Space-Efficient Random Walks on Streaming Graphs

Serafeim Papadias
Technische Universität Berlin
s.papadias@tu-berlin.de

Jorge-Arnulfo Quiané-Ruiz
Technische Universität Berlin
jorge.quiane@tu-berlin.de

Zoi Kaoudi
Technische Universität Berlin
zoi.kaoudi@tu-berlin.de

Volker Markl
Technische Universität Berlin
volker.markl@tu-berlin.de

ABSTRACT

Graphs in many applications, such as social networks and IoT, are inherently streaming, involving continuous additions and deletions of vertices and edges at high rates. Constructing random walks in a graph, i.e., sequences of vertices selected with a specific probability distribution, is a prominent task in many of these graph applications as well as machine learning (ML) on graph-structured data. In a streaming scenario, random walks need to constantly keep up with the graph updates to avoid stale walks and thus, performance degradation in the downstream tasks. We present Wharf, a system that efficiently stores and updates random walks on streaming graphs. It avoids a potential size explosion by maintaining a compressed, high-throughput, and low-latency data structure. It achieves (i) the succinct representation by coupling compressed purely functional binary trees and pairing functions for storing the walks, and (ii) efficient walk updates by effectively pruning the walk search space. We evaluate Wharf, with real and synthetic graphs, in terms of throughput and latency when updating random walks. The results show the high superiority of Wharf over inverted index- and tree-based baselines.

PVLDB Reference Format:
Serafeim Papadias, Zoi Kaoudi, Jorge-Arnulfo Quiané-Ruiz, and Volker Markl. Space-Efficient Random Walks on Streaming Graphs. PVLDB, 16(2): 356-368, 2022. doi:10.14778/3565816.3565835

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/spapadias/wharf.

1 INTRODUCTION

Random walks are used in a large number of graph analysis tasks, such as PageRank [3, 16, 32, 36], SimRank [20], in influence maximization [10, 27, 28, 54], in recommendations [11, 13, 19], in graph embeddings [9, 17, 41], in graph neural networks [60], and even in spatiotemporal processing [24]. For example, random walks-based graph embeddings enable many machine learning (ML) tasks on graphs, e.g., link prediction, vertex classification, and outlier detection. Thus, computing random walks is at the core of many important tasks today.

Yet, real-world graphs are inherently dynamic, entailing continuous additions and deletions of vertices and edges [6, 26, 29]. In many novel applications, such as the Internet of Things and digital twins, graph updates occur with increasingly high frequency, requiring low latency and high throughput processing. For example, Alibaba’s e-commerce platform uses massive graphs to store their data [61]: These graphs consist of billions of vertices (e.g., modelling products, buyers, and sellers) and hundred billions of edges (e.g., representing clicks, orders, and payments). Alibaba reported these graphs are highly dynamic as they receive a high rate of real-time updates.

Thus, it is crucial to keep random walks up-to-date with the continuous changes to not hurt the effectiveness or accuracy of the downstream tasks. This is exacerbated in high-stake applications, such as anomaly and fraud detection, where even a small percentage of higher accuracy is of utter importance. Let us illustrate how the accuracy is affected if random walks are not kept up-to-date. We ran an experiment where we consider a link prediction task on a dynamic social network graph after running node2vec [17]. We executed node2vec in two different settings: static – we train embeddings only on the initial graph and reuse them subsequently, and ideal – we retrain embeddings from scratch at each snapshot (see the setup for this experiment in Section 7.6). Figure 1a shows the accuracy (i.e., AUC score) results. We observe that retraining embeddings from scratch at each new graph snapshot, i.e., after a set of new graph updates have been applied, is mandatory to maintain high accuracy in the downstream ML task (Ideal Learning in the figure). While in the static scenario, the accuracy drops, in the dynamic scenario the accuracy increases as more graph updates arrive. We also ran an experiment where we approximate Personalized PageRank (PPR) scores using [3] on a dynamic citation graph. Figure 1b shows the Symmetric Mean Absolute Percentage Error (SMAPE) when approximating PPR scores. Specifically, we...
illustrate the SMAPE between the actual algorithm in [3], which updates all affected random walks at each snapshot, and a static variant, which uses the existing random walks. We observe that the error in PPR scores is above 80% even after the first snapshot arrives. We thus expect that both the accuracy gap in graph embeddings applications and the estimation error in PPR applications will increase very fast in streaming graphs, where updates arrive at a very fast rate [12, 15, 35, 43, 61].

Despite this importance, the research community has paid little attention to the problem of maintaining up-to-date random walks on streaming (a.k.a. highly dynamic) graphs. We do find a large number of works for efficiently computing random walks [1, 38, 48, 56, 58, 59], but all consider static graphs. Barros et al. [4] present a variety of random walk-based works on Graph Representation Learning (GRL) on dynamic graphs [5, 14, 18, 30, 31, 33, 44, 51, 62]. Some among those, such as [18, 44], consider updating random walks but use simplistic inverted indexes to do so and are, thus, inefficient—they cannot cope with streaming graphs. There are also theoretical works, such as [21, 55], that focus on random walks: [55] studies how to store walks succinctly in an append-only fashion, which is not applicable for the streaming scenario where parts of walks have to be deleted; [21] proposes generating random walks on single-pass graph streams but in an approximate manner.

Updating random walks for streaming graphs is thus an important and open problem. However, doing so is challenging for three main reasons: (i) One should update random walks with both low-latency and high-throughput as streams can become quite bursting and volatile with sudden spikes [23]; (ii) One must enable fast access to (parts of) the walks state. This allows for realizing fast walk updates by efficiently identifying the walks as well as their parts (vertices) to update; (iii) Random walks state should be as succinct as possible especially for applications where the total size of random walks is multiple times larger than the size of the maintained graph, e.g., Graph Representation Learning (GRL) [48].

We propose Wharf, a parallel system that tackles all above-mentioned challenges to maintain stateful streaming random walks. It stores random walks with the graph within a single data structure forming a hybrid tree-of-trees. The main idea is to update walks together with the graph: During a graph update, it identifies the out-of-date walks and updates them in a bulk fashion.

In summary, after giving some preliminaries in Section 2, we make the following major contributions:

(1) We formalize the problem of streaming random walks, which entails storing walks in a space-efficient way, enabling efficient batch walk updates with high-throughput and low-latency, and fast node retrieval in the set of stored walks simultaneously (Section 3).

(2) We propose a novel hybrid-tree that stores random walks together with the graph in a compressed form. Specifically, we represent random walks as triplets which we encode to integer values using pairing functions. This allows us to compact random walks in a lossless manner and maintain the walks state in a way that also serves as an index for efficient walk access. Overall, our structure allows for safe parallelism, fast acquisition of lightweight graph and walks snapshots, and high cache locality.

(3) We devise an output-sensitive algorithm for performing efficient search in the set of walks by leveraging the ordering properties of pairing functions (Section 5). We also present a walk update mechanism that apply updates in batches and we prove that its time complexity is lower than its competitor (Section 6).

(4) We validate Wharf through extensive experiments on a variety of real-world and synthetic graph workloads. The results show that Wharf achieves its goals in terms of throughput and latency when updating random walks, low memory footprint, and effectiveness of the downstream tasks (Section 7).

We then discuss related work in Section 8 where we stress that existing non-streaming random walk systems fail to address the above-mentioned challenges. We conclude the paper in Section 9.

2 PRELIMINARIES

Running example. We use an e-commerce graph that contains users and items extracted from the Taobao e-commerce platform [63] as our running example. The vertices correspond to items that can be purchased through the platform, and the edges denote the item-item relationships, i.e., items that users purchase together. Figure 2a illustrates an excerpt of the graph. We assign integer identifiers to the vertices for simplicity.

Purely-Functional Trees (PF-trees). A PF-tree is a mutation-free tree structure that preserves its former versions when altered and yields a new tree version reflecting the update [34]. Each element of a PF-tree serves as key, and is kept in a separate tree node. Figures 2b illustrates the PF-tree for our example graph.

Compressed Purely-Functional Trees (C-trees). A C-tree [15] is a binary PF-tree [34], which is additionally compressed and stores multiple elements in each vertex. A chunking scheme takes the ordered set of elements to be represented and promotes some of them to heads, which are stored in a purely-functional tree. In more detail, given a set E of elements, one first computes the set of heads \( H(E) = \{ e \in E | h(e) \mod b = 0 \} \), where b is the chunking parameter indicating the number of elements each chunk roughly retains, \( h : K \rightarrow [1, \ldots, N] \) is a hash function drawn from a uniformly random family of hash functions (N is some sufficiently large range). For each \( e \in H(E) \) let its tail be \( t(e) = \{ x \in E | e < x < next(H(E), e) \} \), where \( next(H(E), e) \) returns the next element in \( H(E) \) greater than e. Thus, the rest of the elements are stored in tails that are associated with each head vertex of the tree. It can exist a headless tail containing the smallest elements to be represented, i.e., the prefix. The prefix, as well as the tails, are both called chunks. Figure 2c illustrates the compressed C-tree variant of the PF-tree in Figure 2b for our graph example in Figure 2a. C-trees maintain similar asymptotic cost bounds as the uncompressed trees while improving...
We assume the popular edge stream model, where a graph stream is regarded as a sequence of incoming edges. We consider unbounded graph streams, i.e., there is no limit on the number of graph updates that arrive. A graph update is a set of edge insertions and deletions. In the edge stream model, vertex insertions and deletions happen implicitly via edge updates: A vertex is added when an edge with a vertex not present in the current graph is inserted, while a vertex is deleted only when its degree becomes zero after an edge deletion.

Following this model, a streaming graph is a long sequence of discrete graph snapshots that contain both insertions and deletions of edges. Thus, a streaming graph is a long sequence of discrete graph snapshots containing graph updates that should be applied as they arrive. We formally define a streaming graph as follows:

Definition 1 (Streaming Graph). A streaming graph is a sequence of discrete graph snapshots, \( G^t = \langle V^t, E^t \rangle \), where \( V^t = \{ v^t_1, \ldots, v^t_m \} \) are the vertices, \( E^t = \{ e^t_1, \ldots, e^t_n \} \) are the edges, and \( t \in \mathbb{N} \) is a timestamp.

We now formally define the problem we address in this paper. To do so, we first formalize the streaming graph foundations (Section 3.1), the notion of random walks and walk corpus (Section 3.2), where we also define statistical indistinguishable random walks. Finally, we state the problem of streaming random walks (Section 3.3).

### 3.1 Streaming Graphs

We assume the popular edge stream model, where a graph stream is regarded as a sequence of incoming edges. We consider unbounded graph streams, i.e., there is no limit on the number of graph updates that arrive. A graph update is a set of edge insertions and deletions. In the edge stream model, vertex insertions and deletions happen implicitly via edge updates: A vertex is added when an edge with a vertex not present in the current graph is inserted, while a vertex is deleted only when its degree becomes zero after an edge deletion.

Following this model, a streaming graph is a long sequence of discrete graph snapshots that contain both insertions and deletions of edges. Thus, a streaming graph is a long sequence of discrete graph snapshots containing graph updates that should be applied as they arrive. We formally define a streaming graph as follows:

Definition 1 (Streaming Graph). A streaming graph is a sequence of discrete graph snapshots, \( G^t = \langle V^t, E^t \rangle \), where \( V^t = \{ v^t_1, \ldots, v^t_m \} \) are the vertices, \( E^t = \{ e^t_1, \ldots, e^t_n \} \) are the edges, and \( t \in \mathbb{N} \) is a timestamp.

Note that after applying a graph update \( \delta G^t \) to a graph snapshot \( G^t \) we end up with the new graph snapshot at timestamp \( t + 1 \), i.e., \( G^{t+1} = G^t + \delta G^t \).

### 3.2 Random Walks

A random walk on a graph serves as a sample of the graph and is utilized by many applications, such as PageRank [16, 36] and online influence maximization [10, 27, 28, 54].

Definition 2 (Random Walk). A random walk is a \( k \)-th order Markov chain, where the state space of which is the set of graph vertices \( \mathcal{V} \) and the future state depends on the last \( k \) steps. A random walk \( w \) of length \( l \) contains a sequence of vertices, \( v_0, v_1, \ldots, v_j, \ldots, v_l \), where \( v_j \) is the \( j \)-th vertex in \( w \) and \( j \in \{1, \ldots, l\} \), and every two consecutive vertices are connected with an edge.

In the general case, a random walk \( w \) is generated by sampling a vertex \( v_j \) given the \( k \) previous vertices \( v_{j-k}, \ldots, v_{j-1} \) in \( w \) from the following transition probability distribution: \( \text{prob}(v_j|v_{j-1}, \ldots, v_{j-k}) \). Note that this probability is non-zero only if an edge between vertices \( v_{j-1} \) and \( v_j \) exists. We compute the transition probability following a random walk model, e.g., DeepWalk [41] which is a first-order random walk. In DeepWalk, a walker, which is currently residing at a vertex, consults solely its neighbours to derive the transition probability to select the next vertex for a random walk to visit. For instance, the walker producing walk \( w_0 \) (see Figure 3) moves from vertex \( v_0 \) to \( v_3 \) with probability \( \frac{1}{3} \), as \( v_0 \) has three neighbours, namely \( v_0, v_2 \), and \( v_3 \) (Figure 2a).

The set of random walks extracted from a graph is referred to as walk corpus. Figure 3 shows an excerpt of a walk corpus that contains a set of random walks for \( n_w = 1 \) and \( l = 5 \), where \( n_w \) is the number of walks that initiate from each vertex and \( l \) is the length of each walk in \( W \). As the graph evolves fast, walks can become inconsistent, and even in certain cases invalid: A random walk is inconsistent when it does not reflect the graph transition probabilities correctly; An invalid random walk, in contrast, is an inconsistent walk that has been disrupted by an edge deletion and it cannot be recreated in the updated graph. Note that in the sequel, whenever it is not necessary to differentiate between inconsistent and invalid walks we refer to them simply as affected.

We illustrate both inconsistent and invalid random walks in the walk corpus of Figure 3. Assume that the edge deletion of \( \{4, 7\} \) happens before the edge addition of \( \{2, 8\} \) in the example graph of Figure 2a. In this case, \( w_2 \) becomes inconsistent as the transition probabilities for a walker residing on graph vertex \( v_2 \) change and thus we need to refine all subsequent walk vertices of \( w_2 \). Also,
w4 becomes invalid due to the edge deletion, and at the same time inconsistent as the transition probabilities of vertex v4 change.

Ideally, we want to update only those random walks that become inconsistent or invalid after a graph update. Updating one of these walks means updating all its affected vertices, i.e., those vertices that make the random walk inconsistent or invalid. This is because they might not match the transition probabilities of the updated graph Gt+1. For instance, w2 becomes inconsistent and we need to refine all subsequent walk vertices of walk w2, e.g., producing an updated walk v2, v5, v6, v2. In our work, we choose to refine both invalid and inconsistent walks, to ensure statistical indistinguishability. In specific, we say that a walk corpus is up-to-date when it is statistically indistinguishable [44], which we define as follows:

Property 2 (Statistical Indistinguishability). A walk corpus ‘W’, resulting from updating a walk corpus ‘W’ after a graph update δG, is statistically indistinguishable if it is equi-probable with a new walk corpus that is generated from scratch on graph G′ = G + δG.

3.3 Streaming Random Walks

We now formally define the problem of computing streaming random walks. Specifically, we aim at representing walk corpuses space-efficiently, enabling incremental updates, and allowing for fast walks access. In the sequel, we refer to a specific random walk algorithm as random walk model. Formally:

Problem Statement. Given a random walk model M, we define the problem of streaming random walks as (i) maintaining and storing in main memory a walk corpus ‘W’ that is generated based on M, (ii) ensuring that ‘W’ is always statistically indistinguishable, and (iii) updating ‘W’ incrementally without recomputing it from scratch.

4 GRAPH-WALK STRUCTURE

Our goal is to come up with a data structure that stores the random walk corpus together with the streaming graph. In this way, we can both speed up the update process and reduce the required storage space. In addition, we aim at indexing the random walks so that we can quickly locate the (parts of) random walks that require access. In the sequel, we refer to a specific random walk updates by avoiding calculating all walks from scratch. We make the random walk inconsistent or invalid. This is because they might not match the transition probabilities of the updated graph.

In detail, each vertex v in a corpus W can be uniquely described by the pair (wi, pj), where wi is the walk it participates in, with i ∈ {1, ..., |W|}, and pj is the position of v in wi where j ∈ {1, ..., l}. In other words, a pair (wi, pj) serves as the coordinates of the vertex v in the corpus W and thus, when seeking for v, this pair behaves as its search key. We, thus, denote with viwi,pj the identifier of a vertex v ∈ W. Note that a vertex may appear multiple times in a walk corpus as well as in the same walk. Instead of storing raw walk sequences, we group the walk triplets by vertex identifier and store them in their corresponding walk-trees of the hybrid-tree structure associated with the appropriate vertex of the vertex-tree. This enables fast access to specific vertices of walks in the corpus and space efficiency. However, instantly accessing an affected vertex in the corpus is not enough. We should also be able to traverse a walk so that we can update it efficiently. To achieve this, we maintain the vertex

![Figure 4: WHaRP’s hybrid-tree. The edge-tree of vertex 5 contains its neighbors (v2, v3, v4, v7) as shown in Figure 2a and the walk-tree of vertex 5 contains the encoded triplets (s1, ..., s9) that correspond to v5’s entries in the corpus of Figure 3.](image)
identifier of the next node, \( v_{w,p_{j+1}} \) at position \( p_{j+1} \) in a walk \( w \) as the third element of a walk triplet. We thus represent each vertex \( v_{w,p_j} \) with a walk triplet of the form \((w_i,p_j,v_{w,p_{j+1}})\). Ultimately, these walk triplets serve for both storing and traversing any random walk sequence efficiently. For example, in the corpus of Figure 3, walk \( w_0 \) can be represented as a sequence of the following triplets: \((w_0,p_0,v_1), (w_0,p_1,v_2), (w_0,p_1,v_4), (w_0,p_2,v_4)\). Note that the position numbering starts from 0 and that for the last vertex of a walk, the next vertex of its walk triplet has the same vertex id with the vertex itself denoting the end of the walk\(^1\).

### 4.3 Walk Triplet Pairing

Storing integer values instead of entire triplets objects achieves great space savings. This is not only because of the object footprint but also because it allows for difference encoding. We, thus, convert the walk triplets into integers to store them in the \( C \)-trees. Our main idea is to encode each walk triplet into a unique integer via a pairing function. We could easily achieve this encoding by two invocations of a pairing function: encoding the first two elements of the triplet into a paired value and then encode again the paired value with the third element. Yet, pairing comes at a cost. Recall from Section 2 that for two operands that are up to \( N \) bits, Szudzik pairing function returns a \( 2N \)-bit number. Thus, the fewer pairing function invocations, the smaller the encoded walk triplet values.

We, thus, reduce the number of pairing function invocations by first encoding the walk identifier of a vertex along with its position in the corresponding walk together into a single number. Specifically, for a walk triplet \((w_i,p_j,v_{w,p_{j+1}})\) and given that the length of \( w_i \) is \( l \), we devise the following function to encode \( w_i \) and \( p_j \) into a single integer: \( f(w_i,p_j) = w_i \times l + p_j \). We, then, invoke Szudzik once to pair the output of this function, \( f(w_i,p_j) \), with the identifier of the next node in the walk, \( v_{w,p_{j+1}} \). When we unpair an encoded walk triplet, we retrieve the walk id and position from \( f \) as follows: \( w_i = \frac{f - p_j}{l} \) and \( p_j = f \mod l \).

Note that \( p \) is upper bounded by \( l \), and thus, we can utilize the simple function \( f \) to encode \( w \) and \( p \) as well as revert to the original values with the above-mentioned equations. However, in the streaming setting there is no upper bound for \( v_{w,p_{j+1}} \) so we rely on a pairing function for the final encoding. Specifically, we used the Szudzik function because it ensures that for two \( N \)-bit integer arguments, its value is at most a \( 2N \)-bit integer, and thus, guarantees that there will be no integer overflows. For instance, in our running example for triplet \((w_0,p_0,v_1)\), we invoke \( Szudzik(w_0 \times l + p_0,v_1) \) to get the integer value that we will insert in the \( C \)-tree. Thus, function \( f \), as well as \( v_{w,p_{j+1}} \), must be at most \( N \)-bit numbers. Formally:

\[
  f(w,p) = w \times l + p \leq 2^N - 1 \land v_{w,p_{j+1}} \leq 2^N - 1, \quad \text{where } p_{j+1} \leq l
\]

which dictates the cap of maximum values for \( w, l \), and \( v_{w,p_{j+1}} \). Encoded triplets go in the walk-tree of the vertex they correspond.

Let us now illustrate how the walk- and edge-trees in our running example are populated. Focusing on vertex \( v_5 \), assume that it only appears in the walks that are shown in Figure 3 as well as in the first position of \( w_5 \) with \( v_7 \) as its next vertex (not shown). Figure 4 shows the contents of \( v_5 \)'s walk-tree. The walk triplets of vertex \( v_5 \) are: \((w_0,p_3,v_4), (w_1,p_3,v_6), (w_2,p_2,v_3), (w_3,p_3,v_3), (w_3,p_3,v_9)\), \((w_4,p_1,v_6), (w_5,p_0,v_7), (w_8,p_2,v_4), (w_8,p_4,v_5)\). After the encoding we get the integer values \( s_1, s_2, \ldots, s_9 \), respectively, where \( s_1 < \cdots < s_9 \) holds without loss of generality (w.l.g.) As we see in Figure 4, Wharf stores the encoded triplets monotonically inside the walk-tree and \( s_3, s_6, s_9 \) are selected as head vertices. In addition, Figure 4 shows the edge-tree of \( v_5 \) that contains its neighbours in our running example graph (Figure 2a), which are \( v_2, v_3, v_4, v_7 \). Assuming that \( v_2 < v_5 < v_4 < v_7 \) (w.l.g.), they are monotonically stored inside \( v_5 \)'s edge-tree (\( v_5 \) acts as head).

### 4.4 Walk Triplet Compression

It is worth noting that pairing invocations produce numbers that are much larger than their arguments, which incurs a large storage overhead. Difference encoding (DE) alleviates this problem, as we store only the differences of the integers that correspond to encoded walk triplets in each chunk. More specifically, we exploit the fact that trees store elements (integer values) in sorted order in chunks to further compress the data structure. Given a chunk containing \( d \) integers \( \{i_1,i_2,\ldots,i_d\} \), we compute the differences \( \{i_1-i_2,i_2-i_3,\ldots,i_{d-1}-i_d\} \) and encode them using a variable byte-code [49]. Note that after encoding the walk triplets with the Szudzik, we store them monotonically in increasing order in the \( C \)-trees, and thus, the differences produced by the DE are always non-negative. Clearly, the difference encoding scheme applied to the chunks counterbalances the fact that we store "big" numbers produced by the pairings. Note that each chunk must be processed sequentially, namely decompressed and then re-compressed as a whole. The cost of the sequential decoding does not affect the overall work or depth of parallel tree methods, as the size of each chunk is small \( O(\log n) \) w.h.p. for a constant chunking parameter \( b \). Similarly, chunks must be re-compressed when receiving updates, which has a cost on par with the cost of decompressing the chunks.

### 4.5 Space Complexity

Let us now elaborate on the memory footprint that Wharf needs to store the walks. Assume we use \( B \)-bit integers, Wharf needs \( B \) bits for each encoded walk triplet. The total number of walk triplets in a walk corpus is \(|W| = n \times n_w \times l|\), where \( n \) is the number of vertices in the graph, \( n_w \) is the walks per vertex, and \( l \) is the length of each walk. Therefore, Wharf needs \( \Theta(|W| \times B) \) space to store the walks. This is because we have one encoded walk triplet for each vertex in the walk corpus. On the other hand, a simplistic inverted index-based solution similar to [18] that stores the whole walk corpus sequentially needs \( \Theta(|W| \times B) \) space for the walk sequences. Additionally, for the inverted index that relates each vertex id with the set of walk ids it participates, it needs \( O(2 \times |W| \times B) \). Thus, total space complexity ends up being \( O(3 \times |W| \times B) \).

### 5 OPTIMIZED SEARCH

Before delving into the details of how we update random walks, we first discuss one of the factors that make Wharf highly performant in updating random walks: its capability to search in walk-trees so that walk traversal is possible. Traversing walk-trees is challenging for two reasons. First, a random walk is represented as a set of triplets stored under different vertices of the vertex-tree. Second, the only available information in a walk-tree is unique integer

\(^1\)Note that we can use any other termination identifier such as the integer \(-1\).
values, which are the encoded triplets. In particular, given a vertex \( v \) of a random walk \( w \), the operation for finding the next vertex is essentially searching for a triplet \( (w, p, *) \), but without knowing the actual value of its third element. Thus, when seeking the next vertex in a walk of the corpus, the pair \( (w, p) \) serves as a search key. The fact that walk-trees are filled with integer values representing encoded triplets, prevents us from directly using the search key to find the triplet we are looking for. A trivial way of finding it would be to visit each vertex of the walk-tree, decode its encoded triplets to retrieve the original ones, and check if one of them corresponds to walk \( w \) at position \( p \). In the worst case, we would decode all elements in the tree even if the triplet does not exist. The complexity of this process is \( O(n) \) where \( n \) is the number of elements in a walk-tree, which is prohibitive for large-scale streaming graphs. Next, we describe an efficient search algorithm based on range queries.

### 5.1 Search Space Pruning

We start by describing how we can efficiently searching over walk-trees without necessarily decoding all walk triplets in the worst case. The use of pairing functions is a calculated move, as they have properties that enforce ordering among triplets of a walk-tree. We leverage this ordering property to reduce the search space in a walk-tree and hence achieve efficient search.

Based on Corollary 1, we can construct a search range \( [lb, ub] \) for a vertex of walk \( w \) at position \( p \) where:

\[
lb = \langle w \times l + p, a^{\min}_{w,p+1} \rangle \quad \text{and} \quad ub = \langle w \times l + p, a^{\max}_{w,p+1} \rangle
\]

\( a^{\min}_{w,p+1} \) and the \( a^{\max}_{w,p+1} \) are the minimum and maximum next vertex ids that appear in all the walk triplets of the walk-tree, respectively. We calculate the pair \( \{a^{\min}_{w,p+1}, a^{\max}_{w,p+1}\} \) at the time we construct a tree and refine it when we update the random walks. Conceptually, the search range is a subset of the range \( \{min, max\} \) where:

\[
\min = \langle w_{\min} \times l + p_{\min}, a^{\min}_{w,p+1} \rangle \quad \text{and} \quad \max = \langle w_{\max} \times l + p_{\max}, a^{\max}_{w,p+1} \rangle
\]

The \( \min \) and the \( \max \) are the global minimum and global maximum encoded values that can be possibly found in a walk-tree. Consequently, the range \( \{min, max\} \) encloses all the walk triplets inside the tree. \( [lb, ub] \subset [min, max] \) holds for the two aforementioned ranges. Therefore, if the walk triplet exists in the walk-tree, its encoded value must exist inside the range \( [lb, ub] \).

### 5.2 Next Vertex Search

Algorithm 1 illustrates the \texttt{FindNext} operation which intuitively performs a range query in the reduced search range as defined above. The algorithm receives as input a walk-tree \( W_T \), a walk identifier \( w \), and a position \( p \), and returns the vertex \( v_{w,p+1} \) at position \( p + 1 \) of walk \( w \). We initiate our search from the prefix part of the walk-tree (Lines 3-4). If the triplet is not found inside the prefix, then we continue the search in the tree part of the walk-tree. Note that Algorithm 1 calls the \texttt{TraverseTree} \( (root, w, p, lb, ub) \) procedure (Lines 6 and 9), which recursively traverses the tree part of a walk-tree while searching for matching walk triplets. As pointed out in [15], it is quite important to efficiently compute the first and last elements of a chunk \( c \), i.e., the \( c_{\text{first}} \) and \( c_{\text{last}} \), respectively. Recall a chunk stores the encoded triplets. Of course, \( c_{\text{first}} < c_{\text{last}} \) holds. To avoid scanning whole chunks, the first and last elements are stored at the head of each chunk for fetching \( c_{\text{first}} \) and \( c_{\text{last}} \) in \( O(1) \) work and depth. This modification is important to ensure that \texttt{FindNext} can be done in \( O(\log n + k) \) work and depth w.h.p. on a walk-tree, where \( k \) is the number of encoded triplet values lying within this search range of \( W_T \). We can skip searching in a chunk \( c \) (either in the prefix or in the tree) if \( ub < c_{\text{first}} \) or \( lb > c_{\text{last}} \). All the encoded triplets inside \( c \) are outside the search range (Line 9). It is worth noting that as the tree part of \( W_T \) is actually a binary search tree and its encoded triplets are stored in increasing order, we search it by conducting an in-order traversal.

### 6.0 UPDATING RANDOM WALKS

\texttt{Wharf} applies walk updates in batches and in parallel. It receives graph additions and deletions and buffers them to apply them in bulk. This allows \texttt{Wharf} to perform fast walk updates. It identifies the affected vertices, while updating the graph, and updates all those random walks that contain them. This is important as the number of affected vertices are several orders of magnitude smaller than the total number of vertices. Next, we describe how we compute and update the structure that keeps the affected vertices every time a batch of updates arrives. Then, we present our update algorithm, whose input includes the computed affected vertices structure.
6.1 Map of Affected Vertices

We construct a map of affected vertices (MAV), while processing a graph update, to be able to update only the affected vertices. In a nutshell, the MAV is responsible for bookkeeping key affected vertices in each affected walk. Note that in each affected walk the first encountered affected vertex is of special importance. This is because some of (if not all) the transition probabilities with which we sampled the subsequent vertices do not match the new probabilities in the updated graph. Using walks that do not reflect the graph structure accurately can lead to low accuracy of downstream tasks that rely on them. We thus formally define the MAV as follows:

**Definition 3 (Map of Affected Vertices – MAV).** A MAV is a key-value map that contains affected vertices for each affected walk in a walk corpus $W$: The key is the identifier of an affected walk $w$ and its value is the pair $(v_{\text{min}}, p_{\text{min}})$, with $v_{\text{min}}$ being the first affected vertex in $w$ located at position $p_{\text{min}}$.

We compute the MAV as follows. Assume, without loss of generality, a batch of graph updates, $\delta G$, containing undirected edges: with each edge $e = (s, d) \in \delta G$ being treated as two directed edges, namely, one $e_1$ initiating from a source vertex $s$ to a destination vertex $d$ and another $e_2$ starting from $d$ to $s$. Once an edge is incorporated into the appropriate edge-trees (one for each direction), it may render an existing walk in the maintained corpus inconsistent or even worse invalid. Specifically, based on the edge update (either insertion or deletion), we identify the affected walks and vertices from the walk-trees. We distinguish the following two cases with respect to an updated edge $e_1$ (similarly for the other direction $e_2$):

1. **(1) Edge Insertion:** After the insertion of an edge $e_1$, any walk $w \in W$ containing vertex $s$ becomes inconsistent because its transition probability is not the same anymore. In this case, we insert $(w, (s, p_s))$, where $p_s$ is the position of $s$ in $w$, into the MAV if an entry for $w$ does not exist, otherwise, we update its entry with the pair $(s, p_s)$, if $p_s$ is smaller than the current $p_{\text{min}}$.
2. **(2) Edge Deletion:** After the deletion of an edge $e_1$, any walk $w' \in W$ containing vertex $s$ becomes inconsistent, but it is invalid if it also contains a transition from $s$ to $d$. We update the MAV exactly as in the case of edge insertion. Our hybrid-tree allows us to efficiently update the MAV, as we only have to search the walk-tree of the source vertex that belongs to an edge addition/deletion.

6.2 Batch Walk Update

The main idea is to translate a batch of graph updates into a batch of walk updates. We do so by populating an insertion accumulator which gathers the encoded triplets that correspond to the newly sampled vertices and then bulk-insert them in the corresponding walk trees. A merge process then evicts the obsolete walk triplets.

As a walk corpus must remain statistically indistinguishable, we adopt the following update policy: \(^2\) We update both inconsistent and invalid walks by deleting and re-sampling all affected vertices of an affected walk starting from the first affected vertex at position $p_{\text{min}}$ until the last vertex at position $p_l$. For instance, Figure 5 shows that walk $w_2$ of our running example is inconsistent, and we thus need to re-sample from its second vertex onward.

\(^2\)Yet, note that the walk update policy is orthogonal to our algorithm.

---

**Algorithm 2 BatchWalkUpdate**

1. **Input:** hybrid-tree $\mathcal{H}$, map MAV, walk model $M$
2. **Output:** updated hybrid-tree $\mathcal{H}$
3. **do in parallel**
4. // Sampling of new walk parts
5. for $w \in \text{MAV}$ do in parallel \(\triangleright\) for each affected walk
6. new$_{\text{ver}} = v_{w,p_{\text{min}}}$
7. for $p = p_{\text{min}}, \ldots , l - 1$ do \(\triangleright\) rewalk from $p_{\text{min}} < l$
8. new$_{\text{ver}} = \text{SampleNext}(\text{new$_{\text{ver}}$}, \mathcal{H}, M)$
9. $I = I \cup \text{EncodeTriplet}(w, p, \text{new$_{\text{ver}}$})$
10. end for
11. $I = I \cup \text{EncodeTriplet}(w, l, \text{new$_{\text{ver}}$})$ \(\triangleright\) last vertex
12. **end for in parallel**
13. // Evict obsolete walk triplets
14. if demanded then
15. $\text{Merge}(\mathcal{H}, \text{MAV})$
16. **end do in parallel**
17. $\text{MultiInsert}(\mathcal{H}, I)$
18. **return** $\mathcal{H}$

**Wharf** allows a vertex in the hybrid-tree to have more than one walk-tree version, each containing walk triplets corresponding to a distinct batch edge update. **Wharf** stores the walk-tree versions under a vertex in the order that it creates them. Furthermore, it utilizes a *merge* operation that consolidates all the walk-tree versions of a vertex into a single one, after evicting all the obsolete walk-triplets. In specific, it scans the hybrid-tree in parallel, consults the MAV to check which triplets are still valid, and removes the obsolete ones. **Wharf** uses an on-demand policy, which corresponds to merging walk-trees only when requested, e.g., when a downstream operation requests for the random walks. Different policies with different throughput-memory trade-offs are also possible but we choose the on-demand one because it achieves the highest throughput.

Algorithm 2 shows the pseudocode of the process to update random walks. It takes as input the hybrid-tree, $\mathcal{H}$, the MAV, and the walk model $M$. For each affected walk that appears in the MAV (Line 5), we first initialize the vertex pointer for re-walking $w$ (Line 6). We, then, re-walk from this vertex pointer, i.e., the vertex at the minimum affected position (Line 7), and fill the insertion accumulator $I$ (Lines 8-11). In detail, if we are not yet at the end of $w$ (Line 11), we sample a new vertex with the new transition probability (Line 9). Note that depending on the utilized walk model $M$ we must initialize the MH samplers [59] accordingly. For instance, when we use DeepWalk only the current vertex is needed for sampling the next vertex, however, when we use node2vec we...
need to access the previous vertex id before \( p_{\text{min}} \) for initializing the samplers. Subsequently, we encode the triplet of the new vertex (Lines 9 & 11). As a result, \( I \) maintains all the encoded walk triplets (i.e., integer values) that should be inserted grouped by vertex identifier. In Figure 5, we see \( w_2 \), which is affected, and an excerpt of the hybrid-tree, namely, vertices \( v_1, v_3, v_5 \). The newly sampled vertices (circled in green) are converted into walk triplets and then are batch-inserted to the corresponding walk-trees of \( v_1, v_3, \) and \( v_5 \) (green arrows indicate insertion operations).

While we are running the re-walking process, we run the \textit{merge} process in the background to delete the obsolete walk parts from the walk-trees (Lines 14 & 15). In the example of Figure 5, the merge process is triggered in parallel with the sampling of new vertices to delete \( w_2 \)’s inconsistent walk triplets (circled in orange) from the corresponding walk-trees, e.g., of \( v_3 \) and \( v_5 \) (orange arrows show deletion operations). Note that merge consolidates potentially multiple walk-tree versions, e.g., of \( v_5 \) (more than one yellow trees). Finally, we apply the batch insertions of the newly generated walk parts (Line 17). Note that we use the \textsc{MultiInsert} method for applying batch updates to C-trees [15]. As an outcome, the algorithm produces the hybrid-tree \( \mathcal{H} \) with the updated walk corpus that is statistically indistinguishable from a corpus generated from scratch.

### 6.3 Complexity and Correctness

Let us now elaborate on the time complexity of Algorithm 2 in terms of number of walk triplets that are inserted and deleted. We have to update \( a = |\text{MAV}| \) affected walks. Precisely, we should insert in the hybrid-tree after re-walking, \( |I| = \sum_{i=1}^{n_w}(l - p_{\text{min}}^i) = O(a \times l) \) walk-triplets, where \( p_{\text{min}}^i \) is the minimum affected position of the \( i^{\text{th}} \) affected walk in the MAV. The batch insertion is done by the \textsc{MultiInsert} [15], which has a complexity of \( O(|I| \log |W|) \) work overall, and \( O(\log^3 |W|) \) depth, where \( |W| = n * n_w * l \) is the total number of walk-triplets in a walk corpus, \( n \) is the number of vertices in the graph, \( n_w \) is the walks per vertex, and \( l \) the length of each walk. Therefore, the complexity of Algorithm 2 is \( O(|I| \log |W|) \) work and \( O(\log^3 |W|) \) depth. An inverted index-based solution needs \( \Theta(\sum_{i=1}^{n_w} p_{\text{min}}^i) \) time to construct the MAV, as it traverses an affected walk from its first vertex till its \( p_{\text{min}} \). Additionally, it has to update the affected walk parts like \textsc{Wharf}, and thus, the total complexity is \( \Theta(a \times l) = \Omega(|I|) \). Hence, the complexity of an inverted index solution is greater than that of \textsc{Wharf}.

**Theorem 1 (Correctness).** \textsc{Wharf} updates the random walks in a walk corpus, such that they remain statistically indistinguishable.

**Proof Sketch.** Fix a random walk \( w \in \mathcal{W} \), where \( \mathcal{W} \) is the maintained walk corpus. Let \( (s, d) \) be an undirected edge that gets inserted (w.l.o.g.) into the graph. We discard the following two cases: \( 1) \) if \( s \notin w \) and \( d \notin w \), we are not interested in the transition probabilities with which \( w \) was sampled, using an walk model of up to second-order, do not change. Specifically, in first-order walks (e.g., DeepWalk), a vertex \( v \) uniformly samples one of its neighbors as the next vertex in \( w \). Thus, the transition probabilities between vertices in \( w \) stay intact and hence \( w \) remains valid. In second-order walks (e.g., node2vec), a vertex \( v \) with \( v_{\text{pre}} \) as the previous vertex in \( w \) non-uniformly samples one of its neighbors as the next vertex in \( w \). It does so with a probability that depends on whether the next vertex is (or isn’t) connected with \( v_{\text{pre}} \), or the next vertex is actually \( v_{\text{pre}} \) [17]. As \( s \notin w \) and \( d \notin w \), the transition probabilities of \( w \) remain intact.

\( 2) \) if \( s \notin w \) but \( d \in w \). When one endpoint is not “covered” by \( w \), \textsc{Wharf} incorporates \( d \) into the edge-tree of vertex \( s \) as well as \( s \) into the edge-tree of vertex \( d \) right after the insertion of the undirected edge \((s, d)\). \textsc{Wharf} also checks the walk-tree of \( s \), and the one of \( d \), where it finds the corresponding walk triplet belonging to \( d \), and thus, identifies that \( w \) is affected and proceeds to update it. This holds for both first- and second-order walks.

### 7 EXPERIMENTAL EVALUATION

We evaluate \textsc{Wharf} using a variety of large-scale real-world and synthetic graphs and investigate: how efficient it is in terms of throughput, latency, and space; how it scales to large graphs and batch sizes; how it behaves in the presence of data skew; how its range search and merge policy drives its performance; and whether it can enable high accuracy on downstream tasks.

#### 7.1 Setup

**Hardware.** We ran our experiments on a server with a 24-core Intel(R) Xeon(R) Gold 6126 CPU @ 2.60GHz and 1.5TB of main memory. Our prototype uses the work-stealing scheduler in [15], which is implemented similarly to Cilk for parallelism. We compiled our programs with the \texttt{gcc} compiler (version 9.2.1) having the -03 flag and ran all our experiments five times and report the average.

**Implementation.** We implemented \textsc{Wharf} in C++ on top of Aspen [15] and used weight-balanced trees as the underlying balanced tree implementation [7, 52]. Note that Aspen’s current implementation supports storing up to 64-bit integers in C-trees. Consequently, each Szudzik operand in \textsc{Wharf} should be up to 32 bits. We stress that this is not a limitation of \textsc{Wharf}, but of the 64-bit implementation of Aspen (on which we built). Yet, \textsc{Wharf} can still support much larger graphs for PPR use cases where walks are shorter (around 5-15 vertices long). As explained in [3], the theoretical guarantees are preserved for \( n_w = 10 \) and \( l = 10 \), so \textsc{Wharf} can scale to graphs with up to \( (2^{15} - 1)/100 \approx 42.94M \) vertices, such as the Twitter dataset [25] as we show in our experiments. Note that the walk mixing time [45, 46] depends only on the walking model.

**Table 1: Datasets Statistics.**

| Graph          | Num. Vertices | Num. Edges | Avg. Degree |
|----------------|---------------|------------|-------------|
| com-YouTube    | 1,134,890     | 2,987,624  | 5.30        |
| soc-LiveJournal| 4,847,571     | 85,702,474 | 17.80       |
| com-Orkut      | 3,072,627     | 234,370,166| 76.20       |
| Twitter        | 41,652,230    | 1,468,363,182| 57.70    |

**Datasets.** We used four real-world (Table 1) and eight synthetic graph datasets: \textit{Real Graphs} – \textit{com-YouTube} is an undirected graph of the Youtube social network [57], \textit{soc-LiveJournal} is a directed graph of LiveJournal social network [2], \textit{com-Orkut} is an undirected graph of Orkut social network [57], and \textit{Twitter} is a directed graph of the Twitter network, in which edges model the...
followed by followee relationship [25]. Synthetic Graphs. We generated large-scale synthetic graphs sampled from the R-MAT model [8]. Specifically, we used the TrillionG\(^3\) [40] tool to generate Erdős Rényi, er-k, graphs with \(2^k\) nodes, uniformly distributed edges and with an average vertex degree of 100 by setting the R-MAT parameters to \(a = b = c = d = 0.25\). Additionally, we varied \(k\) from 16 to 22 to evaluate the scalability of Wharf. We also generated a set of skewed graphs, sg-s, with \(2^{20}\) nodes with an average degree of 10, while varying the skew. We set the R-MAT parameters \((a, b, c, d)\) so that the number of edges in the bottom-right part of the matrix is about \(s\) times the top-left part of the matrix. We set \(b = c = 0.25\). Thus, when \(s = 1\), there is no skew, while if \(s > 1\), R-MAT generates power-law graphs. We varied \(s\) from 1 to 7 with a step of 2.

Baseline. As there is no system that maintains streaming random walks, we compared Wharf with approaches proposed in [18, 44], which use an inverted index for maintaining walks. We call this baseline Inverted Index-based (II-based). Specifically, II-based maintains the walks separately from the graph in sequences of vertices stored in vectors. We used a dictionary for storing the walks, where the key is the walk id and the value is the walk sequence. Additionally, II-based maintains an inverted index that relates a vertex id with the set of walk ids it participates. For fairness reasons, we implemented a fully parallel version of II-based by utilizing concurrent hashables\(^4\). We also used a Tree-based baseline, which stores the walk-triplets into parallel balanced binary trees (a.k.a. parallel augmented maps) [52]: a structure that provides highly parallel operations and upon which C-trees are built.

7.2 Overall Performance

Throughput & Latency. We compare Wharf with II-based and Tree-based in terms of throughput, i.e., number of updated walks per second, and latency, i.e., the average time for updating one walk, when updating random walks. If not stated otherwise, we use the DeepWalk [41] walking model with the default parameters, i.e., \(n_w = 10\) walks per vertex of length \(l = 80\). For this experiment, we used the real graphs and generated walks of default length for the first three real datasets, whereas \(l = 10\) for twitter dataset. We also produced batches of 10,000 edges, which we sampled based on the R-MAT [8] model with parameters \(a = 0.5, b = c = 0.1,\) and \(d = 0.3\) to induce graph updates as in \([15]\). We inserted 10 such batches in total and report the average throughput and latency.

Figure 6 illustrates the throughput and latency results. We observe that Wharf is superior to both baselines in all cases. It achieves up to \(\sim 2.6x\) higher throughput and \(2x\) lower latency, which is crucial for streaming applications. This is thanks to its parallel MultiInsert and Merge operations that update different parts of the walk corpus on the hybrid-tree simultaneously. The advantage of Wharf is more evident for the datasets where we used larger walk lengths, such as Livejournal. Contrary to what one may think the Szudzik encoding/decoding function calls required only \(3.66\%\) of the total time for walk updates in com-YouTube, \(10.055\%\) in soc-LiveJournal, \(7.797\%\) in com-Orkut, and \(12.815\%\) in Twitter.

---

\(^1\)https://github.com/chan150/TrillionG

\(^4\)https://github.com/efficient/libcuckoo

As expected, there exists a throughput-memory trade-off: one can achieve higher throughput at the price of a higher memory footprint by merging less frequently.

One may find a discussion on the merge policy in the full version of our paper [47].
Figures 8b and 8c illustrate the total memory footprint when varying the walk length \( l \) and the number of walks per vertex \( n_w \). We report the results only for the soc-LiveJournal dataset because we observed the same behaviour for the other real graphs. Figure 8b shows a linear behaviour in terms of space consumption with respect to the walk length. Wharf requires on average \( \sim 1.5x \) less space than II-based and \( \sim 3.76x \) less space than Tree-based. Figure 8c shows again a linear behaviour with respect to \( n_w \), i.e., Wharf has on average \( \sim 1.6x \) smaller memory footprint compared to II-based and \( \sim 3.77x \) smaller than Tree-based. These space-savings are thanks to the use of pairing functions in combination with differential encoding in the chunks of walk-trees. The reader might think of applying a simple compression technique in II-based, but existing techniques for compressing inverted indexes are neither trivial nor suitable for dynamic data [42]. Therefore, we conclude that our proposed walk-tree structure enables Wharf to store an indexed walk corpus space-efficiently.

Based on all the results above, we decided to discard the Tree-based baseline, and keep only the II-based baseline, in the subsequent experiments: II-based is comparable to Tree-based, in terms of throughput, while occupying much less space.

### 7.3 Scalability

#### Batch Size

We now demonstrate Wharf’s scalability when varying the batch size for edge insertions. We produced batches with sizes 10, 25, 50, 75, and 100 thousand edges that we inserted in com-Orkut. Notice that we omit the results for the other two real datasets because they follow the same trend as for com-Orkut.

Figure 9a illustrates the throughput results, where the black horizontal lines represent the minimum throughput required to generate the walks from scratch. We observe that Wharf is always better than recomputing the random walks from scratch, which is not the case for II-based for batch sizes larger than 25K. This is because II-based can only perform 72.3K walk updates per second. In general, Wharf achieves up to \( \sim 2.6x \) higher throughput than II-based. We also observe that the throughput of both Wharf and of II-based decreases as the batch size increases, namely, \( \sim 22.4\% \) for Wharf and \( \sim 10.6\% \) for II-based going from batch size of 10K to 100K edges. The reason is that the larger the batch size is the higher the average number of affected walks is. Wharf’s throughput decreases a bit more because (i) the time to compute the MAV increases as the walk trees get larger with larger batch sizes, and (ii) the minimum position of the affected walks decreases. During our experiments, we observed that as the batch size increases not only the number of affected walks increases, but also more and more walks are affected at an earlier point of the walk sequence. This leads to more work for updating the random walks. For instance, while inserting 50K edges leads to \( \sim 463K \) affected walks from the first position, inserting 100K edges leads to \( \sim 874K \) affected walks from the first position.

Figure 9b illustrates the latency results. We observe that the latency of Wharf is \( \sim 2x \) lower than the one of II-based. We also see that both Wharf’s and II-based latency increases as the batch size increases because of the increased number of walks. Yet, Wharf’s latency stays considerably low thanks to its on-demand policy for merging. We thus conclude that Wharf is more scalable than the baseline in scenarios with many updates per batch.

#### Input Graph Size

We also study Wharf’s scalability when varying the input graph size w.r.t. the number of vertices. For this experiment, we used the \( er \)-graphs and fixed the batch size to 10K edges. Figures 10a and 10b illustrate the throughput and latency, respectively. We see that Wharf achieves \( 1.9 \sim 2.5x \) higher throughput than II-based, and \( \sim 1.8 \sim 2x \) lower latency. This is attributed to two things: (i) the way Wharf stores the walks in the hybrid-tree that enables updating various parts of the corpus simultaneously, and (ii) its on-demand merge policy. Furthermore, we observe that, as the distribution of vertex degree in the \( er \)-graphs is uniform, the throughput of both solutions remains steady: \( \sim 115K \) walks/second for Wharf and \( \sim 50K \) walks/second for II-based. Consequently, the number of affected walks increases proportionally to the graph size. We conclude that it nicely scales with varying graph sizes.

### 7.4 Performance under Data Skewness

Next, we investigate the effect of graph skew on Wharf’s performance in terms of throughput and memory footprint. Specifically, we use a set of skewed graphs, \( sg-s \), which all have \( 2^{20} \) vertices, where we vary the skew factor \( s = 1, 3, 5, 7 \). We set the batch size to
uses when seeking a specific walk triplet inside a walk-tree. As baseline, we disabled this range search and leave Wharf with the simple search that checks triplets by scanning the entire walk-trees.

Figure 12a illustrates the throughput improvement factor (IF) that Wharf’s range search achieves when running node2vec, with parameters $p = 0.5$ and $q = 2$, on all the real graphs. We observe that the range search feature leads up to $3\times$ higher throughput than the baseline. Note that in smaller graphs, such as com-YouTube, the gains from range search are higher than in larger graphs, such as soc-LiveJournal. This is because smaller graphs have less vertices in their walk-trees, which makes the range search faster as the constructed ranges are smaller. Furthermore, Figure 12b shows the throughput IF for node2vec when varying the batch sizes of edge insertions on the large graph soc-LiveJournal, where the walk-trees contain a huge amount of walk triplets. We get similar results for com-Orkut but we omit them due to space limitations. In this case, range search enables Wharf to achieve on average $\sim 1.7\times$ higher throughput than the simple search. Note that the space overhead for the $\{min, max\}$ bounds in each walk-tree (necessary for our search range search) is negligible (less than 1%). We thus conclude that the range search technique significantly contributes to the high throughput of Wharf with negligible space overhead.

**Benefits of Difference Encoding.** We now explore the impact of difference encoding (DE) on Wharf’s throughput and memory footprint. For this, we disabled the DE in Wharf and used the resulting variant as our baseline. We inserted 10 batches of 10K edges and measured the average throughput and the memory footprint after the merge operation for all our real graphs. We observed that Wharf needs up to $1.4\times$ less space to store the walks than when not using DE. It also achieves quite a similar throughput for all real graphs that is within 5% as the one achieved when not having DE. For instance, on the LiveJournal dataset the throughput with DE is $\approx 207.4$K walks/second, whereas without it is $\approx 214.6$K walks/second. We thus conclude that compression via difference encoding helps improve the memory footprint of Wharf, however, its performance does not stem from compression but from our proposed techniques.

**Vertex Id Distribution.** We now explore the effect that the vertex id distribution has to the space needed to store random walks. Specifically, we used our $er$-18 graph that has 262, 144 vertices as the initial graph $G_1$. In $G_1$, the vertex ids are fully clustered, i.e., they range from 0 to 262, 143, as produced by TrillionG [40]. From $G_1$, we produced $G_{2-\times 20}$ by multiplying the vertex ids by 20 to make the ids of the graph non-clustered, yet ordered. Additionally, we created two more graphs out of $G_1$, namely, $G_{3-\times 1M}$ and $G_{4-\times 5M}$ where we reassigned a unique random id to each vertex drawn from

---

### 7.5 In-Depth Study

**Range vs. Simple Search.** We proceed in exploring the benefits of the output-sensitive FindNext range search algorithm that Wharf
the $[0−1M]$ and $[0−5M]$ ranges, respectively. We observed that the space that Wharf needs to store the walks for $G_1$ is 1.553 GB, for $G_2$ is 1.547 GB, for $G_{3−1M}$ is 1.553 GB, and for $G_{4−5M}$ is 1.547 GB. Thus, the delta encoding scheme ensures the space Wharf needs to store the walks is not affected by the vertex id distribution.

7.6 Effectiveness of Downstream Tasks

Lastly, we measure the accuracy of a vertex classification task and a PPR task on Cora to show Wharf’s effectiveness in maintaining walks. For the former, we implemented an incremental learning approach that uses Wharf: it builds a predictive model, after each graph update (snapshot), from embeddings that use Wharf’s walks. For the latter, we implemented [3] in Wharf for producing and updating the walks used for approximating PPR scores. At each timestep, we ingest a new batch of 250 edges.

**Vertex classification.** As baselines for the vertex classification task, we considered (i) the ideal learning case, i.e., learning a new model at every single snapshot, and (ii) the periodic learning case, i.e., learning a new model every $k$ snapshots (we use $k = 5, 10$). Both ideal and periodic learn embeddings using random walks computed from the scratch. For the incremental learning case (which is based on Wharf), Wharf updates the walks after each batch insertion so that they remain statistically indistinguishable. We, then, incrementally refine the embeddings using yskip [22] with default DeepWalk parameters (i.e., $n_w = 10, l = 80$), and trained 128-sized embedding vectors. We used LogisticRegression for the classification and report the average $F1$ score of three runs. Figure 13a shows that the incremental learning achieves overall the same accuracy as the ideal learning, demonstrating the high effectiveness of Wharf to maintain high-quality random walks. Note that in the periodic learning scenario the accuracy drops significantly in between the snapshots where re-training takes place. The larger the period (e.g., $k = 10$) the lower the accuracy stays. These results are aligned with Figure 1a, where we witnessed a large decrease in the accuracy of a link prediction task if we do not keep the embeddings up-to-date. Such drops in accuracy can have large negative impact in the underlying ML tasks, especially for high-stakes applications, such as fraud detection. We can then also conclude that having statistically indistinguishable random walks is crucial.

**Personalized pagerank.** For the PPR task, we considered a static variant of [3] as baseline, which reuses the existing random walks instead of updating only the affected walks at every snapshot. We generated 10 walks per vertex with a restart probability of 0.2, and report the Symmetric Mean Average Error (SMAPE) between [3] and the static variant (Figure 13b). We observe that as more graph snapshots arrive the SMAPE constantly increases. In fact, even after the first snapshot arrives, the error is already greater than 40%. These results are aligned with Figure 1b, yet, because of the smaller batch sizes we used here, the error gradually increases. These results confirm the vertex classification results: keeping random walks statistically indistinguishable is crucial for the downstream tasks.

8 RELATED WORK

**Random Walk Systems.** KnightKing [58] is a distributed system for computing random walks on static graphs based on a walker-centric computation model, which is able to express various walk algorithms. ThunderRW [50] is a single-node system that conducts in-memory random walks by devising a step-centric computation model, which hides memory access latency by executing multiple queries in an alternating manner. In contrast to the above systems, Wharf is designed for streaming manner, does not impose any main memory constraints, and supports walks of any order.

**Dynamic GRL.** Barros et al. [4] categorize random walk-based GRL methods on dynamic graphs into: (i) random walks on snapshots, (ii) evolving random walks, and (iii) temporal random walks. In the first category, random walks are re-computed at every snapshot so that embeddings are learned from scratch. Wharf falls into the second category, where random walks are not recomputed from scratch after every graph update, but they are updated along with the embeddings of the affected vertices [18, 30, 44]. Yet, Wharf stores, indexes, and updates the walks more efficiently than the competitor approaches as we show in the experimental section.

**Dynamic PageRank.** Bahmani et al. [3] present a method for calculating PPR scores via preprocessing and storing random walks for each node in the graph. The stored walks are not indexed leading to a full scan of walks for each incoming edge update. In contrast, Wharf’s structure offers an index on the walks which leads to faster walk updates. Mo et al. [32] present the Agenda framework for fast and robust Single Source PPR queries on evolving graphs, yet it does not store the entire random walks in main memory, which might require recomputing walks from scratch.

9 CONCLUSION

We presented Wharf (the first step towards Kaixis [39]), a system that produces and updates random walks in a streaming fashion while storing them succinctly. Wharf represents walks concisely by coupling compressed purely functional binary trees and pairing functions and updates the walks efficiently by pruning the search space leveraging the ordering properties of pairing functions. Our experiments show that Wharf can incrementally update walks with up to $2.6\times$ higher throughput and up to $2\times$ lower latency than inverted index-based baselines.

ACKNOWLEDGMENTS

We thank Dordije Krivokapić who helped in the initial stages of Wharf as part of his MSc. thesis, and Dr. Eleni Tzirita Zacharatou for her feedback. This work was funded by the German Ministry for Education and Research as BIFOLD - Berlin Institute for the Foundations of Learning and Data (ref. 01IS18025A and ref. 01IS18037A).
