International Journal of Parallel Programming (2024) 52:147–170
https://doi.org/10.1007/s10766-024-00768-x

GraphTango: A Hybrid Representation Format for Efficient Streaming Graph Updates and Analysis

Alif Ahmed[1] · Farzana Ahmed Siddique[1] · Kevin Skadron[1]

Received: 20 October 2023 / Accepted: 15 April 2024 / Published online: 18 May 2024
© The Author(s) 2024

Abstract

Streaming graph processing performs batched updates and analytics on a time-evolving graph. The underlying representation format of the graph largely determines the throughputs of these updates and analytics phases. Existing representation formats usually employ variations of hash tables or adjacency lists. However, a recent study showed that the adjacency-list-based approaches perform poorly on heavy-tailed graphs, and the hash table-based approaches suffer on short-tailed graphs. We propose GraphTango, a hybrid representation format that provides excellent update and analytics throughput regardless of the graph's degree distribution. GraphTango dynamically switches among three different formats based on a vertex's degree: (i) Low-degree vertices store the edges directly with the neighborhood metadata, confining accesses to a single cache line, (2) Medium-degree vertices use adjacency lists, and (3) High-degree vertices use hash tables as well as adjacency lists. In this case, the adjacency list provides fast traversal during the analytics phase, while the hash table provides constant-time lookups during the update phase. We further optimized the performance by designing an open-addressing-based hash table that fully utilizes every fetched cache line. In addition, we developed a thread-local lock-free memory pool that allows fast growing/shrinking of the adjacency lists and hash tables in a multi-threaded environment. We evaluated GraphTango with the help of the SAGA-Bench framework and compared it with four other representation formats: Stinger, Degree-aware Robin Hood Hashing, and two adjacency list-based formats with different workload balancing scheme. On average, GraphTango provides 4.5x higher insertion throughput, 3.2x higher deletion throughput, and 1.1x higher analytics throughput over the *next best* format. Furthermore, we integrated GraphTango with the state-of-the-art graph processing frameworks DZiG and RisGraph. Compared to the *vanilla DZiG* and *vanilla RisGraph*, [*GraphTango* + *DZiG*] and [*GraphTango* + *RisGraph*] reduces the average batch processing time by 2.3x and 1.5x, respectively.

Alif Ahmed and Farzana Ahmed Siddique have contributed equally to this work.

Extended author information available on the last page of the article

Keywords Streaming graph · Graph processing · Dynamic graph · Hashing

1 Introduction

Streaming graph processing involves performing batched updates and analytics on a time-evolving graph. The update phase handles modifications to the graph topology (e.g., insertion/deletion of edges and nodes), while the analytics phase runs the necessary algorithms on the graph. This is a common scenario in many real-world graph applications such as social network analysis [1, 2], bioinformatics [3, 4], recommendation systems [5, 6], routing and navigation [7], knowledge discovery [8], sensor networks [9], etc. The focus of streaming graph processing is fundamentally different from static graph processing. *Static graphs* are constructed only once, and the construction cost gets amortized over time. Therefore, the overall performance of a static graph processing framework is primarily determined by the analytics throughput. In the case of *streaming graphs*, the graph topology can change very frequently. Hence, both update and analytics throughput is critical for streaming graphs [10].

The most common operation during the update phase is *edge lookup*. The lookup is performed before insertion to avoid duplicate edges[1] and before deletion to find the location of the target edge. On the other hand, the most common operation during the analytics phase is the *neighborhood traversal* of a given vertex. The performance of a streaming graph processing framework is critically dependent on how efficiently the graph storage format can support these lookup and traversal operations. Existing storage formats for streaming graphs usually employ variations of adjacency lists or hash tables [10–14]. Approaches based on adjacency lists [10, 11] provide high update throughput on short-tailed graphs[2] but suffer in heavy-tailed graphs as it requires linear lookup through the edge array [10]. On the other hand, hash-based approaches [12, 13] offer constant-time lookup, providing better update throughput on heavy-tailed graphs. However, they perform poorly on short-tailed graphs because the overhead of hash calculation and several random accesses becomes more expensive than conducting a simple linear search. Furthermore, edges are stored in hash tables relatively sparsely to mitigate collisions. As a result, edge traversal becomes inefficient and negatively impacts their analytics phase's throughput. None of the existing approaches can efficiently handle both short-tailed and heavy-tailed graphs.

This paper proposes GraphTango, a streaming graph representation format that provides excellent performance regardless of the graph's degree distribution. Our key idea is to adaptively switch the underlying data structure based on the vertex degree: (i) *Type1 vertex*: Low-degree vertices where the edges are stored within

[1] In accordance with the prior works [10–13], edges are inserted only after a lookup to avoid duplicate edges.

[2] Following prior work [10], we define heavy/short-tailed graph with respect to an update batch: heavy-tailed graphs have high *maximum degree* within a batch. Short-tail is the opposite.

the same cache line as the neighborhood metadata. Update and edge traversal thus requires only one cache line access, unlike other approaches. (ii) *Type2 vertex*: Medium-degree vertices that store edges as adjacency lists. The degree is too high for this type to fit all edges in a cache line, but small enough so that linear search performs better than hashing. (iii) *Type3 vertex*: High-degree vertices that store edges as adjacency lists, along with hash tables storing indexes to the adjacency lists. In this case, the adjacency list provides optimal edge traversal during the analytics phase, while the hash table provides constant-time lookup during the update phase. The hash tables are not accessed during the analytics phase, avoiding any potential cache pollution. To improve the cache access pattern of the hash table, we designed an open-addressing-based hash table with double hashing that fully utilizes every fetched cache line. Our proposed hashing scheme minimizes cache line fetches and is especially beneficial if the hash tables do not fit into the last level cache (LLC), which is often the case for real-world graph workloads.[3] With this hashing scheme, updates for Type3 vertices are performed with only three cache line accesses for more than 99.2% of the cases. In addition, we developed a thread-local lock-free memory pool that allows fast growing and shrinking of the adjacency lists and hash tables in a multi-threaded environment.

We evaluated GraphTango by integrating it with the SAGA-Bench [10] benchmarking framework. SAGA-Bench integration ensures that all approaches use the same algorithm implementations via a common API. Therefore, any performance improvement comes purely from the data structure standpoint. SAGA-Bench comes with four representation formats: AdListShared, AdListChunked, Stinger [11], and DegAwareRHH [13], each of which is shown to excel in different algorithm and dataset combinations [10]. Details of these formats can be found in Section II. For update operations, GraphTango consistently performed best across all datasets. On average (maximum), GraphTango demonstrates 4.5x (6.6x) higher insertion throughput and 3.2x (5.0x) higher deletion throughput over the *next best* approach. As for analytics, GraphTango offers 1.1x (1.6x) higher throughput than the *next best* approach. Unlike prior approaches, GraphTango provides excellent update and analytics throughput for both short-tailed and heavy-tailed graphs.

Being a storage format, GraphTango is orthogonal to most full-fledged graph processing frameworks and can easily replace the underlying storage formats of those frameworks. To demonstrate, we integrated GraphTango with the state-of-the-art graph processing frameworks DZiG [15] and RisGraph [16]. *DZiG + GraphTango* reduced the overall batch processing runtime by 2.3x (5.2x) on average (maximum) compared to the original DZiG. *RisGraph + GraphTango* reduced the overall batch processing runtime by 1.5x (1.9x) on average (maximum) compared to the original RisGraph.

GraphTango will be made available on GitHub, both as a standalone framework and as an integration with SAGA-Bench, DZiG, and RisGraph.

[3] Even with our smallest dataset of 5 M edges, the LLC miss rate during the update phase is over 49%, indicating that the working set size is larger than the LLC.

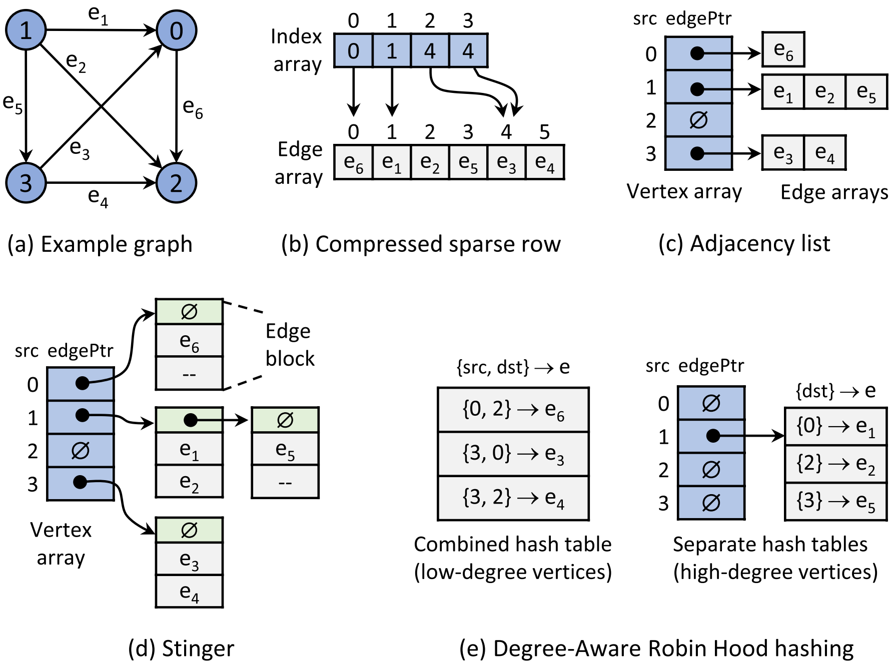

Fig. 1 Example of different graph representation formats. Here, each edge e is an $\{dst, prop\}$ tuple

2 Existing Representation Formats

Figure 1 illustrates how various graph representation formats store vertices and edges. While these examples store only the outgoing edges, the concept is also applicable if storing incoming edges.

2.1 Compressed Sparse Row (CSR)

is one of the most commonly used formats for *static* graphs [17–20]. As shown in Fig. 1b, CSR organizes data in an *edge array* and an *index array*. Edges are stored in the edge array in ascending order - all edges of vertex v_i appear before any edge of v_{i+1}. The index array stores the position of the first edge of every vertex. CSR is widely used for static graphs because it provides a compact representation, increasing spatial locality while traversing the graph. However, inserting or deleting an edge requires reconstructing both the edge array and the index array, making CSR unsuitable for dynamic graphs.

2.2 Adjacency List

stores the edges of every vertex in separate arrays (Fig. 1c). A *vertex array* stores the pointers to these edge arrays. **These edge arrays are assumed to be memory-contiguous (like *std::vector*), rather than a linked list of edges.** This important distinction is used throughout the paper. As each edge array can grow/shrink independently, insertion and deletion operations only modifies the edge array of the corresponding vertex. This property makes adjacency lists a common choice for dynamic graph frameworks [10, 14]. Another advantage of adjacency lists is that the edge traversal during the analytics phase has a sequential access pattern, leading to excellent analytics throughput for vertex-centric algorithms. The downside of adjacency lists is that the edges are not stored in any particular order within an edge array. Therefore, finding an edge requires a linear search through the corresponding edge array, leading to poor update throughput on high-degree vertices.

In adjacency-list-based approaches, parallel updates on multiple vertices are realized in two ways. The first scheme is the shared style multithreading (referred as ***AdListShared***), where the vertex array additionally contains a lock for every vertex. Any thread can process updates on any vertex by acquiring the corresponding lock first. This approach provides fine-grained parallelism. However, if most updates are targeted towards the *same* vertex, it can cause lock contention and is often the case for heavy-tailed graphs. The alternative scheme groups source vertices into chunks and assign each chunk to a fixed thread (referred as ***AdListChunked***). Chunked style multithreading is lock-free. However, it is prone to workload imbalance if the chunks have a high disparity in the number of edges they contain.

2.3 Stinger

Ediger et al. [11] is an adjacency-list-based representation format. As illustrated in Fig. 1d, Stinger stores the edges as linked lists of *edge blocks*. Each edge block can accommodate a fixed number of edges (default is 16). Parallelism in Stinger is achieved by acquiring locks on the edge blocks. The capacity of the edge blocks presents a trade-off between performance and storage requirements. Using smaller capacity edge blocks increase parallelism but makes graph traversal inefficient by increasing the amount of pointer-chasing accesses. On the other hand, larger blocks lead to many unused slots for low-degree vertices. Besides, like adjacency lists, Stinger also suffers from linear lookups on high-degree vertices, stagnating the update throughput.

2.4 Degree-Aware Robin Hood Hashing (DegAwareRHH)

Iwabuchi et al. [13] is a hash-based format. As shown in Fig. 1e, DegAwareRHH maintains two types of hash tables based on the vertex degree. Edges corresponding to low-degree vertices are stored in a combined hash table to improve data locality.

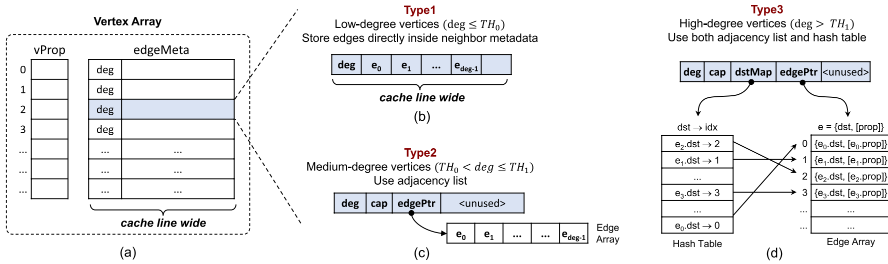

Fig. 2 Proposed hybrid representation format of GraphTango

On the other hand, each high-degree vertex maintains its own hash table. Both of these hash tables use Robin Hood hashing [21], which minimizes probing distance. For parallelism, DegAwareRHH leverages chunked-style multithreading similar to AdListChunked. The constant time lookup enabled by the hash tables makes DegAwareRHH suitable for the update phases on heavy-tailed graphs. However, the sparse storage of edges in the hash table makes DegAwareRHH's edge traversal inefficient, negatively impacting the analytics throughput.

3 GraphTango Data Structure

Figure 2 gives an overview of the GraphTango data structure.[4] GraphTango organizes the vertex data in two arrays: one for storing the vertex properties (*vProp*) and the other for storing neighborhood metadata of the vertex (*edgeMeta*). These arrays are indexed using vertex id. Neighbors of each vertex are stored as an $e_x = \{dst, [prop]\}$ tuple, where $e_x.dst$ is the destination vertex id, and $e_x.prop$ is an optional edge property (e.g., the weight of the edge).

The *edgeMeta* array is aligned to a page boundary,[5] and each element of the array is of cache line size. Therefore, accessing any field of $edgeMeta[i]$ will bring the rest of the fields into the cache. The *deg* field holds the current degree of the corresponding vertex. Depending on the degree, a vertex will fall into one of the following three categories:

3.1 Type1 Vertex

These are low-degree vertices with $deg \leq TH_0$. As illustrated in Fig. 2b, we store the edges directly with the metadata for Type1 vertices. The threshold TH_0 denotes the number of edges that can fit inside the metadata and is defined as:

[4] The description assumes storing only outgoing edges for clarity. In our implementation, we stored both incoming and outgoing edges for directed graphs.

[5] To clarify, only the *edgeMeta* array itself is page boundary aligned, not the edge arrays or hash tables it may point to.

$$TH_0 = \left\lfloor \frac{CACHE_LINE_SIZE - sizeof(deg)}{sizeof(e)} \right\rfloor$$

For example, $TH_0 = 7$ for a typical cache line size of 64 bytes and edges of 8 bytes. The advantage of storing edges with metadata is that all edges are brought into the cache as soon as we access the vertex during the update or analytics phase. When searching for a specific edge, we need to do a linear search. However, the search is extremely fast, as all accesses will be cache hits.

3.2 Type2 Vertex

These are medium-degree vertices with $TH_0 < deg \leq TH_1$, where TH_1 is a user-configurable threshold. Edges for this type of vertices are stored in adjacency lists, as shown in Fig. 2c. To support adjacency lists, *edgeMeta* additionally maintains the current capacity (*cap*) and a pointer to its edge array (*edgePtr*).

Like Type1 vertices, Type2 also requires a linear search when looking for a specific edge. As the linear search on the edge array is prefetcher-friendly and has good spatial locality, it offers better performance than hash-based search up to a certain point (i.e., tuned using the TH_1 threshold). However, the linear nature of the search becomes a performance bottleneck for higher-degree vertices. Hash-based search is preferable in such cases, as explained below.

3.3 Type3 Vertex

These are high-degree vertices with $deg > TH_1$. Figure 2d illustrates this scenario. Here, we maintain *both an adjacency list and a hash table for each Type3 vertex*. The hash table maps an edge's destination vertex id ($e_x.dst$) with its location in the corresponding adjacency list. Maintaining both hash table and adjacency list comes with the following benefits: (i) The hash table enables constant-time lookups during the *update phase*. (ii) The adjacency list provides fast and efficient traversal during the *analytics phase*. Prior hash-based approaches suffer from low analytics throughput due to inefficient edge traversal [10]. GraphTango is free of this issue because it uses only the adjacency lists for edge traversal and does not require accessing the hash tables during the entirety of the analytics phase.

4 GraphTango Basic Operations

4.1 Edge Insertion

The edge insertion procedure is as follows: (i) Retrieve the edge metadata - *edgeMeta*[*srcId*]. (ii) If the current *deg* reaches the current capacity, we double the capacity. The exact steps for capacity doubling will depend upon the current and new type, as demonstrated in Table 1a. In general, capacity doubling involves allocating

Table 1 Vertex type switching steps for insertion/deletions

(a) Insertions triggering type switch or capacity doubling

Direction	New capacity	Alloc new edge array	Edge copy size	Dealloc old edge array	Rehash
Type1 → Type2	nextPow2(TH_0)	✓	deg (=TH_0)	X	X
Type2 → Type2	cap * 2	✓	deg	✓	X
Type2 → Type3	cap * 2	✓	deg (=TH_1)	✓	✓
Type3 → Type3	cap * 2	✓	deg	✓	✓

(b) Deletions triggering type switch or capacity halving

Direction	New capacity	Alloc new edge array	Edge copy size	Dealloc old edge array	Rehash
Type3 → Type3	cap / 2	✓	deg	✓	✓
Type3 → Type2	cap / 2	✓	deg (=TH_1)	✓	X
Type2 → Type2	cap / 2	✓	deg	✓	X
Type2 → Type1	TH_0	X	deg (=TH_0)	✓	X

memory for the larger edge array, copying current edges to the new edge array, and freeing the old array. For Type3, the hash table is also rehashed. The amortized cost of capacity doubling is O(1) [22]. (iii) Search for a duplicate edge using *dst*. As mentioned earlier, for Type1 and Type2, it will involve doing a linear search, and for Type3, the search will be performed using the hash table. (iv-A) If the edge is found, update the property and return. (iv-B) If the edge is *not* found, add the edge at the end of the edge array and increment *deg*. For Type3, we also create an entry in the hash table pointing to the location.

4.2 Edge Deletion

The edge deletion procedure is as follows: (1) Retrieve the edge metadata - *edgeMeta*[*srcId*]. (ii) Search for existing edge using *dst*. (iii-A) If the edge is *not* found, return. (iii-B) If the edge is found, delete the entry from the edge array and hash table (for Type3) and decrement *deg*. We do a ***compaction step*** here to fill the gap. It involves moving the last entry of the edge array to the deleted entry's position and updating the corresponding hash table record. The compaction step is simple and is of constant time complexity. (4) If the *deg* becomes 1/4th of the capacity, we halve the capacity. The steps for capacity halving is given in Table 1b. Similar to the capacity doubling during insertion, the amortized cost of capacity halving is also O(1) [22].

4.3 Edge Traversal

As we store the edges in consecutive memory for all three vertex types,[6] the edge traversal API simply returns a cursor (i.e., position of the iterator) for indexing to the: (i) $edgeMeta[vid]$ for Type1 vertices, or (ii) $edgeMeta[vid].edgePtr$ for Type2/Type3 vertices. GraphTango's traversal mechanism is essentially the same as an adjacency list for Type2/Type3 vertices. As for Type1, GraphTango has a better access pattern as it requires one less indirection.

5 Optimizing GraphTango

5.1 Cache-Friendly Hashing Scheme

The hash table used by the Type3 vertices can be realized in several ways. The most convenient approach is to use *std::unordered_map*. Unfortunately, this approach is not ideal for our purpose because the C++ standard [23] effectively limits the collision resolution of *std::unordered_map* to separate chaining,[7]. With separate chaining, the hash table is constructed as an array of buckets. Each bucket points to a linked list of colliding elements (i.e., keys that hashed to the same bucket). The issue with separate chaining is that it involves multiple random accesses - one to access the bucket and one or more for traversing the linked list. Each of these random accesses is a potential cache miss if the hash table does not fit into the cache. An alternative to separate chaining is open addressing, where all elements are stored in the hash table itself, eliminating the need for linked lists traversals. Prior hash-based graph representation formats [12, 13] leveraged open-addressing-based Robin Hood hashing [21] that minimizes probing distance. For GraphTango, we designed a more cache-friendly open-addressing-based hash table that minimizes the number of cache line accesses, making it especially suitable for real-world graph workloads where the hash table is unlikely to fit into the cache.

The key idea of our hashing scheme is to limit the probes within a single cache line until it is fully searched, before moving onto a different cache line. Figure 3 illustrates this hashing scheme. The hash table itself is composed of an array of $\{key, value\}$ pairs. The index of the i-th probe to the hash table is given by the following hash function:

$$h(key, i) = N \cdot h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right) + h_2(key, i \bmod N)$$

Here, N is the number of $\{key, value\}$ pairs that can fit within a single cache line. The purpose of $h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right)$ is to select a cache line for probing and returns the

[6] Even after deletions, our compaction step ensures that all valid edges of a vertex are stored in consecutive memory.

[7] This constraint is a side effect of mandating *pointer stability* which means that an iterator must remain valid upon inserting or deleting elements.

$$h(key, i) = N * h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right) + h_2(key, i \bmod N)$$

Permutation of $\{0, \ldots, \boldsymbol{M}-1\}$
(selects a cache line)

Permutation of $\{0, \ldots, \boldsymbol{N}-1\}$
(selects offset within the cache line)

$\boldsymbol{M}$: Number of cache lines in the hash table
$\boldsymbol{N}$: Number of $\{key, value\}$ pairs within a single cache line

(a)

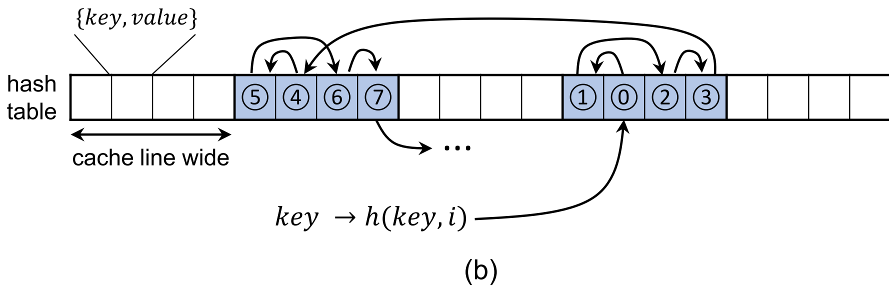

Fig. 3 Proposed hashing scheme. **a** Hash function to determine the index to the hash table. **b** An example probing sequence for $M = 5$ and $N = 4$

base index of that selected cache line. Note that the $\left\lfloor \frac{i}{N} \right\rfloor$ parameter remains the same for every N consecutive probes, thereby selecting the same cache line. As $h_1()$ should eventually explore all cache lines in the hash table, it must be a permutation of $\{0, 1, ..., M-1\}$, where M is the number of cache lines in the hash table. On the other hand, $h_2(key, i \bmod N)$ determines the offset within the cache line and must be a permutation of $\{0, 1, ..., N-1\}$. Any hash function conforming to this permutation requirement can be used to implement $h_1()$ and $h_2()$. In GraphTango, we used double hashing for $h_1()$ to avoid primary/secondary clustering. $h_2()$ uses linear probing to make hash computation simpler. Our hash function is very cheap to compute, with the reference implementation having two multiplications and eight other simple arithmetic/logical instructions. This is because we ensure that both N and M are powers-of-two, converting expensive modulus and division operations to simple shifts. Further optimization is possible by leveraging SIMD instructions to do a parallel comparison on all entries mapped to the same cache line. However, as discussed later, GraphTango demonstrates short probing distance, making iterative comparison just as performant. **Interested readers can find the implementation details of these hash functions in the Appendix** A.

Insertions and deletions to the proposed hash table are similar to other open-addressing-based hash tables. Each location of the hash table can contain either: (i) a valid $\{key, value\}$ pair, or (ii) an *empty* marker, or (iii) a *deleted* marker (i.e., tombstone). We used two reserved values as the empty and deleted marker instead of

using dedicated tag storage. During both insertion and deletion, the table is probed (using the hash function) until the *key* or an empty marker is found. If the *key* is found: (i) For insertion, the corresponding *value* is updated. (i) For deletion, the entry is marked as deleted. Instead of *key*, if an empty marker is found: (i) No action is required for deletion. (ii) For insertion, the $\{key, value\}$ pair is inserted to the location of the first encountered delete marker, or to the current location if no delete marker was encountered.

When using with GraphTango, the hash tables' initial capacity is set to twice the capacity of the corresponding adjacency lists. Upon inserting/deleting edges, both the hash tables and the adjacency lists can grow/shrink in size (see Sect. 4), but the capacity ratio always remains 2. This property sets the maximum load factor (α) of the hash table to 0.5. Assuming uniform hashing,[8] the theoretical average probing distance is: (i) $\frac{1}{1-\alpha} = 2$ for an unsuccessful search. This is often the case for edge insertions in the absence of duplicates. (ii) $\frac{1}{\alpha} ln \frac{1}{1-\alpha} = 1.39$ for a successful search (e.g., deleting existing edges). In GraphTango, we can fit eight $\{key, value\}$ pairs within a single cache line (i.e., $N = 8$). As a result, the hash table needs to access only one cache line as long as the probing distance remains ≤ 8, which provides a large slack over the theoretical average probing distances. We empirically observed the same trend with our graph datasets, where over 99.2% of the insertions had a probing distance ≤ 8. Therefore, almost all edge insertion operations for *Type3* vertices require only three cache line accesses: (i) one for retrieving $edgeMeta[srcId]$ metadata that contains hash table and adjacency list pointers, (ii) one for searching the hash table, and (iii) one for indexing to the adjacency list.

5.2 Memory Allocation Scheme

As discussed in Sect. 4, GraphTango requires frequent growing/shrinking of adjacency lists and hash tables. Calling *malloc*()/*free*() in every such instance can cause high runtime overhead and memory fragmentation. We avoid this issue by designing a fast thread-local lock-free memory pool that supports O(1) allocation and deallocation.

Figure 4 illustrates the data structure of the memory pool. This memory pool allocates chunks in power-of-two sizes. Individual linked lists of available chunks are maintained for each valid size. The heads of the linked lists are stored in the *freePtrs* array. The first 8 bytes of each chunk (highlighted green) hold the pointer to the next free chunk of the same size. This way, no extra storage beside the *freePtrs* array is required to hold the pointers. However, it limits the minimum chunk size to 8 bytes in a 64-bit machine.

Allocation steps are shown in Fig. 4a. For an allocation request of *sz* bytes, the pool will return a chunk of size $newSz = 2^k$, where $k = \lceil log_2(sz) \rceil$ (i.e., nearest power of two that is $\geq sz$). The allocation proceeds as follows: (i) Find the first available free chunk of *newSz*. This is simply given by $ret = freePtrs[k]$. (ii) If *ret* is not a null

[8] Double hashing can demonstrate performance very close to the ideal scenario of uniform hashing [22, 24].

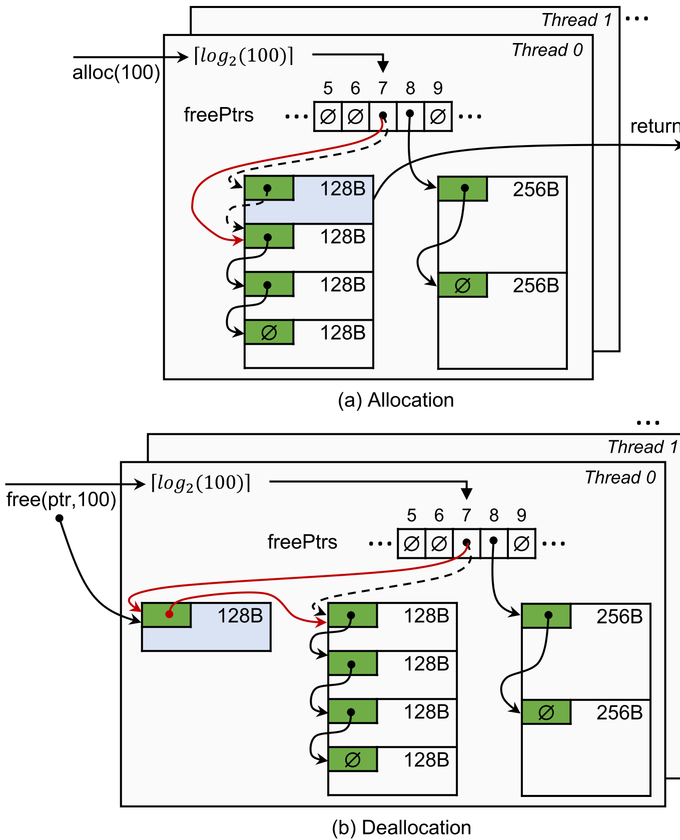

Fig. 4 Allocation and deallocation on the memory pool. Deleted pointers are shown by dashed lines and the modified pointers by red lines

pointer, then it points to a free chunk. In this case, we update *freePtrs*[*k*] to point to the next free chunk, and return *ret*. (iii) If *ret* is a null pointer, this indicates no chunk of the requested size is available. In this case, a large memory block is allocated (of size *max*(4*MB*, *newSz*) and aligned to the page boundary) and then split into a linked list of *newSz* byte chunks. *freePtrs*[*k*] is set to point to the first chunk. At this point, we have free chunks of *newSz*. Therefore, repeating step (ii) will complete the allocation. Note that, once allocated, the full chunk can be used to store data, including the space initially used to hold the pointer to the next chunk.

Deallocation steps are shown in Fig. 4b. Unlike the standard *free*(*ptr*), we provide the size of the allocated chunk as an additional parameter— *free*(*ptr*, *sz*). Using the *sz* parameter, we can directly index to the *freePtrs* array and add *ptr* as a free chunk, as shown in Fig. 4b.

An advantage of the proposed memory pool is that the most recently deallocated chunk will be allocated first, thereby being more likely to reside in the cache.

Furthermore, each thread maintains its own *freePtrs* array. As a result, no lock is required when multiple threads are trying to allocate/deallocate simultaneously. A minor downside is that one thread cannot allocate free chunks from another thread's pool. We found it to be of little consequence in practice because the maximum amount of unused space per thread is O(blockSize). Also, note that the $\lceil log_2(sz) \rceil$ calculation used to index *freePtrs* is very cheap to perform. It only requires *count leading zero* (*clz*) and shift instructions.

5.3 Parallelization

For GraphTango, we experimented with both shared and chunked style multi-threading. We decided to settle for chunked style multithreading as it demonstrated slightly better throughput on our datasets. As we are using chunked style multi-threading, operations concerning any vertices within a partition is mapped to a fixed worker thread. As discussed before, this method eliminates the need of performing atomic operations and requires minimum synchronization, but may suffer from workload imbalance. Exploring advanced load-balancing techniques is left as a potential future work. Another related concern is false sharing, which might occur when multiple threads simultaneously modify a shared data structure. In the case of GraphTango, false-sharing-prone data structures are the vertex property array and the active frontier array. We avoid false sharing by using a partition size multiple of the cache line size (e.g., 512). It ensures that no two partitions' data share the same cache line.

5.4 Determining the TH_1 Threshold

Unlike the TH_0 threshold, which is fixed for a given cache line size and edge element size, the TH_1 threshold is flexible and has a moderate impact on performance and memory usage (Sect. 6.4). As mentioned before, TH_1 should be set to a value for which $O(TH_1)$ linear search through the edge array is likely to perform better than $O(1)$ hash table lookup. The following equation provides an estimate and can be used as a rule of thumb for selecting TH_1:

$$TH_1 = 2^{\lceil log_2(3 \times edgesPerCacheLine) \rceil} \tag{1}$$

This equation sets TH_1 to a value roughly corresponding to four cache line accesses for Type2 vertices. This is slightly above the three cache line accesses of Type3 vertices, as Type2 vertices have favorable sequential access patterns and do not incur hash calculation overheads. As an alternative, we provide a microbenchmark program (*graph dataset agnostic*) with GraphTango that empirically finds a suitable TH_1 threshold.

6 Evaluation

6.1 Experimental Setup

6.1.1 Platform

The experiments are conducted on an AMD Ryzen 3900x @ 3.8GHz machine with 12 physical cores, 64MB of LLC, and 32GB of DDR4 DRAM. Hyper-threading and turbo-boost were disabled for better reproducibility. All experiments are performed with 12 cores.

6.1.2 Implementation

We evaluated GraphTango by integrating it with the SAGA-Bench [10] benchmarking framework. SAGA-Bench comes with four representation formats - AdListShared, AdListChunked, Stinger [11], and DegAwareRHH [13]. Details of these formats can be found in Section II. SAGA-Bench integration facilitates fair comparison, because all approaches must use the exact same algorithm implementations through a common API. The source code is compiled with gcc–9.3.0 and -03 flag.

Both vertex id and edge property are considered to be of 64-bits size. Therefore, GraphTango has $TH_0 = 7$ for unweighted graphs and $TH_0 = 3$ for weighted graphs. TH_1 is set to 32 following the tuning carried out in Sect. 6.4. This TH_1 value also matches the value provided by Eq. 1.

6.1.3 Profiling Methodology

Graph datasets are first randomly shuffled to break any existing ordering of edges. This is done to reflect the realistic scenario where edge updates are unlikely to occur in any pre-defined order. The shuffled dataset is inserted in batches of 1 M edges until the full graph is built and then deleted[9] in batches of 1 M edges until no edges are left to delete. This batch size is similar to prior works [10, 12]. **Analytics is performed on the graph after every batch of insertions and deletions. Reported throughputs are the geometric mean of the per-batch throughputs.** GraphTango dynamically switches between vertex types as edges are inserted/deleted. As discussed in Sect. 4, this switching may involve memory allocation/deallocation, copying, or rehashing. **This switching overhead is included in the reported results.**

6.1.4 Datasets

We have used four real-world datasets in our experiments: Orkut, LiveJournal, Wiki-topcats (referred as Wiki), and Wiki-talk (referred as Talk). Orkut and LiveJournal

[9] The vanilla SAGA-Bench does not support edge deletions. We added deletion support for all representation formats by closely following the corresponding papers or from their source code if available.

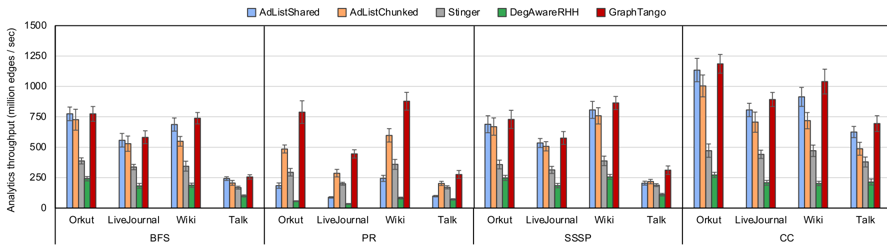

Fig. 5 Comparison of the analytics throughputs. *Higher is better*

are online social media networks, Wiki is a dataset of Wikipedia hyperlinks, and Talk is the Wikipedia communications network. These datasets are part of the SNAP dataset collection [25]. All these datasets are directed except for Orkut. Properties of these datasets are given in Table 2. Orkut and LiveJournal have a much lower per-batch maximum degree compared to Wiki and Talk. Consequently, **Orkut and LiveJournal are characterized as short-tailed graphs while Wiki and Talk are heavy-tailed graphs.**

6.1.5 Algorithms

We used four algorithms in our experiments: (i) Breath-First Search (BFS), (ii) Page Rank (PR), (iii) Single-Source Shortest Path (SSSP), and (iv) Connected Components (CC). Vertex centric incremental compute model is used for these algorithms, where the computation is constrained within the region affected by the update phase instead of the whole graph. The implementations of these algorithms are directly taken from SAGA-Bench without any modification.

6.2 Analytics and Update Performance

6.2.1 Analytics Throughput

Fig. 5 shows the analytics throughput of the representation formats. As mentioned in Sect. 6.1(C), the analytics phase is conducted multiple times as we gradually build the graph. Reported values are the geometric mean of per-batch throughputs. GraphTango outperforms other approaches in every dataset and algorithm combinations. Compared to the *next best* approach (i.e., AdListShared for BFS, SSSP, CC and AdListChunked for PR), GraphTango provides an avg (max) speedup of 1.1x (1.6x). As all these approaches are using the exact same algorithm implementation, their relative performance is primarily determined by their edge traversal efficiency. Adjacency-list-based approaches perform well in this regard, because their edge traversal consists of mostly sequential accesses. GraphTango also uses adjacency lists for medium- and high-degree vertices (Type2 and Type3). For low-degree vertices (Type1), GraphTango has a better access pattern, as it requires one less indirection (i.e., does not need pointer chasing to find the corresponding edge array), thereby offering higher throughput. Stinger, despite using coarse-grained adjacency lists,

Table 2 Evaluated Datasets

Dataset	Vertices (million)	Edges (million)	Max degree[1]		Vertex mapping[2]		
			in	out	Type1	Type2	Type3
Orkut	3.0	117.2	329	329	27.2%	38.6%	34.2%
LiveJournal	4.8	69.0	237	332	63.0%	26.0%	11.0%
Wiki	1.8	28.5	8,504	154	57.8%	33.9%	8.3%
Talk	2.4	5.0	665	20,088	98.3%	1.2%	0.5%

[1] Per-batch maximum degree with batch size of 1 million edges

[2] For $TH_0 = 7$ and $TH_1 = 32$

suffers due to additional pointer chasing between edge blocks. Overall, GraphTango provides an avg (max) speedup of 1.8x (5.1x) over AdListShared, 1.3x (1.6x) over AdListChunked, 2.0x (2.7x) over Stinger, and 5.2x (14.0x) over DegAwareRHH.

6.2.2 Update Throughput

Fig. 6 shows the update (edge insertion and deletion) throughput. Note that the updates are interleaved with analytics phases (see Sect. 6.1.C). The algorithm choice of the analytics phase has little impact on the update throughput, and the reported values are the average across the four algorithms. Here, GraphTango outperforms other approaches by a large margin. Adjacency-list-based approaches perform well on short-tailed graphs. On these graphs, GraphTango provides an avg (max) speedup of 2.5x (2.7x) over the next best approach AdListShared. On the other hand, hash-based DegAwareRHH performs best on the heavy-tailed graphs. Interestingly, AdListShared performed even worse than AdListChunked for heavy-tailed graphs. This is due to the lock contention of shared-style multithreading on AdListShared. On heavy-tailed graphs, GraphTango provides an avg (max) speedup of 6.5x (6.6x) over the next best approach DegAwareRHH. Notably, other approaches are suitable for either short- or heavy-tailed graphs. GraphTango's hybrid nature makes it consistently the best-performing irrespective of the graph's degree distribution.

6.3 Memory Usage

Table 3 shows the average memory usage per edge. The AdListShared and AdListChunked are most efficient in terms of memory usage. Compared to AdListChunked - Stinger, DegAwareRHH, and GraphTango require 5.1x, 4.1x, and 3.4x more memory on average, respectively. For DegAwareRHH, the high memory usage is caused by: (i) Sparse storage of edges in hash tables (to reduce collision), and (ii) Robin Hood hashing mechanism that requires storing the probe distance for each entry. On the other hand, Stinger and GraphTango have a relatively high initial capacity (16 for Stinger and TH_0 for GraphTango) that remains mostly unused for low-degree vertices. This scenario is especially noticeable for the Talk

dataset, where more than 96% of the vertices have a degree ≤ 3, leading to high memory usage. Although GraphTango has higher memory usage compared to the simple adjacency-list-based approaches, the update throughput benefit is significant, especially on heavy-tailed graphs (19.3x to 32.8x speedup on Wiki and Talk datasets). If needed, one way to reduce memory usage of GraphTango is to increase the TH_1 threshold, as discussed in Sect. 6.4. Compared to Stinger and DegAwareRHH, GraphTango requires less memory as well as provides much higher performance.

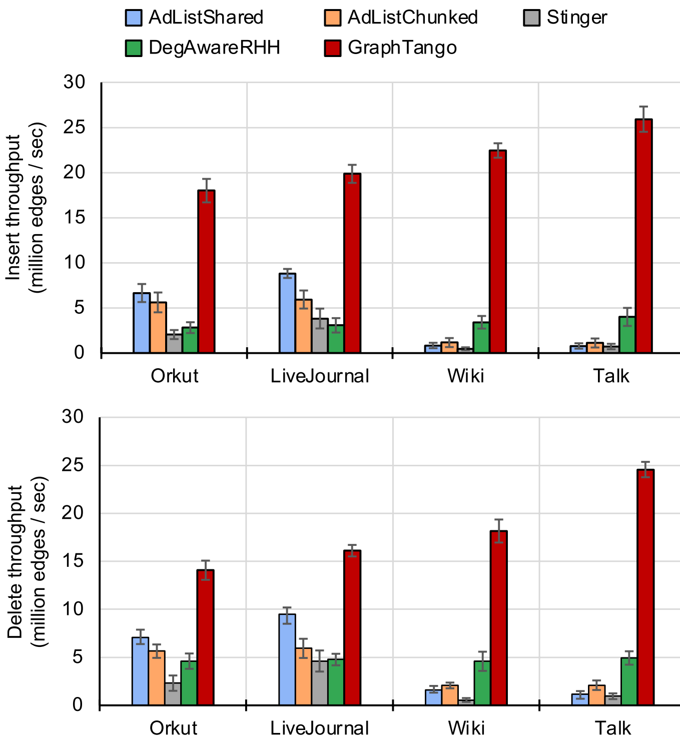

Fig. 6 Comparison of update throughputs. *Higher is better*

Table 3 Average memory usage (Bytes Per Edge)

Dataset	AdList-Shared	AdList-Chunked	Stinger	DegAware RHH	GraphTango
Orkut	13.3	12.0	32.7	43.8	33.6
LiveJournal	16.3	12.9	48.9	57.0	34.6
Wiki	15.7	12.7	44.1	62.9	34.4
Talk	44.8	21.9	230.2	74.6	116.9

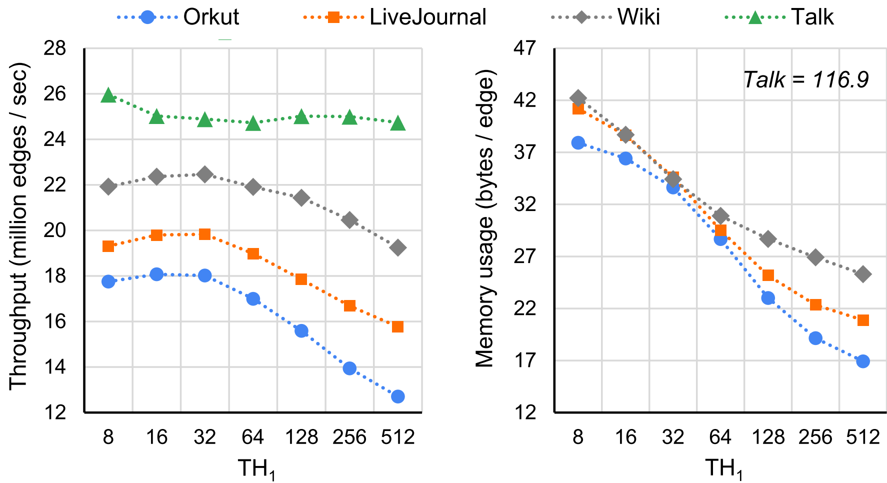

Fig. 7 Impact of TH_1 threshold on update throughput and memory usage

Table 4 Impact of Optimizations on the Update Throughput (Baseline is the proposed hybrid format without any optimizations applied)

Configuration	Format	Allocation Scheme	Hashing Scheme	Speedup STail	Speedup HTail
baseline	Hybrid	malloc[2]	std_map[3]	1.00	1.00
next best[1]	–	–	–	0.76	0.29
opt pool	Hybrid	proposed	std_map[3]	1.12	1.14
opt hash	Hybrid	malloc[2]	proposed	1.71	1.70
GT_Tessil	Hybrid	proposed	Tessil[4]	1.40	1.60
GT_RHH	Hybrid	proposed	RHH[5]	1.35	1.45
GT_Abseil	Hybrid	proposed	Abseil[6]	1.34	1.43
GraphTango	*Hybrid*	*Proposed*	*Proposed*	**1.89**	**1.79**
DegAwareRHH	DegAware	malloc[2]	RHH	0.29	0.29
DegAwareCFH	DegAware	malloc[2]	proposed	0.40	0.38

Bold vlaues corresponds to our proposed approach

[1] AdListShared for STail and DegAwareRHH for HTail

[2] glibc version 2.31

[3] std::unordered_map with libstdc++ version 6.0.28

[4] tsl::robin_map from [26], version 1.0.1

[5] Robin Hood hashing implementation from [27], version 3.11.5

[6] Google's Abseil flat_hash_map version LTS 20211102 [28]

6.4 Impact of TH_1 Threshold

Figure 7 shows the impact of the TH_1 threshold on update throughput and memory usage. Analytics throughput is not shown because the choice of TH_1 does not impact

the analytics performance. TH_1 of 16 and 32 provides the best throughput on three out of the four datasets. We used TH_1 of 32 in all other experiments, as it has lower memory usage.

The TH_1 threshold controls the ratio between Type2 and Type3 vertices. Increasing TH_1 maps more higher-degree vertices to Type2 instead of Type3. As Type2 vertices requires linear search during updates, it eventually becomes a performance bottleneck for $TH_1 > 32$. On the other hand, Type2 vertices do not require maintaining a hash table and thus require less memory than Type3. As a result, increasing TH_1 reduces the memory usage. On average, increasing TH_1 from 8 to 512 reduces the memory usage by 1.9x.

The Talk dataset is an outlier showing negligible variation with TH_1. This is because, for the Talk dataset, 98.3% of the vertices are mapped to Type1 (refer to Table 2), leaving only 1.7% of the vertices that can be affected by changing TH_1.

6.5 Impact of Optimizations

The purpose of this section is to isolate the contribution of the proposed hybrid format as well as the memory allocation and hashing scheme optimizations. Table 4 shows our findings. In this table, the STail and HTail columns show the normalized update throughput over the *baseline* configuration for short-tailed and heavy-tailed graphs, respectively. We show only the update throughput because the memory pool and hashing optimizations have a negligible impact on the analytics throughput.

The *baseline* configuration implements our proposed hybrid format (i.e., Type1, Type2, and Type3 mapping of vertices) but uses sub-optimal malloc for memory allocation and std::unordered_map for hashing. We can observe that using only the hybrid format is sufficient to provide a better performance than the *next best* approach (AdListShared for STail and DegAwareRHH for HTail). Compared to the next best approach, the *baseline* offers 1.3x (3.4x) higher speedup for STail (HTail) graphs. The *opt pool* configuration shows the benefit of the proposed memory allocation scheme. On average, the proposed pool provides 1.13x better performance over the *baseline*. On the other hand, the *opt hash* configuration shows the benefit of the proposed cache-friendly hashing scheme, offering 1.71x speedup over std::unordered_map. With both optimizations enabled, GraphTango provides an average speedup of 1.84x over the *baseline*.

A valid concern at this point is whether we can combine other hashing schemes with our hybrid format to get even better performance. To answer this question, we tried three other open-addressing-based hash-table implementations: (i) *GT_Tessil*: The Robin Hood hashing variation of Tessil (tsl::robin_map) [26]. This is the fastest hash table implementation according to the benchmark results published in [29]. (ii) *GT_RHH*: Another fast implementation of Robin Hood hashing from [27, 29]. Although it is slightly slower than Tessil, it consumes significantly less memory. (iii) *GT_Abseil*: Google's Abseil flat_hash_map [28, 30]. The max load factors of these approaches are set equal to ours (= 0.5) for a fair comparison. On STail (HTail) graphs, the proposed hashing scheme provides 1.35x (1.12x) speedup over

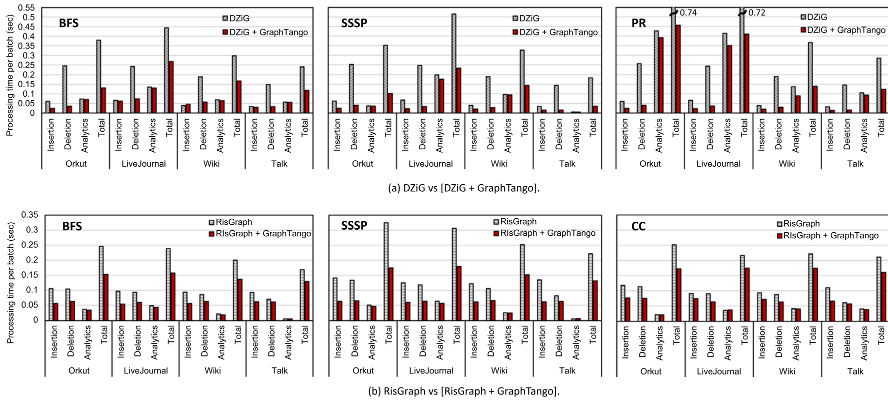

Fig. 8 Batch processing time breakdown of DZiG and RisGraph integration. *Lower is better*

GT_Tessil, 1.4x (1.23x) speedup over *GT_RHH* and 1.41x (1.25x) speedup over *GT_Abseil*. Because our hashing scheme tries to minimize cache line access, it is especially suitable for graph workloads where the hash table is unlikely to reside in the cache.

Finally, we evaluate whether DegAwareRHH can leverage our hashing scheme to outperform GraphTango. *DegAwareCFH* denotes this configuration. *DegAwareCFH* provides 1.37x (1.31x) better throughput over the vanilla DegAwareRHH. However, GraphTango still outperforms it by 4.7x for both STail and HTail graphs.

6.6 Integration with DZiG and RisGraph

This section demonstrates that full-fledged graph processing frameworks can leverage the GraphTango format to improve their performance further. We selected two state-of-the-art graph processing frameworks DZiG [15] and RisGraph [16] for this purpose. We modified their publicly available source code [31, 32] and replaced their storage format with GraphTango. We run the datasets on BFS, PR, and SSSP for DZiG. CC is omitted because its implementation is unavailable in the framework's repository. For the same reason, PR is omitted in case of RisGraph.

Figure 8a shows the comparison results between DZiG and DZiG+GraphTango. DZiG internally uses adjacency list as graph storage. For this reason, analytics time for DZiG and DZiG+GraphTango is similar in most cases. Interestingly, the insertion time is also comparable in some cases. For example, LiveJournal and Wiki datasets for BFS. This is because the original DZiG's edge insertion does not check for duplicate edges. Therefore, the edge insertion becomes as simple as adding an element to the end position of an array.[10] On average, GraphTango provides a 1.9x reduction in insertion time *even though it also checks for duplicate edges*. For deletion, unmodified

[10] There is a flag to enable duplicate edge insertion checking. But that checking is done by sorting the batch as a pre-processing step, thereby incurring heavy overhead.

DZiG performs 6x worse on average. We identified two reasons: (i) Unlike insertions in DZiG that do not search for duplicates, delete operations require a linear search through the neighbor list, incurring higher runtime cost, and (ii) DZiG performs a quicksort on the batch based on the source and destination vertex ids to distribute them among the threads. As we use fixed mapping of vertices in GraphTango, sorting costs are avoided. Overall, DZiG+GraphTango provides an average of 2.3x reduction in total batch processing time compared to the original DZiG.

Figure 8b shows the comparison between RisGraph and RisGraph+GraphTango. RisGraph uses a hybrid graph storage format that uses adjacency list for low/medium degree vertices and adjacency list along with hash table for high degree vertices. Unlike GraphTango, RisGraph does not differentiate between low and medium degree vertices and uses the same data structure for both. Furthermore, RisGraph uses Google's dense hash map and does not attempt to minimize the number of cache accesses as GraphTango does with its proposed cache-friendly hashing scheme. Due to these differences, RisGraph+GraphTango provides on average 1.5x reduction in total batch processing time compared to the vanilla RisGraph.

7 Conclusions

Existing streaming graph representation formats can only support either short-tailed or heavy-tailed workloads efficiently. This paper proposes GraphTango, which aims to solve this issue by adaptively switching formats based on the current degree of a vertex. We also propose a cache-efficient hashing scheme and a fast memory pool. These optimizations work in synergy with GraphTango to provide excellent update and analytics throughput regardless of the graph's degree distribution. Our evaluation on the SAGA-Bench showed that on average (maximum), GraphTango provides 4.5x (6.6x) higher insertion throughput, 3.2x (5.0x) higher deletion throughput, and 1.1x (1.6x) higher analytics throughput over the *next best* approach.

Appendix A Hash Function Implementation

Given these parameters,

M = Number of cache lines in the hash table

N = Number of $\{key, value\}$ pairs within a cache line

Our proposed hash function is of the following form:

$$h(key, i) = N \cdot h_1\left(key, \left\lfloor \frac{i}{N} \right\rfloor\right) + h_2(key, i \bmod N)$$

Here, $h_1()$ selects a cache line inside the hash table array, and $h_2()$ selects an offset within the cache line. Therefore, $h_1()$ must be a permutation of $\{0, 1, ..., M-1\}$ to ensure that all cache lines are eventually selected. Similarly, $h_2()$ must be a permutation of $\{0, 1, ..., N-1\}$ to explore all $\{key, value\}$ pairs within a cache line. Any $h_1()$

and $h_2()$ that meet the permutation requirement can be used. For GraphTango, we used the following:

$$h_1(k,x) = (h_3(k) + x \cdot h_4(k))\ mod\ M$$
$$h_2(k,x) = (k + x)\ mod\ N$$
$$h_3(k) = \lfloor (A \cdot k\ mod\ 2^w)/2^{w-m} \rfloor$$
$$h_4(k) = \lfloor (A \cdot k\ mod\ 2^w)/2^{w-2m} \rfloor\ or\ 1$$

Here, w is the key width in bits, A is a large constant, and $m = log_2(M)$. We use double hashing for $h_1()$ to negate primary/secondary clustering. It is computed with the help of two pairwise independent hashing functions, $h_3()$ and $h_4()$. $h_3()$ and $h_4()$ are computed with multiplicative hashing. As for $h_2()$, we used simple linear probing. Although seemingly complex, the hash can be computed cheaply as we ensure both N and M are powers of two. The following code snippet shows how to calculate the hash value for a 32-bit key:

```
u32 h(u32 key, u32 i){
    u32 y = key * A;
    u32 h3 = y >> (32 - logM);
    u32 h4 = (y >> (32 - (logM << 1))) | 1;
    u32 h1 = (h3 + (i >> logN) * h4) & (M - 1);
    u32 h2 = (key + i) & (N - 1);
    return (h1 << logN) + h2;
}
```

Note that the code does not need any expensive division/modulus operation. When compiled on an x86_64 machine with gcc 9.3.0 and -O3 flag, it resulted in 2 multiplications and 8 other simple arithmetic/logical instructions.

Acknowledgements This work was funded in part by grant 19-1979 from Booz Allen Hamilton and by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

Author Contributions A. Ahmed proposed the initial ideas presented in the manuscript. F. A. Siddique and K. Skadron proposed fundamental improvements over the initial idea. The manuscript text is written by A. Ahmed and F. A. Siddique. All authors reviewed the manuscript.

Declarations

Conflict of interest All (U. Virginia)—plus, Abraham, Jacob (U. Texas), Akel, Ameen (Micron Technologies), Angstadt, Kevin (St. Lawrence U.), Aly, Mohamed El-Hadedy (California Polytechnic), Ceze, Luis (University of Washington), Cheng, Eric (Laboratory for Physical Sciences), Cher, Chen-Yong (Graphen), Cho, Hyungmin (Sungkyunkwan U.), Chung, Sung Woo (Korea University), Clark, Doug (Princeton, retired), Cong, Jason (University of California, Los Angeles), Dickerson, Samuel (University of Pittsburgh), Eilert, Sean (independent), Eliceiri, Kevin (University of Wisconsin), Fazeli, Mahdi (Halmsted University), Gai, Yan (St. Louis University),Gao, Wei (University of Pittsburgh), Gaur, Jayesh (Intel), Gavrilovska, Ada (Georgia Tech), Guo, Xinfei (Shanghai Jiao Tong U.), Hoe, James (CMU), Huang, Hang (University of Maryland), Hwu, Wen-Mei (UIUC), Imani, Mohsen (UC Irvine), Jun, Sang-Woo (UC Irvine), Knight, Rob (University of California, San Diego), Kozyrakis, Christos (Stanford University), Li, Jing (UPenn), Lila, Klas (Robust Chip), Marino, Mario (Leeds Beckett University), Martinez, Jose (Cornell University), Martonosi, Margaret (Princeton), McDaniel, Patrick (Penn State U),

Meyer, Brett (McGill), Mirkhani, Shahrzad (U. Texas), Moshiri, Niema (UCSD), Narayanan, Vijay (Penn State U), Orenstein, Yaron (Bar Ilan U), Page, Brian (LPS), Parkhurst, Jeff (Intel), Patel, Jignesh (Wisconsin University), Pop, Eric (Stanford), Qureshi, Moin (Georgia Tech), Raina, Priyanka (Stanford), Rosing, Tajana (University of California, San Diego), Sadredini, Elaheh (UC Riverside), Salahuddin, Sayeef (Berkeley), Sampson, Adrian (Cornell), Sheaffer, Jeremy (Iowa State), Sivasubramaniam, Anand (Penn State U), Strukov, Dimitri (University of California, Santa Barbara), Subrramaniyan, Arun (Michigan), Subramoney, Sreenivas (Intel), Sun, Yizhou (UCKA), Swift, Michael (Wisconsin), Swanson, Steven (University of California, San Diego), Tabajara, Lucas (Rice), Vardi, Moshe (Rice), Wadden, John “Jack” (University of Michigan), Weimer, Westley (University of Michigan), Witchell, Emmet (U. Texas), Wong, Philip (Stanford), Xie, Yuan (UCSB), Yu, Shimeng (Georgia Tech), Zhang, Yiying (UCSD), Zhang, Zhiru (Cornell University), Zhao, Jishen (University of California, San Diego), Zhou, Peipei (U. Pittsburgh), Zhou, Yuanyuan (University of California, San Diego), Zhu, Song-Chun (University of California, Los Angeles)

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Han, W., et al.: Chronos: a graph engine for temporal graph analysis. In: EUROSYS, pp. 1–14 (2014)
2. Cheng, R., et al.: Kineograph: taking the pulse of a fast-changing and connected world. In: EUROSYS, pp 85–98 (2012)
3. Compeau, P.E.C., et al.: How to apply de bruijn graphs to genome assembly. Nat. Biotechnol. **29**, 987–991 (2011)
4. Zerbino, D.R., et al.: Velvet: algorithms for de novo short read assembly using de bruijn graphs. Genome Res. **18**, 821–829 (2008)
5. Grewal, A., et al.: Recservice: distributed real-time graph processing at twitter. In: HotCloud (2018)
6. Eksombatchai, C., et al.: Pixie: a system for recommending 3+ billion items to 200+ million users in real-time. In: WWW, pp. 1775–1784 (2018)
7. Park, J., Nahrstedt, K.: navigation graph for tiled media streaming. In: ICME, pp. 447–455 (2019)
8. Braun, P., et al.: Knowledge discovery from social graph data. Procedia Comput. Sci. **96**, 682–691 (2016)
9. Borgman, C.L., et al.: Drowning in data: digital library architecture to support scientific use of embedded sensor networks. In: JCDL, pp. 269–277 (2007)
10. Basak, A., et al.: Saga-bench: software and hardware characterization of streaming graph analytics workloads. In: ISPASS, pp. 12–23 (2020)
11. Ediger, D., et al.: Stinger: high performance data structure for streaming graphs. In: HPEC (2012)
12. Jaiyeoba, W., Skadron, K.: Graphtinker: a high performance data structure for dynamic graph processing. In: IPDPS, pp. 1030–1041 (2019)
13. Iwabuchi, K., et al.: Towards a distributed large-scale dynamic graph data store. In: IPDPSW, pp. 892–901 (2016)
14. McCrabb, A., Bertacco, V.: Optimizing vertex pressure dynamic graph partitioning in many-core systems. IEEE Trans. Comput. **70**, 936–949 (2021)
15. Mariappan, M., et al.: Dzig: sparsity-aware incremental processing of streaming graphs. In: EUROSYS, pp. 83–98 (2021)
16. Feng, G., et al.: Risgraph: a real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s. In: SIGMOD, pp. 513–527 (2021)

17. Hu, Y., et al.: Graphlily: accelerating graph linear algebra on hbm-equipped fpgas. In: ICCAD, pp. 1–9 (2021)
18. Ham, T.J. et al.: Graphicionado: a high-performance and energy-efficient accelerator for graph analytics. In MICRO, pp. 1–13 (2016)
19. Sundaram, N., et al.: Graphmat: high performance graph analytics made productive. arXiv:1503.07241 (2015)
20. Gui, C.Y., et al.: A survey on graph processing accelerators: challenges and opportunities. JCS &T **34**, 339–371 (2019)
21. Celis, P., et al.: Robin hood hashing. In: SFCS, pp. 281–288 (1985)
22. Cormen, et al.: Introduction to algorithms. MIT press, Cambridge (2009)
23. ISO/IEC JTC 1/SC 22 technical committee. C++ standard. https://www.iso.org/standard/79358.html (2020)
24. Knuth, D.E.: The art of computer programming. Addison-Westley Publishing, Reading, MA (1973)
25. Leskovec, J., Krevl, A.: Snap datasets: stanford large network dataset collection. https://snap.stanford.edu/data/ (2014)
26. Planchon T.: Tessil github repository. https://github.com/Tessil/robin-map (2022)
27. Ankerl M.: Robin hood hashing github repository. https://github.com/martinus/robin-hood-hashing (2022)
28. Google: Abseil github repository. https://github.com/abseil/abseil-cpp (2022)
29. Ankerl M.: Hashmap benchmarks. https://martin.ankerl.com/2019/04/01/hashmap-benchmarks-01-overview/ (2019)
30. Matt K.: Designing a fast, efficient, cache-friendly hash table, step by step. CPPcon. Standard C++ Foundation (2017)
31. Mariappan, M., et al.: Dzig: sparsity-aware incremental processing of streaming graphs. https://github.com/pdclab/graphbolt/tree/eurosys21-artifact (2021)
32. Feng et al.: Risgraph github repository. https://github.com/thu-pacman/RisGraph (2021)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Authors and Affiliations

Alif Ahmed[1] · Farzana Ahmed Siddique[1] · Kevin Skadron[1]

✉ Alif Ahmed
alifahmed@virginia.edu

Farzana Ahmed Siddique
farzana@virginia.edu

Kevin Skadron
skadron@virginia.edu

[1] University of Virginia, Charlottesville, USA