VEND: Vertex Encoding for Edge Nonexistence Determination

Youhuan Li[1], Hangyu Zheng[1], Lei Zou[2£], Xiaosen Li[3£], Ziming Li[1], Pin Xiao[3], Yangyu Tao[3], Zheng Qin[1]
[1]*College of Computer Science and Electronic Engineering, Hunan University, China;*
[2]*Peking University, China;*
[3]*Tencent Inc., China;*
[1]{liyouhuan,zhenghangyu,zimingli,zqin}@hnu.edu.cn
[2]zoulei@pku.edu.cn, [3]{hansenli,payniexiao,brucetao}@tencent.com

***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 the corresponding 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. 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 performance of graph database. In this paper, 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. With VEND, we can utilize in-memory efficient operations to filter no-result disk accesses for edge query. We also design maintenance algorithms for the proposed solution when data updates happen. Extensive experiments on many real-world datasets confirm the ability of our solution on determining a quite high proportion of non-edge vertex pairs, as well as the acceleration for edge queries.**

***Index Terms*—Graph Database, Edge Query, 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 graph are connected by no edges. In fact, average degree of a large real-world graph tends to be much less than the vertex number [12]. Therefore, for each vertex, there are far fewer vertices adjacent to it than those 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 incur 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.

[£]Corresponding author

In this paper, 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.

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.

A. Applications

1) Relation Retrieval: Relation retrieval over entities is the most basic application of edge query, such as friendship check over social network, transactions search in payment graph and 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. Since triangle is a common sub-structure in graph, we also extend our discussions to SOTA disk based subgraph matching solutions.

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 ($O(|adj(j)|)$).

Algorithm 1: Edge Iterator based Triangle Counting

```
   Input: Graph $G = (V, E)$
1  $count = 0$
2  for each node $i \in V$ 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
             $j' \in adj(i) \wedge j < j'$ then
6              Continue;
7          Get $adj(j)$ from storage disk
8          let $K = \{k \in adj(j) \mid j < k\}$
9          $count = count + |adj(i) \cap K|$
10 return $count$
```

Algorithm 2: Disked based Triangle Counting [14]

```
   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 $p = \lceil |E|/M \rceil$ 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 $i < j \in adj(i)$ do
6          Let $K = \{k \in adj(i) \mid j < k\} \cap [L_P, U_P)$
7          if $VEND(j, k) = NO\_EDGE$ for $\forall k \in K$ 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         $count = count + |J \cap K|$
15 return $count$
```

Trigon [14] 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 for the triangle counting. 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.

Algorithm 3: Diamond Query Evaluation in Graphflow

```
   Input: Graph $G = (V, E)$, diamond subgraph $Q$
1  Compute candidate set $Cand(X)$ for $X \in \{A, B, C, D\}$
2  Let $M_A = Cand(A)$ for each $a \in M_A$ do
3      Get $adj(a)$ from database on disk
4      Let $M_{AB} = \{(a, b) \mid b \in (adj(a) \cap Cand(B))\}$
5      for each $(a, b) \in M_{AB}$ do
6          Let $M_{ABC} = \{(a, b, c) \mid c \in (adj(a) \cap Cand(C))\}$
7          for each $(a, b, c) \in M_{ABC}$ do
8              Get $adj(b)$ from database on disk
9              if $VEND(c, d') = NO\_EDGE$ for each
                 $d' \in adj(b)$ then
10                 break;
11             Get $adj(c)$ from database on disk
12             Let $S = adj(b) \cap adj(c) \cap Cand(D)$
13             Output $\{(a, b, c, d) \mid d \in S\}$
```

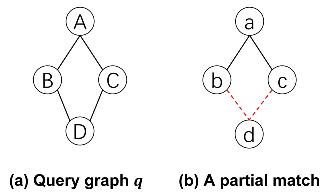


Fig. 1: A running example query in Graphflow [15].

3) Subgraph Matching with WCOJ: Graphflow [15] is the SOTA external-memory subgraph matching solution, which utilizes Worst-Case Optimal Join (WCOJ) over graph databases. Intersection between adjacent lists is a common operation in WCOJ, which is similar to edge iteration based triangle counting. Hence, we extend our discussions on VEND acceleration to Graphflow. For example, Figure 1(a) presents a diamond subgraph Q that is used as a running example query in Graphflow [15]. Let's discuss how Q is evaluated with WCOJ and how VEND could reduce the adjacent list retrievals. Algorithm 3 presents the pseudo codes for the evaluation. Let $Cand(X)$ denotes the candidate vertices of $X \in \{A, B, C, D\}$. For each $a \in Cand(A)$, Graphflow retrieves $adj(a)$ from the database and then conducts $adj(a) \cap Cand(B)$ and $adj(a) \cap Cand(C)$ successively forming partial matches of sub-query $\{A, B, C\}$. Assume that the subgraph $\{a, b, c\}$ in Figure 1(b) is a match of sub-query $\{A, B, C\}$. We know that neither $adj(b)$ nor $adj(c)$ has been loaded from database and hence they are not indexed in memory yet. To extend subgraph $\{a, b, c\}$ into a full match of Q, Graphflow would retrieve both $adj(b)$ and $adj(c)$, and then conduct $adj(b) \cap adj(c) \cap Cand(D)$ for full matches. In fact, at the time when $adj(b)$ is loaded while $adj(c)$ not, we can conduct VEND tests between c and each vertices in $adj(b)$ (Line 9 in Algorithm

3). If c is confirmed to be not adjacent to any vertex in $adj(b)$, the I/O cost of retrieving $adj(c)$ from database can be avoided (Line 11 in Algorithm 3) since $adj(b) \cap adj(c)$ must be $\emptyset$. Note that VEND accelerates the algorithm by reducing I/O cost for loading adjacent lists, which are neither loaded nor indexed in memory.

B. Our contributions

A VEND solution can reduce no-result edge query executions and improve system performance, especially for the graph that is too large to be stored in memory. We summarize our contributions as follows:

- To the best of our knowledge, we are the first to propose and solve VEND problem. We formally define VEND and design an important measure (VEND score) to precisely evaluate the performance of VEND solutions (Section III). VEND problem is important since it can avoid no-result edge query executions, which are primary and frequently used operations in graph computations. We implement many VEND solutions and we find that even the baseline could bring performance improvements. Hence, VEND is important.
- We propose an effective VEND solution. We first design a partial solution (Section IV) which can perfectly determine all no-result edge queries related to a subset of vertices. Then we further design range based and hash based baselines over the partial solution (Section V). We finally propose a uniform hybrid VEND solution incorporating both range based ideas and hash based methods (Section VI). We also design efficient update algorithms over the final version.
- We conduct various experiments to evaluate our solutions. We find that even the basic VEND solution can explicitly improve the corresponding system performance. We also confirm the effectiveness of optimizations we propose when developing our final VEND solution and the corresponding maintenance.

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.

a) 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.

b) 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.

c) 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 use $degr_G(v)$ to denote the degree of vertex v in G. Also, to make the context more clear, 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 $G = (V, E)$ and a dimension number k, encoding function f is defined over V, where for each vertex $v \in V$, $f(v)$ is a k-dimension vector of integers. We use $f(v)[i]$ to denote the corresponding 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 $G = (V, E)$ and v_1, $v_2 \in V$, 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 in G. We use $\mathbb{NE}(G)$ to denotes the set of NEpairs in G, namely:*

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

If (v_1, v_2) is an NEpair, we say that v_1 and v_2 are NEneighbors of each other.

The core target of VEND is to determine NEpairs as many as possible. In consideration of efficiency, determinations from VEND are required to be made just based on vertex vectors of v_1 and v_2. We formally define a determination function over vertex vectors.

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 dimension number k, the proposed problem 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. We call the 2-tuple (f, F) as a VEND solution for G with dimension number k.*

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 paper. There could be some variants of VEND, such as edge (non)existence with direction constraints, length constrained reachability determination with vertex encoding and so on, which could be future works.

a) 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 Section V, we extend the partial VEND into two full versions with range-based encoding and hash-based encoding, respectively. Note that our discussions on these two full versions are to introduce the range-based and hash-based encoding ideas. We do not discuss their maintenance since they are not the final version. In Section VI, we present our final VEND solution incorporating both range-based and hash-based methods. Further more, we discuss how to update the proposed encoding when updates happen. We evaluate our methods in Section VII and conclude in Section VIII.

IV. An Optimal 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. With (f^α, F^α), the design of full VEND solution over G need only consider the induced subgraph over vertices that have not been encoded. We introduce the encoding function f^α in Section IV-A and NDF F^α in Section IV-B.

A. Encoding Function f^α

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

- Step 0: We initial $i = 1$ and build a set of comparative flag τ_i ($\forall i$, $\tau_i < \tau_{i+1}$) that is distinguish 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.,
$$f^\alpha(v) = [\tau_i, v_1, v_2, \cdots, v_{|N_{G_i}(v)|}] \tag{1}$$
where v_1, v_2,$\cdots$,$v_{|N_{G_i}(v)|}$ are neighbors in $N_{G_i}(v)$.
- Step 2: Remove all vertices of 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 of degree less than k, let $i = i+1$ and repeat Steps 1 and 2; otherwise, terminate.

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

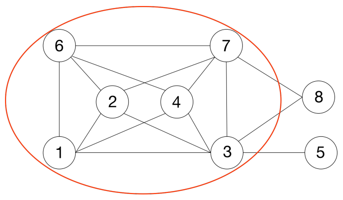


Fig. 2: A data graph example and the core subgraph.

B. NDF in Partial Solution

Let's discuss how to design the NDF F^α over f^α. We know that in each iterator of the construction, the remaining neighbors of each vertex v of 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^α_k, 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^α_k, whether (v_1, v_2) is an NEpair can not be determined by f^α. For example, consider f^α over the graph in Figure 2. $8 \in V^\alpha_3$ 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 followings:

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

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

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

V. Range & Hash based Baselines

Let's consider the VEND solution over core subgraph C^k_G of G. According to previous discussions, any VEND solution over C^k_G can be easily merged with (f^α, F^α), forming a full VEND solution for G. We propose a range-based VEND solution in Section V-A and a hash-based VEND solution in Section V-B. The presentation of these two solutions is to introduce range-based and hash-based ideas that will be used in our finial VEND solution in Section VI.

A. Range-based Encoding

According to previous discussions, vertices in $V(C^k_G)$ may be of degree larger than k, and hence we can not just encode all neighbor IDs into the vector. A straightforward method is to set encoding vector of each vertex v with a subset of $N_{C^k_G}(v)$, leaving other neighbors not recorded. For example, $\forall$ v in $V(C^k_G)$, assuming that $N_{C^k_G}(v)$ $=$ $\{v_1$, v_2, $\cdots$, $v_t\}$ where t $\ge$ k and v_i $<$ v_j for $1 \le i < j \le t$, we can set encoding vector of v with the smallest k neighbor IDs in $N_{C^k_G}(v)$, i.e., $[v_1$, v_2, $\cdots$, $v_k]$. Thus, $\forall$ $v' \in V(C^k_G)$ where $v' \le v_k$, 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^3_G in red cycle in Figure 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 more than one blocks that can be used for build 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.

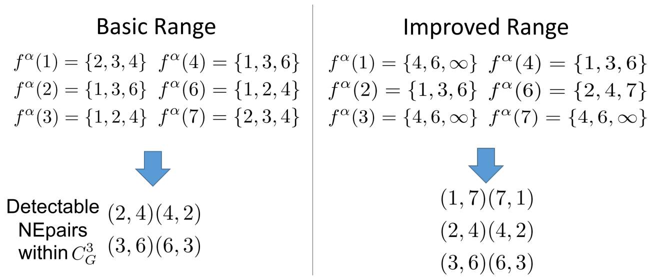

Basic Range
$f^\alpha(1) = \{2,3,4\}$ $f^\alpha(4) = \{1,3,6\}$
$f^\alpha(2) = \{1,3,6\}$ $f^\alpha(6) = \{1,2,4\}$
$f^\alpha(3) = \{1,2,4\}$ $f^\alpha(7) = \{2,3,4\}$
Detectable NEpairs within C^3_G
$(2,4)(4,2)$
$(3,6)(6,3)$
Improved Range
$f^\alpha(1) = \{4,6,\infty\}$ $f^\alpha(4) = \{1,3,6\}$
$f^\alpha(2) = \{1,3,6\}$ $f^\alpha(6) = \{2,4,7\}$
$f^\alpha(3) = \{4,6,\infty\}$ $f^\alpha(7) = \{4,6,\infty\}$
$(1,7)(7,1)$
$(2,4)(4,2)$
$(3,6)(6,3)$

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

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

- *Each nonempty subsequence of* s *is called as a neighbor block of* v*. Neighbor block is usually called as 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*:* $\{-\infty, v_1, \cdots, v_{k-1}\}, \cdots, \{v_{|N_G(v)|-k+2}, \cdots, v_{|N_G(v)|}, \infty\}$*.*
- *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, range of block* $\{-\infty, v_1, \cdots, v_{k-1}\}$ *is 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, vertices 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 $B = \{-\infty, v_1, \cdots, v_{k-1}\}$ 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. Figure 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),
  - 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:

$$\begin{aligned} F^R(v_1, v_2) = &(v_1 \in R_2 \wedge v_1 \notin f^R(v_2)) \\ &\vee (v_2 \in R_1 \wedge v_2 \notin f^R(v_1)) \end{aligned}$$

    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.

B. 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 $f^{hash}(v)[i] = 1$ $(0 \le i < k)$ 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 Figure 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.

VI. Final VEND Solution

We now present our final VEND solution, which is a hybrid one incorporating 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 VI-A, based on which we discuss some important extensions in Section VI-B. We formally introduce our hybrid VEND solution in Section VI-C and discuss the corresponding maintenance in Section VI-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 hybrid version, which together forms the final VEND solution in Section VI-C.

Dynamic selection of block. We can strategically select a uniform k' to encoding vertex in $V(C_G^k)$ by maximizing VEND score as much as possible. However, a uniform k' may not be a 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 VI-C3.

Encoding compression. The essence of a block is an ordered integer sequence. Lots of methods can be used to compress ordered integer sequence 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 $\lceil \log_2(|V|) \rceil \leq I' \leq I$ (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 VI-D on the maintenance of VEND.

One bit to 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 Equation 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 Equation 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: $\{-\infty, v_1, \cdots, v_{k-1}\}, \cdots, \{v_{x-k+2}, \cdots, v_x, \infty\}$. 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 $\{-\infty, v_1, \cdots, v_{k-1}\}$ 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 final VEND solution with those optimizations in Section VI-B. We use (f^{hyb}, F^{hyb}) to denote our final 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^*. Note that we do not need to specify the number of encoded neighbor IDs in the bitset since the remaining unused bits are all cleared. When decoding $f^{hyb}(v)$, we can just terminate when an ID of 0 is encountered. 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 $v \in V(C_G^{k^*+1})$ (i.e., $V - V_{k^*+1}^\alpha$), 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 VI-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 as 1.

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

2) NDF F^{hyb}: Let's discuss the computation of F^{hyb}. For the sake of presentation, we propose an important concept, called NE-test, that will be frequently used in the discussion.

Definition 8 (NE-test). *Consider a data graph G, a dimension k, the corresponding $V_{k^*+1}^\alpha$ 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 $f_{[0]}^{hyb}(v) = 0$, v' is not one of the IDs in $f^{hyb}(v)$.*
- *If $f_{[0]}^{hyb}(v) = 1$ (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 $v' \overset{NE}{\mapsto} f^{hyb}(v)$ to denote that v' can pass the NE-test of $f^{hyb}(v)$. Also, the set of vertices that can pass NE-test of $f^{hyb}(v)$ is denoted as $NT(f^{hyb}(v))$, namely:

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

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 $v \overset{NE}{\mapsto} f^{hyb}(v')$, namely:*

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

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 $C_G^{k^*+1}$.

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, if $v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2)$ and $v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$, then (v_1, v_2) is an NEpair.*

Proof. We prove this by reduction to absurdity. Assume that there is an edge $(v_1, v_2) \in E$ where $v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2)$ and $v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)$.

- ① If $f^{hyb}_{[0]}(v_1) = 0$ and $f^{hyb}_{[0]}(v_2) = 0$, then edge (v_1, v_2) is removed either at the time when we build $f^{hyb}_{[0]}(v_1)$ or at the time building $f^{hyb}_{[0]}(v_2)$. Hence, either v_1 can not pass the NE-test of $f^{hyb}_{[0]}(v_2)$ or v_2 can not pass the NE-test of $f^{hyb}_{[0]}(v_1)$, which is a contradiction to the assumption.
- ② If $f^{hyb}_{[0]}(v_1) = 1$ and $f^{hyb}_{[0]}(v_2) = 1$, then v_1 (v_2, resp.) must be encoded in $f^{hyb}_{[0]}(v_2)$ ($f^{hyb}_{[0]}(v_1)$, resp.) either with range based method or with hash based one according to the corresponding vector constructions. Hence, v_1 (v_2, resp.) can not pass the NE-test of $f^{hyb}_{[0]}(v_2)$ ($f^{hyb}_{[0]}(v_1)$, resp.), which is also a contradiction to the assumption.
- ③ If $f^{hyb}_{[0]}(v_1) = 0$ and $f^{hyb}_{[0]}(v_2) = 1$ (We omit the discussion for the symmetrical case), then v_2 must be encoded in $f^{hyb}_{[0]}(v_1)$ according to the corresponding vector constructions. Hence, v_2 can not pass the NE-test of $f^{hyb}_{[0]}(v_1)$, which is a contradiction to the assumption.

Since the assumption never holds, this theorem is proved. □

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

$$F^{hyb}(v_1, v_2) = \left(v_1 \overset{\text{NE}}{\mapsto} f^{hyb}(v_2)\right) \wedge \left(v_2 \overset{\text{NE}}{\mapsto} f^{hyb}(v_1)\right)$$

3) Block Selection: Let's consider the selection of block to be encoded in $f^{hyb}(v)$ for each vertex v in $V(C^{k^*+1}_G)$. 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 $B \in \mathbb{B}_{C^{k^*+1}_G}(v)$ where $N_{C^{k^*+1}_G}(v) = \{v_1, v_2, \cdots, v_x\}$ ($x = |N_{C^{k^*+1}_G}(v)|$), 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 building $f^{hyb}(v)$ for each block is $O(|N_{C^{k^*+1}_G}(v)|)$. 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 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

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

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 Equation 3. We propose a sliding-window like optimization on computing Z without generating $f^{hyb}(v)$ for each block, which costs only $O(m)$ time for computing $|NT(f^{hyb}(v))|$ for each block, and $O(k^* \cdot |N^{k^*+1}_G(v)| \cdot m)$ time in total for block selection.

Consider all t-size blocks in $\mathbb{B}_{C^{k^*+1}_G}(v)$ over $N_{C^{k^*+1}_G}(v) = \{v_1, v_2, \cdots, v_x\}$ where $x = |N_{C^{k^*+1}_G}(v)|$: $B_1 = \{v_1, \cdots, v_t\}$, $\cdots$, $B_{x-t+1} = \{v_{x-t+1}, \cdots, v_x\}$. We first build an array H_{B_1} of size m where $H_{B_1}[i]$ $(0 \leq i < m)$ record the number of such vertex $v' \in N_{C^{k^*+1}_G}(v) \setminus B_1$ 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 a 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 (Equation

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 $N_{C_G^{k^*+1}}(v) \setminus B_2$ are 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 use 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

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

where $0 \le t \le k^*$. Hence, 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, 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, while for the sake of presentation, we use G^u (G, resp.) to uniformly denote the data graph after (before, resp.) the update. Note that $HybEncode(v, V')$ (See Section VI-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 set 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, similar to the above case, 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 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 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 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 VI-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 on 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 be not 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.

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.

VII. Experimental Evaluation

All methods are implemented in C++ and run on a CentOS machine of 128G memory and two Intel(R) Xeon(R) Silver-4210R 2.40GHz CPUs. Codes are available on Github [36].

We use five real world datasets in our experiments. **As-Skitter** [37] dataset is an Internet topology graph generated from trace-routes. **Wiki-topcats** [38] is a graph of Wikipedia hyperlinks. **Uk-2005** [39] is a 2005 crawl of the .uk domain performed by UbiCrawler [40]. **Gsh-2015** is a large snapshot of the web taken in 2015 by BUbiNG [41]. **Orkut** [42] dataset is created from a free online social network where users form friendships each other. Table I presents the statistical information about these datasets. The default storage backend for adjacent list is RocksDB [43] (on disk). Each graph is taken as undirected and the adjacent list of each vertex contains both in and out neighbors.

TABLE I: Summary of Datasets

Datasets	$\lvert V\rvert$	$\lvert E\rvert$	$d = \frac{2\lvert E\rvert}{\lvert V\rvert}$	$\lvert V_\alpha^k\rvert/\lvert V\rvert$		
				$k=6$	$k=8$	$k=10$
As-Skitter [37]	1,696,415	11,095,298	13	54.08%	73.35%	80.75%
Wiki-topcats [38]	1,791,489	25,444,207	28	21.52%	34.89%	46.08%
Uk-2005 [39]	39,454,463	783,027,125	40	29.25%	34.98%	40.17%
Gsh-2015 [39]	988,490,691	25,690,705,118	52	22.29%	33.09%	37.98%
Orkut [42]	3,072,441	117,185,083	76	7.4%	9.93%	12.56%

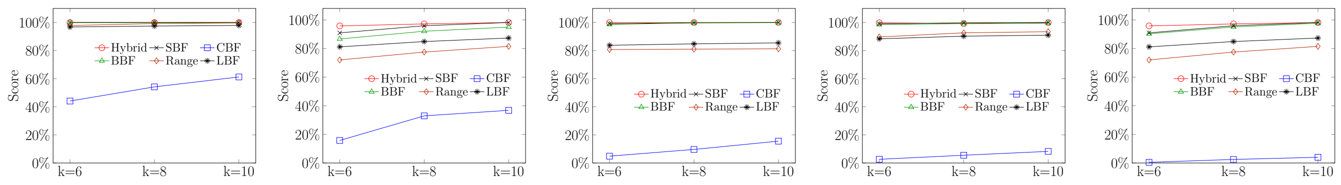
A. Comparative Setting

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 is Counting BF (CBF) [30] where each counter consists of $\lceil \log(|E|) \rceil$ bits. Apparently, edge insertion/deletion over CBF can be done efficiently by increasing/decreasing 1 for the corresponding counters. The third one is Blocked BF (BBF) [33] where bitset are partition into a series of smaller standard BF (blocks) and when a deletion happens, only the affected block need to be reconstructed. We set block size of 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 update without global reconstruction, and hence we only applied standard BF there in view of its highest false positive rate. Apparently, bit-hash version in Section V-B 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 V-A) for comparison, which can be maintained in a similar way to that of hybrid.

We first evaluate these VEND solutions from three aspects: VEND score, edge queries acceleration and the maintenance efficiency (Sections VII-F and VII-D). For each dataset, we set three different dimension numbers k: 6, 8, 10. We then present case studies of VEND in Section VII-F.

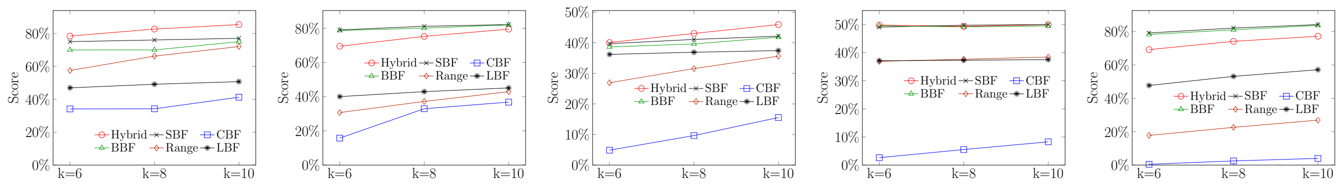
B. VEND Score

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 continuously and randomly generating vertex pairs and compute the latest proportion of detected NEpairs. We terminate this process when the monitored proportion is converged. We find that, for each dataset, the generated vertex pairs are more than one billion. 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,



(a) As-Skitter (b) Wiki-topcats (c) Uk-2005 (d) Gsh-2015 (e) Orkut

Fig. 4: VEND Score over Different Datasets on Randomly Generated Vertex Pairs



(a) As-Skitter (b) Wiki-topcats (c) Uk-2005 (d) Gsh-2015 (e) Orkut

Fig. 5: VEND Score over Different Datasets on Vertex Pairs of Common Neighbor

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 Figures 4 and 5, respectively. We can see that our method and SBF score almost equally highest. Apparently, CBF can hardly be used as a VEND solution since the corresponding score is so much lower than that of others. Score of CBF in Ak-Skitter dataset is much higher than those in other datasets since the average degree of Ak-Skitter dataset (i.e., 13) is close to the slot size (i.e.,k) of CBF. The advantages of our method over range and hash based ones are even more obvious on vertex pairs of common neighbors than that on randomly generated ones, which strengthens the applicability of our method in real world scenarios.

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 are 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 Figure 6. We can see from that all VEND solutions exhibits considerable acceleration on edge queries except CBF. Our method still perform similarly to SBF and outperform the rest. An interesting observation is that our method performs better than SBF on $CommPair$, but the opposite on $RandPair$, which may be because we focus on local neighborhood when building encodings while SBF just hashes edges over a global slot. In view of the locality of edge queries for many graph computation (such as triangle counting and subgraph matching), this observation further confirms advantages of our method over SBF.

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.

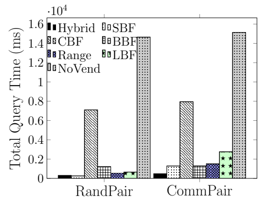
We can see from Figure 7 that although comparative works are more efficient than our method on insertion, we can still address nearly tens of thousands edge insertions per second. An important observation is that performances of SBF and BBF are so terrible that they can hardly be applied in real-world scenarios. In general, our method is apparently the best with high VEND score and efficient maintenance.

TABLE II: Index Construction and Memory Efficiency

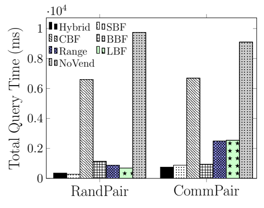
Datasets	Graph Space $\vert G\vert$	Index Space $\vert f\vert$ $(1-\vert f\vert/\vert G\vert)$			Construction Time (k=8)	
		$k=6$	$k=8$	$k=10$	$Thread=1$	$Thread=10$
As-Skitter	169M	38M (77%)	51M(69%)	64M(61%)	33 s	8 s
Wiki-topcats	388M	41M(89%)	54M(86%)	68M(82%)	109 s	16 s
Uk-2005	11.66G	903M(92%)	1203M(89%)	1505M(87%)	64 min	7.65 min
Gsh-2005	382.82G	520M(94%)	694M(92%)	867M(90%)	23.57 h	3.12 h
Orkut	1.74G	70M(96%)	93M(95%)	117M(93%)	9.7 min	1.07 min

E. Index Construction and Space Usage

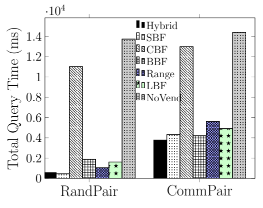
We report the space cost and construction time of the proposed hybrid VEND in Table II. Percentage numbers in red present the corresponding proportion of space saved by VEND. We can see that VEND could save a large proportion of memory usage. We evaluate construction in multi-thread way since encoding of vertices is highly parallelizable. We can see that the construction computation is of good speedup when implemented in multi-thread. For example, construction over Gsh-2005 dataset cost almost 24 hours in single thread while the time cost with 10 threads is a little over 3 hours.



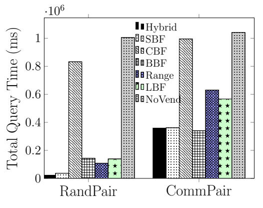
(a) As-Skitter



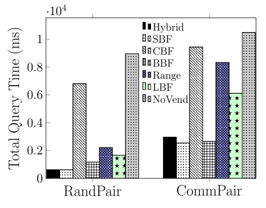
(b) Wiki-topcats



(c) Uk-2005

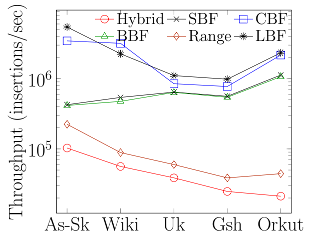


(d) Gsh-2015

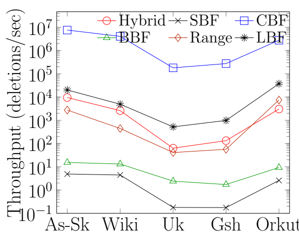


(e) Orkut

Fig. 6: Total Time over Different Edge Query Sets ($k = 8$)

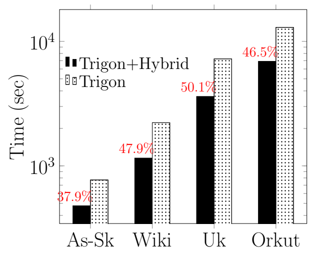


(a) Insertion

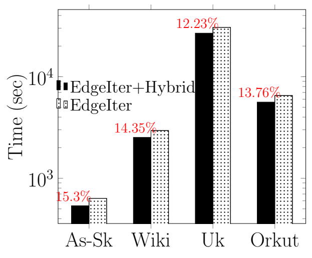


(b) Deletion

Fig. 7: Throughput for Addressing Edge Updates ($k = 8$)



(a) Trigon



(b) EdgeIter

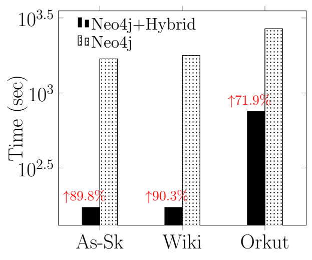
Fig. 8: External-memory Triangle Counting ($k = 8$)

F. Case Study

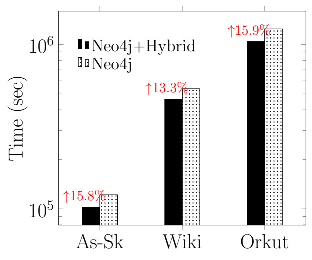
We conduct case studies over acceleration of hybrid VEND on SOTA external triangle counting and acceleration over SOTA graph database Neo4j [1]. Note that we do not report the result on Gsh-2015 (of more than 25 billion edges) here since it does not finish on triangle counting in 100 hours and the estimated time is more than one month. We fix $k = 8$.

We implement Trigon [14] and the external-memory version fo edge iterator based solution (denoted as EdgeIter) for evaluation. We set the memory available to Trigon exactly as that of VEND. We can see from Figure 8 that our method can accelerate Trigon by 40%∼50% and EdgeIter by 10%∼15%. In fact, Trigon is the SOTA external triangle counting method that outperforms EdgeIter, and hence the acceleration of VEND is of great significance.

We also evaluate the VEND acceleration for edge query and triangle counting of SOTA graph database Neo4j [1] (Community Edition 3.4.5 with C++ driver), which is the well-known and widely-used graph database all over the world. Neo4j always ranks first in DB-Engines Ranking of graph database [44]. Without the ability to access the underlying storage of Neo4j, only edge iterator based triangle counting is implemented and we build index for improving the efficiency of adjacent list retrieval. Even with index, it takes much more time to retrieve adjacent list in well functional Neo4j than that in lightweight Key-Value store. Hence, we only report the results over three smaller datasets, namely, As-Skitter, Wiki-topcats and Orkut. We can see from Figure 9 that VEND improves the overall edge query performance of Neo4j by more than 71.9%, and 10%∼15% for triangle counting.



(a) Edge Query Acceleration



(b) Triangle Acceleration

Fig. 9: VEND Acceleration over Neo4j ($k = 8$)

VIII. Conclusions

Edge query is one of the fundamental operations in graph databases. We first proposes to study vertex encoding for edge nonexistence determination (VEND) for accelerating edge queries by efficiently filtering no-result ones (vertex pairs connected by no edges). We formally define VEND as designing vertex encoding f and NEpair determination function (NDF) F. We propose VEND score to evaluate the performance of VEND. 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 precisely detected. We also illustrate range-based and hash-based extensions over the optimal partial version, after which we propose a final hybrid VEND solution incorporating range and hash ideas. Furthermore, we propose efficient maintenance algorithm over the hybrid VEND solution. Extensive experiments show that our solution performs well on real-world datasets and is able to detect most no-result edge queries. Since there is no previous work on VEND, our work is highly innovative and efficient.

Acknowledgment

This work was supported by Natural Science Foundation of China under grant 62102142 and Fundamental Research Funds for the Central Universities under grant 531118010657 and 531118010703.

REFERENCES

[1] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the southern association for information systems conference, Atlanta, GA, USA*, vol. 2324, no. 36, 2013.

[2] A. Deutsch, Y. Xu, M. Wu, and V. Lee, "Tigergraph: A native mpp graph database," *arXiv preprint arXiv:1901.08248*, 2019.

[3] "Nebula graph," https://nebula-graph.io/, 2022, [Online; accessed 15-January-2022].

[4] "Janusgraph," https://janusgraph.org/, 2022, [Online; accessed 15-January-2022].

[5] T. Neumann and G. Weikum, "The rdf-3x engine for scalable management of rdf data," *The VLDB Journal*, 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," *Proceedings of the VLDB Endowment*, vol. 4, no. 8, pp. 482–493, 2011.

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

[8] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

[9] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles in massive networks," in *Proc. 22nd ACM International Conference on Information & Knowledge Management*. ACM, 2013, pp. 529–538.

[10] K. Tangwongsan, A. Pavan, and S. Tirthapura, "Parallel triangle counting in massive streaming graphs," in *Proc. 22nd ACM International Conference on Information & Knowledge Management*, 2013, pp. 781–786.

[11] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang, "Efficient subgraph matching by postponing cartesian products," in *Proceedings of the 2016 International Conference on Management of Data*, 2016, pp. 1199–1214.

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

[13] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song, "Heterogeneous graph neural networks for malicious account detection," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, pp. 2077–2085.

[14] 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, 2022. [Online]. Available: https://doi.org/10.1109/TKDE.2020.3003259

[15] C. Kankanamge, S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu, "Graphflow: An active graph database," in *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*, S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, and D. Suciu, Eds. ACM, 2017, pp. 1695–1698. [Online]. Available: https://doi.org/10.1145/3035918.3056445

[16] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "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 computation*, 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 *Proceedings of the 22nd international conference on World Wide Web*, 2013, pp. 37–48.

[20] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2014, pp. 701–710.

[21] A. Grover and J. Leskovec, "node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and 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 *Proceedings of the 24th international conference on world wide web*, 2015, pp. 1067–1077.

[23] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 2017, pp. 1025–1035.

[24] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio, "Graph attention networks," *arXiv preprint arXiv:1710.10903*, 2017.

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

[26] M. E. Newman, "Clustering and preferential attachment in growing networks," *Physical review E*, vol. 64, no. 2, p. 025102, 2001.

[27] G. Kossinets, "Effects of missing data in social networks," *Social networks*, 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," *CoRR*, vol. abs/1804.04777, 2018. [Online]. Available: http://arxiv.org/abs/1804.04777

[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, 2000. [Online]. Available: https://doi.org/10.1109/90.851975

[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, 2010. [Online]. Available: https://doi.org/10.1109/LCOMM.2010.06.100344

[32] H. Lim, J. Lee, H. Y. Byun, and C. Yim, "Ternary bloom filter replacing counting bloom filter," *IEEE Commun. Lett.*, vol. 21, no. 1, pp. 278–281, 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, 2009. [Online]. Available: https://doi.org/10.1145/1498698.1594230

[34] S. B. Seidman, "Network structure and minimum degree," *Social networks*, 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 *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 993–1008.

[36] "Codes," https://github.com/hnuGraph/VEND, 2022.

[37] J. Leskovec, J. Kleinberg, and C. Faloutsos, "Graphs over time: densification laws, shrinking diameters and possible explanations," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining*, 2005, pp. 177–187.

[38] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, 2017, pp. 555–564.

[39] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "uk-2005: A crawl of the .uk domain performed by ubicrawler," https://law.di.unimi.it/webdata/uk-2005/, 2005.

[40] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Ubicrawler: a scalable fully distributed web crawler," pp. 711–726, 2004. [Online]. Available: https://doi.org/10.1002/spe.587

[41] 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. [Online]. Available: https://doi.org/10.1145/3160017

[42] J. Yang and J. Leskovec, "Defining and evaluating network communities based on ground-truth," *Knowledge and Information Systems*, vol. 42, no. 1, pp. 181–213, 2015.

[43] S. Dong, A. Kryczka, Y. Jin, and M. Stumm, "Rocksdb: Evolution of development priorities in a key-value store serving large-scale applications," *ACM Transactions on Storage (TOS)*, vol. 17, no. 4, pp. 1–32, 2021.

[44] "Db-engines," https://db-engines.com/en/ranking/graph+dbms, 2022, [Online; accessed 15-January-2022].