hyb+
As-Sk	84M	13M(85%)	25M(69%)	51M(38%)	103M(N/A)	207M(N/A)	33 s	35.23 s
Wiki	194M	13M(93%)	27M(86%)	54M(71%)	109M(43%)	218M(N/A)	109 s	119.53 s
Uk	5.83G	301M(95%)	602M(89%)	1.17G(80%)	2.35G(60%)	4.70G(20%)	64 min	69 min
Gsh	191.41G	7.3G(96%)	14.7G(92%)	29.4G(84%)	58.9G(69%)	117G(38%)	23.57 h	26.12 h
Orkut	890.8M	23M(97%)	46M(95%)	93M(89%)	187M(78%)	375M(58%)	9.7 min	10.85 min
Cage	206M	11M(94%)	23M(88%)	46M(77%)	91M(55%)	183M(11%)	40.32s	49.63 s

are marked as N/A when the k is larger than the average degree, where VEND turns invalid according to our problem scenarios.

VIII. CONCLUSION

Edge query is one of the fundamental operations in graph databases. We propose vertex encoding based edge nonexistence determination (VEND) for accelerating edge queries. A VEND solution consists of a vertex encoding f and NEpair determination function (NDF) F, with which we efficiently filtering no-result edge queries, i.e., vertex pairs connected by no edges. We first design an efficient optimal partial VEND solution over a subset of vertices such that no-result edge queries related to these vertices can be efficiently and precisely detected. We also illustrate range-based and hash-based extensions over the optimal partial version, after which we propose a hybrid VEND solution incorporating both range and hash ideas, as well as an efficient maintenance algorithm over the hybrid version when edge insertions/deletions happen. Furthermore, NDF is a fine-grained and frequently used operation, and we incorporate SIMD acceleration over our hybrid VEND solution, forming a hyb+ version. Extensive experiments show that our solution performs well on real-world datasets and is able to detect most no-result edge queries.

REFERENCES

[1] J. J. Miller, "Graph database applications and concepts with Neo4j," in *Proc. southern Assoc. Inf. Syst. Conf.*, Atlanta, GA, USA, 2013, vol. 2324, no. 36, pp. 141–147.

[2] A. Deutsch, Y. Xu, M. Wu, and V. E. Lee, "Aggregation support for modern graph analytics in tigergraph," in *Proc. Int. Conf. Manage. Data*, D. Maier, R. Pottinger, A. Doan, W.-C. Tan, A. Alawini, and H. Q. Ngo, Eds., Portland, OR, USA, Jun. 14–19, 2020, pp. 377–392. [Online]. Available: https://doi.org/10.1145/3318464.3386144

[3] "Nebula graph," 2022, Accessed: Jan. 15 . [Online]. Available: https://nebula-graph.io/

[4] "Janusgraph," 2022, Accessed: Jan. 15, . [Online]. Available: https://janusgraph.org/

[5] T. Neumann and G. Weikum, "The RDF-3X engine for scalable management of RDF data," *VLDB J.*, vol. 19, no. 1, pp. 91–113, 2010.

[6] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao, "gStore: Answering SPARQL queries via subgraph matching," *Proc. VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

[7] S. N. Soffer and A. Vazquez, "Network clustering coefficient without degree-correlation biases," *Phys. Rev. E*, vol. 71, no. 5, 2005, Art. no. 057101.

[8] M. A. Hasan and V. S. Dave, "Triangle counting in large networks: A review," *Wiley Interdiscipl. Rev.: Data Mining Knowl. Discov.*, vol. 8, no. 2, 2018, Art. no. e1226.

[9] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles in massive networks," in *Proc. 22nd ACM Int. Conf. Inf. Knowl. Manage.*, 2013, pp. 529–538.

[10] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel triangle counting in massive streaming graphs," in *Proc. 22nd ACM Int. Conf. Inf. Knowl. Manage.*, 2013, pp. 781–786.

[11] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proc. Int. Conf. Manage. Data*, 2016, pp. 1199–1214.

[12] J. Leskovec and A. Krevl, "SNAP datasets: Stanford large network dataset collection," Jun. 2014. Accessed: Sep. 15, 2023. [Online]. Available: http://snap.stanford.edu/data

[13] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song, "Heterogeneous graph neural networks for malicious account detection," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 2077–2085.

[14] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 1695–1698. [Online]. Available: https://doi.org/10.1145/3035918.3056445

[15] Y. Cui, D. Xiao, D. B. H. Cline, and D. Loguinov, "Improving I/O complexity of triangle enumeration," *IEEE Trans. Knowl. Data Eng.*, vol. 34, no. 4, pp. 1815–1828, Apr. 2022.

[16] J. Zhou et al., "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[17] S. T. Roweis and L. K. Saul, "Nonlinear dimensionality reduction by locally linear embedding," *Science*, vol. 290, no. 5500, pp. 2323–2326, 2000.

[18] M. Belkin and P. Niyogi, "Laplacian eigenmaps for dimensionality reduction and data representation," *Neural Comput.*, vol. 15, no. 6, pp. 1373–1396, 2003.

[19] A. Ahmed, N. Shervashidze, S. Narayanamurthy, V. Josifovski, and A. J. Smola, "Distributed large-scale natural graph factorization," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 37–48.

[20] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proc. 20th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2014, pp. 701–710.

[21] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proc. 22nd Int. Conf. Knowl. Discov. Data Mining*, 2016, pp. 855–864.

[22] J. Tang, M. Qu, M. Wang, M. Zhang, J. Yan, and Q. Mei, "Line: Large-scale information network embedding," in *Proc. 24th Int. Conf. World Wide Web*, 2015, pp. 1067–1077.

[23] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. 31st Int. Conf. Neural Inf. Process. Syst.*, 2017, pp. 1025–1035.

[24] P. Velickovic, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *Stat*, vol. 1050, p. 4, 2018.

[25] L. Lü and T. Zhou, "Link prediction in complex networks: A survey," *Physica A: Stat. Mechanics Appl.*, vol. 390, no. 6, pp. 1150–1170, 2011.

[26] M. E. Newman, "Clustering and preferential attachment in growing networks," *Phys. Rev. E*, vol. 64, no. 2, 2001, Art. no. 025102.

[27] G. Kossinets, "Effects of missing data in social networks," *Social Netw.*, vol. 28, no. 3, pp. 247–268, 2006.

[28] L. Katz, "A new status index derived from sociometric analysis," *Psychometrika*, vol. 18, no. 1, pp. 39–43, 1953.

[29] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing bloom filter: Challenges, solutions, and comparisons," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 2, pp. 1912–1949, Secondquarter 2019.

[30] L. Fan, P. Cao, J. M. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281–293, Jan. 2000.

[31] C. E. Rothenberg, C. A. B. Macapuna, F. L. Verdi, and M. F. Magalhães, "The deletable Bloom filter: A new member of the Bloom family," *IEEE Commun. Lett.*, vol. 14, no. 6, pp. 557–559, Jun. 2010.

[32] H. Lim, J. Lee, H. Y. Byun, and C. Yim, "Ternary bloom filter replacing counting bloom filter," *IEEE Commun. Lett.*, vol. 21, no. 2, pp. 278–281, Feb. 2017. [Online]. Available: https://doi.org/10.1109/LCOMM.2016.2624286

[33] F. Putze, P. Sanders, and J. Singler, "Cache-, hash-, and space-efficient bloom filters," *ACM J. Exp. Algorithmics*, vol. 14, pp. 4.4–4.18, 20010. [Online]. Available: https://doi.org/10.1145/1498698.1594230

[34] S. B. Seidman, "Network structure and minimum degree," *Social Netw.*, vol. 5, no. 3, pp. 269–287, 1983.

[35] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson, "An experimental study of bitmap compression vs. inverted list compression," in *Proc. ACM Int. Conf. Manage. Data*, 2017, pp. 993–1008.

[36] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2002, pp. 145–156. [Online]. Available: https://doi.org/10.1145/564691.564709

[37] Wikipedia Contributors, "Binary search tree – Wikipedia, the free encyclopedia," 2023, Accessed: Jan. 15, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Binary_search_tree

[38] D. Lemire, N. Kurz, and C. Rupp, "Stream vbyte: Faster byte-oriented integer compression," *Inf. Process. Lett.*, vol. 130, pp. 1–6, 2018.
[39] "Codes," 2022. Accessed: Sep. 15, 2023. [Online]. Available: https://github.com/hnuGraph/VEND
[40] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: Densification laws, shrinking diameters and possible explanations," in *Proc. 11th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2005, pp. 177–187.
[41] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2017, pp. 555–564.
[42] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "uk-2005: A crawl of the. uk domain performed by ubicrawler," 2005. Accessed: Sep. 15, 2023. [Online]. Available: https://law.di.unimi.it/webdata/uk-2005/
[43] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: A scalable fully distributed web crawler," *Softw.: Pract. Experience*, vol. 34, no. 8, pp. 711–726, 2004. [Online]. Available: https://doi.org/10.1002/spe.587
[44] P. Boldi, A. Marino, M. Santini, and S. Vigna, "Bubing: Massive crawling for the masses," *ACM Trans. Web*, vol. 12, no. 2, pp. 12:1–12:26, 2018.
[45] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowl. Inf. Syst.*, vol. 42, no. 1, pp. 181–213, 2015.
[46] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proc. AAAI Conf. Artif. Intell.*, 2015, pp. 4292–4293.
[47] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "RocksDB: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Trans. Storage*, vol. 17, no. 4, pp. 1–32, 2021.
[48] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *Proc. 40th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2019, pp. 918–934. [Online]. Available: https://doi.org/10.1145/3314221.3314598
[49] L. Takac and M. Zabovsky, "Data analysis in public social networks," in *Proc. Int. Sci. Conf. Int. Workshop Present Day Trends Innovations*, 2012, pp. 1–6.

Hangyu Zheng received the BS degree from Nanchang University, Nanchang, China, in 2021. He is currently working toward the MS degree in graph search from the College of Computer Science and Electronic Engineering, Hunan University, China. His research interests include graph query and graph database.

Youhuan Li (Member, IEEE) received the BS degree and the PhD degree in graph stream management from Peking University, China, in 2013 and 2018, respectively. From 2018 to 2020, he was a postdoc with Peking University and Tencent, Shenzhen, China, respectively. He is currently an associate professor with Hunan University. His research focuses on graph data management.

Fang Xiong received the BS degree in computer science and technology from Hunan Normal University, China, in 2002 and the PhD degree from the National University of Defense Technology, Changsha, China, in 2022. He is currently with Network Information Department, Xiangya Hospital, Central South University. His research mainly focuses on medical Big Data.

Xiaosen Li received the BS degree in software from Central South University, China, in 2012 and the master's degree in software engineering from Peking University, China, in 2015. In 2018, he was a senior research engineer with Tencent, China. His research interests mainly include data mining and machine learning.

Lei Zou is currently a professor with the Institute of Computer Science and Technology, Peking University, China. He is also a faculty member with Big Data Center, Peking University and Beijing Institute of Big Data Research, respectively. His research interests include graph database and semantic data management.

Peifan Shi received the BS degree from the Beijing University of Chemical Technology, China, in 2021. She is currently working toward the MS degree in distributed graph search with Hunan University. Her research research focuses on graph search.

Zheng Qin received the PhD degree in cyber-security from Chongqing University, China, in 2001. He is currently a professor with Hunan University. His research interests mainly include network and data security and privacy.