CuckooGraph: A Scalable and Space-Time Efficient Data Structure for Large-Scale Dynamic Graphs

Zhuochen Fan*†, Yalun Cai†, Zirui Liu†, Jiarui Guo†, Xin Fan‡, Tong Yang†, Bin Cui†
*Pengcheng Laboratory, China †School of Computer Science, Peking University, China ‡Wuhan University, China
{fanzc,caiyalun,zirui.liu,ntguojiarui,yangtong,bin.cui}@pku.edu.cn, xin.fan@whu.edu.cn

***Abstract*—Graphs play an increasingly important role in various big data applications. However, existing graph data structures cannot simultaneously address the performance bottlenecks caused by the dynamic updates, large scale, and high query complexity of current graphs. This paper proposes a novel data structure for large-scale dynamic graphs called CuckooGraph. It does not require any prior knowledge of the upcoming graphs, and can adaptively resize to the most memory-efficient form while requiring few memory accesses for very fast graph data processing. The key techniques of CuckooGraph include TRANSFORMATION and DENYLIST. TRANSFORMATION fully utilizes the limited memory by designing related data structures that allow flexible space transformations to smoothly expand/tighten the required space depending on the number of incoming items. DENYLIST efficiently handles item insertion failures and further improves processing speed. Our experimental results show that compared with the most competitive solution Spruce, CuckooGraph achieves about $33\times$ higher insertion throughput while requiring only about 68% of the memory space.**

I. INTRODUCTION

A. Background and Motivation

Graphs can intuitively represent various relationships between entities and are widely used in various big data applications, such as user behavior analysis in social/e-commerce networks [1]–[3], financial fraud detection in transactional systems [4]–[6], network security and monitoring in the Internet [7]–[9], and even trajectory tracking of close contacts of the COVID-19 epidemic [10], [11], *etc.* Correspondingly, graph analytics systems also play an increasingly significant role, responsible for storing, processing, and analyzing graph-like data well.

As an essential part of graph analytics systems, graph storage schemes are facing challenges introduced mainly by the following properties of graph data: ① Fast update: graphs always arrive quickly and are constantly dynamic [12]–[14]. This requires the storage structure to be updated at high speed. ② Large data scale: graphs can even reach hundreds of millions of edges [15]–[17]. This requires the storage structure to be flexibly adapted to the data scale. ③ High query complexity: graphs have complex topologies, and their node degrees are often unevenly distributed and follow a power-law distribution [18]–[20]. This means that graphs usually consist of mostly low-degree nodes and a few high-degree nodes. Querying the neighbors of high-degree nodes takes longer, while low-degree nodes take less time. However, the former is more likely to be queried and updated than the latter. This imbalance leads to poor query performance and hinders further optimization. In summary, an ideal graph storage scheme can achieve memory-saving, fast processing speed, and good update and expansion performance to deal with any unknown graphs.

Currently, most existing graph storage and database schemes [21]–[41] use the following two as basic data structures: adjacency list and compressed sparse row (CSR), but neither of them directly supports large-scale dynamic graphs. The most widely used adjacency list represents node connections intuitively and is easy to edit (such as adding or removing edges). However, due to its non-contiguous memory allocation and inefficiency in accessing non-neighbor edges, this pointer-intensive data structure is prone to significant space-time overhead as the graph size grows. The CSR provides a more compact array-based representation that is more memory efficient and suitable for fast traversal. However, the CSR is inherently static and struggles with updates for dynamic graphs, as its update usually requires completely rebuilding the CSR structure, which is computationally expensive and inefficient. In order to accommodate large-scale dynamic graphs, the data structures of many state-of-the-art graph storage schemes are evolved from adjacency lists or CSRs. Unfortunately, they cannot completely avoid the above-mentioned shortcomings of the adjacency list or CSR itself, so that they cannot solve the above ①②③ at the same time or have various limitations, and there is still room for improvement. For example, Spruce [36], the most competitive solution whose basic data structure is based on adjacency lists, still needs to record many pointers.

B. Our Proposed Solution

In this paper, we propose a novel data structure for storing large-scale dynamic graphs, namely **CuckooGraph**. It has the following advantages: 1) It is memory-saving and can be flexibly expanded or contracted according to actual operations; 2) It basically maintains the fastest running speed in a series of graph analytics tasks; 3) It works for any graph of unknown size without knowing any information about it in advance.

The design philosophy of CuckooGraph is as follows. Instead of using the traditional adjacency list or CSR, we choose a hash-array-based data structure to improve time and space efficiency when handling large-scale dynamic graphs.

Co-first authors: Zhuochen Fan, Yalun Cai, and Zirui Liu. Corresponding author: Tong Yang (yangtong@pku.edu.cn).

Specifically, we utilize a (large) cuckoo hash table (L-CHT) [42] as the basic data structure with a finer-grained partitioning of the space in each bucket. We assume that the edge $\langle u, v\rangle$ is is mapped and will be stored in this bucket. Initially, part of the bucket space is used to store node u, and the other part, which is divided into an even number of small slots, is used to store node(s) v. Then, L-CHT decides whether to perform our TRANSFORMATION technique based on the degree (the number of incoming v) of the u: 1) When the node degree is small: this sparsity is more consistent with most graphs in reality, then we sequentially store the incoming v into the small slots. 2) When the node degree exceeds the specified number of small slots: these small slots merged in pairs to form several large slots with one of them deposited into the first pointer to the first (small) cuckoo hash table (S-CHT) that has just been activated, and all v is transferred into that large-capacity S-CHT to accommodate more incoming v. 3) When the node degree is even larger: S-CHT is incremented with some regularity, to cope with the large increase of v; and of course, it can also be decremented with some regularity to handle v deletions. In short, our TRANSFORMATION smoothly expands or shrinks the space through a series of spatial transformations to adapt to the increase or decrease of v, while ensuring that few accesses are required even in the worst case. It greatly reduces the number of pointers and makes full use of limited memory space while ensuring speed.

Although we seem to fully guarantee time and space efficiency through the well-designed L/S-CHT, the shortcomings of cuckoo hashing itself have not yet been addressed. As the memory space becomes tight due to the increase in incoming items, item replacement caused by hash collisions may occur frequently and may cause insertion failures. On the one hand, not handling insertion failures may result in CuckooGraph no longer being error-free, which is unacceptable; on the other hand, we can also address it by directly expanding CuckooGraph every time an insertion failure occurs, but this may make it slower. Therefore, we further propose the DENYLIST optimization, which aims to cooperate with the TRANSFORMATION technique to efficiently accommodate those items that fail to be inserted. Our ablation experiments have verified that this optimization can improve insertion and query throughput with almost no additional memory overhead. For more details on CuckooGraph's TRANSFORMATION technique and DENYLIST optimization, please refer to § III-A1 and § III-A2, respectively. Further, we also propose an extended version of CuckooGraph for streaming scenarios to support duplicate edges in § III-B.

The rest of this paper is organized as follows. We theoretically prove that the time and memory cost of CuckooGraph is desirable through mathematical analysis in § IV. We conduct extensive experiments on 7 large-scale graph datasets with different characteristics to evaluate the performance of CuckooGraph on basic tasks and graph analytics tasks. The results clearly show that CuckooGraph has the fastest speed and the lowest memory overhead on almost all tasks. Finally, CuckooGraph is integrated into Redis and Neo4j databases as use cases. See § V for more details. All relevant source code is already open source on Github [43].

Main Experimental Results: 1) For basic tasks (§ V-D), the insertion and query throughput of CuckooGraph is on average $32.66\times$ and $133.62\times$ faster than those of Spruce, respectively, while its memory usage is on average $1.47\times$ less than that of Spruce ($i.e.$, only 68.03% of its); 2) For graph analytics tasks (§ V-E), the running time of CuckooGraph on 7 typical tasks (Breadth-First Search, Single-Source Shortest Paths, Triangle Counting, Connected Components, PageRank, Betweenness Centrality, and Local Clustering Coefficient) is on average $0.73\times$, $168.45\times$, $21.33\times$, $1.07\times$, $1.03\times$, $16.17\times$, and $5.80\times$ faster than those of Spruce, respectively.

II. Related Work

In this section, we introduce graph storage schemes whose main contribution lies in data structure design and Cuckoo Hash Table (CHT)—the most basic data structure of CuckooGraph, in § II-A and § II-C, respectively.

A. Existing Solutions

There are many existing works with the concept of dynamic graph storage, only some of which focus on optimizing graph updates (insertion, deletion, and attribute change, $etc.$) at the algorithm level. Most of them have data structures based on or improved on adjacency lists or CSRs. Of course, there are also other or hybrid ones. We only select representatives to introduce for the sake of space.

Adjacency list-based: GraphOne [29] uses a complementary combination of adjacency lists and edge log lists. The adjacency list stores snapshots of old data, while the edge log records the latest updates, which periodically transfers data into the adjacency list in batches. LiveGraph [30] uses the proposed Transactional Edge Log (TEL) and Vertex Blocks (VB) to store edge information and nodes, respectively. In TEL, edge insertions and updates are performed in the form of log entries in a specified order. RisGraph [31] uses the proposed indexed adjacency lists and sparse arrays. The indexed adjacency list is a dynamic array including arrays and edge indexes for continuous storage, while the sparse array is used for updates to avoid unnecessary data access. Sortledton [34] uses a customized adjacency list, including the expandable adjacency index and adjacency set, to store the mapping from nodes to edge sets and the neighbors of each node, respectively. Wind-Bell Index (WBI) [35] consists of an adjacency matrix and many adjacency lists, where each bucket of the matrix is associated with an adjacency list through a pointer. It selects the shortest hanging list through multiple hashes to address the slow query caused by node degree imbalance that is not considered in existing graph databases. Spruce [36] consists of an edge-storage part and a node-indexing part similar to the vEB tree. The edge-storage part is based on the adjacency list and is used to store the edges as well as attributes. The node-indexing part includes a hash table and a bit vector. It is used to record node identifiers and map nodes to their connected edges. Also, it divides the

TABLE I: Symbols used in this paper.

Notation	Meaning
$\langle u, v\rangle$	A distinct graph item
L-CHT	The large cuckoo hash table
$H_1(.)$, $H_2(.)$	Two hash functions associated with L-CHT
S-CHT	The small cuckoo hash table
$h_1(.)$, $h_2(.)$	Two hash functions associated with S-CHT(s)
n	The length of 1st S-CHT
R	The number of large slots in Part 2 of each cell
LR	The loading rate
G	The preset LR threshold for expansion
Λ	The preset overall LR threshold for contraction
L-DL	Denylist for L-CHT(s)
S-DL	Denylist for S-CHT(s)
T	Maximum number of loops in L/S-CHT
w	Weight, or number of times $\langle u, v\rangle$ is repeated

8-byte identifier into 4, 2, 2, where 4 is stored in the hash table to share the same hash address, and two 2s are stored in the bit vector associated with the edge storage part. In this way, Spruce achieves low memory consumption and efficient dynamic operations, but it still needs to record quite a few pointers to any graph.

CSR-based: To address the problem that CSR is not suitable for dynamic graphs, PCSR [26] replaces the array storing neighbors in CSR with the packed memory array (PMA)[1] [44], which essentially maintains an implicit complete binary search tree. To avoid frequent rebalance of PCSR, VCSR [33] maintains PMA by reserving empty slots between nodes in proportion to the current node degree. Teseo [45] mainly uses PMAs as leaf nodes of improved B+ trees for graph updates, but only supports undirected graphs. Each PMA is divided into several expandable segments, and a hash table maps nodes to locations within the segment.

Other-based: Terrace [13] decides which data structure to use for storage based on the node degree: 1) Nodes with few neighbors are stored in an sorted array; 2) Nodes with a medium number of neighbors are stored in a PMA; 3) Nodes with many neighbors are stored in a B+ tree. However, its biggest drawback is that the number of nodes for the workload must be known in advance.

B. Orthogonal Work

Recently, there are notable works such as VEND [46] that aim to accelerate edge queries by filtering no-result edges, which are orthogonal to graph storage/databases. Since most node pairs in the real world have only no-edge connections, VEND introduces a novel data structure for nodes to store redundant neighbor information based on range and hash solutions. We leave the possibility of applying VEND to CuckooGraph as future work.

C. Cuckoo Hashing

Cuckoo Hashing (or Cuckoo Hash Table, CHT) [42] contains two hash tables, each associated with a hash function. Each newly inserted item is mapped to two candidate buckets (one in each table), and one of the two buckets will be selected to store it. If at least one of the two candidate buckets is empty, CHT stores the item in an empty bucket. If both candidate buckets are full, CHT randomly selects one of the stored items to kick out, and the kicked out item will be re-inserted into its another candidate bucket. This process is repeated until each item finds a bucket to settle down, or reaches the maximum kick-out threshold and has to exit, which means that there is an item insertion failure. To query an item, CHT only needs to directly check the two candidate buckets through two hash functions. Therefore, CHT has a high loading rate and $O(1)$ query time complexity. However, in the worst case, item insertions take a lot of time while still failing as described above, especially when CHT has a high loading rate.

III. CuckooGraph Design

In this section, we present the basic version of CuckooGraph that does not support duplicate edges in § III-A and the extended version of CuckooGraph that supports duplicate edges in § III-B, respectively. The symbols (including abbreviations) frequently used in this paper are shown in Table I.

A. Basic Version

*1) **Foundation Stage (Transformable Data Structures)**:*

As shown in Figure 1 (Foundation Stage), the basic data structure of CuckooGraph consists of one (or more) large cuckoo hash table(s) (denoted as L-CHT(s)) and many small cuckoo hash tables (denoted as S-CHTs) associated with it/them, all of which are specially designed.

L-CHT has two bucket arrays denoted as B_1 and B_2, respectively, associated with two independent hash functions $H_1(.)$ and $H_2(.)$, respectively[2]. Each bucket has d cells, each of which is designed to have two parts: Part 1 and Part 2. For any arriving graph item $\langle u, v\rangle$, assuming it is mapped to bucket B_H and deposited in the cell C, then: Part 1 of C is used to store u, while Part 2 of C is directly used to store v **or** the pointer(s) pointing to S-CHT which is used to actually store v. In Foundation Stage (red box), Part 2 is designed as a structure that can be flexibly transformed in a manner determined by *the number of* v (denoted as l) corresponding to u in Part 1, as shown below: ① Part 2 is initialized to $2R$ small slots, $i.e.$, up to $2R$ v can be recorded, to handle the situation where $l \leq 2R$; ② When $l > 2R$, $2R$ small slots are merged in pairs to form R large slots dedicated to storing R pointers usually with more bytes, and 1st large slot is deposited with a pointer that points to 1st S-CHT; Then, all the current v are stored into this S-CHT of length[3] n.

Next, we proceed to describe the transformation strategies of S-CHT and L-CHT to efficiently cope with the increasing l as follows: 1) If the growing l causes the loading rate (LR)

[1]PMA is a dynamic array used to maintain an ordered collection of items. It balances item insertion and deletion operations by interspersing empty slots within the array to optimize access and modification performance.

[2]The basic structure of S-CHT is no different from that of L-CHT, except for the length and the association with two other independent hash functions $h_1(.)$ and $h_2(.)$.

[3]We define the length of CHT as the number of buckets in the array with more buckets.

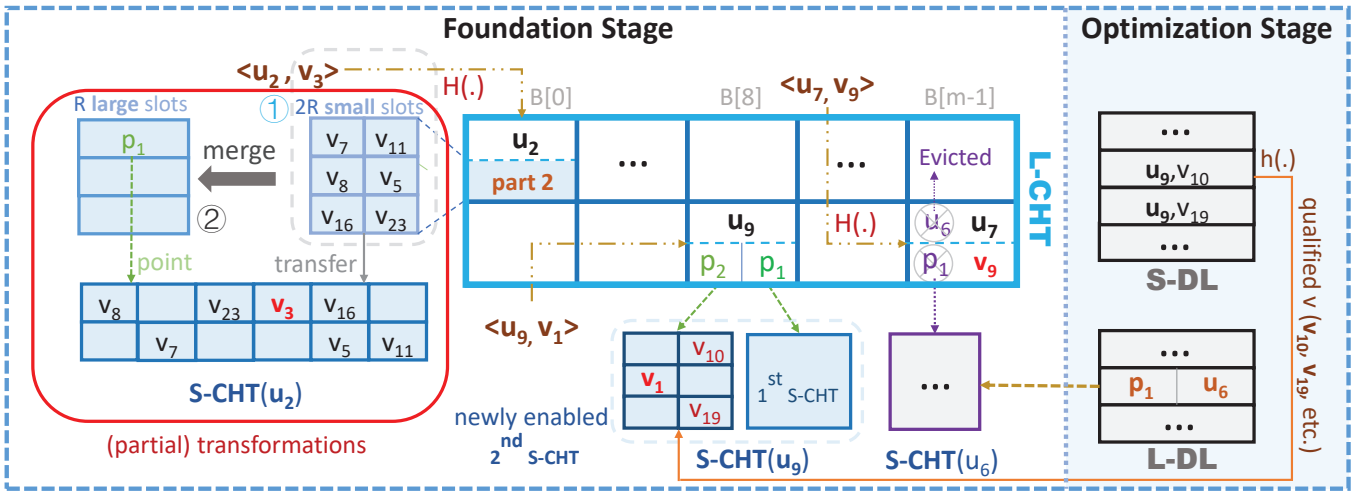


Fig. 1: Data structure and examples of CuckooGraph. For clarity, only one hash table per L/S-CHT is shown.

of the 1st S-CHT to reach the preset threshold G before the current v arrives, we enable the 2nd pointer and store it in the 2nd large slot, as well as simultaneously enables the 2nd S-CHT; 2) By analogy, when LR of the $(R-1)$-th S-CHT also reaches G, we continue to enable the R-th pointer and S-CHT similarly. Here, we allocate the length of each newly enabled S-CHT in a memory-efficient manner, which is specifically related to the R value. We illustrate the transformation rule of length with $R = 3$, when there are at most 3 S-CHTs, as shown in Table II. When $0 \rightarrow 1$ and $1 \rightarrow 2$ occur, the 2nd and 3rd S-CHTs with a length of $0.5n$ are enabled in turn; When $2 \rightarrow 3$ occurs, the 1st, 2nd and 3rd S-CHTs are merged at once into a new 1st S-CHT of length $2n$ on the 1st pointer, and the new 2nd S-CHT with length n is enabled on the 2nd pointer; and so on. In summary, different R values correspond to different transformation rules, and such rules can also be applied to L-CHT to better handle unpredictable large-scale graphs.

Reverse Transformation: We introduce reverse transformation strategies for S-CHT and L-CHT to efficiently cope with the decreasing l. Similar to the situation where l increases, if the deletion of the current v happens to cause the overall loading rate of the S-CHT chain[4] to be lower than another threshold Λ, then we delete/compress the S-CHT where the v was originally located as follows: 1) If there are two or more S-CHTs on the S-CHT chain, we delete the current S-CHT and transfer the previously stored v on it to other S-CHTs; 2) If there is only one S-CHT left on the S-CHT chain, we compress the length of the S-CHT to half of the original length. *The above processing can be applied similarly to L-CHT.*

*2) **Optimization Stage (Denylist)**:*

There is one aspect of our design that has not been considered so far: the original CHT may suffer from item insertion failures. A straightforward solution is to extend CuckooGraph via the transformations described above whenever an insertion failure occurs. To address this issue more efficiently, we further propose an optimization called DENYLIST (DL), as shown in Figure 1 (Optimization Stage).

[4]For convenience, we call all S-CHT(s) associated with the pointers corresponding to each u a S-CHT chain.

TABLE II: An Example of Transformation Rule

# $LR > G$	the 1st S-CHT	the 2nd S-CHT	the 3rd S-CHT
0	n	null	null
1	n	n/2	null
2	n	n/2	n/2
3	2n	n	null
4	2n	n	n
5	4n	2n	null
6	4n	2n	2n
7	8n	4n	null
...	...	...	...

DL is actually a vector with a size limit. In CuckooGraph, all S-CHT(s) and L-CHT(s) are each equipped with a DL, denoted as S-DL and L-DL, respectively. However, S-DL and L-DL are organized differently: 1) Each unit of S-DL records a complete graph item, *i.e.*, a $\langle u, v \rangle$ pair; 2) While L-DL's is consistent with that of each cell of L-CHT(s), so that even if u is kicked out during item replacement, the associated S-CHT(s) does not need to be copied/moved. S-DL and L-DL are used to accommodate those that are ultimately unsuccessfully inserted into S-CHT(s) and L-CHT(s), respectively. Let's take S-DL, which cooperates with S-CHT(s), as an example for a more detailed explanation, as follows: 1) Initially, we assume that v is attempted to be inserted into an arbitrary S-CHT; 2) When the total number of kicked out exceeds the threshold T and there is still an unsettled v', the insertion fails, then v' and its corresponding u' is placed in S-DL; 3) Each time it is the S-CHT's turn to expand, we insert those v'' in S-DL whose u'' exactly match the u'' present in the current S-CHT into the new S-CHT. 4) Subsequently, S-DL continues to accommodate all failed insertion items as usual.

*3) **Operations**:*

By introducing the operations of CuckooGraph below, we aim to show how it can handle dynamically updated large-scale graphs.

Insertion: The process of inserting a new graph item $e = \langle u, v \rangle$ mainly consists of three steps, as follows:

- Step 1: We first query whether e is already stored in CuckooGraph through the **Query** operation below. If so, e is no longer inserted; otherwise, proceed to Step 2.
- Step 2: By calculating hash functions, we map u to a bucket B_H of L-CHT and try to store it in one of the cells. There are three cases here: ① u is mapped to B_H for the first time and there is at least one empty cell in the bucket. Then, we store u in Part 1 of an arbitrary empty cell, and store v in Part 2. See § III-A1 for extensions of related data structures that may be triggered by the arrival of v. ② u is mapped to B_H for the first time but the bucket is full. Then, we randomly kick out the resident u' in one of the cells and store u in it. The remaining operations are the same as ①. For this kicked-out u', we just re-insert it. ③ If u has been recorded in B_H, we directly store v in Part 2.
- Step 3: For any u and v that were not successfully inserted into Part 1 and Part 2 (in case of S-CHT), respectively, we store the relevant information in L-DL and S-DL, respectively.

Next, we illustrate the above insertion operations through the examples in Figure 1. For convenience, we assume that $R = 3$ and that there is only one bucket array for each L-CHT and S-CHT, associated with hash functions $H(.)$ and $h(.)$, respectively.

Example 1: For the new item $\langle u_2, v_3 \rangle$, it is mapped to $B[0]$. There is one cell in $B[0]$ that has already recorded u_2, but the small slots in Part 2 has recorded $2R$ v in total. Then, we transfer all v to the 1st S-CHT by computing the hash function $h(.)$.

Example 2: For the new item $\langle u_9, v_1 \rangle$, it is mapped to $B[8]$. u_9 has been recorded in $B[8]$, but LR of the 1st S-CHT exceeds G. Then, we map the qualified v (v_{10} and v_{19}) in S-DL and v_1 to the 2nd S-CHT with half the previous length.

Example 3: For the new item $\langle u_7, v_9 \rangle$, it is mapped to $B[m-1]$. Since $B[m-1]$ is already full, we randomly kick out an unlucky u_6 and stores u_7 in Part 1 of the new empty cell, and stores v_9 in the 1st small slot of Part 2. Then, we try to map u_6 into another bucket. Suppose one u is not settled in the end and it happens to be u_6. We have to place it in L-DL, along with the pointer associated with it.

Query: The process of querying a new graph item $e = \langle u, v \rangle$ mainly consists of two steps, as follows:

- Step 1: *Query whether u is in L-CHT, otherwise check whether it is in L-DL.* Specifically, we first calculate hash functions to locate a bucket B_H that may record u, and then traverse B_H to determine whether u exists. If so, we directly execute Step 2; otherwise, we further query L-DL: if u is in L-BS, proceed to Step 2; otherwise, return `null`.
- Step 2: *Query whether u has a corresponding v in Part 2.* If so, we directly report it; otherwise, we further query S-DL: if v is in S-DL, report it; otherwise, return `null`.

Deletion: To delete a graph item, we first query it and then delete it. For compression of related data structures that may be caused by deleting this item, see **Reverse Transformation** in § III-A1.

B. Extended Version

The base version of CuckooGraph can be easily extended into a new version that efficiently supports storing duplicate edges, as designed for streaming scenarios.

Data Structure: We only need to make a few customized modifications based on the transformable data structure proposed in Section III-A1. Specifically for S-CHT, each small slot in Part 2 needs to change from storing only v to storing both v and weight w. As more information is recorded, the number of small slots changes from $2R$ to R accordingly, *i.e.*, the space of two small slots is used to store $\langle v, w \rangle$.

Next, we describe the operations related to the weighted version of CuckooGraph, focusing only on its differences from the basic version for better intuition.

Insertion: The main difference from the basic version is that when it is initially discovered that the item $\langle u, v \rangle$ already exists, it changes from doing nothing to incrementing the corresponding w by 1 (or other defined value, the same below) and then returning.

Query: Report the item and return the value of w.

Deletion: We decrement the w of the item by 1 and delete the item when the weight is reduced to 0.

IV. Mathematical Analysis

In this section, we theoretically analyze the performance of CuckooGraph (basic version). Specifically, we show its time and memory complexity.

A. Time Cost of CuckooGraph

In this part, we assume that there is only insertion operation in CuckooGraph. First, we show a theorem with respect to multi-cell cuckoo hash tables. Then, we analyze the insertion time complexity of CuckooGraph based on it. Finally, we analyze the time cost of the expansion process.

Theorem 1. *Assume that a cuckoo hash table has m buckets, each bucket has d cells, and there are n distinct items to insert. Let $dm = (1+\varepsilon)n$. If $d \geq \max\left\{8, 15.8 \ln \frac{1}{\varepsilon}\right\}$, then the expected time complexity for inserting an item is $(\frac{1}{\varepsilon})^{O(\log d)}$.*

This theorem is based on the relevant proof in [47].

Since the LR is defined as $\frac{n}{dm}$, by setting $\varepsilon = \frac{dm}{n} - 1 \geq \frac{1}{G} - 1$, the expected time cost for inserting an item can be calculated as $(\frac{G}{1-G})^{O(\log d)}$. If we set T as the maximum number of loops for L-CHT, then the worst-case insertion time cost can be further written as $O(T)$, or $O(1)$ if T is not very large. Here, we provide an experiment to verify the above: We expand CuckooGraph starting from the minimum length and insert all the edges of NotreDame Dataset into it in sequence. It can be calculated that the average number of insertions per item in L-CHT and S-CHT considering the expansion is about 1.017 and 1.006, respectively, which is much less than T ($T = 250$ in the experiments in § V).

TABLE III: Comparison of complexity between different solutions.

Algorithm	Amortized Time Complexity		Space Complexity
	Insert Edge $\langle u, v\rangle$	**Query Edge** $\langle u, v\rangle$	
LiveGraph [30]	$O(1)$	$O(\deg(v))$	$O(\|E\|)$
Spruce [36]	$O\left(\frac{\|E\|}{\|V\|}\right)$	$O\left(\log \frac{\|E\|}{\|V\|}\right)$	$O(\|E\|)$
Sortledton [34]	$O(\log \|E\|)$	$O(\log \|E\|)$	$O(\|E\|)$
WBI [35]	$O(1)$	$O(\frac{\|E\|}{K^2})$	$O(K^2 + \|E\|)$
CuckooGraph (Ours)	$O(1)$	$O(1)$	$O(\|E\|)$

Then, we analyze the amortized cost of inserting N edges into CuckooGraph. We assume that two hash functions H_1, H_2 in L-CHT are the same modular hash functions. The insertion time complexity of CuckooGraph can be summarized as follows:

Theorem 2. *Assume that the L/S-DL are never full during insertion procedure and inserting an edge into L-CHT (not triggering L-CHT expansion) costs 1 dollar, then the price of inserting N edges into L-CHT will not exceed $3N$ dollars, and its expectation will not exceed $2.25N$ dollars.*

Proof. We first analyze the cost of merging and expansion: Assume that the 1st, 2nd, 3rd L-CHT stores x, y, z distinct u respectively. When merging L-CHT, we re-hash every u and re-insert it into the merged L-CHT if its hash value does not match its bucket index. The hash functions are the same modular hash functions, and the size of the merged L-CHT is 2 times larger than 1st L-CHT, and 4 times larger than 2nd and 3rd L-CHT. Hence, the probability to re-insert every u is $\frac{1}{2}$ in 1st L-CHT, and $\frac{3}{4}$ in 2nd and 3rd L-CHT. In conclusion, the price of merging operation will not exceed $x + y + z$ dollars, and its expectation is $\frac{1}{2}x + \frac{3}{4}(y + z)$ dollars.

Then, we assume that after inserting N edges, the 1st L-CHT has $2^k n$ cells, and it has n cells before insertion. Assume that the 1st, 2nd, 3rd L-CHT stores x_i, y_i, z_i distinct u before i-th merging and expansion, then $x_i \leq 2^{i-1}Gn, y_i, z_i \leq 2^{i-2}Gn$. Hence, the total cost of merging and expansion will not exceed

$$2Gn + 4Gn + 8Gn + \cdots + 2^k Gn = 2(2^k - 1)Gn,$$

and its expectation is

$$\frac{5}{4}Gn + \frac{5}{2}Gn + \cdots + 5 \cdot 2^{k-3}Gn = \frac{5}{4}(2^k - 1)Gn.$$

If $N \leq 2Gn$, then the insertion costs N dollars in total; otherwise, the LR of L-CHT is smaller than G but greater than $\frac{2}{3}G$, hence $N \geq \frac{2}{3}G \cdot (2^k n + 2^{k-1}n) = 2^k Gn$ and the total cost of merging and expansion will not exceed $2N$. In conclusion, the price of inserting N edges into L-CHT will not exceed $N + 2N = 3N$ dollars, and its expectation will not exceed $N + \frac{5}{4} \cdot N = 2.25N$. □

B. Memory Cost of CuckooGraph

In this part, we take both insertion and deletion operations into consideration. We first define a stable state for L/S-CHT and analyze its property. Then, we show the memory cost of CuckooGraph under the stable state. We assume that $\Lambda \leq \frac{2}{3}G$ in this part.

Definition 3. *We define a group of L/S-CHTs as **stable**, if its overall loading rate (LR) is at least Λ.*

The property of stable state is that, once a group of L/S-CHTs is stable, then it will be stable with high probability.

Lemma 4. *Assume that the number of graph items inserted into the L/S-CHTs at time t are l and s, respectively. If a group of L/S-CHTs is on stable state at time t, then it will always stay on stable state if the number of items in this group of L/S-CHTs is at least l and s.*

Proof. Since a group of L/S-CHTs is on stable state, its LR must be greater than Λ. Then, once its LR is greater than G, then the hash table will expand $\frac{4}{3}$ or $\frac{3}{2}$ times to its original size. And once its LR is less than Λ, the hash table will contract if possible. As a result, if the number of items in this group of L/S-CHTs is at least l and s, then the size of the L/S-CHTs will not be smaller after time t. Therefore, its LR is still greater than Λ after expansion. □

Theorem 5. *Assume that the L/S-DL are never full during insertion procedure and the L/S-CHTs are all on stable state. The upper bound of cells is $\frac{|V|}{\Lambda}$ for L-CHT and $\frac{|E|}{\Lambda}$ for all S-CHT (not including the L/S-DL), where $|V|$ denotes the number of distinct nodes, and $|E|$ denotes the number of distinct edges.*

Proof. We first analyze the number of cells in L-CHT: Since there are $|V|$ distinct nodes in the graph, they occupy at most $|V|$ cells in L-CHT. The lower bound of LR is Λ on stable state, so L-CHT has at most $\frac{|V|}{\Lambda}$ cells.

Then, we analyze the number of cells in S-CHT: Assume that $V = \{u_1, \cdots, u_{|V|}\}$, and the number of edges starting from u_i is f_i. Since all groups of S-CHTs are on stable state, then the S-CHT for u_i has at most $\frac{f_i}{\Lambda}$ cells. Hence, all S-CHT occupy at most $\frac{f_1}{\Lambda} + \cdots + \frac{f_{|V|}}{\Lambda} \leq \frac{|E|}{\Lambda}$ cells. □

C. Discussions

In this part, we provide Table III to summarize the time and space complexities of CuckooGraph and some state-of-the-art schemes ($i.e.$, competitors mentioned in § V). Here, K refers to the length/width of the matrix, which is a parameter of WBI. In summary, our CuckooGraph has an insertion and query time

complexity of $O(1)$ when Theorem 1 holds and T is not very large, while its space complexity is still $O(|E|)$. The previous analysis in § IV-B also proves that its space overhead is very small.

V. Evaluation

In this section, we evaluate the performance of CuckooGraph through extensive experiments, which are briefly described as follows: 1) We introduce the experimental setup in § V-A; 2) We evaluate how key parameters affect CuckooGraph in § V-B; 3) We verify the effect of DENYLIST optimization through ablation experiments in § V-C; 4) We evaluate the insertion, query, and deletion throughput as well as memory usage of CuckooGraph and its competitors in § V-D; 5) We evaluate the running time of CuckooGraph and its competitors on graph analytics tasks (BFS, SSSP, TC, CC, PR, BC, LCC) in § V-E; 6) We deploy CuckooGraph on Redis and Neo4j databases and evaluate the speed in § V-F and § V-G, respectively.

A. Experimental Setup

Platform: We conduct all the experiments on a 18-core CPU server (36 threads, Intel(R) Core(TM) i9-10980XE CPU @ 3.00GHz) with 128GB DRAM memory. It has 64KB L1 cache, 1MB L2 cache for each core, and 24.75MB L3 cache shared by all cores.

Implementation: We implement CuckooGraph and the other competitors with C++ and build them with g++ 7.5.0 and -O3 option. The hash functions we use are 32-bit Bob Hash (obtained from the open-source website [48]) with random initial seeds. For CuckooGraph, we set $R = 3$, as well as the ratio of the number of buckets in the two arrays of L/S-CHT is 2:1, and whether the basic or extended version of CuckooGraph is used depends on whether the dataset has repeated edges.

Competitors: Since there are many related works on dynamic graph storage, we rigorously select some of the SOTA ones from recent years for experimental comparison: LiveGraph [30], Sortledton [34], Wind-Bell Index (WBI) [35], and Spruce [36].

Datasets: We use various graph datasets to comprehensively evaluate the performance of CuckooGraph and its competitors, and the details are shown in Table IV. 1) The CAIDA dataset [49] is streams of anonymized IP traces collected by CAIDA. Each flow is identified by a five-tuple: source and destination IP addresses, source and destination ports, protocol. The source and destination IP addresses in the traces are used as the start and end nodes of the graphs, respectively. 2) The NotreDame dataset [50] is a web graph collected from University of Notre Dame. Nodes represent web pages, and directed edges represent hyperlinks between them. 3) The StackOverflow dataset [51] is a collection of interactions on the stack exchange website called Stack Overflow. Nodes represent users and edges represent user interactions. 4) The WikiTalk dataset [52] is a collection of user communications obtained from English Wikipedia, and the nodes and edges refer to the same as above. 5) The Weibo dataset [53] is captured from Sina Weibo Open Platform APIs, and the definitions of nodes and edges are similar to those of StackOverflow. 6) We synthesize the DenseGraph dataset. 7) We synthesize the SparseGraph dataset.

Metrics: We use the following key metrics.

- **Throughput:** It is defined as Million Operations Per Second (Mops). We use Throughput to evaluate the average insertion, query, and deletion speed.
- **Memory Usage:** It is defined as the memory used to store a specified amount of edges.
- **Running Time:** It is defined as the time spent performing the specified graph analytics tasks.

B. Experiments on Parameter Settings

In this subsection, we measure the effects of some key parameters for CuckooGraph, namely, the number of cells per bucket in L/S-CHT $\boldsymbol{d}$, the preset LR threshold for expansion $\boldsymbol{G}$, and the maximum number of loops in L/S-CHT $\boldsymbol{T}$. This experiment evaluates the effects by: 1) We first batch inserting edges in the CAIDA dataset into CuckooGraph and then batch querying them from CuckooGraph, and measure the average throughput separately; 2) We measure the memory usage by continuously inserting edges.

Effects of d (Figure 2(a))-2(c): *Our experimental results show that the optimal values of d is 4 and 8.* We find that $d = 8$ and $d = 4$ enable the fastest insertion and query throughput of CuckooGraph, respectively. Also, the memory usage of CuckooGraph with $d = 4$ and $d = 8$ is the least and second least, respectively. Considering that smaller d means smaller LR, we set $d = 8$.

Effects of G (Figure 3(a))-3(c): *Our experimental results show that the overall performance is best when the value of G is 0.9.* We find that the insertion and query throughput of CuckooGraph with G of 0.8, 0.85, and 0.9 are very close to each other, and all are faster than the one at G of 0.95. In addition, the larger G is, the smaller the memory usage of CuckooGraph is. Thus, we set $G = 0.9$ after the above considerations.

Effects of T (Figure 4(a))-4(c): *The experimental results show that CuckooGraph achieves most ideal performance at T of 150 and 250.* We find that CuckooGraph has the fastest insertion and query throughput when T is 150 and 250, respectively. Meanwhile, different values of T make no difference to the memory usage of CuckooGraph. Hence, we set $T = 250$.

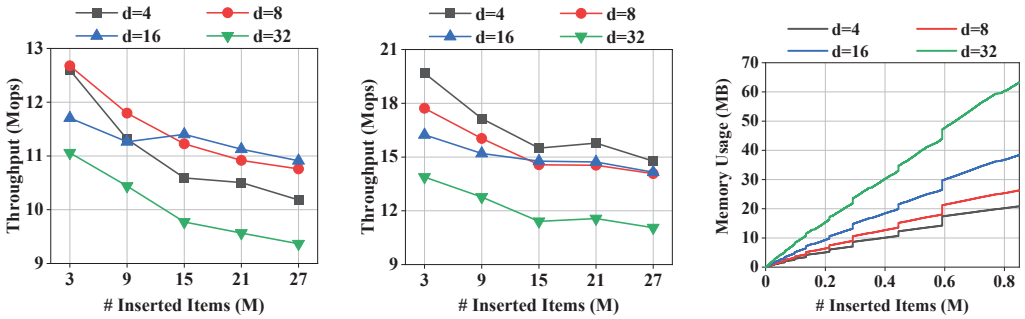
C. Ablation Experiments

In this subsection, we conduct ablation experiments to evaluate the individual effects of DENYLIST (DL) optimization on CuckooGraph performance to verify its effectiveness. Our methodology is that every time an insertion failure occurs, we expand the size of CuckooGraph to $1.5\times$ its original size. We use the CAIDA dataset and evaluate the effects in terms of insertion and query throughput as well as memory usage.

Effects of DL (Figure 5): *The experimental results show that DL indeed further speeds up insertion and querying*

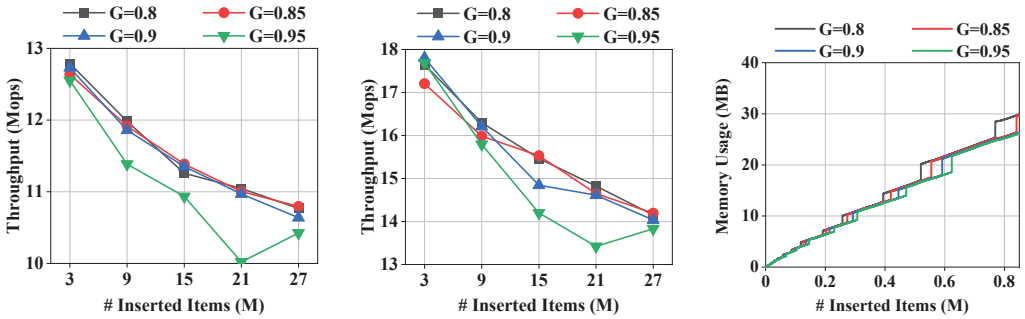
TABLE IV: A brief analysis of the graph datasets used.

Graph Dataset	Weighted?	# Nodes	# Edges	# Edges (dedup)	Avg. Deg.	Max. Deg.	Edge Density
CAIDA	✔	0.51M	27.12M	0.85M	1.66	17950	3.26×10^{-6}
NotreDame	✘	0.33M	1.50M	1.50M	4.60	10721	1.41×10^{-5}
StackOverflow	✔	2.60M	63.50M	36.23M	13.92	60406	5.35×10^{-6}
WikiTalk	✔	2.99M	24.98M	9.38M	3.14	146311	1.05×10^{-6}
Weibo	✘	58.66M	261.32M	261.32M	4.46	278491	7.60×10^{-8}
DenseGraph	✘	8K	57.59M	57.59M	7199.16	14537	0.90
SparseGraph	✘	5M	30M	30M	6	6	1.20×10^{-6}



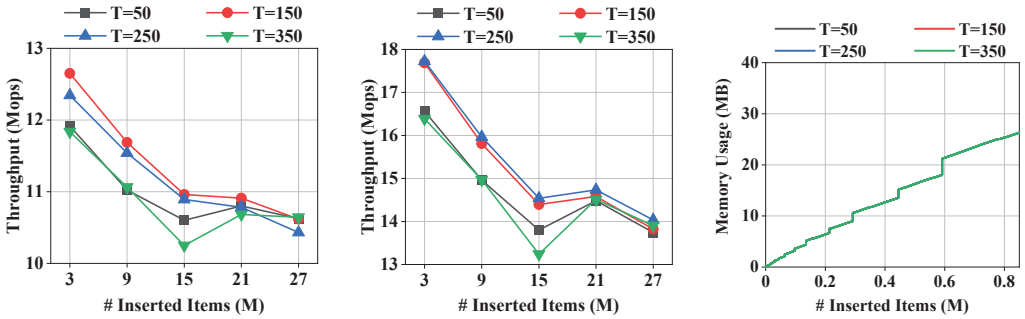
(a) Insertion Throughput (b) Query Throughput (c) Memory Usage

Fig. 2: Tuning experiments for parameter d.



(a) Insertion Throughput (b) Query Throughput (c) Memory Usage

Fig. 3: Tuning experiments for parameter G.



(a) Insertion Throughput (b) Query Throughput (c) Memory Usage

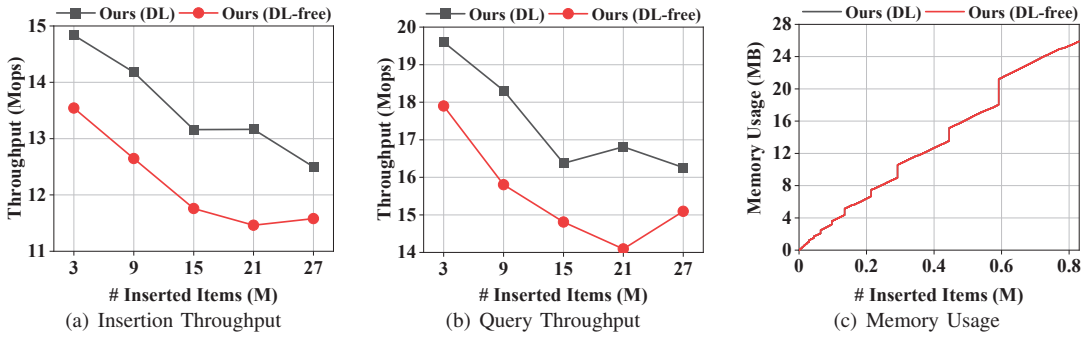
Fig. 4: Tuning experiments for parameter T.

with almost no additional memory overhead. We find that the insertion and query throughput of CuckooGraph with DL optimization is 1.11× and 1.12× faster than that of CuckooGraph without DL optimization, respectively. Also, the memory usage of CuckooGraph with DL optimization is only about 4KB more than that of CuckooGraph without DL optimization when all items are inserted.

D. Experiments on Throughput and Memory Usage

In this subsection, we evaluate the performance of CuckooGraph and its competitors in terms of insertion, query, and deletion throughput and memory usage on various graph datasets.

Methodology: 1) We insert all edges from the graph dataset into an empty graph structure, and calculate the average



(a) Insertion Throughput (b) Query Throughput (c) Memory Usage

Fig. 5: Ablation experiments: CuckooGraph with and without DL optimization.

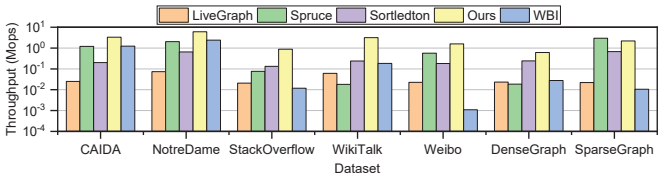


Fig. 6: Insertion throughput on different datasets.

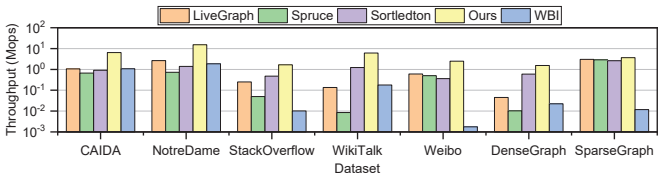


Fig. 7: Query throughput on different datasets.

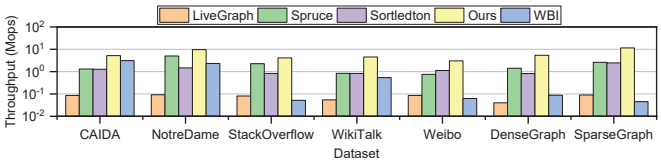


Fig. 8: Deletion throughput on different datasets.

insertion throughput; 2) We then query all edges from the graph structure and calculate the average query throughput. 3) We delete edges one by one and calculate the throughput of the process after deletions. 4) We first de-duplicate the datasets to obtain non-duplicated edges, and then insert them into each scheme one by one. After each insertion, the physical memory overhead at that moment is output.

Insertion throughput (Figure 6): The results show that, on the seven datasets, the insertion throughput of CuckooGraph is $72.17\times$, $32.66\times$, $8.60\times$, and $253.32\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Query throughput (Figure 7): The results show that, on the seven datasets, the query throughput of CuckooGraph is $14.69\times$, $133.62\times$, $5.34\times$, and $287.48\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Deletion throughput (Figure 8): The results show that, on the seven datasets, the deletion throughput of CuckooGraph is $85.47\times$, $3.63\times$, $5.01\times$, and $65.55\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: 1) Thanks to the novel data structure of CuckooGraph, when inserting or querying an edge, even in the worst case, only 6 buckets in L-CHT and S-CHT, as well as two Denylists, are accessed. Since the size of the bucket and Denylist is fixed, the upper limit on the number of memory accesses is also fixed and small. Therefore, no matter how the incoming dataset changes, CuckooGraph can achieve fast insertion and query. In contrast, other competitors are designed based on the adjacency list or its variants, so an edge insertion/query operation often requires multiple memory accesses and cannot adapt well to changes in the amount and characteristics of the dataset. 2) For deletions, other schemes simply delete the target when it is found, while CuckooGraph may involve additional contraction operations.

Memory Usage (Figure 9(a)-9(g)): The results show that, on the seven datasets, the memory usage of CuckooGraph when all item insertions are completed is $5.92\times$, $1.47\times$, $4.89\times$, and $2.34\times$ less than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

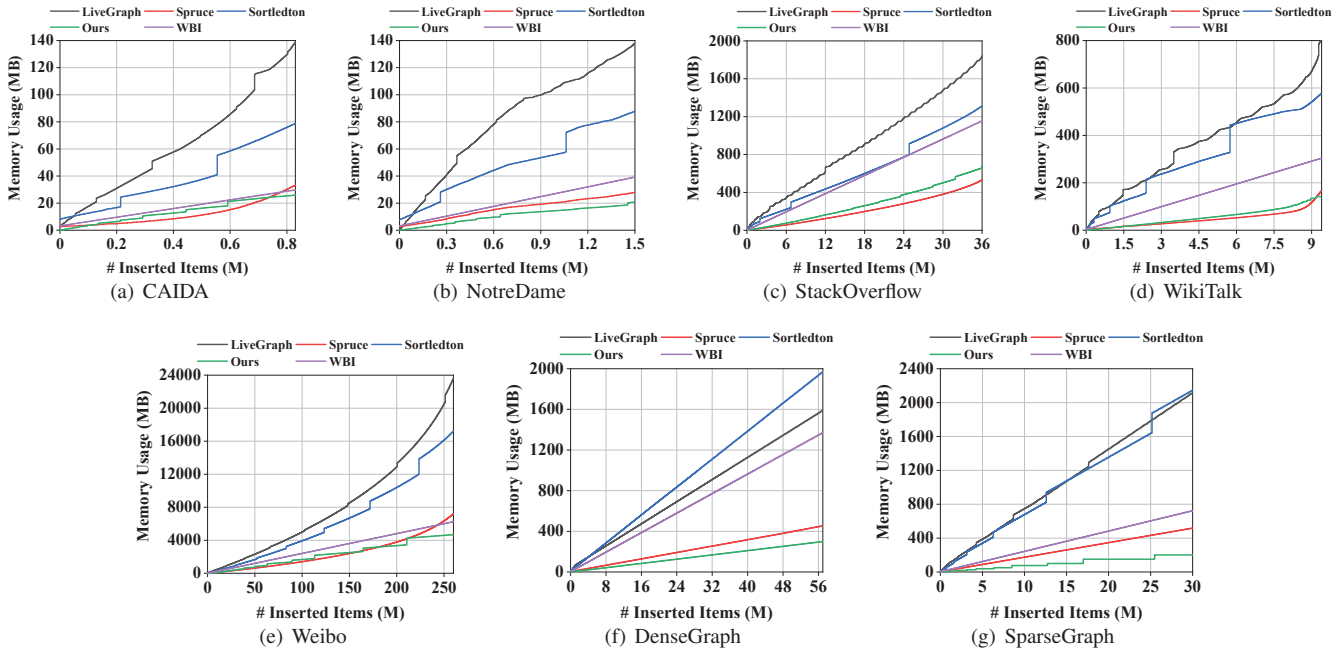
Analysis: CuckooGraph is a customized design based on CHT, so it does not need to store a large number of pointers like the schemes based on adjacency lists, which significantly reduces space overhead. In addition, since CHT has a high loading rate, CuckooGraph achieves a high loading rate and minimizes space waste.

E. Experiments on Graph Analytics Tasks

In this subsection, we evaluate the performance of CuckooGraph and its competitors in terms of running time on the graph datasets in Table IV through the following typical graph analytics tasks: Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), Triangle Counting (TC), Connected Components (CC), PageRank (PR), Betweenness Centrality (BC), and Local Clustering Coefficient (LCC). Note that some competitors did not complete the experiments within the given time, so their results are not shown in the provided figures.

*1) **Breadth-First Search**:*

Methodology: We first insert all the edges of the entire dataset. Then, we select a specific number of nodes with the largest total degree (*i.e.*, the sum of out-degree and in-degree, the same below), and perform a BFS on these nodes, returning



(a) CAIDA (b) NotreDame (c) StackOverflow (d) WikiTalk

(e) Weibo (f) DenseGraph (g) SparseGraph

Fig. 9: Memory usage on different datasets.

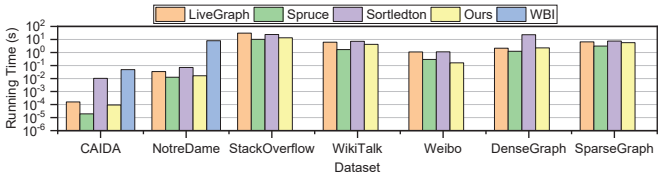


Fig. 10: Running time of BFS on different datasets.

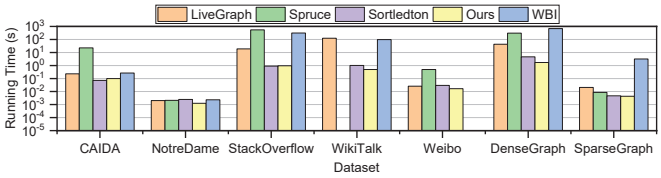


Fig. 11: Running time of SSSP on different datasets.

each node and the number of nodes obtained in the order of BFS traversal. Finally, we calculate the average time taken for these BFS tasks.

Results (Figure 10): We find that, on the seven datasets (two for WBI), the running time of CuckooGraph on BFS is $2.34\times$, $0.73\times$, $19.83\times$, and $504.81\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: The most frequently used function of each scheme in this task is its successor query function. The structure of CuckooGraph is based on hash tables, so it has good spatial locality. Therefore, during the process of querying and traversing the hash tables, the algorithm can achieve excellent spatial locality, which greatly increases the cache hit rate. Most other adjacency list-based schemes need to store data in different memory addresses and then use pointers to link them. When querying for successors, the time and space locality is poor and the cache hit rate is low, requiring frequent memory access, which reduces query efficiency. It is worth noting that WBI not only has the above shortcomings, but also needs to access many other redundant edges when querying successors, so it performs the worst. The advantage of Spruce may be that its end nodes for finding neighbors can be approximately regarded as being stored more continuously than the other 3.

*2) **Single-Source Shortest Paths**:*

Methodology: We first insert all the edges of the entire dataset. Then, we select a specific number of nodes with the largest total degree to extract subgraphs, and select the 10 nodes with the largest total degree among these nodes. Note that this refers to the 10 nodes with the largest total degree on the original graphs, not on the subgraphs. After that, we use these 10 nodes as sources to perform Dijkstra algorithm [54] 10 times and calculate the average time.

Results (Figure 11): We find that, on the seven datasets (six for Spruce & WBI), the running time of CuckooGraph on SSSP is $43.64\times$, $168.45\times$, $1.62\times$, and $278.0\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: The most frequently used function of each scheme in this task is edge query function. As described in the analysis in § V-D, CuckooGraph has a huge advantage over other adjacency list-based schemes in edge query, so CuckooGraph achieves the best performance in the SSSP task.

*3) **Triangle Counting**:*

Methodology: TC means given a node, return the number of triangles in the graph that contain that node. First, we perform successor queries to find all 2-hop successors of the node. Then, we enumerate all possible edges $\langle \textit{2-hop successor}, \textit{node} \rangle$ composed of the node's 2-hop successors and the node itself to perform edge queries. Finally, the number of successful queries is the results of TC.

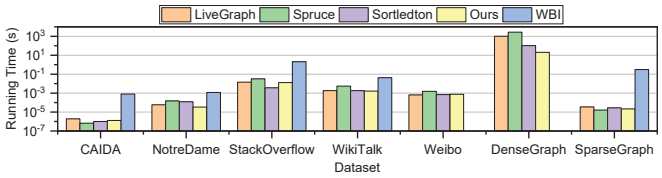


Fig. 12: Running time of TC on different datasets.

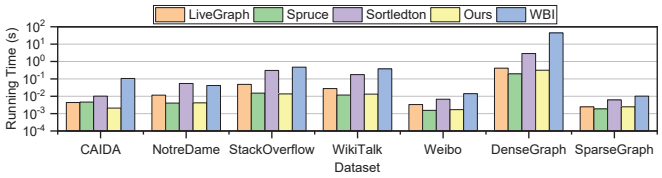


Fig. 13: Running time of CC on different datasets.

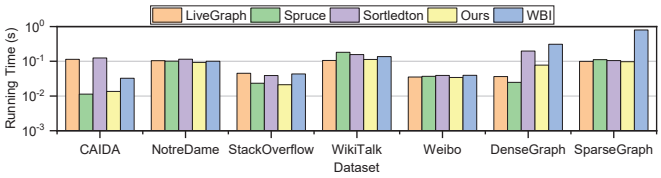


Fig. 14: Running time of PR on different datasets.

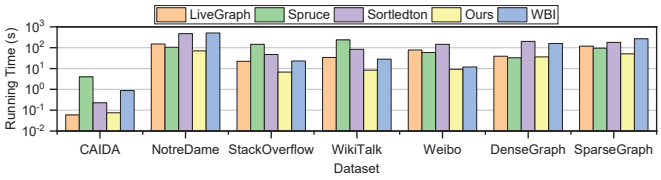


Fig. 15: Running time of BC on different datasets.

Results (Figure 12): We find that, on the seven datasets (five for WBI), the running time of CuckooGraph on TC is $8.23\times$, $21.33\times$, $1.86\times$, and $3015.11\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: The edge query and successor query functions are the most frequently used functions in each scheme in this task. As analyzed in § V-E1 and § V-E2, CuckooGraph can achieve good performance in these two functions, so it also performs very well in this task.

*4) **Connected Components**:*

Methodology: All edges for the entire dataset are inserted. We first select a specific number of nodes with the largest total degree to extract subgraphs, and then insert the subgraphs into each scheme. *Note that the above steps also apply to the last 3 tasks.* After that, we run the Tarjan algorithm [55] on the subgraphs using each scheme and return the connected components and their number.

Results (Figure 13): We find that, on the seven datasets, the running time of CuckooGraph on CC is $2.11\times$, $1.07\times$, $9.91\times$, and $39.7\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: The most frequently used function of each scheme in this task is its successor query function. As analyzed in § V-E1, CuckooGraph performs well in this function, so it has an advantage over other schemes in this task.

*5) **PageRank**:*

Methodology: The initial steps are the same as those in § V-E4. Then, we use the successor query function of each scheme to assist in constructing the matrix required to solve the PageRank (PR), and iterate 100 times on the matrix to find the PR of each node on the subgraphs.

Results (Figure 14): We find that, on the seven datasets, the running time of CuckooGraph on PR is $2.16\times$, $1.03\times$, $2.62\times$, and $2.87\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: Each scheme in this task frequently uses the successor query function to construct the matrix required to calculate PR. As analyzed in § V-E1, CuckooGraph performs well in successor query, so it also shows advantages over other schemes in this task.

*6) **Betweenness Centrality**:*

Methodology: The initial steps are the same as those in § V-E4. Then, we run the Brandes algorithm [56] on the subgraphs using each scheme.

Results (Figure 15): We find that, on the seven datasets, the running time of CuckooGraph on BC is $3.15\times$, $16.17\times$, $7.33\times$, and $5.23\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: Similar to the analysis in § V-E4.

*7) **Local Clustering Coefficient**:*

Methodology: The initial steps are the same as those in § V-E4. Then, we pre-compute all neighbors of each node and run the Local Clustering Coefficient (LCC) algorithm, which is implemented in [57].

Results (Figure 16): We find that, on the seven datasets (six for WBI), the running time of CuckooGraph on LCC is $2.06\times$, $5.80\times$, $3.94\times$, and $4.21\times$ faster than that of LiveGraph, Spruce, Sortledton, and WBI on average, respectively.

Analysis: Similar to the analysis in § V-E4.

F. Redis Implementation

Methodology: We utilize Redis Module [58] to register our CuckooGraph module, adding the data structure of CuckooGraph to the original Redis. This allows Redis to store graphs in addition to supporting the five original data structures. Specifically, we implement Redis Module API (including `save_rdb`, `load_rdb`, `aof_rewrite` and other interfaces) on top of CuckooGraph to support Redis persistence operations. Meanwhile, we also provide extended commands for CuckooGraph (including `insert`, `del`, `query` and `getneighbors`). We compile our interface implementation into a dynamic link library, and simply import CuckooGraph library with `--loadmodule` when Redis starts.

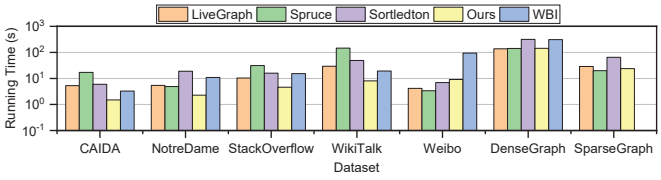


Fig. 16: Running time of LCC on different datasets.

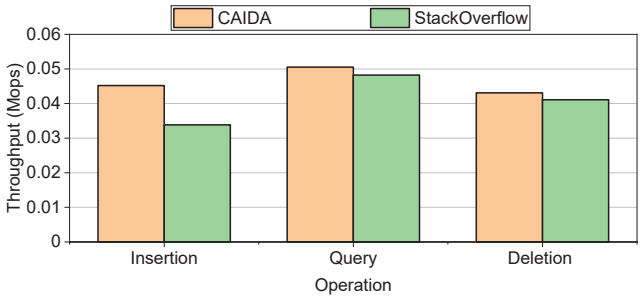


Fig. 17: The throughput of CuckooGraph on Redis.

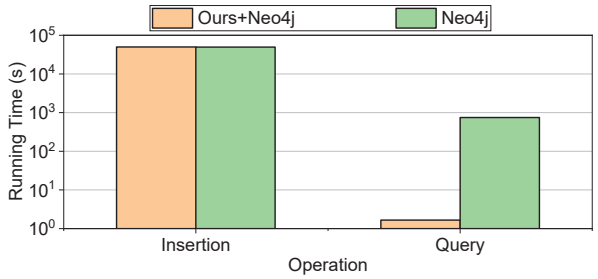


Fig. 18: Running time of Neo4j with and without CuckooGraph.

Setup: We conduct experiments on the CAIDA and StackOverflow datasets to test the throughput performance of CuckooGraph on Redis.

Results & Analysis (Figure 17): The results show that the insertion, query, and deletion throughput of CuckooGraph on Redis is around $0.04 \sim 0.05$ Mops. There is some performance loss compared to those of CuckooGraph on CPU, which is mainly caused by the Redis system. We also run Redis benchmark on the server, and the peak throughput of native Redis is only around 0.16 Mops. Considering that CuckooGraph itself inevitably has overhead, its performance on Redis is completely acceptable.

G. Neo4j Implementation

Methodology: If we want to store an edge $\langle u, v\rangle$ in Neo4j [37], nodes u and v each maintain a adjacency list that stores all the edges associated with that node, so the information about $\langle u, v\rangle$ is stored in the adjacency lists of both u and v. If we want to query an edge $\langle u, v\rangle$, we have to find the adjacency list of u, and then traverse the list and compare the edges one by one until we find $\langle u, v\rangle$. Obviously, it is inefficient. Once the degree of u is high, querying edge $\langle u, v\rangle$ has to access a large number of unrelated other edges, causing additional redundancy overhead. To speed up edge queries, we introduce the CuckooGraph query interface to obtain an edge without traversing the adjacency list of the node. Since multiple edges (with the same u and v but not the same edge) are allowed in Neo4j, the data structure of CuckooGraph needs some adjustments for this. Compared to the weighted version on the CPU, we change the weight field in each S-CHT small slot from a counter that records the number of edges to a linked list consisting of a series of edges with the same nodes u and v. The linked list is as long as the number of edges corresponding to $\langle u, v\rangle$ in that small slot. In this way, the query interface of CuckooGraph returns an iterator, through which the linked list can be traversed to obtain all the edges between $\langle u, v\rangle$.

Setup: We deploy the above CuckooGraph on top of the original Neo4j and evaluate the performance by running time, as shown below. 1) For the insertion experiments, we insert the first 1M edges from the CAIDA dataset into Neo4j. Whenever an edge is inserted into Neo4j, we also need to insert that edge into the CuckooGraph structure, which requires a little extra time overhead. 2) For the query experiments, we first deduplicate the 1M edges, and then query the CuckooGraph structure. For comparison, related operations on pure Neo4j do not introduce CuckooGraph.

Results & Analysis (Figure 18): 1) Thanks to the good performance of our data structure, our insertions are very fast and require only a little extra overhead, so our insertion time is almost the same as pure Neo4j. **2)** Since the time cost of CuckooGraph's query to obtain the iterator of the linked list is $O(1)$, the query speed of the version with CuckooGraph is very fast. It can be predicted that the query speed of Neo4j with CuckooGraph will be improved more significantly as the data scale increases. In pure Neo4j, many irrelevant/redundant edges must be traversed, and this additional overhead time is not $O(1)$, which ultimately causes the query time of pure Neo4j to be much slower than that with the assistance of CuckooGraph.

VI. Conclusion

In this paper, we propose a novel data structure designed for large-scale dynamic graphs, called CuckooGraph, which includes two key techniques, Transformation and Denylist. Thanks to them, CuckooGraph can be flexibly resized based on actual operations to achieve memory efficiency while keeping few memory accesses to achieve fast processing speed without any prior knowledge of the upcoming graphs. Our mathematical analysis theoretically proves that CuckooGraph is time and space efficient. Our experimental results show that CuckooGraph significantly outperforms 4 state-of-the-art schemes. In particular, compared with Spruce, CuckooGraph achieves $32.66\times$ faster insertion throughput while reducing memory space by about 32%, and $168.45\times$ less running time on the SSSP task. Finally, we integrate CuckooGraph in Redis and Neo4j databases to extend its practicality.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments. This work was supported by the National Natural Science Foundation of China (NSFC) (No. 62402012, 62372009), China Postdoctoral Science Foundation (No. 2023TQ0010, GZC20230055, 2024M750102), research grant No. SH-2024JK29, and High-Performance Computing Platform of Peking University.

References

[1] J. Li, X. Wang, K. Deng, X. Yang, T. Sellis, and J. X. Yu, "Most influential community search over large social networks," in *ICDE*, 2017, pp. 871–882.

[2] Y. Matsunobu, S. Dong, and H. Lee, "Myrocks: Lsm-tree database storage engine serving facebook's social graph," *PVLDB*, vol. 13, no. 12, pp. 3217–3230, 2020.

[3] J. Zhang, C. Gao, D. Jin, and Y. Li, "Group-buying recommendation for social e-commerce," in *ICDE*, 2021, pp. 1536–1547.

[4] D. Wang, J. Lin, P. Cui, Q. Jia, Z. Wang, Y. Fang, Q. Yu, J. Zhou, S. Yang, and Y. Qi, "A semi-supervised graph attentive network for financial fraud detection," in *ICDM*, 2019, pp. 598–607.

[5] J. Jiang, Y. Li, B. He, B. Hooi, J. Chen, and J. K. Z. Kang, "Spade: a real-time fraud detection framework on evolving graphs," *PVLDB*, vol. 16, no. 3, pp. 461–469, 2022.

[6] X. Huang, Y. Yang, Y. Wang, C. Wang, Z. Zhang, J. Xu, L. Chen, and M. Vazirgiannis, "Dgraph: A large-scale financial dataset for graph anomaly detection," *NeurIPS*, vol. 35, pp. 22 765–22 777, 2022.

[7] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network monitoring using traffic dispersion graphs (tdgs)," in *IMC*, 2007, pp. 315–320.

[8] T. Wang and L. Liu, "Privacy-aware mobile services over road networks," *PVLDB*, vol. 2, no. 1, pp. 1042–1053, 2009.

[9] M. Simeonovski, G. Pellegrino, C. Rossow, and M. Backes, "Who controls the internet? analyzing global threats using property graph traversals," in *WWW*, 2017, pp. 647–656.

[10] Y. Ma, P. Gerard, Y. Tian, Z. Guo, and N. V. Chawla, "Hierarchical spatio-temporal graph neural networks for pandemic forecasting," in *CIKM*, 2022, pp. 1481–1490.

[11] X. Zhu, X. Huang, L. Sun, and J. Liu, "A novel graph indexing approach for uncovering potential covid-19 transmission clusters," *ACM Transactions on Knowledge Discovery from Data*, vol. 17, no. 2, pp. 1–24, 2023.

[12] J. Mondal and A. Deshpande, "Managing large dynamic graphs efficiently," in *SIGMOD*, 2012, pp. 145–156.

[13] P. Pandey, B. Wheatman, H. Xu, and A. Buluc, "Terrace: A hierarchical graph container for skewed dynamic graphs," in *SIGMOD*, 2021, pp. 1372–1385.

[14] J. Hou, Z. Zhao, Z. Wang, W. Lu, G. Jin, D. Wen, and X. Du, "Aeong: An efficient built-in temporal support in graph databases," *PVLDB*, vol. 17, no. 6, pp. 1515–1527, 2024.

[15] M. Potamias, F. Bonchi, A. Gionis, and G. Kollios, "K-nearest neighbors in uncertain graphs," *PVLDB*, vol. 3, no. 1-2, pp. 997–1008, 2010.

[16] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos, "Gbase: an efficient analysis platform for large graphs," *The VLDB Journal*, vol. 21, pp. 637–650, 2012.

[17] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *PVLDB*, vol. 11, no. 4, pp. 420–431, 2017.

[18] Y. Shao, B. Cui, L. Chen, L. Ma, J. Yao, and N. Xu, "Parallel subgraph listing in a large-scale graph," in *SIGMOD*, 2014, pp. 625–636.

[19] Z. Wei, X. He, X. Xiao, S. Wang, Y. Liu, X. Du, and J.-R. Wen, "Prsim: Sublinear time simrank computation on large power-law graphs," in *SIGMOD*, 2019, pp. 1042–1059.

[20] J. Guo, B. Chen, K. Yang, T. Yang, Z. Liu, Q. Yin, S. Wang, Y. Wu, X. Wang, B. Cui, T. Li, X. Peng, R. Chen, and G. Zhang, "Hourglasssketch: An efficient and scalable framework for graph stream summarization," in *ICDE*, 2025.

[21] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *HPEC*, 2012, pp. 1–5.

[22] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *PPoPP*, 2013, pp. 135–146.

[23] L. Zou, M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao, "gstore: a graph-based sparql query engine," *The VLDB Journal*, vol. 23, pp. 565–590, 2014.

[24] W. Sun, A. Fokoue, K. Srinivas, A. Kementsietsidis, G. Hu, and G. Xie, "Sqlgraph: An efficient relational-based property graph store," in *SIGMOD*, 2015, pp. 1887–1901.

[25] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer, "Llama: Efficient graph analytics using large multiversioned arrays," in *ICDE*, 2015, pp. 363–374.

[26] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *HPEC*, 2018, pp. 1–7.

[27] L. Dhulipala, G. E. Blelloch, and J. Shun, "Low-latency graph streaming using compressed purely-functional trees," in *PLDI*, 2019, pp. 918–934.

[28] S. Firmli, V. Trigonakis, J.-P. Lozi, I. Psaroudakis, A. Weld, D. Chiadmi, S. Hong, and H. Chafi, "Csr++: A fast, scalable, update-friendly graph data structure," in *OPODIS*, 2020.

[29] P. Kumar and H. H. Huang, "Graphone: A data store for real-time analytics on evolving graphs," *ACM Transactions on Storage*, vol. 15, no. 4, pp. 1–40, 2020.

[30] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, "Livegraph: a transactional graph storage system with purely sequential adjacency list scans," *PVLDB*, vol. 13, no. 7, pp. 1020–1034, 2020.

[31] G. Feng, Z. Ma, D. Li, S. Chen, X. Zhu, W. Han, and W. Chen, "Risgraph: A real-time streaming system for evolving graphs to support sub-millisecond per-update analysis at millions ops/s," in *SIGMOD*, 2021, pp. 513–527.

[32] A. Mhedhbi, P. Gupta, S. Khaliq, and S. Salihoglu, "A+ indexes: Tunable and space-efficient adjacency lists in graph database management systems," in *ICDE*, 2021, pp. 1464–1475.

[33] A. A. R. Islam, D. Dai, and D. Cheng, "Vcsr: Mutable csr graph format using vertex-centric packed memory array," in *CCGrid*, 2022, pp. 71–80.

[34] P. Fuchs, D. Margan, and J. Giceva, "Sortledton: a universal, transactional graph data structure," *PVLDB*, vol. 15, no. 6, pp. 1173–1186, 2022.

[35] R. Qiu, Y. Ming, Y. Hong, H. Li, and T. Yang, "Wind-bell index: Towards ultra-fast edge query for graph databases," in *ICDE*, 2023, pp. 2090–2098.

[36] J. Shi, B. Wang, and Y. Xu, "Spruce: a fast yet space-saving structure for dynamic graph storage," *PACMMOD*, vol. 2, no. 1, pp. 1–26, 2024.

[37] "Neo4j website." [Online]. Available: https://neo4j.com/

[38] "OrientDB website." [Online]. Available: http://orientdb.org/

[39] "ArangoDB website." [Online]. Available: https://www.arangodb.com/

[40] "JanusGraph website." [Online]. Available: https://janusgraph.org/

[41] "GraphDB website." [Online]. Available: https://www.ontotext.com/products/graphdb/

[42] R. Pagh and F. F. Rodler, "Cuckoo hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[43] "The source codes related to CuckooGraph." [Online]. Available: https://github.com/pkufzc/CuckooGraph

[44] M. A. Bender and H. Hu, "An adaptive packed-memory array," *ACM Transactions on Database Systems*, vol. 32, no. 4, pp. 26–es, 2007.

[45] D. De Leo and P. Boncz, "Teseo and the analysis of structural dynamic graphs," *PVLDB*, vol. 14, no. 6, pp. 1053–1066, 2021.

[46] Y. Li, H. Zheng, L. Zou, X. Li, Z. Li, P. Xiao, Y. Tao, and Z. Qin, "Vend: Vertex encoding for edge nonexistence determination," in *ICDE*, 2023, pp. 328–340.

[47] M. Dietzfelbinger and C. Weidling, "Balanced allocation and dictionaries with tightly packed constant size bins," *Theoretical Computer Science*, vol. 380, no. 1-2, pp. 47–68, 2007.

[48] "Hash website." [Online]. Available: http://burtleburtle.net/bob/hash/evahash.html

[49] "CAIDA Anonymized Internet Traces 2018 Dataset." [Online]. Available: https://www.caida.org/catalog/datasets/overview/

[50] "Note Dame web graph." [Online]. Available: http://snap.stanford.edu/data/web-NotreDame.html

[51] "Stack Overflow temporal network." [Online]. Available: http://snap.stanford.edu/data/sx-stackoverflow.html

[52] "Wikipedia talk (en)." [Online]. Available: http://konect.cc/networks/wiki_talk_en/

[53] R. Rossi and N. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, vol. 29, no. 1, 2015. [Online]. Available: https://networkrepository.com

[54] E. W. Dijkstra, *A Note on Two Problems in Connexion with Graphs*, 1st ed. ACM, 2022, pp. 287–290. [Online]. Available: https://doi.org/10.1145/3544585.3544600

[55] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[56] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[57] A. Iosup, A. Musaafir, A. Uta, A. P. Pérez, G. Szárnyas, H. Chafi, I. G. Tănase, L. Nai, M. Anderson, M. Capotă *et al.*, "The ldbc graphalytics benchmark," *arXiv preprint arXiv:2011.15028*, 2020.

[58] "Redis modules API." [Online]. Available: https://redis.io/docs/latest/develop/reference/modules/