# Spruce: A Fast yet Space-saving Structure for Dynamic Graph Storage

JIFAN SHI, University of Science and Technology of China, China
BIAO WANG, University of Science and Technology of China, China
YUN XU*, University of Science and Technology of China, China

Dynamic graphs have been gaining increasing popularity across various application domains. With the growing size of these graphs, the update performance as well as space occupancy is becoming a crucial aspect of dynamic graph storage. Although existing dynamic graph systems can handle massive streaming updates (e.g., insertions and deletions), they cannot achieve both high throughput and low memory footprint. Drawing inspiration from the basic operations of the van Emde Boas (vEB) tree in double-logarithmic time, we designed Spruce, a high-performance yet space-saving in-memory structure to store dynamic graphs. Spruce uses a compact representation to construct the tree-like multilevel structure, which shares the common prefixes of vertices and has no merging or splitting of nodes to achieve the requirements of low memory consumption and high-efficiency dynamic operations. Furthermore, Spruce incorporates a read-optimized concurrency protocol, which refines ROWEX and Optimistic Locking, to facilitate efficient simultaneous read/write operations. Our experiment demonstrates that compared to Sortledton (the best of competitors), Spruce is up to 2.4× faster in ingesting graph updates, while saving up to 38.5% of memory space. As for graph analytics, Spruce shows high adaptability to different analytical workloads, and achieves comparable performance to other state-of-the-art dynamic graph structures.

CCS Concepts: • **Information systems → Graph-based database models**; *Data structures.*

Additional Key Words and Phrases: Graph data structures, Index, Streaming

**ACM Reference Format:**
Jifan Shi, Biao Wang, and Yun Xu. 2024. Spruce: A Fast yet Space-saving Structure for Dynamic Graph Storage. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 27 (February 2024), 26 pages. https://doi.org/10.1145/3639282

## 1 INTRODUCTION

Graph analytical techniques have been under rapid development and are widely implemented in our daily lives. They are used in various fields, including economics, physics [74], chemistry [37] and biology (for DNA analysis) [25], etc. For example, Facebook and Twitter use social networks to manage social relationships among users [48], while JPMorgan Chase uses the graph database to help generate risk management measures [61]. These graphs not only have large scales, containing up to billions of vertices and trillions of edges [36, 46], but also are continuously changing [7, 31], with streaming updates reaching at a high rate [51]. Nowadays, lots of applications are built on evolving graphs, such as dynamic social recommend systems [66] and real-time events discovery

*Corresponding author.

Authors' addresses: Jifan Shi, University of Science and Technology of China, Hefei, China, shijifan8@mail.ustc.edu.cn; Biao Wang, University of Science and Technology of China, Hefei, China, wangbiao0814@mail.ustc.edu.cn; Yun Xu, University of Science and Technology of China, Hefei, China, xuyun@ustc.edu.cn.

ACM 2836-6573/2024/2-ART27
https://doi.org/10.1145/3639282

[4]. Therefore, there is an urgent need for both high-performance and space-saving dynamic graph data systems.

The storage scheme is a crucial part of graph systems, which affects the memory footprint, graph analytics run time, and update performance. Most existing graph systems organize graph data based on two types of data structures: *adjacency list* [35] and *Compressed Sparse Row (CSR)* [47]. The adjacency list organizes each vertex's adjacency edges in a linked list, which is easy to edit and suitable for streaming updates. But the pointers in lists not only take up extra space but also lead to pointer chasing (irregular memory access) when accessing adjacency edges [8]. Besides, it stores vertices in a static array, which is unsuitable for storing evolving graphs. CSR is a kind of compressed sparse matrix, which stores the non-zero elements of the matrix in several arrays [68]. Intrinsically, CSR uses a *compact storage scheme* that stores elements consecutively in dense arrays. It has good spatial locality as well as space efficiency, making it ideal for static graph storage [63, 64]. However, when it comes to dynamic graphs, CSR is inconvenient because it has to reconstruct the whole array when performing insertions or deletions on the graph.

Recently, many data structures have been developed from the adjacency list and CSR to better meet the growing need for evolving graph storage. These structures mainly focus on three aspects of efforts: (1) enabling fast modification and mapping of vertices, (2) providing good editability and accessibility of adjacency edges, and (3) supporting concurrent read/write operations. For (1), recent works usually construct a high-performance index for vertices, including hash table, B+ tree, radix tree, purely-Functional tree and multilevel vectors [3, 22–24, 29, 43]. Regrettably, these indexes cannot avoid reconstructing, splitting or merging part of their structures when the vertices suffer from frequent changes, which leads to performance degradation on write-heavy workloads. For (2), typical methods include using skip lists to accelerate queries, adopting block-based list structure to improve locality, pre-allocating spaces in arrays for upcoming insertions, etc [27, 29, 43, 72, 73]. For (3), multi-version concurrency control (MVCC) along with optimistic locking is widely adopted [29, 43, 75] to satisfy the growing requirement of running graph analytics along with graph updates. However, existing data structures for evolving graph storage are difficult to juggle both space consumption and dynamic performance, which still presents opportunities for improvement.

In this paper, we present Spruce, a high-performance yet space-saving structure for dynamic graph storage. Our method borrows the idea from van Emde Boas tree (vEB tree) [69], a tree data structure that implements an associative array and guarantees operations in $O(lglg(u))$ time when using $O(u)$ space to store $n$ elements of universe $U = \{0, 1, 2, ..., u - 1\}$ [71]. Spruce takes advantage of the shared prefix nodes and compact format of the vEB tree to save space, as well as the no splitting and merging of nodes to improve operational performance. Moreover, we use a block-based strategy to organize vertices to optimize their space consumption. For storage of adjacency edges, we introduce a combination of dynamically changeable buffers and compact sorted blocks like CSR to balance space overhead and operation performance. Besides, we design a lightweight concurrency protocol that benefits from the features of no splitting and merging of index nodes, significantly improving system concurrency performance and throughput.

We compared Spruce with other state-of-the-art dynamic graph systems on different datasets. The experiment result shows that Spruce notably outperforms all existing graph systems for insertion and deletion throughput while using the least runtime memory. Also, Spruce provides comparable graph analytics performance on read-only workloads and supports graph pattern matching algorithms (e.g., local clustering coefficient). As for mixed workloads, Spruce achieves over 10× speed up compared to LiveGraph when ingesting updates, and has lower run times for concurrent graph analytics.

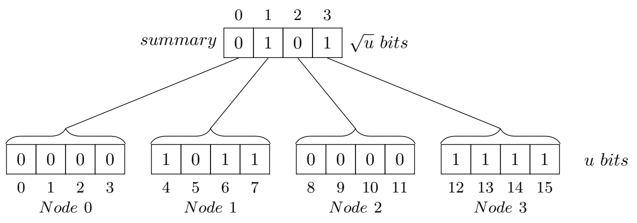


Fig. 1. The prototype of vEB tree. This instance represents the set $\{\mathbf{4}, \mathbf{6}, \mathbf{7}, \mathbf{12}, \mathbf{13}, \mathbf{14}, \mathbf{15}\}$ of universe $\mathbf{U} = \{\mathbf{0}, \mathbf{1}, \mathbf{2}, ..., \mathbf{15}\}$.

## 2 PRELIMINARIES

### 2.1 Van Emde Boas tree

A *bitvector* is a bit array $B$ that compactly stores bits, which is the base structure of the vEB tree. The bitvector can efficiently map a range of integers to values in the set $\{0, 1\}$ using less space (e.g., the set $A = \{0, 2, 5, 6, 7\}$ of 5 bytes can be represented as the 8-bit bitvector $B[8] = \{1, 0, 1, 0, 0, 1, 1, 1\}$). Van Emde Boas tree (vEB tree) [69] supports dynamic operations on a set $S$ of $n$ keys chosen from the universe $U = \{0, 1, 2, ..., u - 1\}$ in worst case time $O(lglg(u))$. The prototype of the vEB tree is a multiway tree consisting of bitvectors using $O(u)$ space, as shown in Figure 1. The bitvectors at the bottom contain $u$ bits in total, where each bit indicates the presence or absence of the corresponding integer in $U$. The upper nodes are treated as a bitvector $summary[0, ..\sqrt{u} - 1]$. If $summary[i]$ is 0, the bits in the bitvector of size $\sqrt{u}$ at the lower level are all 0. Otherwise, at least one of these bits is 1.

Dynamic operations are easy to perform on this structure. Take the vEB tree in Figure 1 as an example. To insert 2, we set the corresponding bit in $Node$ 0 to 1 and also set the $summary[0]$ to 1. To delete 15, we first set the 4th bit in $Node$ 3 to 0, then set $summary[3]$ the result of logical-or of the bits in $Node$ 3, which is 1. These operations only take $O(lglg(u))$ time on the vEB tree since the size of total bits in bitvectors shrinks by the square root at each level. Unlike B tree [10], B$^+$ tree [20], prefix tree [28], and many of their variants [14, 50, 54], these operations do not cause resizing splitting or merging of nodes, which demonstrates superiority over them.

A standard vEB tree requires $O(u)$ space to store a subset of size $n$ in the integer universe $U$. This upper bound can reduced to $O(nlglg(u))$ by using hash tables to replace bitvectors and only allocating space and entries for non-empty nodes (i.e., nodes that have at least one value) [6]. Each level takes at most $O(n)$ space, and there are $O(lglgu)$ levels. Therefore, the total space bound is $O(nlglg(u))$. However, such a representation requires a lot of different hash tables, which is still expensive for space and lowers the vEB tree's performance.

### 2.2 Optimistic Locking and ROWEX

Optimistic locking [39] is a concurrency control method based on the optimistic assumption that there will be few concurrent modifications on the same resources. An optimistic lock usually consists of an exclusive lock and a timestamp. For each writer, it modifies the data exclusively and updates the timestamp at each write. For each reader, it takes the timestamp as an argument and reads data when the lock is free. After reading, it rechecks the timestamp to make sure no changes have occurred during the reading process. If the check detects stale data, the read result is invalid, and corresponding operations need to be restarted. The advantages of optimistic locking are that it can prevent deadlocks and avoids the overhead of frequently locking the data.

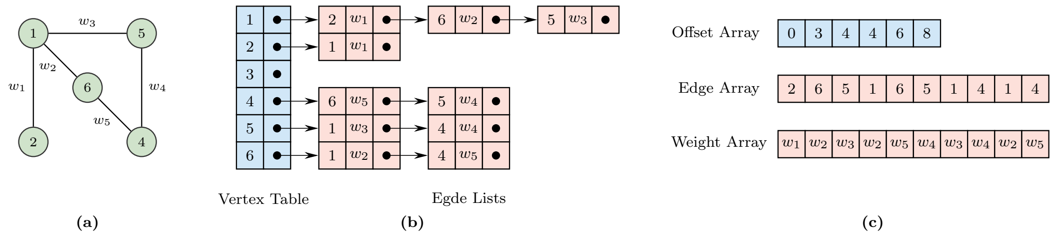


Fig. 2. Graph Representation: (a) an example of a weighted graph; (b) adjacency list of (a); (c) CSR of (a).

Optimistic locking behaves well when few conflicts occur among writers, but the restart strategy may result in a significant performance slowdown for reads on write-heavy workloads. *Read-Optimized Write EXclusion (ROWEX)* [41] was proposed and firstly used in *adaptive radix tree (ART)* [40] to solve this problem. The main component of ROWEX is the write lock. This lock only provides exclusion among writers, and readers can read data ignoring the locks. To avoid restarts while ensuring the correctness of reads, any read or write must execute atomically. Besides, the internal data structures in ROWEX generally need some modifications to coordinate with atomic operations. For example, to apply ROWEX, ART adds a lock for each node in the trie. It takes the read-copy update (RCU) for updates because ensuring consistency across the entire node structure is difficult within a single atomic operation. To update a node, ART first locks the node and its parent. After that, a copy of the node is created, and updates are done on the copy. Then, the pointer points to the old node is changed to the new copy using an atomic store. Finally, the old node is marked obsolete, and the locks of the old node and its parent are released.

## 3 BACKGROUND AND MOTIVATIONS

### 3.1 Classical Methods of Graph Representation

Adjacency list [35] and Compressed Sparse Row (CSR) [47] are two of the classical graph representation methods. For the example of weighted graph in Figure 2(a), the adjacency list is composed of a *vertex table* and *edge lists*, as shown in Figure 2(b). The vertex table is a static array indexed by vertex identifiers, and the edge list is a singly linked list that stores information of connected edges, including adjacency vertices and edge properties. The adjacency list has the advantage that it is easy to perform insertion or deletion on edge lists. However, it cannot efficiently support the changes of vertices as the size of the vertex table is determined by the scale of the initial graph. Besides, the pointers in the lists occupy considerable space and usually cause irregular memory access.

CSR is more space-efficient compared to the adjacency list. As shown in Figure 2(c), CSR only has several static arrays, i.e., *offset array*, *edge array*, and *edge property arrays*. The offset array keeps the offsets of the beginning and the end of edges in the edge array for each vertex. The edge array consecutively stores the outgoing edges of vertices, and the edge property array maintains the properties (weight, name, etc.) of these edges. Insertions and deletions are expensive in CSR since all arrays need to be reconstructed, making it unsuitable for storing evolving graphs.

### 3.2 Existing Solutions and Limitations

The key to dynamic graph storage is to enable concurrent *graph updates* (insertion, deletion, property change, etc.) with low latency. Most of the existing dynamic graph systems refine the structure of the adjacency list or CSR to store evolving graphs.

Dynamic graph systems based on the adjacency list mainly focus on two efforts: (1) make vertex table support dynamic operations and (2) improve access and update speed of edge lists. For (1), most systems use an index to map explicit vertex identifiers to logical vertex identifiers and store logical vertices in arrays. For example, hash tables, adaptive radix tree, and multilevel vectors are used to index logical vertices. Hash tables are known to take up much more space than other indexes to deal with collisions, and existing tree-like indexes for graph systems cannot avoid merging, splitting, or resizing nodes when executing graph updates, reducing their efficiency. For (2), the methods are diverse. Stinger [24] aggregates nodes in edge lists as blocks to improve read efficiency and uses a hash table as the secondary index to accelerate point queries. Graphone [38] novelly combines adjacency list with edge log array. Any changes on the graph are firstly appended to the tail of the edge log array and then merged into the adjacency list in batches. Sortledton [29] applies unrolled skip list [56] to replace edge lists. An unrolled skip list manages multiple edges in the list nodes and borrows the concept of the skipped list to provide logarithmic time complexity for lookup, insertion, and deletion. Aspen [26] and Pac-trees [22] develop a new compressed purely-functional search tree data structure, which could store compressed edge data with low space occupation at the cost of slowdown in updates. Regrettably, none of these methods could both well reduce the space usage of edge lists and gain high throughput for updates.

There are three typical ways to make CSR support graph updates. The first one is dividing CSR into segments [27] to avert the reconstruction of the whole array. The vertices are stored in separate segments indexed by a global array, and each vertex's edges are stored in an edge array. The insertion only needs to edit a single segment and shift values in the corresponding edge array. Though such a design enables CSR for updates, the cost of shifting values at each insertion restricts the system's throughput to some extent. The second one is using snapshots for graph updates [49]. Any batch changes on the initial graph are written to a new snapshot, and the query on the graph may iterate through multiple snapshots. The snapshots would grow rapidly and take up a lot of space when graphs are frequently updated. Thus, this method only works well with seldom-changed graphs. The third one is using packed memory arrays (PMA) [13] to store edges [73]. Empty slots or gaps are arranged between elements in PMAs, so there is no need to shift existing elements when inserting new elements. However, these gaps need to be rebalanced when the gaps are few or distributed very unevenly, which leads to significant latency when the array is large. Besides, the gaps occupy extra space and fail to embody the space efficiency of CSR.

In addition to the above findings, the concurrent protocol design is critical to the performance of the graph system. Early works only provide parallelization support [18, 53, 62]. In order to improve concurrency efficiency, these systems generally execute updates and queries in turn, which means that updates wait for all queries to finish before updating the graph, and queries wait for updates to finish before accessing the graph. Recent research shows a new concern for concurrent execution of graph updates along with queries [22, 29, 43, 75]. Most studies use the optimistic locking strategy and multiversion concurrency control (MVCC) to allow running graph algorithms on continuously evolving graphs. For example, LiveGraph uses two timestamps per edge to preserve the sequential nature of scans on graphs, and Teseo uses a variant of the optimistic latch for transactions. We noted that such a method could be optimized with ROWEX, which can improve read efficiency.

## 3.3 Opportunities

Given the limitations of existing graph systems, we now identify several opportunities for our design.

**Opportunity One: Replace vertex table with the vEB-tree-like structure to support changes of vertices.** Compared to other indexes for the vertex table, the vEB tree has a remarkable advantage in that its nodes do not merge, split, or copy for updates. Moreover, graph data are usually extracted using graph labeling techniques [30], where unique, densely packed integers are used as vertex identifiers [3, 32, 43] to provide efficiency for storage and computation. Such a feature could benefit from the bitvector representation. However, the scale of bitvectors in the original vEB-tree is only related to the universe size of the given integers. For 64-bit integers, there are 6 levels of bitvectors, and the bottom bivectors should have $2^{64}$ bits in total, resulting in unbearable space consumption. To address it, we borrow the concept of the vEB tree and design a block-based, tree-like vertex index, which has a lower memory footprint and access latency.

**Opportunity Two: Redesign the edge list consisting of bufferblock and sortedblock to strike a balance between space usage and update speed.** The adjacency list offers good support for graph updates, while CSR offers excellent memory access performance. The majority of real-world evolving graphs are sparse [19, 44], where most vertices have relatively low degrees. Thus, for each vertex, we maintain a sortedblock that compactly stores its neighbors like CSR in sorted order and use a bufferblock indexed by an indicator to replace the list structure for fast insertions. The high-degree nodes usually take only a very small part of the vertices, and the overall maintenance cost for them is relatively low, so we do not specially design complicated structures for them.

**Opportunity Three: Develop a lightweight concurrency control protocol to support concurrent operations.** The most distinctive advantage of ROWEX is that readers are never blocked. To achieve this goal, traditional designs of ROWEX usually adopt the exclusive lock, CAS, and read-copy update (RCU) techniques [41]. However, the implementation of RCU will bring additional space and time costs when copying blocks. We note that there has been no prior work on supporting parallelism on vEB trees [5], and the nonexistence of node splitting and merging in the vEB tree provides opportunities to refine ROWEX by editing data in nodes/blocks without RCU. For the range-scan consistency needed for getting neighbors of a vertex, optimistic locking is an ideal solution. Hence, a blend of ROWEX and optimistic locking is well suited to our tree-like data structure.

## 4 DATA STRUCTURE DESIGN

The data structure of Spruce is composed of a vEB-tree-like vertex index and edge storage blocks (bottomblocks), as shown in Figure 3. Spruce's index is composed of a hash table and multi-level bitvectors. The bitvectors are segmented into fixed-sized blocks (topblocks and middleblocks), which are allocated as needed. The vertices' edges, along with their properties, are stored in the bottomblocks that are connected to the middleblocks. For vertex identifiers, Spruce employs 8-byte integers, which is enough to accommodate up to $2^{64}$ vertices and is sufficient to support large-scale graphs commonly used in practice. For edge properties, Spruce supports fixed-length properties (or fixed-length pointers to variable-length properties). In the following subsections, we give the structure of Spruce and its operations in detail.

### 4.1 Vertex Index

Spruce stores the explicit vertex identifiers in its index structure, which map vertices to their connected edges. We divide the 8-byte vertex identifier into 4 + 2 + rest bytes and store them separately in the hash table, topblocks, and middleblocks. By such division, the maximum scale of each level shrinks by the square root: a middeblock can cover at most $2^{16}$ vertices, a topblock can cover at most $2^{32}$ vertices, and the hashtable at the top of our index covers at most $2^{64}$ vertices. Thus, there are $O(lglg(u))$ levels in the index, and dynamic operations (e.g., insertion, deletion) can be performed in $O(lglg(u))$ time. All the blocks in these levels are allocated and deallocated in need, which allows Spruce to dynamically fit in the size of the evolving graph.

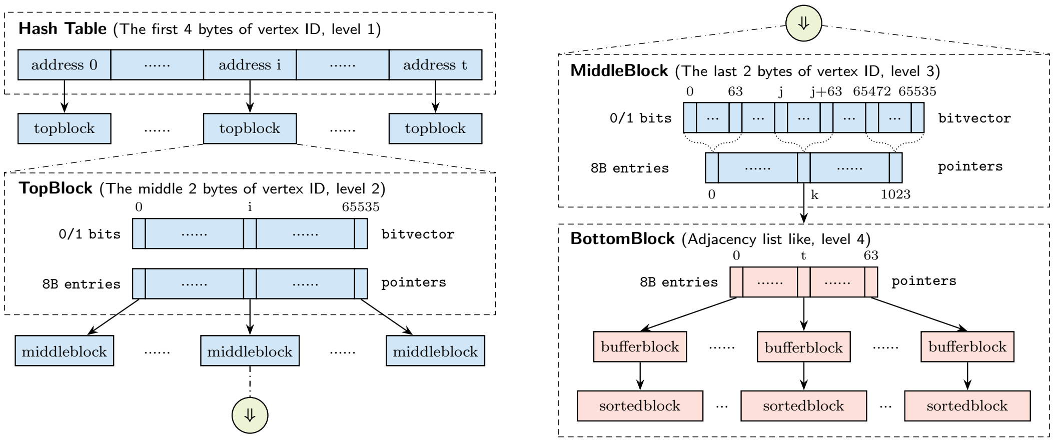

Hash Table (The first 4 bytes of vertex ID, level 1)
address 0
address i
address t
topblock
TopBlock (The middle 2 bytes of vertex ID, level 2)
0/1 bits
bitvector
8B entries
pointers
middleblock
MiddleBlock (The last 2 bytes of vertex ID, level 3)
BottomBlock (Adjacency list like, level 4)
bufferblock
sortedblock

Fig. 3. The data structure of Spruce for 8-byte vertex identifiers. Spruce is based on adjacency lists and is divided into four levels, which are the hash table, topblocks, middleblocks, and bottomblocks, respectively. The first three levels take the function of vertex table, and the last level takes the function of edge lists.

The first level is a hash table. The first 4 bytes of the vertex identifier are hashed to various cells in the hash table, which is the address of the blocks storing the middle 2 bytes (i.e., topblocks). That is, all vertices with the same 4-byte prefix are mapped to the same topblock. We have two reasons for choosing the hash table technology: (1) the vertex identifiers with the same 4-byte prefixes share the same hash address, which can significantly save space, and (2) lookups in a hash table could be accomplished in $O(1)$ time. In Spruce, we use Junction hash [57], which is a concurrent hash map using linear probing for collisions.

The middle 2 bytes are encoded in topblocks. Each topblock contains a bitvector of $2^{16}$ bits denoting the middle 2 bytes and an array storing pointers. Each bit in the bitvector indicates the existence of the 2-byte value and the corresponding pointer to the blocks storing the last 2 bytes (i.e., middleblocks), like the global bitvector of the vEB tree. By using the bitvector, only 1 bit is needed to represent a value. Here, we use 1 for present and 0 for absent. For example, if the bitvector in topbolck is the sequence of 000100..., then the fourth bit indicates that the value 0x0003 of the middle 2 bytes exists, and the fourth pointer points to the corresponding middleblock. The bitvector is designed to accelerate the range-scan of vertices, which is used in the deletion (see Section 4.3 for details) and many graph algorithms (e.g., PageRank, WCC, and LCC). Scanning a bitvector is much faster than scanning an array of pointers due to the fewer memory accesses.

The remaining bytes are encoded in middleblocks, which are similar to topblocks. A middleblock only contains a bitvector having $2^{16}$ bits and a pointer array of $2^{10}$ group pointers. The $j$th group pointer points to the $j$th bottomblock, which contains the aggregated adjacency edges for the vertex identifier values from $(2^6 \cdot j)$ to $(2^6 \cdot (j+1) - 1)$ of the last 2 bytes. In our index, if the $i$th bit of the topblock's bitvector is 1, then the $i$th middleblock exists, and there is at least a 1 in the middleblock's bitvector. That is, the upper level is a summary of the lower level. Note that the number of topblocks shrinks by a factor of $2^{16}$ compared to that of middleblocks, so we do not use aggregation techniques to reduce their space.

## 4.2 Edge Storage

The bottomblock is an adjacency-list-like structure that stores adjacency edges for a group of vertices (corresponding to the last 6 bits of vertex identifiers). A bottomblock is composed of three

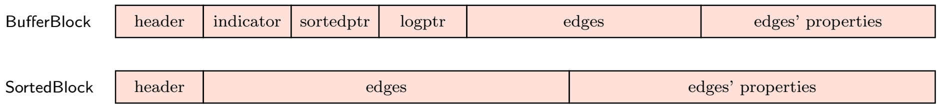


Fig. 4. The data layouts for bufferblock and sortedblock. For a directed edge $(u, v)$, the bufferblock stores the identifier $v$ and the property (optional) of the edge. Here $n = 2^k$ for $k$ in $\{2, 3, ..., lgC\}$, and $m$ is the multiple of $n$.

parts: a pointer array, bufferblocks, and sortedblocks. As shown in Figure 3, the pointer array in the bottomblock pointed by the $k$th group pointer has 64 subpointers, each pointing to a bufferblock which stores adjacency edges for a value of vertex's last 2 bytes in range $[j, j + 63]$. The adjacency information is stored in bufferblock and sortedblock, and the bufferblock is an insertion buffer of the sortedblock to improve writing efficiency.

The bufferblock is designed to temporarily store the edges. The data layout for bufferblock is shown in Figure 4. A bufferblock includes a header, an 8-byte indicator, two pointers (sortedptr and logptr, one to sortedblock and one to logs), and two arrays (one for edges and the other for edges' properties). The header indicates the types of the bufferblock and contains lock-related information (which we will discuss in the next section). If the number of a vertex's edges is less or equal to the maximum capacity $C$ of a bufferblock (we set $C$ to 64 for alignment with the indicator), all the edges are stored in the bufferblock. Otherwise, a sortedblock is needed to support storing more edges. In bufferblock, neighbor identifiers and edge properties (optional) are stored following the same order in two arrays, and the indicator shows the occupancy status of the arrays. We set several base types for the array to allow it to dynamically change with the adjacency edges' scale, and the type is stored in the header. If the $i$th bit of the indicator is 0, then the $i$th cell in the neighbor identifier array is empty. Otherwise, the cell stores corresponding adjacent information. Such representation provides the following advantages:

- Accelerate insertions: In order to find an empty position in the block for insertion, Spruce only needs to do a bitwise operation on the indicator instead of scanning the entire array sequentially.
- Improve read efficiency: When accessing a vertex's adjacency edges, Spruce uses the indicator to directly extract data from non-empty cells, avoiding checking them one by one.

The sortedblock is designed to store edges as an expansion of the bufferblock. A sortedblock includes a header, a neighbor identifier array, and an edge property array. The header indicates the capacity of this block and the number of deleted edges in the block. The adjacency edges (except deleted edges) in the sortedblock are stored consecutively without gaps, and they are sorted from the smallest to the largest by vertex identifiers. When an edge is deleted, Spruce will simply mark the edge invalid. When the edges from a full bufferblock are moved to the sortedblock, a merge sort will be triggered to merge newly added edges. The advantages of sortedblocks are as follows:

- Speed up queries: An edge could be located in logarithmic time using binary search. The ordered sequence is beneficial to basic pattern-matching algorithms like LCC (Local Clustering Coefficient).
- Minimize space usage: Sortedblock compactly stores edges in arrays to efficiently use the space. Compared to the linked list structure, it avoids the extra overhead of pointers and enhances data locality, while maintaining considerable update performance by organizing insertions in batches.

```
Algorithm 1: Insert Edge into Bufferblock
Input: the pointer $b$ to the bufferblock , the directed edge $e = (v_1, v_2, w)$
Output: the pointer $b$ to the bufferblock
1  if ! isfull($b->indicator$, C) then
2      $idx$ = rank$_0$($b \longrightarrow indicator$, 0) ;
3      if $idx > b \longrightarrow size$ then
4          $b$ = expand_buffer($b$) ; // double its size
5      end
6  else
7      $idx$ = 0 ;
8      if !($b \longrightarrow sortedblock$) then
9          $b \longrightarrow sortedblock$ = new sortedblock ;
10     end
11     sort ($b \longrightarrow edges$) ;
12     merge_move ($b \longrightarrow edges, b \longrightarrow sortedblock$) ;
13     clear_bitvector ($b \longrightarrow indicator$) ;
14 set_value($b \longrightarrow edges[idx], e$) ;
15 set_bit($b \longrightarrow indicator, idx$) ;
16 return $b$;
```

## 4.3 Operations in Spruce

In Spruce, undirected graphs are implemented by directed graphs, where each undirected edge is viewed as two directed edges. Generally speaking, dynamic graph operations include insertion, deletion, update, and query of vertices and edges, which we will introduce as follows:

**Vertex Operations.** Vertex operations include *Locate, Insert, Delete* and *GetNeighbours. Locate (Query)* is the fundamental operation in Spruce. It checks whether a vertex $v$ exists in Spruce and returns the pointer to the bufferblock storing its adjacency edges. To locate $v$, Spruce will first use the 4 bytes of its vertex identifier to hash to the address of the topblock. Then, it will continue to use the middle 2 bytes to locate the middleblock by checking the bitvector inside the topblock. A similar process will be done in the middleblock and the bottomblock to finally locate the bufferblock. *Insert* adds a vertex into the index. The execution process for *Insert* is similar to *Locate*. The difference is that *Insert* will set bits in the bitvectors of different levels' blocks according to the vertex identifier. If the block does not exist, Spruce will allocate a new one and store the corresponding pointer in the upper level. *Delete* removes a vertex in the index. It will first locate the vertex position. After that, the vertex's bufferblock and sortedblock (if they existed) are freed, and the subpointer in the bottomblock is set to NULL. Then, the process will check whether the bottomblock is empty through the bitvector. If true, the bottomblock is freed, and the corresponding bit and the group pointer in the middleblock are cleared. Finally, similar maintenance will be done for the topblock and the hash table. *GetNeighbours* accesses a vertex's adjacency edges and gets the vertex's all adjacency vertices' identifier. It will first locate the vertex position, then traverse the bufferblock and sortedblock to extract all valid neighbor vertices' identifiers.

**Edge Operations.** Edge Operations include *Insert, Update* and *Delete. Insert* $(v_1, v_2, w)$ adds a new edge into the bufferblock. It will first locate vertex $v_1$'s bufferblock, then get an empty position through the indicator. If the bufferblock is full, the following operations are executed to generate spaces: If the size of the bufferblock is smaller than $C$, the bufferblock will be expanded to twice the

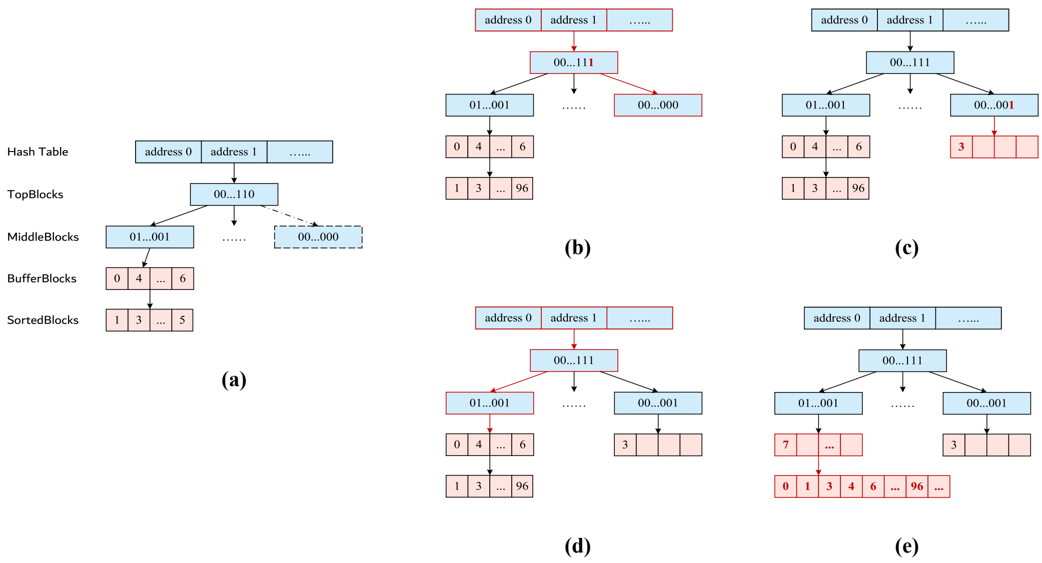


Fig. 5. A running example of Spruce (auxiliary structures and entries are ignored) : (a) the initial state, (b) ~(c) insertion of edge $\mathbf{(0,3)}$, (d) ~ (e) insertion of edge $\mathbf{(2^{17},7)}$. A vertex's identifier is broken into three parts to locate its bufferblock. Here, 0, 0, 0 is used for vertex 0, and 0, 2, 0 is used for vertex $2^{17}$.

original size. Otherwise, Spruce will merge the bufferblock's edges into the sortedblock, then empty the bufferblock. Finally, the neighbor identifier $v_2$ and edge property $w$ will be inserted into the arrays of the bufferblock. The pseudocode for modification on bufferblock is shown in algorithm 1. *Update* $(v_1, v_2, w)$ updates an edge's property. After locating the vertex $v_1$'s position, it will check the non-empty cells in the bufferblock to find the edge $(v_1, v_2)$. If the edge is not in the bufferblock, a binary search will be performed on the sortedblock to find it. Then, the corresponding property $w$ for the edge will be updated. *Delete* $(v_1, v_2)$ delete an edge from the bufferblock or sortedblock. It uses the same approach as the update to find the edge. If the edge to be deleted is in the bufferblock, the bit of the indicator corresponding to its position is set to 0. If the edge to be deleted is in a sortedblock, the value in its storage position is marked as invalid.

A running example of insertion is shown in Figure 5. To insert edge $(0, 3)$, Spruce first needs to locate its bufferblock. Since the bufferblock's parent middleblock does not exist, Spruce allocates a new middleblock and sets the corresponding bit to 1 in the bitvector of the topblock (Figure 5(b)). Then, it allocates a new bufferblock and sets the corresponding bit to 1 in the bitvector of the middleblock. Finally, it insert 3 directly into the bufferblock (Figrue 5(c)). To insert $(2^{17}, 7)$, Spruce locates $2^{17}$'s bufferblock (Figrue 5(d)). Since the bufferblock is full, the edges in the bufferblock are merged into the sortedblock to spare spaces. Then, the value 7 is inserted into the bufferblock (Figrue 5(e)).

## 4.4 Discussions

In this subsection, we specify the following advantages that distinguish Spruce from other dynamic graph data structures:

**Operation Complexity.** We analyze the time complexity of fundamental operations on Spruce and compare them with previous state-of-the-art work in Table 1. For vertex operations, Spruce's time complexity is second only to Stinger, which uses a single hash table to index all vertices. For

Table 1. Time complexity of fundamental operations. $\boldsymbol{n}$ is the number of vertices, $\boldsymbol{u}$ is the size of integer universe and $\boldsymbol{d}$ is the average degree of vertices.

| Operation | Stinger | Teseo | Sortledton | Spruce |
|---|---|---|---|---|
| locate_vertex | $O(1)$ | $O(lg(u))$ | $O(lg(n))$ | $O(lglg(u))$ |
| insert_vertex | $O(1)$ | $O(lg(u))$ | $O(lg(n))$ | $O(lglg(u))$ |
| delete_vertex | $O(d)$ | $O(lg(u)+d)$ | $O(lg(n)+d)$ | $O(lglg(u)+d)$ |
| insert_edge | $O(d)$ | $O(lg(u)+lg(d))$ | $O(lg(n)+lg(d))$ | $O(lglg(u)+d)$ |
| delete_edge | $O(d)$ | $O(lg(u)+lg(d))$ | $O(lg(n)+lg(d))$ | $O(lglg(u)+lg(d))$ |
| update_edge | $O(d)$ | $O(lg(u)+lg(d))$ | $O(lg(n)+lg(d))$ | $O(lglg(u)+lg(d))$ |
| get_neighbours | $O(d)$ | $O(lg(u)+d)$ | $O(lg(n)+d)$ | $O(lglg(u)+d)$ |

edge operations (which need first to locate the edges' corresponding vertices), Spruce demonstrates superiority over other methods in terms of edge updates and deletion. For insertions, the cost for inserting an edge into a bufferblock and merging a bufferblock into a sortedblock is $O(1)$ and $O(d)$ ($d$ is the degree of the vertex), respectively. As real-world graphs are sparse [19, 44], the merge operations are not frequently triggered for most vertices.

**Memory consumption.** Spruce uses the vEB-tree-like index and represents it by blocks sharing the same prefixes, which reduces the levels and saves space compared to the original vEB-tree. The most significant difference of this index compared to other graph indexes is that in Spruce, the vertex identifiers are separated and stored in different levels, sharing prefixes to reduce memory consumption, while others need to store the whole identifiers. Besides, the compact data structures (bitvector and sortedblock) help further reduce memory consumption. We note that the space savings also result in performance improvement for most operations due to performing fewer memory accesses.

Graph data are extracted using graph labeling techniques, which map vertex in the graphs to a set of integers [30] (vertex identifiers). Generally, these integers are nearly sequentially numbered to provide efficiency for storage and computation [3, 32]. We define the *gap* as the difference of two contiguous integers in an ordered set. Let graph $\mathcal{G} = (V, E)$, $n_v = |V|$, $n_e = |E|$, and $d_i$ be the number of edges connected to vertex $i$. Let $U = \{0, 1, 2, ..., u-1\}$ be the set of integers in the universe and $G, B$ be the average gap between contiguous vertex identifiers in $V$ and the size of a bitvector in a block, respectively. For vertex identifier set $V = \{v_1, v_2, ...\}$ where $v_i < v_{i+1}$, $G = (\sum_{i=1}^{n_v-1}(v_{i+1} - v_i))/n_v = (v_{n_v} - v_1)/n_v$. Thus, $Gn_v$ denotes the scale of bits needed to store all keys within the range $[v_1, v_{n_v}]$. In the middleblock level, we store bits in blocks of bitvectors, so there should be no more than $n_v$ blocks. Therefore, the total space of middleblock level considering the block size is $O(min(B\lceil Gn_v/B\rceil, Bn_v)) = O(min(Gn_v, Bn_v))$. Since the size of upper levels shrinks by square root, the upper bound of the total space complexity is $O(min(\sum_{i=0}^{lglg(u)-1}(Gn_v)^{2^{-i}}, Bn_v lglg(u)))$. Ignoring lower-order terms, it is $O(min(Gn_v, Bn_v lglg(u)))$. For the common case when a large amount of 8-byte vertex identifiers in graphs are sequentially numbered ($G$=1~2), $Gn_v$ is far less than $Bn_v lglg(u)$, so the space complexity of Spruce is about $O(Gn_v)$ and the consumption in practice is low. According to Spruce index, when $G$=1~2, only 1~2 bits of the bitvector in middleblock level are used to represent a vertex identifier, a bit of the bitvector in topblock level can be shared by $2^{15}$~$2^{16}$ vertex identifiers and a slot of the hash table is shared by at most $2^{31}$~$2^{32}$ vertex identifiers. The worst case occurs when the gap between every two vertex identifiers is greater than $u^{1/2}$, where each block (excluding the hash table) has only one element. However, such a case is rare in practice and can be resolved by relabeling them [43]. For edge storage, the buffer block and sorted block dynamically adjust their size according to the number of edges stored in them, so vertex $i$ uses

$O(d_i)$ space to store edges. Since $n_e = \sum_{i=1}^{n_v} d_i$, we can easily deduce that the total space for storing edges in graph $\mathcal{G}$ is $O(n_e)$.

**Efficiency of data access.** Compared to other methods, Spruce accesses fewer blocks when getting adjacency edges. For example, to find a directed edge and then delete it, Spruce only needs to access the source vertex's bufferblock and sortedblock. Stinger and GraphOne need to traverse $O(d/B)$ ($d$ and $B$ references to the vertex's degree and block capacity, respectively) blocks in the linked list, while Teseo and Sortledton need to access $O(lg(d/B))$ blocks. Thus, the compact storage scheme for bufferblocks and sortedblocks provides better data locality when accessing memory, which further benefits its performance.

## 5 CONCURRENCY SUPPORT

The method of concurrency control is another important factor affecting the performance of a graph system. An ideal concurrency protocol for a graph system should have good scalability and space efficiency while well satisfying the requirement of running updates concurrently with computations. For this, we take advantage of the vEB tree's feature of no splitting and merging of nodes, present a lightweight protocol using a combination of ROWEX and optimistic locking, and further introduce MVCC to support transactions concurrent operations.

### 5.1 General Idea

The core architecture of Spruce consists of two main components: the vertex index and the edge storage blocks. We have implemented concurrency control mechanisms for two components independently. For the vertex index, as discussed in section 4, Spruce's index is designed such that there's no need for merging, splitting, or resizing of blocks. The operations only relevant to the index are block allocation and deallocation. Based on these, we introduced the *Improved ROWEX*, which is an enhancement of ROWEX, using only exclusive locks and atomic operations for write procedures, thereby eliminating the need for the RCU technique. The underlying principle of Improved ROWEX is simple: when a block needs modification, the writer acquires only that block's lock (as opposed to acquiring locks for both the block and its parent). The modification is executed through atomic operations. During this process, an additional verification checks that the target data hasn't been altered by another process after the lock's acquisition, ensuring concurrency correctness. If this verification fails, the writer releases the lock and restarts the operation for the block. In our designed data structure, this verification does not require a timestamp.

For edge storage blocks, there is a requirement to ensure the correctness of neighbor queries, which works similarly to the range scan. To achieve this, we implement an optimistic locking strategy. However, a limitation of this strategy is that when a reader accesses frequently modified data, it might undergo numerous restarts. To address this, we've refined the lock to be upgradeable. Specifically, if the number of restarts surpasses a predetermined threshold, the reader updates its optimistic lock to an exclusive one. To further enhance the system's capabilities, we've integrated version logs, enabling transactions in Spruce.

### 5.2 Concurrency Protocol

In the topblock and middleblock, every 64 bits in the bitvector as well as their corresponding pointers share a single exclusive lock. In the bottomblock, the subpointer array uses the same lock with the pointer to the bottomblock, and each vertex's bufferblock and sortedblock has an optimistic lock, which includes an exclusive lock and a timestamp.

**Concurrency Protocol for Vertex Index.** We first discuss the protocol for writers. In Spruce, only insertion and deletion can modify the index structure. The protocol for modifying the topblock

**Algorithm 2:** Concurrently Insert Value into MiddleBlock

```
Input: the pointer $midblock$ to the middleblock, the value $sufidx$ (last 2 bytes of vertex id)
        to be inserted.
Output: the pointer $bufblock$ to the value's corresponding bufferblock.
1  acquire($midblock \rightarrow locks[sufidx/64]$) ;
2  if $midblock \rightarrow obsolete\_flag$ then
3      release($midblock \rightarrow locks[sufidx/64]$) ;
4      return NULL ; // restart from upper level
5  end
6  if $!(midblock \rightarrow bitvector.\text{get}(sufidx))$ then
7      if $!(midblock \rightarrow pointers(sufidx/64))$ then
8          $botblock$ = new bottomblock ;
9          $midblock \rightarrow pointers[sufidx/64] = botblock$ ;
10     else
11         $botblock = midblock \rightarrow pointers[sufidx/64]$;
12     $bufblock$ = new bufblock ;
13     $botblock \rightarrow pointers[sufidx\%64] = bufblock$ ;
14     $midblock \rightarrow bitvector.\text{set}(sufidx)$ ;
15 else
16     $bufblock = (midblock \rightarrow pointers[sufidx/64]) \rightarrow pointers[sufidx\%64]$ ;
17 release($midblock \rightarrow locks[sufidx/64]$) ;
18 return $bufblock$;
```

and middleblock is similar, and we use the middleblock as an example. As the pseudocode shown in algorithm 2, the modification process on a middleblock during the insertion is performed in 4 steps:

(1) Acquire the lock based on the vertex identifier.
(2) Check whether the middleblock is obsolete. If the block is invalid (has been removed before being locked), the insertion restarts from the upper level; else, continue the process.
(3) Reread the bitvector to check whether the bufferblock to be inserted in the lower level already exists. If the block does not exist, allocate and insert a new block and edit associated pointers and bitvector. In this process, the pointer array in the bottomblock may need to be allocated or edited. If the block already exists, the pointer to the bufferblock is got directly.
(4) Release the lock and continue the insertion process in the bufferblock.

The modification on a middleblock during the deletion is triggered when a vertex is deleted, which is performed differently from insertion:

(1) Acquire the lock based on the vertex identifier.
(2) Edit the bitvector and pointer. If the pointer array in the bottomblock is empty, the bottomblock is marked obsolete.
(3) Release the lock.
(4) Check the whole bitvector in the block. If all bits are 0s, acquire all the locks in the block and go to step 5. Otherwise, the deletion will be finished. The locks should be acquired sequentially for each writer to avoid deadlock.
(5) Recheck the obsolete flag and the whole bitvector in the block. If the block is already marked as obsolete, then the process completes directly. If all bits in the bitvector are 0s, clear the block and mark it as obsolete, then continue the maintenance process in the upper level.

Otherwise, the deletion completes because the bitvector indicates that new data has been inserted between step 3 and 5.

For readers, they simply read data atomically, ignoring the locks. A similar check of the obsolete flag is needed to ensure concurrency.

**Concurrency Protocol for Edge Blocks.** To modify a buffer-block or sortedblock, the writer will first gain the exclusive lock before accessing the data, then edit the data without checking the timestamp. The timestamp will be changed twice during this process: one before the modification and the other after the modification. For readers, each holds a restart counter for the read. When accessing a vertex's bufferblock and sortedblock, the reader checks the timestamps before and after the read. If the timestamps are not changed, then the read is successful. Otherwise, the reader restarts reading these blocks and the restart counter plus one. If the restart count exceeds the pre-defined threshold, the reader will acquire the exclusive lock for reading.

## 5.3 Correctness of Improved ROWEX

This subsection shows the correctness of our improved ROWEX on Spruce. Here we take the middleblock as an example, and the proofs of ROWEX on the bottomblock and the topblock are similar. Assume that two threads $T_1$ and $T_2$ concurrently access the same middleblock. If they both perform *Locate* (Lookup), no conflict occurs and the result is obviously correct. If they both perform *Insert* and need to set bits in the middleblock's bitvector and corresponding pointers, the exclusivity is guaranteed by the lock, and the recheck process in step (3) of *Insert* makes sure that the former inserted value would not be overwritten by the latter.

If $T_1$ and $T_2$ both perform *Delete* on the middleblock, the cases are similar to that of *Insert*. The only difference is that after deleting the value in the bitvector and the pointer array, the thread needs to decide whether the block is empty and should be deleted at step (3) of *Delete*. If the block is empty, the thread will acquire all locks in that block and delete the whole block. If $T_1$ and $T_2$ both have detected that the block is empty, the one who gets the locks later will find the obsolete flag true and quit the delete process since the block has already been deleted.

Then consider that $T_1$ performs *Delete* and $T_2$ performs *Insert*. If the target delete value is not the only value in the middleblock, the cases are similar to the above; else, the following cases may occur:

- **Case 1: *Delete* completes before *Insert*.** In this case, the middleblock is marked obsoleted before *Insert*. $T_2$ will find that the obsolete flag is true and restart *Insert* from the upper level to get a new middleblock for insertion.
- **Case 2: *Insert* completes before *Delete*.** After $T_2$ inserts the value, $T_1$ will find that the bitvector is not empty at step (4) or step (5) of deletion and finish *Delete* without clearing the whole middleblock.

When $T_1$ performs *Locate* and $T_2$ performs *Insert*/*Delete*, the cases are more complicated. The cases for *Insert* and *Delete* are similar. Due to the page limit, we only discuss the cases for concurrent *Locate* and *Delete* on a middleblock. *Delete* firstly set the corresponding bit in the bitvector to 0 (then modify the pointer in the pointer array) while *Locate* reads the bitvector and the pointer sequentially. If $T_2$ has deleted the whole middleblock before $T_1$ reads it, $T_1$ will find the block obsoleted and restart reading from the upper level to get the latest result. For other cases, assume that $T_1$ attempts to read $i$ and $T_2$ attempts to delete $j$. If $i \neq j$, the correctness is obvious since the operations are done atomically. Otherwise, the possible interleavings of $T_1$ and $T_2$ are listed as following:

- **Case 1: $T_2$ modified the bitvector before $T_1$ reads it.** $T_2$ has deleted the value before $T_1$ accesses it, thus $T_1$ will return false, indicating that the value does not exist in the index.

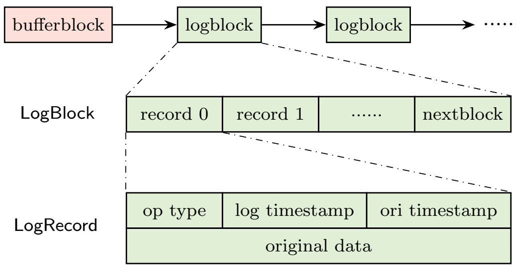


Fig. 6. Version design of Spruce.

- **Case 2: $T_2$ modified the bitvector after $T_1$ reads it.** In this case, $T_1$ may get an empty pointer or a pointer to an obsoleted bottomblock, which will trigger $T_1$ to restart reading the middleblock to get the latest correct result.

## 5.4 Version Management

Many real-time graph analytics need to access a consistent snapshot of the graph using transactions. In Spruce, MVCC is supported to enable transnational concurrent operations. One of the goals of Spruce is to save space. Thus, we gather all the versioned edges of a vertex in a log list structure rather than maintaining a version list for each edge. As shown in Figure 6, each vertex has associated a pointer in its bufferblock, which points to a linked list of logblocks storing the versions. Each version record in the logblock is composed of 4 fields: (1) `op_type`, which indicates the operation type on the data (e.g., edge deletion/update), (2) `log_timestamp`, which indicates the time when this operation was committed, (3) `ori_timestamp`, which indicates the last modification time before this record, (4) `original_data`, which stores the versioned edge/vertex data corresponding to the `ori_timestamp`. Note that the initial insertions of edges or vertices are not recorded in logblocks. Instead, we store them along with their timestamp directly in the bufferblock and sortedblock to save space.

## 6 EVALUATION

**Experimental Setup.** We do our experiments on a dual-socket machine equipping Intel Xeon Gold 5120 @ 2.20GHz processors and 500GB DDR4 RAM. Each CPU has 19.25 MB L3 Cache, 14 cores, and supports at most 28 threads. All the codes were complied with GCC 11.3.0 and the O3 parameters.

**Graph Data.** A variety of graphs are used in our experiment to evaluate the time and space performance, as shown in Table 2. *Com-Livejournal* is an undirected graph of the LiveJournal social network [16]. *DOTA League* is a weighted graph representing the relationship between many game entities [33]. *Orkut* is a relationship graph of ground-truth communities from the Orkut social network. *Yahoo-Songs* is a bipartite person–song rating network, which contains over 250 million ratings performed by over 1 million users [60]. The *Graph500-X* and *Uniform-X* datasets are undirected power-law graphs and uniform-law graphs, respectively. *Com-friendster* is an undirected graph of the Friendster social network [45].

**Graph Benchmark**. We run the experiment for Spruce and its competitors based on the GFE (Graph Framework Evaluation) Driver, a C++ driver to evaluate updates and analytics on dynamic structural graphs [42]. For graph algorithms, we implement kernels from LDBC Graphalytics [34], which is an industrial-grade benchmark including several algorithms that cover real-world

Table 2. Analysis of graph datasets. "K", "M", "B" stand for "thousand", "million" and "billion", respectively. The "Top 0.1% Degree" means that this degree surpasses 99.9% of all the other vertices' degrees.

| Graph Dataset | \|V\| | \|E\| | Avg. Deg. | Max. Deg. | Top 0.1% Deg. | Avg. Gap | Max. Gap | Size(GB) |
|---|---|---|---|---|---|---|---|---|
| com-livejournal | 4M | 34M | 17.35 | 14815 | 473 | 1.010 | 86 | 0.47 |
| dota-league | 61K | 51M | 1663.24 | 17004 | 12988 | 5.194 | 720 | 0.76 |
| orkut | 3M | 117M | 76.28 | 33313 | 1174 | 1.000 | 12 | 1.6 |
| yahoo-songs | 1.6M | 256M | 513.04 | 468425 | 23751 | 1.000 | 1 | 3.0 |
| graph500-24 | 9M | 260M | 58.70 | 406416 | 7019 | 1.891 | 18 | 4.1 |
| uniform-24 | 9M | 260M | 58.70 | 103 | 84 | 1.500 | 2 | 4.1 |
| graph500-26 | 33M | 1B | 64.13 | 1003338 | 5340 | 2.046 | 21 | 17.5 |
| uniform-26 | 33M | 1B | 64.13 | 111 | 90 | 1.500 | 2 | 17.5 |
| com-friendster | 65M | 2B | 55.06 | 5214 | 1612 | 1.903 | 1874 | 30.1 |

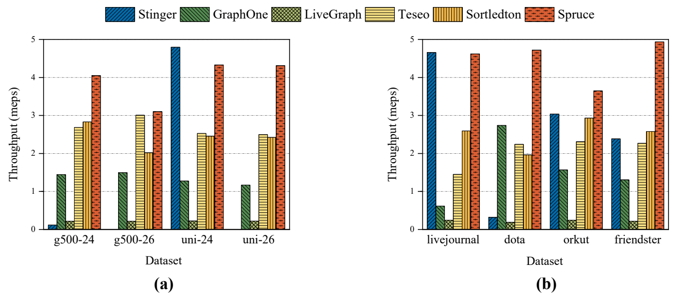


Fig. 7. Insertion with random access pattern on (a) distribution-specific graphs and (b) real-world graphs.

workloads: breadth-first search (BFS), single-source shortest paths (SSSP), PageRank (PR), local triangle counting (LCC), etc.

**Competitors.** We choose the recent state-of-the-art dynamic graph data structures for comparison: Stinger [24], GraphOne [38], LiveGraph [75], Teseo [43] and Sortledton [29]. LiveGraph, Teseo, and Sortledton are transactional graph systems, while Stinger and GraphOne only provide parallelization support for operations.

## 6.1 Dynamic Operations

In this subsection, we evaluate the throughput of dynamic operations (including insertion and deletion) on a variety of graphs with multiple threads. Note that some methods did not finish the experiment within the given time, so their results are not depicted in the provided figures.

*6.1.1 Insertions.* In this experiment, we randomly insert all edges (along with vertices) with weights (they are generated randomly if the raw data does not contain weights) from the graph dataset to an empty graph structure and report the average throughput. Figure 7(a) shows the insertion throughput on graphs with certain distributions. For power-law graphs, Spruce performs best among all the methods. For uniform-law graphs, though Stinger achieves the highest throughput on *unifrom-24*, it cannot load *uniform-26* and *graph500-26* within the time limit. This is because Stinger performs a linear search to check whether the edge exists, which leads to unbearable running time for very large graphs. Spruce gets first place and second place on *uniform-26* and *uniform-24*, respectively. We note that support for transactions does have performance overhead, which is an

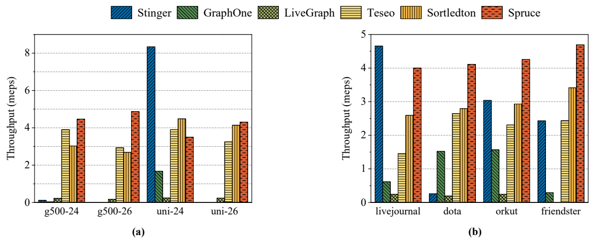


Fig. 8. Deletion with random access pattern on (a) distribution-specific graphs and (b) real-world graphs.

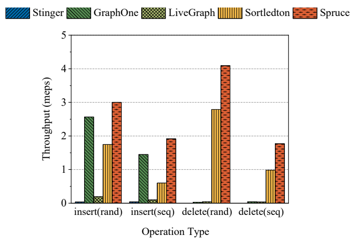


Fig. 9. Comparison of sequential access pattern and random access pattern.

important reason why transactional structures like Spruce and Sortledton have lower throughput on some of the datasets (e.g., *uniform-24*) compared to non-transactional structures (Stinger).

Different from graphs with specific distributions, real-world graphs present a more practical and complex form of data distribution. Therefore, we also test the performance of operations on real-world graphs. Figure 7(b) depicts the throughput for insertion on four different real-world graphs. The results demonstrate that Spruce has greater adaptability to real-world graphs, whose insertion speed is up to 7.5× faster than GraphOne, 3.2× faster than Teseo, and 2.4× faster than Sortledton. LiveGraph inserts no more than 1 million edges per second, which owes much to its intricate transaction design.

Compared to some recent state-of-the-art techniques, such as the fat tree in Teseo and multilevel vectors in Sortledton, our index doesn't involve merging, splitting, or resizing of nodes and has a better time complexity, enhancing the dynamic performance of the vertex index. The design of bufferblocks and compact storage scheme for sortedblocks also provides edibility and good data locality for edge modifications, which is suitable for real-world graphs where a majority of vertices tend to have lower degrees (as indicated in Table 2).

*6.1.2 Deletions.* The deletions experiment starts with the data structure storing the input graphs and deletes all the edges in the graph in random order.

Figure 8(a) shows the deletion throughput on distribution-specific graphs for all methods. For power-law graphs, Spruce is up to 1.7× faster than Teseo and 1.9× faster than Sortledton. For uniform graphs, Teseo and Sortledton are slightly faster than Spruce on *uniform-24* while slower on *uniform-26*. Stinger does not support edge versioning. On small, uniform graphs, the time required for a linear search to find the edges that need to be deleted and the time for maintaining its hash index are both relatively short. Thus, it outperforms all other competitors on *uniform-24*.

Figure 8(b) reports the deletion throughput on the real-world graphs. Spruce outperforms most of the competitors with the throughput of 4.0 ∼ 4.6 meps, which is 0.9× ∼ 15.8× of Stinger and 1.4× ∼ 1.5× of Sortledton. Other competitors show relatively low throughput, less than 3 meps.

It is noted that in the deletion experiment, most of the modification operations are concentrated at the adjacency edges of vertices, and an artful vertex index could provide quick access to these vertices. GraphOne and LiveGraph do not perform well in this experiment due to the lack of an efficient index. The performance of Stinger on different datasets varies widely because it leaves the adjacency edges unsorted, and the time cost for linear search in linked lists mainly depends on the degree of vertices. Spruce outperforms other methods thanks to the high-performance index and compact storage scheme for sorted edges.

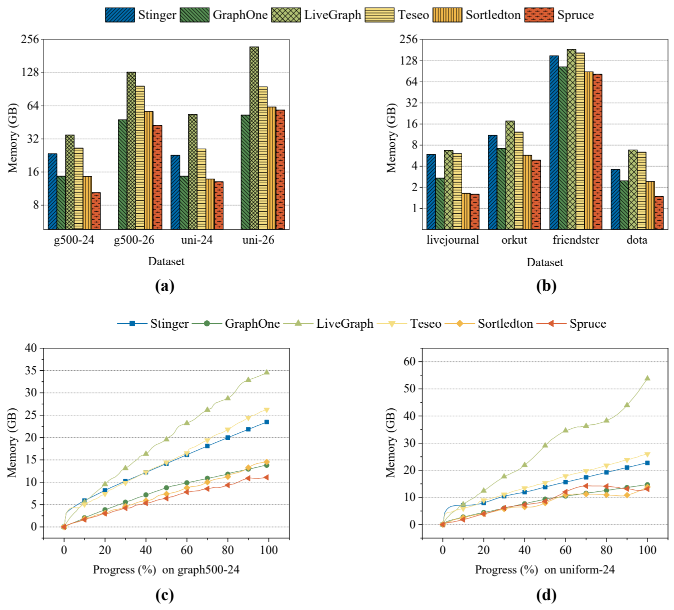


Fig. 10. Memory consumption of different methods. (a) shows the memory consumption on graphs with certain distributions. (b) shows memory consumption on real-world graphs. (c) and (d) report the memory footprint of the system while experiments on *graph500-24* and *uniform-24* progressed, respectively.

*6.1.3 Sequential Access Pattern.* A number of real-world dynamic graphs show a sequential access pattern. That is, the modification of edges within a certain period of time relates to the same group of vertices and exhibits a strong temporal locality. In this experiment, we evaluate sequential operation performance on *Yahoo-Songs*, which presents a strong sequential access pattern (vertices and edges being inserted/deleted in sequential order of vertex identifiers). In the experiment, the edges are inserted/deleted continuously with no sleep time, and we report the average throughput of the whole process.

Figure 9 shows the result for evaluation in the form of a bar chart. GraphOne, LiveGraph, Sortledton, and Spruce are able to finish all operations within the time limit. The result demonstrates that Spruce takes the first place among all the methods when executing sequential operations. For sequential insertion, Spruce reaches 3.0 meps per second, which is 1.9× faster than GraphOne. For sequential deletion, Spruce reaches 1.8 meps per second, which is 1.8× faster than Sortledton. Compared to random execution, all methods experience different degrees of performance degradation with sequential execution. The slowdown is mainly caused by competition for accessing resources, especially for methods using the vertex-centric locking strategy such as Livegraph and Sortledton. GraphOne uses a global edge array to arrange insertions in batches, thus gaining considerable throughput for sequential insertion. Even though Spruce uses one lock per vertex, the buffer design undercut the performance degradation when inserting edges.

## 6.2 Memory Consumption

Here, we evaluate the amount of physical memory used to store dynamic graphs for different methods. The *Resident Set Size (RSS)* in `/proc/[pid]/statm` [65] is used to trace the memory footprint of the program or application. In the following, we first analyze the memory consumption for loading the graphs and then evaluate the memory footprint during the loading process.

Table 3. Performance of different graph algorithms. For baseline, we report the explicit execution time. For all the methods, we report the slow-down over the baseline. The "–" means the graph algorithm does not finish within the time limit.

| Graph | Method | BFS | PR | SSSP | WCC | LCC | Graph | Method | BFS | PR | SSSP | WCC | LCC |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| graph500-24 | baseline | 0.54s | 2.11s | 5.56s | 8.24s | 81.31s | uniform-24 | baseline | 0.51s | 3.14s | 6.39s | 1.12s | 3.29s |
| | Stinger | 1.75× | 3.30× | 1.75× | 4.09× | – | | Stinger | 2.27× | 2.87× | 2.12× | 2.31× | 18.45× |
| | GraphOne | 5.27× | 3.67× | 1.68× | 2.17× | – | | GraphOne | 11.45× | 3.37× | 2.46× | 55.21× | 19.70× |
| | LiveGraph | 5.75× | 3.67× | 1.69× | 2.35× | – | | LiveGraph | 4.27× | 3.21× | 3.02× | 2.25× | 12.55× |
| | Teseo | 4.19× | 3.52× | 2.47× | 3.35× | 5.34× | | Teseo | 2.65× | 3.18× | 3.33× | 2.32× | 22.46× |
| | Sortledton | 2.80× | 2.26× | 1.34× | 3.55× | **1.72×** | | Sortledton | 1.91× | 2.42× | 2.50× | 1.48× | **2.05×** |
| | Spruce | **1.45×** | **1.80×** | **1.07×** | **1.91×** | 2.90× | | Spruce | **1.36×** | **1.55×** | **1.43×** | **1.39×** | 5.50× |
| orkut | baseline | 0.13s | 0.54s | 1.72s | 0.16s | 2.80s | dota-league | baseline | 0.12s | 2.04s | 0.48s | 0.33s | 61.41s |
| | Stinger | 3.06× | 5.58× | 2.16× | 6.99× | 63.89× | | Stinger | 8.90× | 4.80× | 2.61× | 6.28× | 19.13× |
| | GraphOne | 40.46× | 12.23× | 4.70× | 161.54× | 33.06× | | GraphOne | 383.40× | 24.70× | 10.52× | 133.84× | 6.14× |
| | LiveGraph | 5.02× | 4.91× | 2.21× | 4.35× | 31.70× | | LiveGraph | 9.94× | 2.09× | 1.37× | 2.51× | 7.92× |
| | Teseo | 3.81× | 3.32× | 2.38× | 5.33× | 196.1× | | Teseo | 5.19× | 1.52× | 1.25× | 2.45× | 8.57× |
| | Sortledton | 3.77× | 3.78× | 2.13× | 5.14× | **2.63×** | | Sortledton | 3.63× | 1.23× | **0.92×** | **1.95×** | 1.04× |
| | Spruce | **1.69×** | **2.97×** | **1.23×** | **3.04×** | 5.22× | | Spruce | **3.04×** | **1.01×** | 1.11× | 2.39× | **1.01×** |

Figure 10(a) and Figure 10(b) give the results of memory consumption for loading the whole graph. The y-axis is the memory consumption shown in the logarithmic scale, and the x-axis represents different graph datasets. We note that the physical memory used by all systems are higher than the raw data size because (1) an undirected edge is stored as two directed edges, (2) random 8-byte weights are generated during the insertion process (3) additional structures such as timestamp and pointers are taken into consideration. For graphs with certain distributions (Figure 10(a)), Spruce uses the least memory to dynamically load the graphs. Compared to the best of competitors (Sortledton), Spruce saves about 5.5% ~ 28.7% of memory. For real-world graphs (Figure 10(b)), this gap is further widened: Spruce's memory consumption is up to 38.5% lower than Sortledton and 41.0% lower than GraphOne, respectively. LiveGraph suffers from severe space overhead because it stores two timestamps per edge. Although Teseo implements the Compressed Sparse Row (CSR) to store graphs, the memory consumption is still high due to the extra cost of gaps in packed memory arrays. Spruce surpasses other methods owing to its compressed dynamic index and compactly stored edges.

Figure 10(c) and Figure 10(d) record the variation of memory footprint with the progress of insertions on two datasets of graphs having the same scale: *graph500-24* and *uniform-24*. All of the methods have an approximately linear relationship between the memory consumption and the edges stored in the system, while Spruce gains the lowest slope on average. The curve of LiveGraph rises more sharply on *uniform-24* than *graph500-24* because when a block is full, LiveGraph will copy its data to a new block of twice its current size [75], which is unsuitable for uniform-law graphs.

## 6.3 Graph Analytics

Now we evaluate the read performance of all the methods on different graphs using the benchmark kernels of LDBC Benchmark Suite mentioned at the beginning of section 6. BFS, SSSP, PR, WCC, and LCC are included to test the performance for various data access patterns. Table 3 reports the run times of these graph algorithms. We use CSR as the baseline, and the run times of the other methods are normalized to it. For each algorithm, we took the average run time of 5 trials, and the best results excluding the baseline in the table are marked in bold font.

**Breath-first Search and Single Source Shortest path.** For all methods, we test the basic BFS algorithms without direction optimization [11] and delta-stepping SSSP [52]. Both algorithms show strong random access patterns of vertices and sequential access patterns of adjacency edges. As

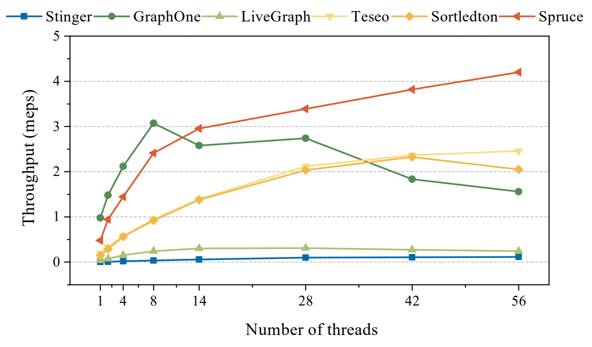


Fig. 11. The multi-thread scalability of different methods on *graph500-24*.

shown in table 3, Spruce gets the best run time among all the methods, except for the second best of SSSP on *dota-league*, which owes to the fast locating of vertices in the index and good data locality of neighbors.

**PageRank and Weakly Connected Components.** The data access patterns of PR and WCC are similar, where all the vertices are sequentially accessed. The run times of PR on Spruce are about 1.01× ∼ 2.97× of the baseline, which outperforms all other methods. For WCC, Spruce gives the best results on all the datasets except the second best results on *dota-league* (3.17× baseline) following Sortledton, which indicates that for index on small graphs, the superiority of $O(\lg \lg u)$ time over $O(\lg u)$ time is not significantly manifested.

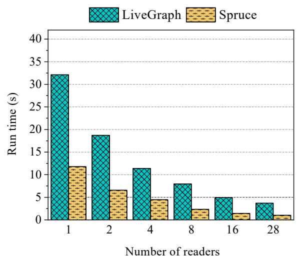
**Local Clustering Coefficient.** For Spruce, we implement the same LCC algorithm as Teseo and Sortledton, which requires that the adjacency edges of vertices are sorted. Table 3 shows that Sortledton and Spruce are significantly better than other methods on all datasets. Specifically, Spruce's run time ties the first place on *dota-league*, while Sortledton takes the first place on other graph datasets. The execution times of other methods have obvious gaps behind those of Sortledton and Spruce. Spruce is slower than Sortledton on 3 of 4 datasets, which is mainly caused by the disorder of edges in bufferblocks, and sorting them in LCC affects the performance of our method.

## 6.4 Concurrent Performance

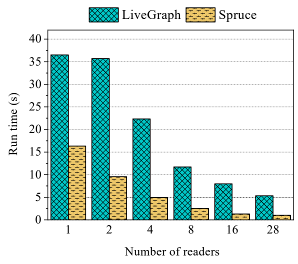
This subsection reports the concurrent performance of dynamic graph storage structures, including the scalability and the performance of concurrent read-write workloads.

*6.4.1 Multi-thread Scalability.* To evaluate multi-thread scalability, we test the insertion performance with 1 ∼ 56 write threads on *graph500-24*. Figure 11 depicts the multi-thread performance for all methods. Spruce, Stinger, and Teseo can scale up to 56 threads. Spruce scales better than Teseo and Stinger at a peak throughput of 4.2 meps because its index uses a more lightweight concurrency protocol, where no read-copy updates are needed when the graph is evolving. Sortleton and LiveGraph only scale up to 42 and 28 threads due to the fierce competition from vertex-centric locks. GraphOne merely scales to 8 threads, and its performance slows down significantly above 28 threads, which is due to the imbalanced workload between workers.

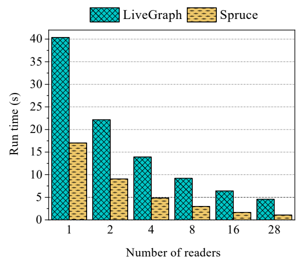
*6.4.2 Concurrent Read and Write.* In this part, we evaluate the performance of concurrent read-write workloads. The setup of the experiment is the same as Teseo [43] as follows. For the writers, they perform about $10 \cdot |E|$ update operations, where $|E|$ is the number of edges in the original graph. The first 10% operations load the graph, and the other 90% operations are insertions and



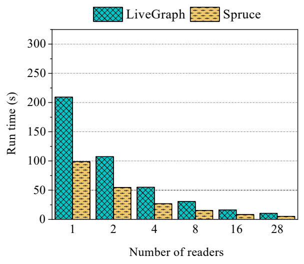
**(a) BFS run times with no writer.**



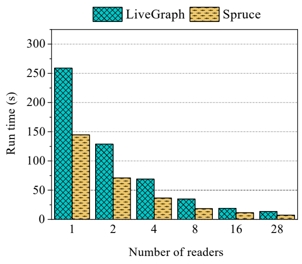
**(b) BFS run times with 16 writers.**



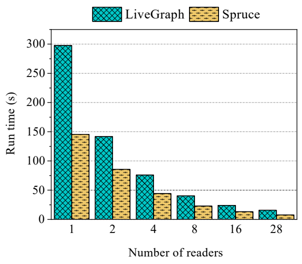
**(c) BFS run times with 28 writer**



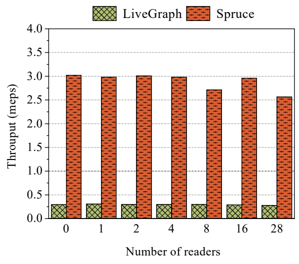
**(d) PR run times with no writer.**



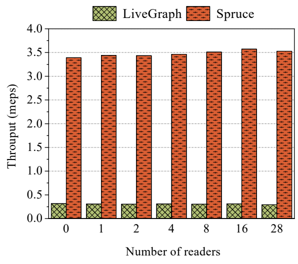
**(e) PR run times with 16 writers.**



**(f) PR run times with 28 writers.**



**(g) Update throughput with 16 writers.**



**(h) Update throughput with 28 writers.**

Fig. 12. Mixed read and write workload on *graph500-24*.

deletions that are balanced to keep the scale and data distribution of the graph stable. For the readers (analytical threads), they execute graph analytical algorithms (BFS/PR) once the graph is loaded. We choose the LiveGraph for comparison, which is a state-of-the-art graph system supporting concurrent operations by snapshots. We report the run times and throughput of various graph algorithms under different read-write workloads as follows.

Figure 12(a)~(c) reports the BFS latency with 0, 16, and 28 writers, respectively. Both Spruce and LiveGraph show good support for parallel graph analytics on evolving graphs by reason that their run times drop down significantly as the number of readers (analytical threads) increases. Compared to the case that has no writers, the self-slowdown is 1.04× and 1.24× for Spruce and LiveGraph on the heaviest write workload (28 writers), respectively. For PR, we get similar results (Figure 12(d)~(f)).

Figure 12(g) and Figure 12(h) depict the update throughput of 16 and 28 writers. When the number of writers is kept the same, the update throughput of both methods is not significantly affected by the number of readers. When the number of writers increases from 16 to 28, LiveGraph shows a minor increase in throughput (3.3% on average), while Spruce has a significant increase (21.7% on average). We attribute such a difference to the different concurrency protocols used in the two structures. The fine-grained MVCC protocol in LiveGraph for transactions puts heavy burdens on writers and restricts their throughput, while the lightweight locking strategy in Spruce makes it easier for the writers to modify data.

## 7 RELATED WORK

Relevant work includes tree-based indexes, concurrency control methods, and graph databases.

**Tree-based Indexes.** Lots of graph storage structures speed up graph queries as well as updates using tree-based indexes, B+ tree and its variants are used in many graph systems [3, 55]. They have high fanout and large node size, which is optimized for disk-based storage, and the basic operations (put, get, delete, etc.) on B+ tree can be performed in $O(lgn)$ time [9, 12, 58, 70]. Radix tree (radix trie) is a kind of space-optimized trie where data insertion, deletion, and query operations can be done in nearly constant time. Much recent work has further optimized this data structure and used them for key-value store [15, 40] as well as graph representation [22, 59]. Teseo [22] picks Adaptive radix trie (ART) [41] as its primary index, which designs different sizes of nodes to dynamically adjust the fanout of internal nodes and use path compression technique to reduce the space usage. Binary search trees (BST) are used for some in-memory graph storage methods [22, 23, 73]. Aspen [23] uses the BST to store vertices. It introduces the hash technique to balance BST and improve space efficiency by aggregating nodes in chunks. Pac-trees [22] further use BST to store edges and apply difference encoding for tree leaves.

**Graph Databases.** The design of existing graph databases can be divided into two categories. Databases in the first category (native graph databases) use specially designed structures for graph storage [1–3], while those in the second category (non-native graph databases) rely on relational database management systems (RDBMS) or key-value storage systems to support storing graphs [17, 21, 67]. Most of these databases adopt tree-based structures for index construction and pointer-based structures for edge storage. Thus, they can benefit from using Spruce as the basic storage structure to save space and achieve higher throughput.

## 8 CONCLUSION

In this paper we present Spruce, a tree-based graph storage structure that is characterized by low height, compression capabilities, and concurrency-friendly attributes. Spruce uses a novel index developed from the vEB tree to reduce levels as well as compress identifiers for vertex index, and combines the concept of adjacency list with CSR to store edges. Besides, we design a mixed locking strategy from ROWEX and optimistic locking to allow Spruce for concurrent updates and analytics on evolving graphs. In future work, we plan to extend Spruce's structure to allow storing dynamic graphs both in-memory and out-of-memory, which is expected to introduce a variation of LSM-Tree and batch-updated logs to optimize for disk I/O. Furthermore, we are interested in expanding Spruce to distributed systems and using *Remote Direct Memory Access (RDMA)* technology to improve its performance.

## ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their feedback and suggestions. The work was supported by the Nature Science Foundation of China (61672480), the 111 Project 2.0 (BP0719016).

## REFERENCES

[1] 2015. TITAN: Distributed Graph Database. https://titan.thinkaurelius.com.
[2] 2020. OrientDB Community. https://orientdb.org.
[3] 2022. Neo4j Graph Database Platform. https://neo4j.com/product/neo4j-graph-database.
[4] Manoj Agarwal, Krithivasan Ramamritham, and Manish Bhide. 2012. Real Time Discovery of Dense Clusters in Highly Dynamic Graphs: Identifying Real World Events in Highly Dynamic Environments. *Proceedings of the VLDB Endowment* 5 (06 2012). https://doi.org/10.14778/2336664.2336671

[5] Kunal Agrawal, Julian Shun, Yan Gu, Ziyang Men, Zheqi Shen, Yihan Sun, and Zijin Wan. 2023. Parallel Longest Increasing Subsequence and van Emde Boas Trees. *Proceedings of the 35th ACM Symposium on Parallelism in Algorithms and Architectures* (2023), 327–340. https://doi.org/10.1145/3558481.3591069

[6] A. Amir, A. Efrat, P. Indyk, and H. Samet. 2001. Efficient Regular Data Structures and Algorithms for Dilation, Location, and Proximity Problems. *Algorithmica* 30, 2 (01 Jun 2001), 164–187. https://doi.org/10.1007/s00453-001-0013-y

[7] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 1185–1196. https://doi.org/10.1145/2463676.2465296

[8] Nikolas Askitis and Justin Zobel. 2005. Cache-Conscious Collision Resolution in String Hash Tables. In *String Processing and Information Retrieval*, Mariano Consens and Gonzalo Navarro (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 91–102.

[9] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proc. VLDB Endow.* 7, 14 (oct 2014), 1881–1892. https://doi.org/10.14778/2733085.2733094

[10] R. Bayer and E. McCreight. 1970. Organization and Maintenance of Large Ordered Indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Houston, Texas) *(SIGFIDET '70)*. Association for Computing Machinery, New York, NY, USA, 107–141. https://doi.org/10.1145/1734663.1734671

[11] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing Breadth-First Search. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–10. https://doi.org/10.1109/SC.2012.50

[12] M.A. Bender, E.D. Demaine, and M. Farach-Colton. 2000. Cache-oblivious B-trees. In *Proceedings 41st Annual Symposium on Foundations of Computer Science*. 399–409. https://doi.org/10.1109/SFCS.2000.892128

[13] Michael A. Bender and Haodong Hu. 2007. An Adaptive Packed-Memory Array. *ACM Trans. Database Syst.* 32, 4 (nov 2007), 26–es. https://doi.org/10.1145/1292609.1292616

[14] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 521–534. https://doi.org/10.1145/3183713.3196896

[15] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Trans. Database Syst.* 47, 1, Article 3 (apr 2022), 46 pages. https://doi.org/10.1145/3506692

[16] P. Boldi and S. Vigna. 2004. The Webgraph Framework I: Compression Techniques. In *Proceedings of the 13th International Conference on World Wide Web* (New York, NY, USA) *(WWW '04)*. Association for Computing Machinery, New York, NY, USA, 595–602. https://doi.org/10.1145/988672.988752

[17] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX Association, San Jose, CA, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[18] Zhuhua Cai, Dionysios Logothetis, and Georgos Siganos. 2012. Facilitating Real-Time Graph Mining. In *Proceedings of the Fourth International Workshop on Cloud Data Management* (Maui, Hawaii, USA) *(CloudDB '12)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/2390021.2390023

[19] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. [n. d.]. *R-MAT: A Recursive Model for Graph Mining*. 442–446. https://doi.org/10.1137/1.9781611972740.43 arXiv:https://epubs.siam.org/doi/pdf/10.1137/1.9781611972740.43

[20] Douglas Comer. 1979. Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (jun 1979), 121–137. https://doi.org/10.1145/356770.356776

[21] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8 (aug 2013), 22 pages. https://doi.org/10.1145/2491245

[22] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. PaC-Trees: Supporting Parallel and Compressed Purely-Functional Collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 108–121. https://doi.org/10.1145/3519939.3523733

[23] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-Latency Graph Streaming Using Compressed Purely-Functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 918–934.

https://doi.org/10.1145/3314221.3314598

[24] David Ediger, Rob McColl, Jason Riedy, and David A Bader. 2012. STINGER: High Performance Data Structure for Streaming Graphs. *2012 IEEE Conference on High Performance Extreme Computing* 1 (2012), 1–5. https://doi.org/10.1109/hpec.2012.6408680

[25] Barış Ekim, Bonnie Berger, and Rayan Chikhi. 2021. Minimizer-space de Bruijn graphs: Whole-genome assembly of long reads in minutes on a personal computer. *Cell Systems* 12, 10 (2021), 958–968.e6. https://doi.org/10.1016/j.cels.2021.08.009

[26] Guanyu Feng, Zixuan Ma, Daixuan Li, Shengqi Chen, Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2021. RisGraph: A Real-Time Streaming System for Evolving Graphs to Support Sub-Millisecond Per-Update Analysis at Millions Ops/s. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 513–527. https://doi.org/10.1145/3448016.3457263

[27] Soukaina Firmli, Vasileios Trigonakis, Jean-Pierre Lozi, Iraklis Psaroudakis, Alexander Weld, Dalila Chiadmi, Sungpack Hong, and Hassan Chafi. 2021. CSR++: A Fast, Scalable, Update-Friendly Graph Data Structure. In *24th International Conference on Principles of Distributed Systems (OPODIS 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 184)*, Quentin Bramas, Rotem Oshman, and Paolo Romano (Eds.). Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 17:1–17:16. https://doi.org/10.4230/LIPIcs.OPODIS.2020.17

[28] Edward Fredkin. 1960. Trie Memory. *Commun. ACM* 3, 9 (sep 1960), 490–499. https://doi.org/10.1145/367390.367400

[29] Per Fuchs, Domagoj Margan, and Jana Giceva. 2022. Sortledton: a universal, transactional graph data structure. *Proceedings of the VLDB Endowment* 15, 6 (2022), 1173–1186. https://doi.org/10.14778/3514061.3514065

[30] Joseph A Gallian. 2022. A Dynamic Survey of Graph Labeling. *The Electronic Journal of Combinatorics* 1000 (2022). https://doi.org/10.37236/11668

[31] Sanchit Garg, Trinabh Gupta, Niklas Carlsson, and Anirban Mahanti. 2009. Evolution of an Online Social Aggregation Network: An Empirical Study. In *Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement* (Chicago, Illinois, USA) *(IMC '09)*. Association for Computing Machinery, New York, NY, USA, 315–321. https://doi.org/10.1145/1644893.1644931

[32] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 599–613.

[33] Yong Guo and Alexandru Iosup. 2012. The Game Trace Archive. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. 1–6. https://doi.org/10.1109/NetGames.2012.6404027

[34] Alexandru Iosup, Tim Hegeman, Wing Lung Ngai, Stijn Heldens, Arnau Prat-Pérez, Thomas Manhardto, Hassan Chafio, Mihai Capotă, Narayanan Sundaram, Michael Anderson, Ilie Gabriel Tănase, Yinglong Xia, Lifeng Nai, and Peter Boncz. 2016. LDBC Graphalytics: A Benchmark for Large-Scale Graph Analysis on Parallel and Distributed Platforms. *Proc. VLDB Endow.* 9, 13 (sep 2016), 1317–1328. https://doi.org/10.14778/3007263.3007270

[35] Andrew Lumsdaine Jeremy Siek, Lie-Quan Lee. 2022. The Boost Graph Library (BGL). https://www.boost.org/doc/libs/1_80_0/libs/graph/doc/index.html.

[36] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. 2012. Gbase: An Efficient Analysis Platform for Large Graphs. *The VLDB Journal* 21, 5 (oct 2012), 637–650. https://doi.org/10.1007/s00778-012-0283-9

[37] Steven Kearnes, Kevin McCloskey, Marc Berndl, Vijay Pande, and Patrick Riley. 2016. Molecular graph convolutions: moving beyond fingerprints. *Journal of Computer-Aided Molecular Design* 30, 8 (2016), 595–608. https://doi.org/10.1007/s10822-016-9938-8

[38] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-Time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4, Article 29 (jan 2020), 40 pages. https://doi.org/10.1145/3364180

[39] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (jun 1981), 213–226. https://doi.org/10.1145/319566.319567

[40] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 38–49. https://doi.org/10.1109/ICDE.2013.6544812

[41] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of Practical Synchronization. In *Proceedings of the 12th International Workshop on Data Management on New Hardware* (San Francisco, California) *(DaMoN '16)*. Association for Computing Machinery, New York, NY, USA, Article 3, 8 pages. https://doi.org/10.1145/2933349.2933352

[42] Dean De Leo and Peter Boncz. 2021. GFE Driver. https://github.com/cwida/gfe_driver.

[43] Dean De Leo and Peter Boncz. 2021. Teseo and the analysis of structural dynamic graphs. *Proceedings of the VLDB Endowment* 14, 6 (2021), 1053–1066. https://doi.org/10.14778/3447689.3447708

[44] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. 2010. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.* 11 (mar 2010), 985–1042.

[45] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[46] Panagiotis Liakos, Katia Papakonstantinopoulou, Theodore Stefou, and Alex Delis. 2022. On Compressing Temporal Graphs. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. 1301–1313. https://doi.org/10.1109/ICDE53745.2022.00102

[47] Weifeng Liu and Brian Vinter. 2015. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing* (Newport Beach, California, USA) *(ICS '15)*. Association for Computing Machinery, New York, NY, USA, 339–350. https://doi.org/10.1145/2751205.2751209

[48] C. Lu, W. Lam, and Y. Zhang. 2012. Twitter user modeling and tweets recommendation based on wikipedia concept graph. *AAAI Workshop - Technical Report* (01 2012), 33–38.

[49] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *2015 IEEE 31st International Conference on Data Engineering*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[50] Markus Mäsker, Tim Süß, Lars Nagel, Lingfang Zeng, and André Brinkmann. 2019. Hyperion: Building the Largest In-Memory Search Tree. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1207–1222. https://doi.org/10.1145/3299869.3319870

[51] Robert Campbell McColl, David Ediger, Jason Poovey, Dan Campbell, and David A. Bader. 2014. A Performance Evaluation of Open Source Graph Databases. In *Proceedings of the First Workshop on Parallel Programming for Analytics Applications* (Orlando, Florida, USA) *(PPAA '14)*. Association for Computing Machinery, New York, NY, USA, 11–18. https://doi.org/10.1145/2567634.2567638

[52] U. Meyer and P. Sanders. 2003. Delta-Stepping: A Parallelizable Shortest Path Algorithm. *J. Algorithms* 49, 1 (oct 2003), 114–152. https://doi.org/10.1016/S0196-6774(03)00076-2

[53] Derek G. Murray, Frank McSherry, Michael Isard, Rebecca Isaacs, Paul Barham, and Martin Abadi. 2016. Incremental, Iterative Data Processing with Timely Dataflow. *Commun. ACM* 59, 10 (sep 2016), 75–83. https://doi.org/10.1145/2983551

[54] Sai Tung On, Haibo Hu, Yu Li, and Jianliang Xu. 2009. Lazy-Update B+-Tree for Flash Devices. In *2009 Tenth International Conference on Mobile Data Management: Systems, Services and Middleware*. 323–328. https://doi.org/10.1109/MDM.2009.48

[55] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluc. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1372–1385. https://doi.org/10.1145/3448016.3457313

[56] Kenneth Platz, Neeraj Mittal, and S. Venkatesan. 2019. Concurrent Unrolled Skiplist. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*. 1579–1589. https://doi.org/10.1109/ICDCS.2019.00157

[57] Jeff Preshing. 2018. junction. https://https://github.com/preshing/junction.

[58] Jun Rao and Kenneth A. Ross. 1999. Cache Conscious Indexing for Decision-Support in Main Memory. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 78–89.

[59] Pedro Ribeiro and Fernando Silva. 2014. G-Tries: a data structure for storing and finding subgraphs. *Data Mining and Knowledge Discovery* 28, 2 (2014), 337–377. https://doi.org/10.1007/s10618-013-0303-4

[60] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com

[61] Neha Sharma and Prithwis Kumar De. 2023. *Application of Machine Learning to Predict Climate Change Consequences Arising Due to Investments by Banks in Fossil Fuel Sectors*. Springer Nature Singapore, Singapore, 49–90. https://doi.org/10.1007/978-981-19-5244-9_3

[62] Xiaogang Shi, Bin Cui, Yingxia Shao, and Yunhai Tong. 2016. Tornado: A System For Real-Time Iterative Analysis Over Evolving Data. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 417–430. https://doi.org/10.1145/2882903.2882950

[63] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. *SIGPLAN Not.* 48, 8 (feb 2013), 135–146. https://doi.org/10.1145/2517327.2442530

[64] Julian Shun, Laxman Dhulipala, and Guy E. Blelloch. 2015. Smaller and Faster: Parallel Processing of Compressed Graphs with Ligra+. In *2015 Data Compression Conference*. 403–412. https://doi.org/10.1109/DCC.2015.8

[65] Chris Siebenmann. 2012. Understanding Resident Set Size and the RSS problem on modern Unixes. https://utcc.utoronto.ca/ cks/space/blog/unix/UnderstandingRSS.

[66] Weiping Song, Zhiping Xiao, Yifan Wang, Laurent Charlin, Ming Zhang, and Jian Tang. 2019. Session-Based Social Recommendation via Dynamic Graph Attention Networks. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining* (Melbourne VIC, Australia) *(WSDM '19)*. Association for Computing Machinery, New York, NY, USA, 555–563. https://doi.org/10.1145/3289600.3290989

[67] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. SQLGraph: An Efficient Relational-Based Property Graph Store. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1887–1901. https://doi.org/10.1145/2723372.2723732

[68] Mahammad Valiyev Tum and Mahammad Valiyev. 2017. Graph Storage : How good is CSR really ?

[69] P. van Emde Boas. 1975. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. 75–84. https://doi.org/10.1109/SFCS.1975.26

[70] Jeffrey Scott Vitter. 2001. External memory algorithms and data structures: dealing with massive data. *ACM Computing Surveys (CSUR)* 33, 2 (2001), 209–271. https://doi.org/10.1145/384192.384193

[71] Biing-Feng Wang and Chien-Hsin Lin. 2011. Improved Algorithms for Finding Gene Teams and Constructing Gene Team Trees. *IEEE/ACM Transactions on Computational Biology and Bioinformatics* 8, 5 (2011), 1258–1272. https://doi.org/10.1109/TCBB.2010.127

[72] Brian Wheatman and Randal Burns. 2021. Streaming Sparse Graphs using Efficient Dynamic Sets. *2021 IEEE International Conference on Big Data (Big Data)* 00 (2021), 284–294. https://doi.org/10.1109/bigdata52589.2021.9671836

[73] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. *2018 IEEE High Performance extreme Computing Conference (HPEC)* 00 (2018), 1–7. https://doi.org/10.1109/hpec.2018.8547566

[74] Pengfei Zhang, Hang Dong, Yu Gao, Liangtian Zhao, Jie Hao, Jean-Yves Desaules, Qiujiang Guo, Jiachen Chen, Jinfeng Deng, Bobo Liu, Wenhui Ren, Yunyan Yao, Xu Zhang, Shibo Xu, Ke Wang, Feitong Jin, Xuhao Zhu, Bing Zhang, Hekang Li, Chao Song, Zhen Wang, Fangli Liu, Zlatko Papić, Lei Ying, H. Wang, and Ying-Cheng Lai. 2022. Many-body Hilbert space scarring on a superconducting processor. *Nature Physics* (2022), 1–6. https://doi.org/10.1038/s41567-022-01784-9

[75] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. 2020. LiveGraph: a transactional graph storage system with purely sequential adjacency list scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351

Received July 2023; revised October 2023; accepted November 2023