LSGraph: A Locality-centric High-performance Streaming Graph Engine

Hao Qi[1], Yiyang Wu[1], Ligang He[2], Yu Zhang[1], Kang Luo[1], Minzhi Cai[1], Hai Jin[1], Zhan Zhang[3], Jin Zhao[1]

[1] National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2] Department of Computer Science, University of Warwick, United Kingdom [3] Zhejiang Lab, China
{theqihao,wuyiyang}@hust.edu.cn,ligang.he@warwick.ac.uk,{zhyu,luokang2000,caimz,hjin}@hust.edu.cn,
zhanzhang@zhejianglab.com,zjin@hust.edu.cn

Abstract

Streaming graph has been broadly employed across various application domains. It involves updating edges to the graph and then performing analytics on the updated graph. However, existing solutions either suffer from poor data locality and high computation complexity for streaming graph analytics, or need high overhead to search and move graph data to ensure ordered neighbors during streaming graph update.

This paper presents a novel locality-centric streaming graph engine, called LSGraph, to enable efficient both graph analytics and graph update. The main novelty of this engine is a differentiated hierarchical indexed streaming graph representation approach to achieve efficient data search and movement for graph update and also maintain data locality and ordered neighbors for efficient graph analytics simultaneously. Besides, a locality-aware streaming graph data update mechanism is also proposed to efficiently regulate the distance of data movement, minimizing the overhead of memory access during graph update. We have implemented LSGraph and conducted a systematic evaluation on both real-world and synthetic datasets. Compared with three cutting-edge streaming graph engines, i.e., Terrace, Aspen, and PaC-tree, LSGraph achieves 2.98×-81.08×, 1.46×-12.56×, and 1.26×-10.31× speedups during graph update, while obtaining 1.02×-4.28×, 1.58×-3.55×, and 1.20×-2.72× speedups during graph analytics, respectively.

CCS Concepts: • **Theory of computation → Dynamic graph algorithms**; **Data structures design and analysis**.

Keywords: Streaming graph, Graph engine, Graph update, Graph analytics, Data locality

EuroSys '24, April 22–25, 2024, Athens, Greece

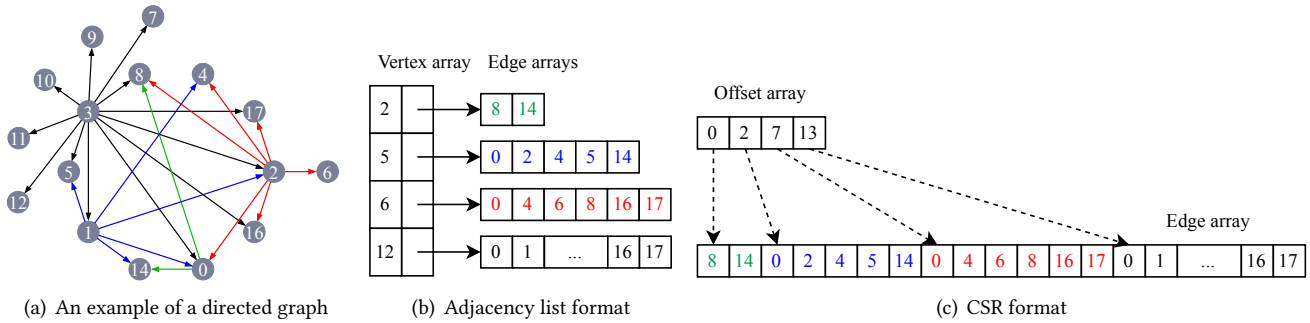
ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3650076

ACM Reference Format:
Hao Qi[1], Yiyang Wu[1], Ligang He[2], Yu Zhang[1], Kang Luo[1], Minzhi Cai[1], Hai Jin[1], Zhan Zhang[3], Jin Zhao[1]. 2024. LSGraph: A Locality-centric High-performance Streaming Graph Engine. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3627703.3650076

1 Introduction

Graph analytics has been used in various domains, including social networks [48], machine learning [4], and bioinformatics [19]. Streaming graphs are pervasive in real-world scenarios. Over time, streaming graphs require frequent updates (i.e., insertion, deletion, or modification of vertices and edges). For example, the relationships of users in Twitter and Facebook social networks change every day [6, 72]. Graph updates continue to arrive and are buffered in batches [31, 49, 50, 67]. After that, buffered updates are applied to the previous graph snapshot to construct the new snapshot, which will be utilized by graph analytics tasks to obtain the updated results. The graph update and graph analytics are usually alternately performed [1, 3, 23, 26, 29, 52, 60, 65, 67, 68].

A streaming graph system, particularly its internal data representation, is critical to support streaming graph analytics applications. Designing a streaming graph representation necessitates efficient support for graph analytics (e.g., continuous and ordered data storing) and graph update (e.g., data searching and data moving). In other words, to run graph analytics applications efficiently, it is essential to 1) quickly access the neighbors of each vertex and 2) ensure that the vertex's neighbors are ordered. Firstly, this is because traveling the neighbors of vertices is a core operation of these applications. For example, in the PageRank algorithm [81], vertices are required to access the scores of their neighboring vertices in order to update their own scores. Secondly, streaming graph analytics relies on the ordered neighbors of each vertex for low computation complexity [7, 11–13, 55]. For example, with ordered neighbors, cutting-edge *Graph Pattern Mining* (GPM) systems [55] can efficiently process set computations, which typically are the major performance



(a) An example of a directed graph (b) Adjacency list format (c) CSR format

Figure 1. An example graph, in which edges of each vertex are depicted using the same color, and its popular data formats

bottleneck. Note that streaming graph update based on the above ordered graph data representation requires (1) efficient search operations to find the position and (2) minimizing the overhead of data movement when this position already stores data.

However, existing streaming graph representations cannot address the combination of the aforementioned concerns. Most prior work does not guarantee ordered neighbors [2, 23, 25, 36, 47, 81], resulting in high computation complexity for streaming graph analytics. To ensure ordered neighbors and maintain locality for efficient streaming graph analytics, the *Packed Memory Array* (PMA) [5, 71], i.e., an ordered gapped array, is recently adopted by streaming graph representations [60, 68, 70, 71] to store graph data. But these representations cannot efficiently ingest large graph updates due to excessive data movement overhead. Note that Aspen [18] and PaC-tree [17] leverage the search tree to provide efficient graph updates and ensure ordered neighbors, but they suffer from numerous random memory accesses for graph analytics. Terrace [54] is further designed to adopt multiple data structures, including PMA [71] and B-tree [14], yet still suffers from massive data movement when many insertions are conducted.

We evaluate the performance of the cutting-edge streaming graph systems and observe that the PMA can support graph analytics well, but its update performance is poor when inserting large batch edges. Therefore, we focus on storing graphs using ordered gapped arrays to ensure the performance of graph analytics and then trying to enhance the efficiency of graph update. We further analyze the characteristics of PMA and observe that PMA suffers from ineffective search and numerous data movements. Specifically, PMA searches the data using binary search with data dependencies and poor spatial locality, and utilizes a single array to store a large amount of graph data, resulting in a significant overhead of data movement in order to guarantee ordered neighbors.

To address the above issues, we design a locality-centric streaming graph engine called *LSGraph* to support efficient both graph update and graph analytics. LSGraph presents a differentiated hierarchical indexed streaming graph representation based on ordered gapped arrays to achieve efficient search and maintain locality simultaneously. In particular, it introduces *Redundant Indexed Array* (i.e., RIA) to efficiently sample the graph data associated with low-degree vertices to construct a compact index array for improving search efficiency. For the graph data associated with high-degree vertices, it also designs a *Hybrid Indexed Tree* (i.e., HITree) to enable efficient search through integrating machine learning models and RIA. LSGraph further regulates the distance of data movement with a locality-aware mechanism to reduce the overhead of memory access during graph update.

We implemented and evaluated LSGraph with four real-world graphs and one synthetic graph. The results show that compared with three cutting-edge streaming graph engines, i.e., Terrace, Aspen, and PaC-tree, LSGraph gains the speedups of 2.98×-81.08×, 1.46×-12.56×, and 1.26×-10.31× during graph update, while achieving the speedups of 1.02×-4.28×, 1.58×-3.55×, and 1.20×-2.72× during graph analytics, respectively. In summary, LSGraph makes the following contributions:

- We design a differentiated hierarchical indexed streaming graph representation to enable efficient search and maintain locality simultaneously.
- We regulate the distance of data movement with a locality-aware mechanism for the above representation to reduce the overhead of memory access.
- We implement and evaluate a prototype of LSGraph[1] and the results demonstrate the efficacy of LSGraph.

2 Background and Motivation

2.1 Static Graph Representation

Figure 1 illustrates the example graph and its two popular static graph data formats. First, the *compressed sparse row* (CSR) consists of two arrays: the edge array and the offset array. The edge array contains all the outgoing edges of the vertices, while the offset array stores the index of the first

[1]The source code is available at https://github.com/CGCL-codes/LSGraph

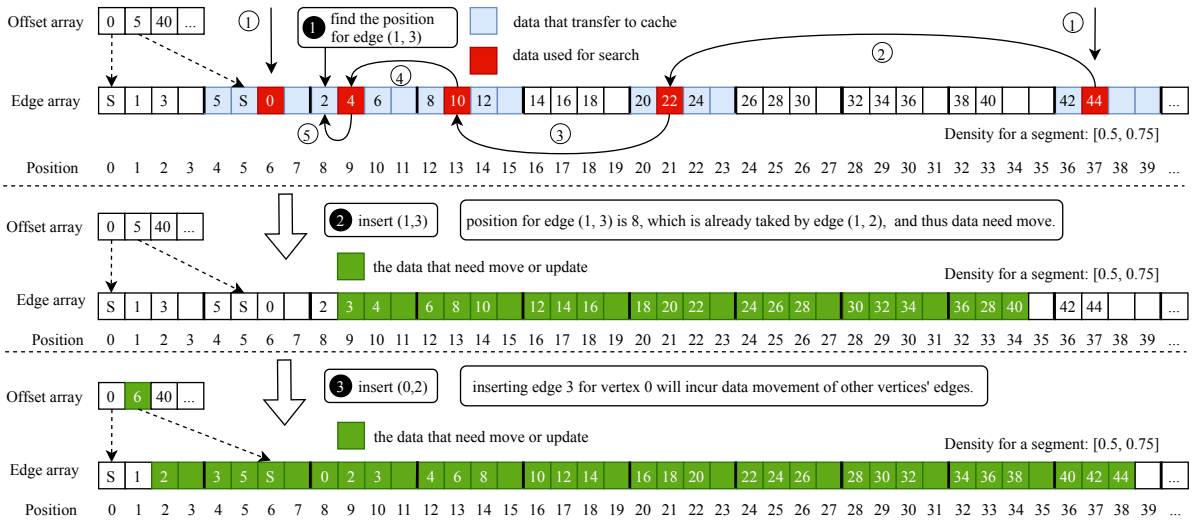


Figure 2. An example to illustrate inserting data into PMA. The "*S*" in the edge array is a sentinel entry.

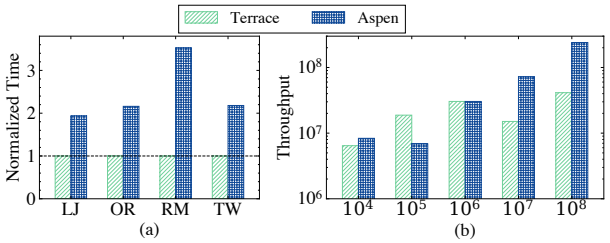


Figure 3. Performances of graph analytics and graph updates in Terrace and Aspen: (a) time to run BFS normalized to Terrace; (b) throughput (edges/second) for insertion with varying batch sizes on OR

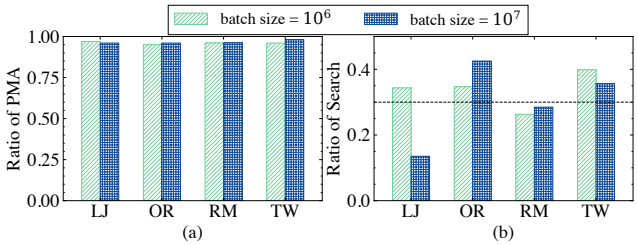


Figure 4. Performance analysis on different graphs for Terrace in large insertions: (a) time proportion of PMA [71] for insertion; (b) time proportion of search for insertion, which includes search and data movement in the evaluation

edge in the edge array for each vertex. The utilization of CSR is prevalent in several graph analytics systems due to its space efficiency [46, 53, 62]. Second, the *adjacency list* (AL) stores the edges of each vertex independently, and the vertex array is employed to store the degree of each vertex, together with a pointer to the corresponding edge array of the vertex. Compared to CSR, AL is more compatible with handling streaming graph updates since a graph update only impacts the edge array of a single vertex.

2.2 PMA-based Streaming Graph Representation

To ingest graph updates and maintain locality, several systems [60, 68, 71] use the *Packed Memory Array* (PMA) [5] to replace the edge array in the CSR format and store a sentinel entry for each vertex in the PMA for updating the offset array. As shown in Figure 2, PMA utilizes an ordered gapped array to store data, where gaps between data represent the unused space for insertion operations. The PMA maintains an implicit complete binary tree. The size of the leaf nodes in a binary tree is denoted as $log(N)$ (N is the size of the array), which is also referred to as the segment size. Each node of the binary tree has a lower and upper density limit. When the density of a node is below the lower limit or above the upper limit, PMA will redistribute the data to neighboring nodes in order to attain an appropriate density. Although the PMA reserves unused space to facilitate insertion, all edges of a graph are stored in an array, resulting in significant data movement overhead when a large number of insertions occur.

2.3 Limitations of Existing Solutions

Designing a streaming graph representation demands effective support for graph updates (including efficient search and data movement) and graph analytics (including quick access to the neighbors of each vertex and maintaining the ordered neighbors of each vertex). Nevertheless, existing streaming graph systems face challenges when addressing the combination of the aforementioned concerns.

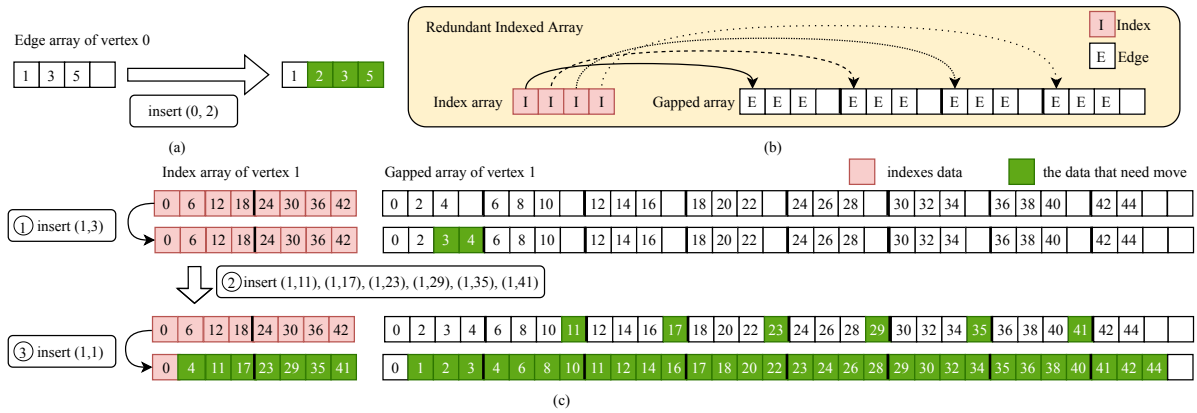


Figure 5. Illustration of inserting edges into an array and RIA: (a) insert edge (0, 2) into the edge array of vertex 0; (b) illustration of RIA; (c) insert edges into RIA

To illustrate these problems, we evaluate two cutting-edge streaming graph systems, namely Aspen [18] and Terrace [54], both of which ensure ordered neighbors. Aspen [18] leverages the compressed search tree to provide efficient graph updates and reduce memory usage. Although applying the block technique to trees, it still suffers from numerous random memory accesses, resulting in low performance in graph analytics [54]. To tradeoff the performance of graph updates and graph analytics, Terrace [54] adopts multiple data structures. Specifically, Terrace stores edges of high-degree vertices in B-tree [14] to support updates efficiently, while other edges are stored in PMA [71] and vertex blocks to enable locality. Note that vertex blocks are a data structure used in Terrace to store a few (or all) neighbors and the metadata (e.g., degree and a pointer to the B-tree) of each vertex. However, Terrace still suffers from ineffective search and numerous data movements.

The details of the platform and datasets used in the evaluation are presented in Section 6.1. Figure 3 shows that Terrace outperforms Aspen by at least 2×, up to 3.5× when running the *breadth-first search* (BFS) algorithm, while Aspen significantly outperforms Terrace when dealing with large insertions. Large insertions are even more important given the sharp rise of data in the era of big data. There are various scenarios in real-world streaming graphs where frequent insertion of edges occurs. For example, users on Twitter post tweets every day [57].

Aspen's low performance in graph analytics motivates us to focus on Terrace and improve its graph update efficiency when handling large insertions. To further investigate the reasons for inefficient updates in Terrace, we estimate the time proportion of updates for different data structures in Terrace in large batch sizes, using a single thread to eliminate the effect of multi-thread competition. We observe that the PMA accounts for the most percentage (up to 97%) of the total update time, as shown in Figure 4(a). This high overhead is caused mostly by two reasons: ineffective search and numerous data movements.

Ineffective Search. The example in Figure 2 demonstrates this problem, assuming that both a cache line size and a segment size of the PMA are four, and the lower and upper densities of each segment in the PMA are 0.5 and 0.75, respectively. To search the position of edge (1, 3) (❶), the range for edges of vertex 1 is first located using the offset array (the step ①), i.e., [6, 37]. After that, the binary search algorithm is used to find the position of 3 in the range. Specifically, since 21 is the middle position between 6 and 37, we compare the element 22 in position 21 with 3 to determine the range (i.e., [6, 21], left of position 21 because 22 is larger than 3) of subsequent comparisons (the step ②). Then, we continue with this process (the steps ③, ④, ⑤) until we find position 8 for 3. However, such processes suffer from data dependencies and poor spatial locality. In particular, each cache line transfer (i.e., transfer data to cache from memory) depends on the result of the last transfer. Besides, most data of each transfer fetched into the cache is not actually required (e.g., 1/4 is used for comparison of binary search in the example), leading to the underutilization of cache resources and memory bandwidth.

We analyze the impact of search on the overall insertion performance. As illustrated in Figure 4(b), the results show that in most cases, search time accounts for over 30% of the overall time, reaching up to 43% in the worst case. Such significant search-time proportions should not be neglected. This motivates us to design efficient search strategies to quickly find the appropriate position for each graph update.

Numerous Data Movements. We illustrate this issue using the example of inserting edge (1,3) (❷) in Figure 2. After searching position 8 for 3, we discover that the position is already taken by 2, which we call *position conflict*. As a result, data movement is necessary to store 3. The current

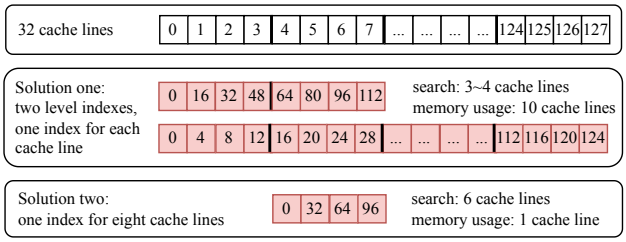


Figure 6. Two solutions to build indexes for elements with 32 cache lines size

segment cannot store 3 since its upper density is 0.75, and thus we need to find a nearby segment where the upper density does not exceed 0.75 after storing one element, i.e., segment 9. Finally, all data in the range [9, 34] needs movement to guarantee ordered neighbors. Note that, as seen in Figure 2, when inserting the edge (0,2) (❸) of vertex 0, it also leads to abundant data movements for edges of vertex 1.

Massive data movements result in a significant increase in memory accesses, particularly for the scenario with a large number of insertions, thereby decreasing the performance of graph updates. Figure 4(b) also illustrates the high data movement overhead in Terrace. Furthermore, data movements incur the occurrence of multi-thread contention issues since two threads could try to update the same range of the PMA simultaneously, resulting in the high overhead of lock and cache coherence. In addition, the PMA maintains an upper density bound for each segment to guarantee that segments are never totally filled, ensuring that threads can always insert without blocking or waiting [71]. However, this underutilizes the space and increases data movement. These problems drive us to design data structures that support efficient data movements when handling graph updates.

3 Design of LSGraph

To overcome the above limitations, we first build indexes on arrays with minimal memory consumption and computation cost to improve search efficiency and maintain locality simultaneously, and then regulate the distance of data movement to reduce memory access overhead.

3.1 Indexed Gapped Arrays

We store the edges of vertices based on the AL-based format in Figure 1(b) to avoid inefficient data movement between edges of vertices. Figure 5(a) shows that inserting (0,2) only moves 3 elements, instead of 37 elements in Figure 2(❸). A significant advantage of CSR over AL is the sequential access to the edge array when executing the graph analytics algorithms by the order of vertices' ID. However, the scenarios for this computation approach are quite restricted in streaming graphs. First, since streaming graph updates only affect a fraction of the graph, rather than full graph computation, most recent streaming graph systems employ incremental computation [8, 31, 49, 50, 67], where most of the accesses to the starting addresses in the edge array of a vertex's neighbors are random accesses because graph updates arrive randomly. Second, even though computing the entire graph, several systems [56, 77–79] asynchronously process graph vertices with priority scheduling to speed up the convergence of graph analytics algorithms.

To reduce the overhead of binary search as discussed in Section 2.3, we build indexes with redundant data on gapped arrays, which we call *Redundant Indexed Array* (RIA), to speed up the search process by performing an efficient search in the indexes at the beginning of the search. As illustrated in Figure 5(b), RIA consists of an index array and a gapped array. The gapped array is organized as a number of blocks (block size is 4 in Figure 5) with contiguous address space. The data in RIA are divided evenly among blocks according to data size, and the first elements within each block are copied and combined to form the index array. Thus, there are no empty blocks in the gapped array, and consequently, RIA is memory-efficient. Different from PMA [71], we do not maintain the upper density for each block since we assign updates of a vertex to a single thread to eliminate the issue of multi-thread competition (details in Section 5).

For example, to search (1,3) in the RIA of Figure 5(c) (in step ①), we first search the index array of vertex 1 to decide which block should store 3. Since 3 is less than 6 (which is the first element in the second block), it will be stored in the first block. After that, we search for the position of 3 and then insert it in the first block. Therefore, we only need 2 cache transfers for search and 2 element movements, rather than 5 cache transfers and 24 element movements in Figure 2(❷).

Limitations of RIA. Streaming graphs follow the power-law degree distribution [54, 68], which means that the majority of vertices have a small number of edges, while a few vertices have a very large number of edges. However, storing edges of high-degree vertices in RIA will result in either an inefficient search or a high overhead in memory usage. Figure 6 illustrates two solutions for building indexes for data with 32 cache line sizes. Solution one creates an index for each cacheline's data and builds two-level indexes to enable efficient searching (transfer 3~4 cache lines, 1~2 for first level indexes, 1 for second level indexes, 1 for searching in a block), but it consumes a high amount of memory usage (10 cache lines). Solution two builds an index for eight cache lines and uses less memory usage (1 cache line) than solution one, but it raises the search overhead (transfer 6 cache lines) since the search continues throughout the eight cache lines (transfer 5 cache lines).

To alleviate these problems, we build learned indexes [34] on arrays for the edges of high-degree vertices, which refers to *Learned Indexed Array* (LIA) (details in Section 3.2). The learned index is motivated by the observation that the *Cumulative Distribution Function* (CDF) can be efficiently approximated by a model that predicts the position within an

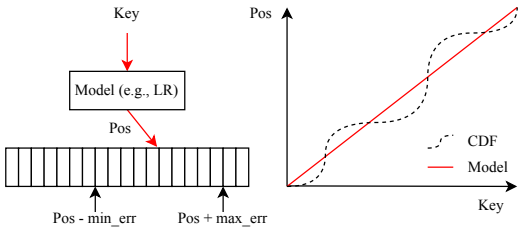


Figure 7. Illustration of learned index [34]

ordered array given a key. Based on this, prior work [34] leverages *Machine Learning* (ML) models, such as *Liner Regression* (LR), to estimate the CDF to find the key's position. As shown in Figure 7, given a key, the position (Pos) predicted by the model (LR) is imprecise with a min-error and max-error, therefore, a local search (e.g., the exponential search used by ALEX [20]) is needed to obtain the actual position. Note that we do not create learned indexes for low-degree vertices' edges because a few edges make it difficult to provide sufficient information to build a model, and building models for a large number of low-degree vertices could lead to significant computation overhead.

More importantly, when dealing with continuing insertions, RIA suffers from massive data movement, similar to PMA. As illustrated in Figure 5(c), after adding several edges in RIA (the setp ②), inserting edge 1 will result in a large amount of data movement (the setp ③). This drives us to regulate data movement to minimize the overhead of data movement.

3.2 Regulate the Data Movement

During inserting an element into an ordered data structure, after searching the position, the key issue is how the element and these elements in the structure should be moved when a position conflict occurs between the new element and the existing elements. The PMA [71] employs a strategy wherein elements are inserted into leaf nodes that exceed their upper density, and subsequently will be moved to neighboring nodes with lower density, which we refer to as ***horizontal movement*** (HM). While the PMA does ensure cache locality, the amount of data movement becomes enormous as the inserted edges increase. On the contrary, the B-tree can dynamically request small memory spaces for moving the conflicted elements and then create a pointer to them, which reduces the amount of data movement, which we refer to as ***vertical movement*** (VM). But it cannot handle graph analytics effectively due to its enormous data movements over long distances.

Large amounts of data movement can result in frequent memory accesses, leading to bandwidth bottlenecks. In addition, the frequent data movements with long distances, which are typically random memory accesses, could lead to an increased occurrence of cache misses [24], DRAM row

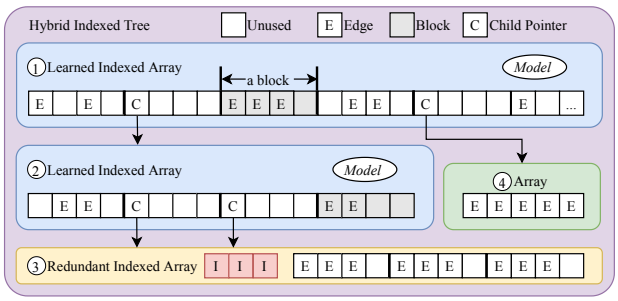


Figure 8. The design overview of HITree

buffer misses [9], and even TLB misses [33], thereby degrading the performance of graph analytics and graph updates. Generally speaking, when handling the position conflict, HM conducts more data movements than VM, while HM moves the data for shorter distances on average than VM. Therefore, we prefer HM with limited short distances for cache locality, and perform VM, particularly when frequent continuous insertions for high-degree vertices, to reduce the amount of data movement.

To reduce frequent data movements of insertions in the case where there are few gaps in RIA (the step ③ in Figure 5(c)), we set an upper bound for the distance of data movement. When exceeding the upper bound, we expand the space of RIA with more gaps to reduce the data movement overhead caused by subsequent insertions. In general, more gaps will reduce the data movement, enabling faster updates, but it will also increase the memory usage and memory access of traversing edges of vertices in graph analytics. To adjust the density of the gaps in the new expanded space, we define the *space amplification factor* (denoted by α), and the size of the new space is (old_size * α) (old_size is the size of elements before expanding space).

Note that the PMA could move data across the entire array based on its density limit in the worst case, and expanding the space of the PMA is extremely expensive since it stores most edges of the graph in Terrace [54]. Therefore, the PMA reserves enormous gaps at initialization to reduce the chance of expanding space. However, this results in significant memory usage (details in Section 6.4).

When storing edges of high-degree vertices which relate to frequent updates in streaming graphs [54, 68], RIA also suffers from high overhead of horizontal data movement (even expanding the entire space), similar to the PMA. Therefore, vertical data movement is needed to reduce the overhead of horizontal data movement. This motivated us to design the *Hybrid Indexed Tree* (HITree) that combines the two types of movement to support efficient data movement.

HITree. As shown in Figure 8, the internal node (non-leaf node) of HITree employs the LIA (① and ②), while the leaf nodes can be either a RIA (③) or an array (④). Firstly, the RIA is preferred to LIA as leaf nodes mainly due to two reasons: (1)

modeling small-sized data will lead to an enormous amount of random memory accesses and training overhead; and (2) when a position conflict occurs during insertion, the RIA can efficiently move data horizontally or expand its capacity without training overheads to reduce the depth of HITree. Secondly, the LIA is designed to enable the creation of a child node (vertical movement) to reduce the amount of horizontal data movement when frequent position conflicts occur.

To reduce random memory accesses resulting from creating child nodes, we allow horizontal data movement with a fixed space size (i.e., a block) in LIA. Specifically, instead of vertical movement, we solve the position conflict problem by moving data horizontally within a block, which not only ensures efficient graph update performance, but also guarantees cache locality in graph analytics. In other words, if there are position conflicts in LIA, the data are moved within a distance (i.e., HM). If there still exist conflicts, a child data structure (LIA, RIA, or array) is created (i.e., VM). Note that LIPP [75] and AFLI [76] directly create a child node when a position conflict occurs, thereby leading to a significant number of random memory accesses.

Specifically, each LIA is comprised of three main components: an array of entries, a model that takes the key as input and outputs a position in the array, and a bit vector that represents the types of entries. LIA has four types of entries:

- **Unused (U)**: This entry is used to insert a new edge. After that, the entry's type is changed to Edge.
- **Edge (E)**: The entry stores the destination vertex ID of an edge. When a data conflict occurs at the entry during insertion, its type is changed to Block.
- **Block (B)**: The entry is designed to prevent the direct creation of a child node in the event of a position conflict, hence reducing random memory accesses. It appears consecutively in a block size, where the beginning of a block consists of the Edge type and the subsequent is the Unused type. When there is no space left in a block during insertion, the entry types of the whole block are changed to Child Pointer.
- **Child Pointer (C)**: The entry contains a pointer to a LIA (②), a RIA (③), or an array (④) in the next level to construct the tree structures.

The model's high accuracy usually requires higher training and prediction costs. We balance the trade-off between accuracy and speed. Specifically, we use the LR models instead of the *piecewise linear regression* (PLR) models, which are commonly employed in several learned index systems [42, 43, 66]. While the PLR model exhibits higher predictive accuracy than LR models, the training and prediction overheads incurred by the PLR model are significantly greater than those of the LR model. For instance, when HITrees are employed to store graph data of LJ in Table 1, it has been observed that the graph update performance of the LR model is an order of magnitude better than that of the PLR model.

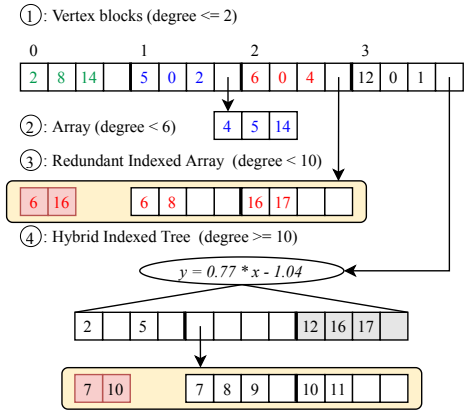


Figure 9. The example graph stored in LSGraph

4 Graph Representation and Operations

4.1 Graph Representation

Based on the above design, we present a differentiated hierarchical indexed streaming graph data representation, as shown in Figure 9. We store edges of vertices in several data structures, depending on the vertices' degrees. First, we store the degree, some (or all for low-degree vertices) edges, and the pointer to other edges for each vertex in the vertex blocks (①), like Terrace [54]. Then, in the case of edges of a vertex that exceed the space allocated to it in the vertex blocks (the threshold is denoted by L), the remaining edges of the vertex are stored into an individual space according to the degree of the vertex. Finally, RIA (③) and HITree (④) are specifically designed for storing edges of vertices with different degrees (L+M is the threshold). To reduce the memory usage of indexes, we store the edges of a vertex in an array (②) instead of a RIA when the degree of the vertex is relatively small (L+A is the threshold).

Putting It All Together. As demonstrated in Figure 9, the neighbors of each vertex are stored in at most two data structures. First, each vertex has a small number of neighbors stored using the vertex blocks. For example, all the neighbors of vertex 0, which has two neighbors, are in the vertex blocks, while vertices 1, 2, and 3 store only a part of their neighbors in the vertex blocks since their degrees are larger than the threshold L (i.e., 2). Vertices 1, 2, and 3 use array, RIA, and HITree to store their remaining neighbors (thresholds L+A and L+M are 5 and 10), respectively. Vertex 2 employs RIA with indexes 6 and 16, which correspond to the first elements of the two blocks in the gapped array. Vertex 3 has 12 elements (larger than the threshold 10), which are stored in HITree (including a LIA and a RIA).

4.2 Graph Operations

We employ the operations of HITree to demonstrate the support for graph analytics and graph update since HITree also contains the operations of RIA and the operations for

Algorithm 1: The BulkLoad Algorithm for HITree

Input: *ns*: an array of ordered elements; α: the space amplification factor; *M*: the threshold using RIA or LIA; *BKS*: the size of each block.
Output: *np*: the current node pointer.

```
1  if ns.size <= M then
2      np.gapped_array = malloc(ns.size * α);
3      np.index_array = malloc(⌈ns.size * α / BKS⌉);
4      DistributeData(np.gapped_array, ns);
5      BuildIndex(np.index_array, np.gapped_array);
6  else
7      np.array = malloc(ns.size * α);
8      np.model = BuildModel(ns, np.array);
9      poss = PredictedAllPositions(ns, np.model);
10     foreach subns, subposs in a BKS of np.array ∈ ns, poss
         do
11         ba = BlockAddress(subposs, np.array);
12         if is_unique(subposs) then
13             foreach u, pos ∈ subns, subposs do
14                 np.array[pos] = u; SetType(pos, E);
15         else if subns.size <= BKS then
16             StoreBlock(np.array, ba, subns);
17             SetTypes(ba, B);
18         else if subns.size > BKS then
19             child = BulkLoad(subns, α, M, BKS);
20             np.array[ba] = child; SetTypes(ba, C);
21     MergeAdjacentChildren();
22 return np;
```

the vertex blocks and arrays are relatively simple. Given a vertex v or an edge (v, u), the main operations of HITree can be categorized as follows:

- *BulkLoad*: it uses HITree to store neighbors of vertex v for graph initialization.
- *Traverse*: it traverses the HITree and implements the function f for neighbors of vertex v to support graph analytics.
- *Insert (Delete)*: it inserts (deletes) the edge (v, u) in the HITree during graph update.

BulkLoad. Bulkload stores an ordered array *ns* into HITree and returns a pointer to the root node, as described in Algorithm 1. In the first step, we select RIA or LIA depending on the threshold M (line 1), and develop a separate implementation for each of the two cases. For RIA, the sizes of the gapped array and index array are initially obtained by multiplying the size of the *ns* and the amplification factor α (the lower density is $1/\alpha$), and the number of blocks in the gapped array, respectively (lines 2–3). After allocating space for the gapped array and index array, the elements of the *ns* are evenly distributed across the blocks of the gapped array (line 4). Finally, the first element of each block in the gapped array is copied to build the index array (line 5).

For LIA, we first allocate the space for *np.array* and build a linear model based on *ns* and *np.array.size* (lines 7–8). The model is then used to predict the positions (*poss*) in *np.array* of all the elements in *ns* (line 9). After that, we iteratively construct the HITree using the elements (*subns*) mapped to a continuous *BKS* space in the *np.array* as the basic processing group (line 10). If there is no same predicted position in the *BKS*, we directly store the *subns* in their predicted positions and then set their types to **E** (lines 12–14). Otherwise, we calculate the starting address (*ba*) in *np.array* of the current block (line 11) and then compare the size of the *subns* to the *BKS*. If *BKS* is larger than *subns.size*, which indicates that the current block has the capacity to accommodate them, we place the elements of *subns* at the beginning of the current block in *np.array* and set types to **B** (lines 15–17), thereby eliminating the need to create a new child node. On the other hand, we store a pointer at the beginning of the current block in *np.array* to a child node created by *subns* and then set types to **C** (lines 18–20). To further reduce random memory access, we merge their children when multiple consecutive blocks are **C** type (line 21). As shown in Figure 8 (②), there are two child pointers in the LIA pointing to the same RIA.

Figure 9 shows that the HITree (④) for vertex 3 consists of a LIA and a RIA. The array's size in LIA is three blocks. The first one has no position conflict and is made up of **E** and **U** types, which are determined by the LR model ($y = 0.77 * x - 1.04$). Since the number of position conflicts exceeds the size of a block, the second block creates a pointer to a RIA. The last one has position conflicts, but since the total number of elements in the block does not exceed the block size, all elements predicted by the LR model in the block are placed consecutively at the beginning of the block (i.e., 12, 16, and 17).

Traverse. For RIA, we apply function f to elements in the gapped array by scanning the array and skipping unused space. For the LIA, we continue to traverse the elements in *np.array* and check their types. For the **U** type, we continue the iterative process. For the **E** type, we apply f to the corresponding element. For **B** types, we traverse the beginning elements in the current block. For the **C** type, we follow the pointer to traverse the next level of the HITree.

Insert. As illustrated in Algorithm 2, for RIA, we first search the block position (*bid*) which stores u in the index array, and then try to insert u into the block (lines 2–3). If the insertion succeeds (e.g., the block has unused space), we update the index array if necessary (lines 4–5). Otherwise, we move the elements to the nearby blocks. Specifically, based on the greedy algorithm, we continually traverse the left or right blocks to find a block with unused spaces. We set the upper bound on the number of blocks to be searched to $log(num\_block)$ by default (num_block is the number of blocks in *np.index_array*). If we find free space

Algorithm 2: The Insert Algorithm for HITree

Input: *np*: the current node pointer; *u*: the insert element; α: the space amplification factor; *M*: the threshold using RIA or LIA; *BKS*: the size of each *Block*.

```
1  if np.size <= M then
2      bid = SearchIndex(np.index_array, u);
3      insert_ok = InsertBlock(np.gapped_array, bid, u, BKS);
4      if insert_ok then
5          UpdateIndex(np, bid, BKS);
6      else
7          move_ok, range = MoveNearBlocks(np, bid, BKS);
8          if move_ok then
9              UpdateIndexes(np, range, BKS);
10         else
11             ns = MergeData(np.gapped_array, u);
12             np = BulkLoad(ns, α, M, BKS);
13 else
14     pos = Predicted(np.model, u);
15     type = GetType(pos);
16     ba = BlockAddress(pos, np.array);
17     if type == U then
18         np.array[pos] = u; SetType(pos,E);
19     else if type == E or type == B then
20         ns = MergeDataBlock(np.array, pos, u);
21         if ns.size <= BKS then
22             StoreBlock(np.array, ba, ns); SetTypes(ba, B);
23         else
24             child = BulkLoad(ns, α, M, BKS);
25             np.array[ba] = child; SetTypes(ba, C);
26     else if type == C then
27         Insert(np.array[ba], u, α, M, BKS);
```

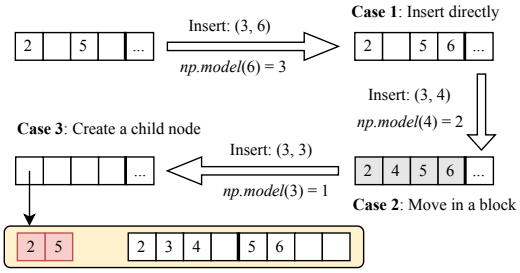


Figure 10. Insertions in the LIA

within this upper bound, then we will move the data in the *range* and update the index array (lines 7–9). Otherwise, we merge elements of *np.gapped_array* and *u*, and then expand *np.gapped_array* with the space amplification factor α (lines 10–12).

For LIA, we first use the *np.model* to predict *u*'s position (*pos*) in the *np.array*, and then identify the type of *u* (lines 14–15). If type is **U**, we store *u* in the *pos* and set **E** type for the *pos* (lines 16–18). If the type is **E** or **B**, we merge the elements within the current block in *np.array* and *u* to an ordered array *ns* and then either store *ns* in the current block or create a child node for *ns* depending on the size of *ns* (lines 19–25). Finally, if the type is **C**, the child node in the next level will recursively handle *u* (lines 26–27).

An Example of Using LIA. The procedure of inserting edges into LIA is illustrated in Figure 10, where the first four spaces (a block) in LIA of vertex 3 are used as an example for the explanation. There are primarily three cases. In **Case 1**, when inserting edge (3,6), the position of 6 in LIA should be 3, denoted as *np.model*(*v*) which indicates the position of *v* predicted by the model in LIA. Then, we directly store 6 in *np.array*[3] because the type of position 3 is **U** type. In **Case 2**, the predicted position of 4 is 2, which is already used by 5, and then we move elements horizontally within a block. In **Case 3**, the block does not have enough space for 3, and thus it is necessary to create a child node.

Delete. Similar to the insert operation, the delete operation involves finding the position of data, then changing its type, and finally moving data. Specifically, after searching the element, the **E** type will be changed to **U**. When all spaces in a block are unused, we move data horizontally (in RIA) and change types to **U** (in LIA).

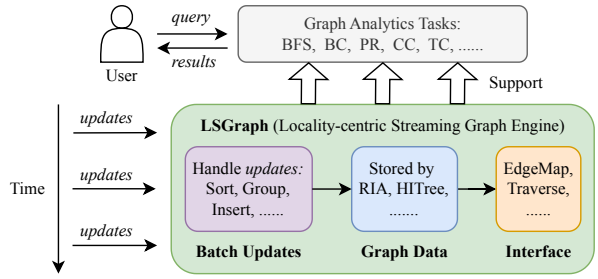


Figure 11. System architecture of LSGraph

5 Implementation of LSGraph

Based on the proposed graph representation and its operations, we developed LSGraph, a shared-memory streaming graph engine. Figure 11 shows the system architecture, where LSGraph ingests *updates* and then provides support for graph analytics tasks through the interface, finally supporting user queries. In particular, LSGraph consists of three main components:

Batch Updates. Given a batch of updates, we first sort them by the ID of the source vertices and then sort them by the ID of the destination vertices. After that, we divide the update edges of each source vertex into groups, each of which will be assigned to the same thread for batch processing, thereby improving locality and avoiding lock overhead. Finally, we insert these updates into the graph data. The

Table 1. A list of graph datasets with their number of vertices and edges, and their *average degree* (Avg.Deg) for evaluation

Graph	Vertices	Edges	Avg.Deg
LiveJournal (LJ)	4,847,571	85,702,474	17.7
Orkut (OR)	3,072,627	234,370,166	76.2
rMat (RM)	8,388,608	1,098,754,156	130.9
Twitter (TW)	61,578,415	2,405,026,092	39.1
Friendster (FR)	124,836,180	3,612,134,270	28.9

above steps are executed in parallel, and their time is included in the overall time for graph updates to obtain the throughput of LSGraph in Section 6.2.

Graph Data. We adopt the above graph representation to store graph data, and its configuration is as follows considering cache locality. First, each vertex is assigned the size of a single cache line within the vertex blocks. Additionally, the *BKS* in RIA and LIA also fits within a cache line. Next, the size of each array in RIA and HITree is set to multiple cache line sizes, while ensuring that the starting memory addresses of all arrays are aligned with a cache line size. Since inserting an edge into RIA requires at least two cache line transfers (one for index and one for block), we set the threshold A to the size of two cache lines. The default threshold M and amplification factor α are 4096 and 1.2, respectively, which will be discussed in Section 6.5.

Interface. To support graph analytics algorithms, we expand the interface of the *EdgeMap* primitive proposed by Ligra [62] and implement it based on HITree's *Traverse* operation.

6 Evaluation

6.1 Experimental Setup

Baselines. We compared LSGraph with Terrace [54], Aspen [18], and PaC-tree [17], three cutting-edge streaming graph systems. Different from Aspen, which stores arrays in each node of the tree and randomizes chunk sizes, PaC-tree stores arrays only in the leaf node and optimizes the selection of chunk sizes with stronger theoretical bounds for several operations. We compared them with the throughput of graph updates, the performance of different graph analytics algorithms, and memory footprints. Note that we have conducted the experiments to compare PaC-tree and Sortledton [28]. The results show that Pac-tree outperforms Sortledton by 40.56×-142.53×. Thus, we use PaC-tree as a baseline instead of Sortledton.

Environments. We implemented LSGraph as a C++ library. Both LSGraph and Terrace used Cilk [40] for parallelism and the OpenCilk [58] compiler (version 1.0) based on the Tapir/LLVM [59] branch of the LLVM [39] compiler (version 10.0.1). We compiled Aspen and PaC-tree using the g++ compiler (version 9.4). We ran all experiments on a dual-socket machine equipped with a 64-core 2-way hyper-threaded Intel Xeon Platinum 8358 CPU @ 2.60GHz, which has 96MB of L3 cache and 1 TB of memory. To avoid the effects of NUMA and ensure a fair comparison, we performed all experiments on a single CPU socket with 32 physical cores and 64 hyper-threads, like Terrace [54].

Graph Analytics Algorithms. For a fair comparison, we use the same four algorithms from Terrace: *breadth-first search* (BFS), single-source *betweenness centrality* (BC), *PageRank* (PR), and *connected components* (CC). Besides, we implement and optimize the *triangle counting* (TC) algorithm, a well-known GPM algorithm that involves massive intersection computation, based on our graph representation.

Datasets. Table 1 summarizes the graphs utilized in our experiments. We perform evaluations on four real-world graphs (i.e., LJ [41], OR [41], TW [38], and FR [37]) and one synthetic graph (i.e., RM). We generate RM using rMat generator [10] with $a = 0.5; b = c = 0.1; d = 0.3$, which is the same as Aspen [18]. To ensure a fair comparison with Terrace and Aspen, we also test all the graphs with their symmetrized versions.

6.2 Graph Update Throughput

To make a fair comparison of graph update throughput, our experimental steps are consistent with Aspen and Terrace. First, all edges of an existing graph are inserted into LSGraph. Then, a batch of updated edges is inserted into the graph and deleted from the graph subsequently, ensuring that the original graph structure will not be changed during the insertion and deletion for each batch size. Those updated edges are generated by the rMat generator with the same parameters as the RM graph. We report the results averaged over 5 trials, each of which inserts the different edges.

Figure 12 illustrates the insertion's throughput of Terrace, Aspen, PaC-tree, and LSGraph with batch sizes ranging from 10^4 to 10^8. As shown in Figure 12, the graph update performance for all cases of LSGraph outperforms that of Terrace, Aspen, and PaC-tree, with speedups of 2.98×-81.08×, 1.46×-12.56×, and 1.26×-10.31×, respectively (7.18×, 4.77×, and 4.80× on average). The throughput of LSGraph is to reach a maximum of $3.5 * 10^8$ edges per second on LJ when handling the batch of 10^8 updates. Note that, for edge deletions, LSGraph also outperforms Terrace, Aspen, and PaC-tree by 3.59×-133.52×, 1.97×-26.77×, and 1.58×-24.41×, respectively. Additionally, LSGraph also exceeds the baselines when dealing with smaller batch sizes. For example, when the batch size is 10, LSGraph outperforms Terrace, Aspen, and PaC-tree by 1.05×-2.04×, 2.28×-3.58×, and 1.88×-2.52×, respectively.

The primary reason for Terrace's poor performance is attributed to the excessive data movement overhead and inefficient search of PMA [71]. Although Aspen and PaC-tree employ the tree structure to reduce the amount of data movement, their numerous random memory accesses result in lower performance than LSGraph. LSGraph accelerates the

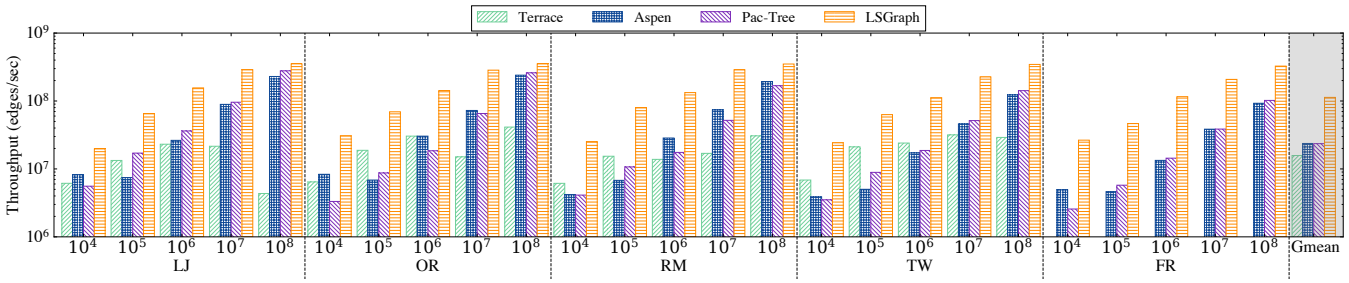


Figure 12. Throughput (edges/second) for insertion with varying batch sizes on all graphs in Terrace, Aspen, Pac-Tree, and LSGraph. Throughputs of the FR graph for Terrace are omitted because of time constraints.

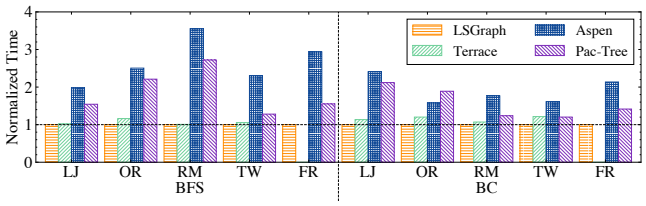


Figure 13. Time to run BFS and BC normalized to LSGraph in LSGraph , Terrace, Aspen, and Pac-Tree

search by designing the index-based structures and proposes the data movement strategies with a locality-aware approach to reduce the overhead of memory access.

We observe that there is a steady improvement in the overall performance of LSGraph, Aspen, and PaC-tree as the batch size increases, whereas Terrace's performance does not. The main reason for this phenomenon is that Terrace suffers from significant data movement overhead as the batch size increases because the PMA in Terrace stores most edges of a graph in a single array [54], which leads to frequent massive data movements and even expansion of the entire array, in particular when dealing with large batch sizes on small graphs, such as 10^8 on LJ, which has been validated by our experiments. Unlike Terrace, Aspen uses trees to store graph data, and LSGraph uses RIA and HITree to store edges of vertices and optimizes their insertion strategies.

We also evaluate the contribution of RIA, HITree, and LIA in the performance of LSGraph. In detail, to evaluate the performance improvement contributed by RIA, we implement a version that uses PMA instead of RIA. For HITree, we implement another version that stores the edges of high-degree vertices using RIA instead of HITree. Finally, for LIA, we implement the third version that uses the binary search instead of the learned index in LIA. We compare the three versions with LSGraph. The results show that RIA, HITree, and LIA contribute 60.9%-83.4%, 6.9%-21.5%, and 1.8%-7.2% of performance improvement, respectively. Note that, when inserting 10^8 edges, the number of changes from RIA to HITree ranges from 29 (on LJ) to 1599 (on OR), which only causes 0.2%-3.1% runtime overhead. Without the changes, it will take a longer time to perform graph updates.

6.3 Graph Analytics Performance

We compare the performance of graph analytics algorithms running on four systems, and the results are shown in Figure 13 and Table 2. The results of PR, CC, and TC for Aspen and PaC-tree are omitted due to missing implementations in their open-source codes. The results are reported on the average of five trials for each graph analytics algorithm.

Figure 13 illustrates the running time normalized to LSGraph on BFS and BC of all systems. LSGraph outperforms Terrace, Aspen, and PaC-tree by 1.02×-1.16×, 1.98×-3.55×, and 1.28×-2.72× on BFS, and 1.07×-1.21×, 1.58×-2.14×, and 1.20×-2.12× on BC, respectively. LSGraph and Terrace perform better than Aspen and PaC-tree in all cases since the latters store all edges of the graph in the tree structure, which leads to poor cache locality. LSGraph outperforms Terrace mainly since the HITree designed by LSGraph has better cache locality than B-tree used by Terrace, and we also reduce the scenarios of HITree for higher locality due to the good update performance of RIA (details in Section 6.5) while reducing B-tree in Terrace heavily decreases its update throughput [54].

Table 2 illustrates the running times and speed up of LSGraph and Terrace on PR, CC, and TC. LSGraph achieves 1.24×-1.69×, 1.04×-1.53×, and 1.45×-4.28× speedup over Terrace on PR, CC, and TC, respectively. In general, PR exhibits superior speedup than BFS, BC, and CC since PR traverses the edges of the whole graph at the beginning and high-degree vertices usually converge after many rounds of iterations [79], leading to the problem of poor cache locality of B-tree more prominent. TC is a computationally intensive algorithm that repeatedly traverses the edges of the same vertices several times, especially the edges of high-degree vertices [55]. Terrace implements TC through multiple intersection operations by traversing different data structures. To improve locality and reduce programming complexity, we first store the edges of the vertices into arrays and then perform the intersection operation based on these arrays. Due to the fast traversal of our RIA and HITree, the overhead

Table 2. Execution times (in seconds) of LSGraph and Terrace on PR, CC, and TC. T/L denotes the speedup of LSGraph with respect to Terrace, Traversal and Tra/L denote the traversal time of LSGraph and the time ratio of traversal to LSGraph.

Graph	PR			CC			TC				
	LSGraph	Terrace	T/L	LSGraph	Terrace	T/L	LSGraph	Traversal	Terrace	T/L	Tra/L
LJ	0.184	0.239	1.30	0.053	0.055	1.04	1.335	0.148	2.031	1.52	10.99%
OR	0.352	0.593	1.69	0.099	0.152	1.53	3.535	0.689	5.116	1.45	19.48%
RM	1.782	2.205	1.24	0.309	0.348	1.13	13.151	2.351	22.130	1.68	17.88%
TW	8.553	11.902	1.39	2.187	2.677	1.22	792.797	5.044	3394.430	4.28	0.64%

Table 3. Memory usage (GB) of graphs in Table 1 on the different systems (T, A, P denoted as Terrace, Aspen, PaC-tree), and the ratio of index overhead (denoted as I/L) to LSGraph. T/L is the ratio of Terrace's memory usage to LSGraph.

Graph	LSGraph	T	A	P	T/L	I/L
LJ	0.61	1.51	0.58	0.35	2.48	2.90%
OR	1.27	2.51	0.89	0.73	1.98	4.96%
RM	5.67	18.02	3.99	3.47	3.18	5.43%
TW	16.58	44.70	12.36	8.92	2.70	3.16%
FR	23.66	51.72	22.76	14.99	2.19	4.06%

of storing the graph data in arrays is lightweight compared to the overall time (0.64%-19.48%).

6.4 Memory Footprint

Table 3 presents the memory usage of the four systems. The memory usage of Terrace is 1.98×-3.18× larger than that of LSGraph, primarily due to Terrace's utilization of a lower density of (0.125, 0.25) for the PMA, which corresponds to an amplification factor of (4, 8), whereas LSGraph uses an amplification factor of 1.2 by default. LSGraph consumes more memory than Aspen and PaC-tree, since they employ compressed data structures, whereas LSGraph takes uncompressed data structures. Although LSGraph consumes more memory than Aspen and PaC-tree, LSGraph achieves significantly higher performance. For batch sizes ranging from 10^4 to 10^7, the speedup-to-memory usage ratio of LSGraph is better than Aspen and PaC-tree by 2.27×-9.66× and 1.73×-6.53×, respectively. In addition, we analyze the amount of memory space occupied by index in LSGraph, which is the total of the index array of RIA and the model size of LIA. The results in Table 3 demonstrate that the memory footprint overhead of index in LSGraph is lightweight, with ratios ranging from 2.90% to 5.43%.

6.5 Sensitivity Analysis

To analyze the impact of α and M, we evaluate the performance of graph update and graph analytics of LSGraph on LJ, RM, and TW datasets, with a range of α between 1.1 and 2.0, and M ranging from 2^{12} to 2^{16}.

Table 4. A list of real-world streaming graph datasets with their characteristics

Graph	Vertices	Edges
mathoverflow (MO) [41]	24,818	506,550
askubuntu (AU) [41]	159,316	964,437
superuser (SU) [41]	194,085	1,443,339
wiki-talk-temporal (WT) [41]	1,140,149	7,833,140

Impact of Space Amplification Factor α. As shown in Figure 14, we observe that a drop in the value of α leads to a decrease in the overall performance of graph updates, which is particularly obvious when α changes from 1.2 to 1.1. The reason for this is that lots of unused space decreases the amount and distance of data movement during insertions, resulting in improved performance. Large α will incur more memory access when traversing edges of vertices, thereby lowering the performance of graph analytics, as shown in Figure 15. There is little difference in the performance of graph analytics when α is below 1.3. Therefore, we set α to 1.2 by default to trade off the performance of graph update and graph analytics (we do not use 1.3 since it increases memory usage).

Impact of Threshold M. Larger M results in more edges stored by RIA and a large number of horizontal data movements during insertion, which degrades graph update performance, particularly when there are a few unused spaces in the graph store. For example, as shown in Figure 14, when α = 1.1, the performance of TW continues to drop as the M increases since edges of these high-degree vertices are stored in HITree. The graph update performance of LJ and RM is smooth after 2^{12} because they have few high-degree vertices. Figure 15 illustrates that the performance of graph analytics exhibits a consistent and steady trend starting from 2^{12}. Therefore, we set M to 2^{12} by default, primarily with the aim of optimizing the efficiency of graph analytics. Note that even if we set M to 2^{10} that is the same as Terrace, our graph analytics performance also outperforms Terrace (e.g., when running PR on TW, LSGraph and Terrace take 9.03 and 11.90 seconds, respectively), mostly due to the superior cache locality of HITree compared to B-tree.

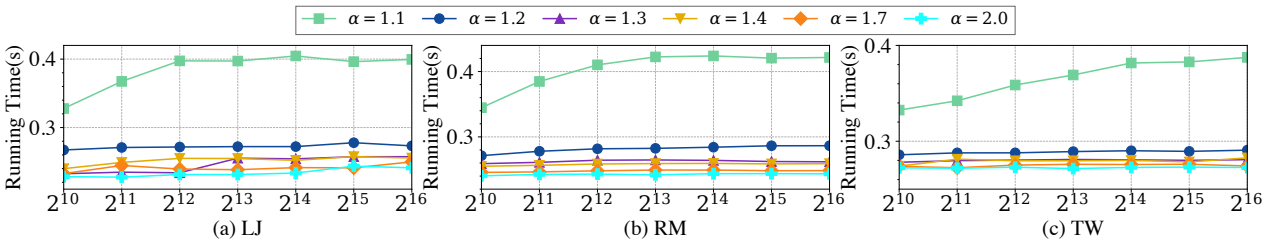


Figure 14. Running times (in seconds) of inserting 10^8 edges in LSGraph on LJ, RM, TW with different M and α

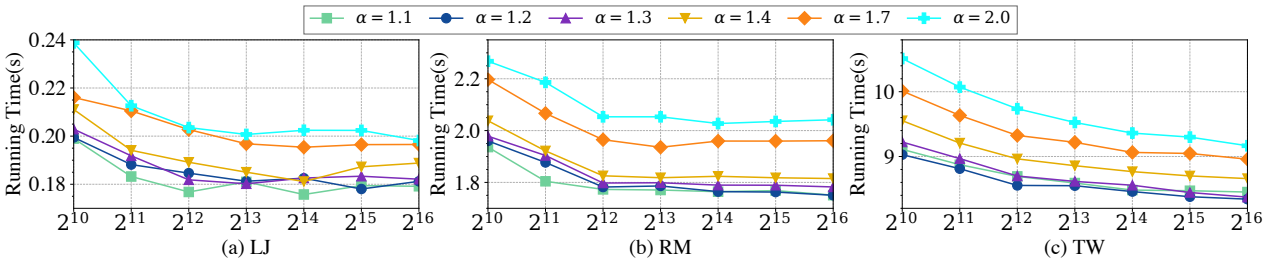


Figure 15. Running times (in seconds) of PR algorithm in LSGraph on LJ, RM, TW with different M and α

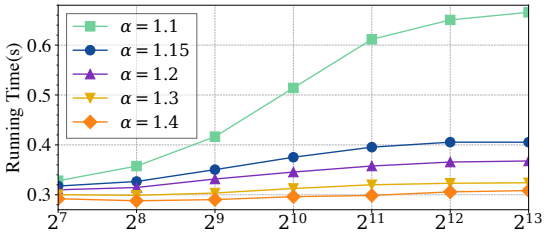


Figure 16. Average (5 trials) running times (in seconds) of continuous inserting 10^8 edges in LSGraph on OR with different M and α

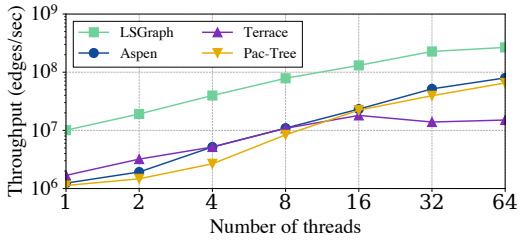


Figure 17. Scalability of multi-threading when inserting 10^7 edges in Terrace, Aspen, Pac-Tree, and LSGraph on OR in different number of threads

Scenarios with Frequent Insertions. To evaluate our update performance in such scenarios, we continuously insert five times in LSGraph on OR, with a batch size of 10^8 each time. We report the average running time in Figure 16, and the performance of the graph update degrades as the utilization of HITree decreases, particularly when α is quite small, due to HITree's vertical movement, which lowers large-scale data movements. This demonstrates the significance of our HITree design.

Scenarios with Real-world Streaming Graphs. We also evaluate the performance on real-world streaming graphs (see Table 4) with realistic data arrival patterns. Like previous work [50, 68], we treated 10% of the graph datasets as streaming edge additions and ran the test five times. The results show that LSGraph outperforms Terrace, Aspen, and PaC-tree by 1.63×-2.95×, 1.05×-2.42×, and 1.02×-1.82×, respectively.

Scenarios with Larger Graph Datasets. To evaluate the performance of LSGraph on larger graph datasets, we use the graph500 generator [51] to generate a graph dataset with 1 billion vertices and then convert it into a symmetrized version that contains 4.3 billion edges. The results show that LSGraph outperforms Aspen and PaC-tree by 4.64×-10.22× and 2.88×-29.37×, respectively.

6.6 Scalability

Figure 17 illustrates that LSGraph , Aspen, and Pac-Tree scale well across different numbers of threads, while Terrace does not scale beyond 16 threads. The reason for this phenomenon is mainly that several threads collaborate to insert data into a single array (i.e., PMA [71]) in Terrace, and they could change the same range of the array due to data movements, resulting in a high overhead of lock and cache coherence. Instead, we use an AL-based structure where edges of different vertices do not affect each other and regulate the distance of the data movement, thereby reducing the amount of data movement. Besides, the updated edges for a vertex are assigned to a

single thread for processing, which avoids the lock overhead and enhances cache locality.

7 Related Work

The Data Representations for Streaming Graph Systems. To support streaming graph updates, several graph representations have been proposed. Graph representations based on the CSR utilize one ordered gapped array (i.e., PMA [5]) for ingesting graph updates [60, 68, 70, 71]. The AL-based graph representations represent the edge sets of a vertex in various methods, such as block-based linked list [23, 26, 73, 74], array [3, 29, 81], hash table [2], and hybrid data structure [25, 28]. Tree-based graph representations leverage the search tree to provide efficient graph updates [16, 18]. Terrace [54] and GraphOne [36] trade-off the performance between graph updates and graph analytics using multiple data structures. However, these solutions are not efficient in simultaneously addressing the problems associated with graph update and graph analytics, including search, data movement, cache locality, and ordered neighbors.

Note that, similar to Terrace, LSGraph's RIA also uses the gapped array as Terrace's PMA. However, PMA suffers from high searching and moving overhead. Thus, RIA is not built on top of PMA, or implemented by adapting PMA. Rather, RIA is designed as a new data layout from scratch, which integrates a novel set of strategies to reduce data movement and searching. Further, LSGraph designs LIA and HITree to optimize the update of high-degree vertices. Although Sortledton [28] also targets similar memory access patterns as LSGraph (set operations for GPM), compared with LSGraph, Sortledton employs the array and the block-based skip list, which suffer from high data searching and moving overhead.

In summary, the above data representations encompass the design choices in three main directions: (1) one that maintains contiguous edge storage and ordered edges' IDs for graph analytics while reducing data searching and movement for graph update [17, 18, 54] , (2) one that focuses on graph update optimizations [36] (e.g., edge-list in differential dataflow [52]), and (3) one that only focuses on graph analytics optimizations [50]. LSGraph falls in the first direction, and outperforms other latest design choices in this direction. Note that several solutions are proposed to support transactions while running graph analytics on graph databases [22, 32, 35, 61], which is mostly orthogonal to our work.

The Learned Structures for Memory Systems. To support updated operations of the learned index, DPGM [27], XIndex [66], and FINEdex [42] adopt the delta-buffer insert strategy, which maintains delta buffers for inserting data and merges the buffers into index structure periodically, and ALEX [20], LIPP [75], and AFLI [76] use the in-place insert strategy, which keeps gaps in arrays for inserting [64]. In addition to the dynamic workloads, prior works also apply the learned index to various scenarios, such as LSM-based [15, 45] and RDMA-based KV stores [43, 69], persistent for NVM [80], flash-based SSD [63], and some learned systems [21, 30, 44]. However, they do not address the locality problem of streaming graph updates and analytics.

8 Conclusion

This paper proposes LSGraph, a novel locality-centric streaming graph engine to efficiently support both graph update and graph analytics. We design a differentiated hierarchical indexed streaming graph representation to enable efficient search and maintain locality. We further regulate the distance of data movement when inserting into the above representation to reduce the overhead of memory access. Our evaluation results indicate that LSGraph outperforms cutting-edge streaming graph systems on both graph updates and graph analytics.

Acknowledgments

We would like to thank our shepherd H. Howie Huang and all anonymous reviewers for their constructive comments and suggestions. Yu Zhang (zhyu@hust.edu.cn) is the corresponding author of this paper. This paper is supported by National Key Research and Development Program of China (No. 2022YFB2404202), Key Research and Development Program of Hubei Province (No. 2023BAB078), Knowledge Innovation Program of Wuhan-Basi Research (No. 2022013301015177), and Huawei Technologies Co., Ltd (No. YBN2021035018A6).

References

[1] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proceedings of the VLDB Endowment* 11, 6 (2018), 691–704.

[2] Muhammad A. Awad, Saman Ashkiani, Serban D. Porumbescu, and John D Owens. 2020. Dynamic Graphs on the GPU. In *Proceedings of the 2020 IEEE International Parallel and Distributed Processing Symposium*. 739–748.

[3] Abanti Basak, Zheng Qu, Jilan Lin, Alaa R. Alameldeen, Zeshan Chishti, Yufei Ding, and Yuan Xie. 2021. Improving Streaming Graph Processing Performance Using Input Knowledge. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1036–1050.

[4] Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, and Torsten Hoefler. 2019. A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning. In *Proceedings of the 2019 IEEE International Parallel and Distributed Processing Symposium*. 66–77.

[5] Michael A. Bender and Haodong Hu. 2007. An Adaptive Packed-Memory Array. *ACM Transactions on Database Systems* 32, 4 (2007), 26:1–26:43.

[6] Maciej Besta, Marc Fischer, Vasiliki Kalavri, Michael Kapralov, and Torsten Hoefler. 2023. Practice of Streaming Processing of Dynamic Graphs: Concepts, Models, and Systems. *IEEE Transactions on Parallel and Distributed Systems* 34, 6 (2023), 1860–1876.

[7] Maciej Besta, Raghavendra Kanakagiri, Grzegorz Kwasniewski, Rachata Ausavarungnirun, Jakub Beránek, Konstantinos Kanellopoulos, Kacper Janda, Zur Vonarburg-Shmaria, Lukas Gianinazzi, Ioana

Stefan, Juan Gómez Luna, Jakub Golinowski, Marcin Copik, Lukas Kapp-Schwoerer, Salvatore Di Girolamo, Nils Blach, Marek Konieczny, Onur Mutlu, and Torsten Hoefler. 2021. SISA: Set-Centric Instruction Set Architecture for Graph Mining on Processing-in-Memory Systems. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 282–297.

[8] Laurent Bindschaedler, Jasmina Malicevic, Baptiste Lepers, Ashvin Goel, and Willy Zwaenepoel. 2021. Tesseract: Distributed, General Graph Pattern Mining on Evolving Graphs. In *Proceedings of the 16th European Conference on Computer Systems*. 458–473.

[9] Mahdi Nazm Bojnordi and Farhan Nasrullah. 2019. ReTagger: An Efficient Controller for DRAM Cache Architectures. In *Proceedings of the 56th Annual Design Automation Conference*. 1–6.

[10] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. 2004. R-MAT: A Recursive Model for Graph Mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*. 442–446.

[11] Qihang Chen, Boyu Tian, and Mingyu Gao. 2022. FINGERS: Exploiting Fine-Grained Parallelism in Graph Mining Accelerators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 43–55.

[12] Xuhao Chen and Arvind. 2022. Efficient and Scalable Graph Pattern Mining on GPUs. In *Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation*. 857–877.

[13] Xuhao Chen, Tianhao Huang, Shuotao Xu, Thomas Bourgeat, Chanwoo Chung, and Arvind. 2021. FlexMiner: A Pattern-Aware Accelerator for Graph Pattern Mining. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*. 581–594.

[14] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. The MIT Press.

[15] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. 2020. From WiscKey to Bourbon: A Learned Index for Log-Structured Merge Trees. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 155–171.

[16] 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.

[17] Laxman Dhulipala, Guy E. Blelloch, Yan Gu, and Yihan Sun. 2022. Pactrees: Supporting Parallel and Compressed Purely-functional Collections. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 108–121.

[18] 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*. 918–934.

[19] Luisa Di Paola, Micol De Ruvo, Paola Paci, Daniele Santoni, and Alessandro Giuliani. 2013. Protein Contact Networks: An Emerging Paradigm in Chemistry. *Chemical Reviews* 113, 3 (2013), 1598–1613.

[20] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. 2020. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[21] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska. 2020. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. *Proceedings of the VLDB Endowment* 14, 2 (2020), 74–86.

[22] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proceedings of the VLDB Endowment* 9, 11 (2016), 852–863.

[23] David Ediger, Rob McColl, Jason Riedy, and David A. Bader. 2012. STINGER: High Performance Data Structure for Streaming Graphs. In *Proceedings of the 2012 IEEE Conference on High Performance Extreme Computing*. 1–5.

[24] Priyank Faldu, Jeff Diamond, and Boris Grot. 2020. Domain-Specialized Cache Management for Graph Analytics. In *Proceedings of the 2020 IEEE International Symposium on High Performance Computer Architecture*. 234–248.

[25] 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*. 513–527.

[26] Guoyao Feng, Xiao Meng, and Khaled Ammar. 2015. DISTINGER: A Distributed Graph Data Structure for Massive Dynamic Graph Processing. In *Proceedings of the 2015 IEEE International Conference on Big Data*. 1814–1822.

[27] Paolo Ferragina and Giorgio Vinciguerra. 2020. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. *Proceedings of the VLDB Endowment* 13, 8 (2020), 1162–1175.

[28] 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.

[29] Oded Green and David A. Bader. 2016. cuSTINGER: Supporting Dynamic Graph Aigorithms for GPUs. In *Proceedings of the 2016 IEEE High Performance Extreme Computing Conference*. 1–6.

[30] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, Not from Queries! *Proceedings of the VLDB Endowment* 13, 7 (2020), 992–1005.

[31] Xiaolin Jiang, Chengshuo Xu, Xizhe Yin, Zhijia Zhao, and Rajiv Gupta. 2021. Tripoline: Generalized Incremental Graph Processing via Graph Triangle Inequality. In *Proceedings of the 16th European Conference on Computer Systems*. 17–32.

[32] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. 2017. ZipG: A Memory-efficient Graph Store for Interactive Queries. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1149–1164.

[33] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *Proceedings of the 2019 IEEE International Symposium on High Performance Computer Architecture*. 596–608.

[34] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The Case for Learned Index Structures. In *Proceedings of the 2018 International Conference on Management of Data*. 489–504.

[35] Pradeep Kumar and H. Howie Huang. 2016. G-Store: High-Performance Graph Store for Trillion-Edge Processing. In *Proceedings of the 2016 International Conference for High Performance Computing, Networking, Storage and Analysis*. 830–841.

[36] Pradeep Kumar and H. Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 249–263.

[37] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *Proceedings of the 22nd International Conference on World Wide Web*. 1343–1350.

[38] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web*. 591–600.

[39] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*. 75–86.

[40] Charles E. Leiserson. 2009. The Cilk++ Concurrency Platform. In *Proceedings of the 46th Annual Design Automation Conference*. 522–527.

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

[42] Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. 2021. FINEdex: A Fine-Grained Learned Index Scheme for Scalable and Concurrent Memory Systems. *Proceedings of the VLDB Endowment* 15, 2 (2021), 321–334.

[43] Pengfei Li, Yu Hua, Pengfei Zuo, Zhangyu Chen, and Jiajie Sheng. 2023. ROLEX: A Scalable RDMA-oriented Learned Key-Value Store for Disaggregated Memory Systems. In *Proceedings of the 21st USENIX Conference on File and Storage Technologies*. 99–114.

[44] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2119–2133.

[45] Kai Lu, Nannan Zhao, Jiguang Wan, Changhong Fei, Wei Zhao, and Tongliang Deng. 2022. TridentKV: A Read-Optimized LSM-Tree Based KV Store via Adaptive Indexing and Space-Efficient Partitioning. *IEEE Transactions on Parallel and Distributed Systems* 33, 8 (2022), 1953–1966.

[46] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. 2017. MOSAIC: Processing a Trillion-Edge Graph on a Single Machine. In *Proceedings of the 12th European Conference on Computer Systems*. 527–543.

[47] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient Graph Analytics Using Large Multiversioned Arrays. In *Proceedings of the 2015 IEEE International Conference on Data Engineering*. 363–374.

[48] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-Scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*. 135–146.

[49] Mugilan Mariappan, Joanna Che, and Keval Vora. 2021. DZiG: Sparsity-Aware Incremental Processing of Streaming Graphs. In *Proceedings of the 16th European Conference on Computer Systems*. 83–98.

[50] Mugilan Mariappan and Keval Vora. 2019. GraphBolt: Dependency-Driven Synchronous Processing of Streaming Graphs. In *Proceedings of the 14th European Conference on Computer Systems*. 1–16.

[51] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. 2010. Introducing the Graph 500. *Cray Users Group (CUG)* 19 (2010), 45–74.

[52] 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 (2016), 75–83.

[53] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*. 456–471.

[54] 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*. 1372–1385.

[55] Hao Qi, Yu Zhang, Ligang He, Kang Luo, Jun Huang, Haoyu Lu, Jin Zhao, and Hai Jin. 2023. PSMiner: A Pattern-Aware Accelerator for High-Performance Streaming Graph Pattern Mining. In *Proceedings of the 60th ACM/IEEE Design Automation Conference*. 1–6.

[56] Shafiur Rahman, Nael Abu-Ghazaleh, and Rajiv Gupta. 2020. GraphPulse: An Event-Driven Hardware Accelerator for Asynchronous Graph Processing. In *Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture*. 908–921.

[57] David Sayce. 2022. The Number of tweets per day in 2022. https://www.dsayce.com/social-media/tweets-day.

[58] Tao B. Schardl and I-Ting Angelina Lee. 2023. OpenCilk: A Modular and Extensible Software Infrastructure for Fast Task-Parallel Code. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. 189–203.

[59] Tao B. Schardl, William S. Moses, and Charles E. Leiserson. 2019. Tapir: Embedding Recursive Fork-join Parallelism into LLVM's Intermediate Representation. *ACM Transactions on Parallel Computing* 6, 4 (2019), 1–33.

[60] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proceedings of the VLDB Endowment* 11, 1 (2017), 107–120.

[61] Bin Shao, Haixun Wang, and Yatao Li. 2013. Trinity: A Distributed Graph Engine on a Memory Cloud. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 505–516.

[62] Julian Shun and Guy E. Blelloch. 2013. Ligra: A Lightweight Graph Processing Framework for Shared Memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 135–146.

[63] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. 2023. LeaFTL: A Learning-Based Flash Translation Layer for Solid-State Drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 442–456.

[64] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.

[65] Toyotaro Suzumura, Shunsuke Nishii, and Masaru Ganse. 2014. Towards Large-Scale Graph Stream Processing Platform. In *Proceedings of the 23rd International Conference on World Wide Web*. 1321–1326.

[66] Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. 2020. XIndex: A Scalable Learned Index for Multicore Data Storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 308–320.

[67] Keval Vora, Rajiv Gupta, and Guoqing Xu. 2017. KickStarter: Fast and Accurate Computations on Streaming Graphs via Trimmed Approximations. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*. 237–251.

[68] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. 2021. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 149–159.

[69] Xingda Wei, Rong Chen, and Haibo Chen. 2020. Fast RDMA-based Ordered Key-Value Store using Remote Learned Cache. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation*. 117–135.

[70] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *Proceedings of the 2018 IEEE High Performance Extreme Computing Conference*. 1–7.

[71] Brian Wheatman and Helen Xu. 2021. A Parallel Packed Memory Array to Store Dynamic Graphs. In *Proceedings of the 2021 Workshop on Algorithm Engineering and Experiments*. 31–45.

[72] Charith Wickramaarachchi, Alok Kumbhare, Marc Frincu, Charalampos Chelmis, and Viktor K. Prasanna. 2015. Real-time Analytics for Fast Evolving Social Graphs. In *Proceedings of the 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. 829–834.

[73] Martin Winter, Daniel Mlakar, Rhaleb Zayer, Hans-Peter Seidel, and Markus Steinberger. 2018. faimGraph: High Performance Management of Fully-Dynamic Graphs Under Tight Memory Constraints on the GPU. In *Proceedings of the 2018 International Conference for High Performance Computing, Networking, Storage and Analysis*. 754–766.

[74] Martin Winter, Rhaleb Zayer, and Markus Steinberger. 2017. Autonomous, Independent Management of Dynamic Graphs on GPUs. In *Proceedings of the 2017 IEEE High Performance Extreme Computing Conference*. 1–7.

[75] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. 2021. Updatable Learned Index with Precise Positions. *Proceedings of the VLDB Endowment* 14, 8 (2021), 1276–1288.

[76] Shangyu Wu, Yufei Cui, Jinghuan Yu, Xuan Sun, Tei-Wei Kuo, and Chun Jason Xue. 2022. NFL: Robust Learned Index via Distribution Transformation. *Proceedings of the VLDB Endowment* 15, 10 (2022), 2188–2200.

[77] Yanfeng Zhang, Qixin Gao, Lixin Gao, and Cuirong Wang. 2013. Maiter: An Asynchronous Graph Processing Framework for Delta-Based Accumulative Iterative Computation. *IEEE Transactions on Parallel and Distributed Systems* 25, 8 (2013), 2091–2100.

[78] Yu Zhang, Xiaofei Liao, Lin Gu, Hai Jin, Kan Hu, Haikun Liu, and Bingsheng He. 2020. AsynGraph: Maximizing Data Parallelism for Efficient Iterative Graph Processing on GPUs. *ACM Transactions on Architecture and Code Optimization* 17, 4 (2020), 29:1–29:21.

[79] Yu Zhang, Xiaofei Liao, Hai Jin, Ligang He, Bingsheng He, Haikun Liu, and Lin Gu. 2021. DepGraph: A Dependency-Driven Accelerator for Efficient Iterative Graph Processing. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 371–384.

[80] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proceedings of the VLDB Endowment* 16, 2 (2022), 243–255.

[81] 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.