GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing

Hongfu Li*
Northeastern Univ., China
lihongfu@stumail.neu.edu.cn

Qian Tao*
Tongyi Lab, Alibaba Group
qian.tao@alibaba-inc.com

Song Yu
Northeastern Univ., China
yusong@stumail.neu.edu.cn

Shufeng Gong
Northeastern Univ., China
gongsf@mail.neu.edu.cn

Yanfeng Zhang
Northeastern Univ., China
zhangyf@mail.neu.edu.cn

Feng Yao
Northeastern Univ., China
yaofeng@stumail.neu.edu.cn

Wenyuan Yu
Tongyi Lab, Alibaba Group
wenyuan.ywy@alibaba-inc.com

Ge Yu
Northeastern Univ., China
yuge@mail.neu.edu.cn

Jingren Zhou
Tongyi Lab, Alibaba Group
jingren.zhou@alibaba-inc.com

ABSTRACT

An efficient data structure is fundamental to meeting the growing demands in dynamic graph processing. However, the dual requirements for graph computation efficiency (with contiguous structures) and graph update efficiency (with linked list-like structures) present a conflict in the design principles of graph structures. After experimental studies of state-of-the-art dynamic graph structures, we observe that the overhead of cache misses accounts for a major portion of the graph computation time. This paper presents `GastCoCo`, a system with graph storage and coroutine-based prefetch co-design. By employing software prefetching via stackless coroutines and designing a prefetch-friendly data structure `CBList`, `GastCoCo` significantly alleviates the performance degradation caused by cache misses. Our results show that `GastCoCo` outperforms state-of-the-art graph storage systems by 1.3× - 180× in graph updates and 1.4× - 41.1× in graph computation.

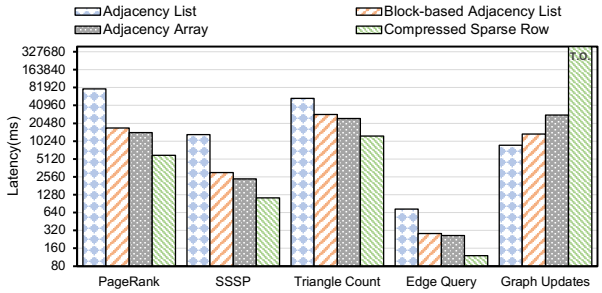
PVLDB Reference Format:
Hongfu Li, Qian Tao, Song Yu, Shufeng Gong, Yanfeng Zhang, Feng Yao, Wenyuan Yu, Ge Yu, and Jingren Zhou. GastCoCo: Graph Storage and Coroutine-Based Prefetch Co-Design for Dynamic Graph Processing. PVLDB, 17(13): 4827 - 4839, 2024.
doi:10.14778/3704965.3704986

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

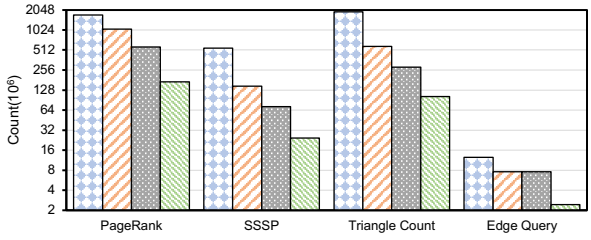
1 INTRODUCTION

With the increase in data scale and diversification of data types, dynamic graph processing has become a critically important issue in various domains such as e-commerce, financial technology, and social networks [4, 45, 49]. Specifically, dynamic graph processing involves executing graph computation algorithms on graphs that experience frequent structural changes, such as millions of edge updates per hour [24, 42]. For instance, there are more than 400 million behaviors per day [1] between users and items on Taobao [2]. The platform should employ a combination of graph analytics and interactive graph traversals for fraud detection or interactive pattern mining [17]. Such applications on dynamic graphs require not only efficient graph computations but also supporting the large volume of updates per second [23, 38, 43].

Regrettably, the need for computation efficiency and frequent update support presents a contradictory challenge, leading to a dilemma in designing storage structures for dynamic graphs. Take the widely used graph storage structures, Adjacency List (AL) and Compressed Sparse Row (CSR) [41], as an example. AL is a graph data structure where the neighbors of each vertex are stored in a



(a) Algorithm execution time and graph update time.



(b) CPU cache stall count.

Figure 1: The execution time for graph algorithms (and graph updates) and the CPU cache stall count on different data structures (T.O.: graph updates cannot finish in 24 hours).

Proceedings of the VLDB Endowment, Vol. 17, No. 13 ISSN 2150-8097.
doi:10.14778/3704965.3704986

*Hongfu Li and Qian Tao contributed equally.

linked list. Since the nodes in the linked list are non-contiguous memory fragments connected by pointers, it is easy to insert/delete nodes without data movements on the AL structure, which is suitable for graph updates. On the other hand, CSR has been widely used for graph storage in static graphs due to its memory and computational efficiency. Figure 1a reports the execution time for graph computations (PageRank, SSSP, Triangle Count, and Edge Query) and updates (Insertion) under a single thread on the LiveJournal dataset [5]. We could observe that CSR is more efficient than AL on graph algorithms, but spends more time for updating graphs.

To effectively support both graph computation and graph updates, some efforts propose designing graph structures that maintain a moderate degree of contiguity [18, 22, 30, 36, 52]. For example, Adjacency Array (AA) [52] stores all neighbors of each vertex in a separate array and Block-based Adjacency List (BAL) [18] stores the neighbors of each vertex in a block-based linked list (*i.e.,* a node contains more than one neighbors). The degrees of contiguity in these two structures fall between CSR and AL. More recently, more complex and detailed implementations, *e.g.,* the state-of-the-art dynamic graph storage systems like GraphOne [30], Stinger [18], Terrace [36], and Sortledton [22], maintain the graphs with a linked data structure fragmented in memory with various degrees of contiguity as a halfway house. However, such a halfway-house solution can only balance the performance of graph computation and graph updates, which sacrifices the efficiency of graph computation to enhance graph update throughput, or vice versa.

To avoid differences in optimization and implementation across systems, we use uniformly implemented simple data structures for the cache stall count experiments in Figure 1b. As representatives of "halfway-house solutions", BAL and AL exhibit a large number of cache misses in graph computation tasks. This motivates us to seek a novel perspective to improve both the efficiency and dynamic graph support of dynamic graph processing: *Can we mitigate the overhead of cache misses from the linked data structures to improve the efficiency for both computation and updating?*

To answer the question, we consider a co-design of the prefetching techniques and graph structures. Prefetching techniques allow us to load data, which will be accessed later, from memory into the CPU cache, thereby reducing the data access latency. These techniques consist of both hardware prefetching, where the CPU automatically loads data and instructions, and software prefetching, where programmers explicitly insert instructions into the code to prefetch data. Although hardware prefetching techniques are commonly equipped on standard hardware, software prefetching techniques have recently demonstrated their effectiveness in many applications [9, 25, 50]. However, when it comes to the practical dynamic graph processing domain, there are still challenges.

Our approach. This paper presents `GastCoCo`, an in-memory system with GrAph STorage and COroutine-based prefetch CO-design, which is designed for dynamic graph processing applications to alleviate the performance overhead caused by cache misses. ①In terms of hardware prefetching, the prefetching effectiveness varies across different data structures. `GastCoCo` proposes a prefetch-aware data structure `CBList` for dynamic graph processing. ②Software prefetching is used to compensate for cases where hardware prefetching fails, so the benefits of applying software prefetching are our focus. `GastCoCo` develops a set of hybrid prefetching strategies to avoid fetching reduplicated data by both software and hardware prefetching. To minimize the overhead of applying software prefetching and increase the prefetch success rate, `GastCoCo` leverages C++20 [27] stackless coroutines to prefetch the graph data that the program requires to access, benefiting both graph computation and graph updates by mitigating the cache miss overhead. ③Finally, `GastCoCo` designs a set of adaptive coroutine scheduling and task allocation strategies tailored to different graph tasks, allowing graph processing using software prefetching to perform better on `GastCoCo`.

Contributions. To sum up, the contributions of this paper include:

- The first to employ the instruction stream interleaving execution mode composed of coroutines and software prefetching to reduce cache misses in dynamic graph processing, thereby enhancing the performance of both graph computation and graph updates.
- A dynamic graph data structure `CBList`, specifically designed for efficient dynamic graph processing, which not only improves the effectiveness of hardware prefetching but also facilitates the implementation of software prefetching via coroutines.
- An efficient graph storage system `GastCoCo` that is equipped with a set of optimizations, *e.g.,* task allocation, coroutine scheduling, and hybrid prefetching, to minimize the overhead of software prefetching via coroutine as much as possible on different graph tasks and runtime environments. Our results show that `GastCoCo` outperforms state-of-the-art graph storage systems by 1.3× - 180× in graph updates and 1.4× - 41.1× in graph computation.

2 PRELIMINARIES

In this section, we first summarize the data access patterns in graph computations and then discuss how hardware and software prefetching work in graph processing.

2.1 Graph Operations and Data Access Patterns

There are numerous types of dynamic graph processing tasks and their performances are usually regarded as memory access bounded [7, 19]. Most of these tasks can be deconstructed into a bunch of data access operations on vertices and edges. The operations to access vertices are listed as follows.

- *scan_vertices()* traverses all vertices in the entire graph.
- *scan_vertices(cond)* makes certain conditional filtering during the traversal, and the vertices that meet the conditions *cond* can undergo subsequent operations.
- *read_vertex(v)* reads a specific vertex v.

Processing edges can be considered as first locating the edges via the source vertex v_{src} and then processing the neighbors of v_{src}. Since locating the source vertex can be achieved by *read_vertex(v)*, our focus is exclusively on the process of neighbors.

- *scan_edges(v_{src})* traverses all neighbors of a vertex v_{src}.
- *read_edge(v_{src},v_{dst})* reads a particular edge from a source vertex v_{src} to a destination vertex v_{dst}.

For example, in each iteration, Single Source Shortest Path (SSSP) algorithm [10] requires processing all neighbors of vertices whose status was updated in the previous iteration. This can be represented as a combination of *scan_vertices(cond)* and *scan_edges(v_{src})*. The edge query [42] between two vertices involves *read_vertex(v)* and *read_edge(v_{src},v_{dst})* operations.

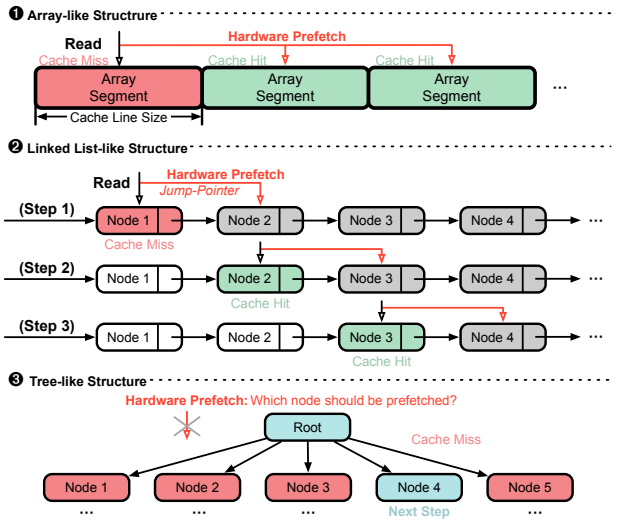


Figure 2: Hardware prefetching.

In summary, these operations can be categorized as sequential and random memory access.

Sequential Data Access. Some tasks require sequential data access, *i.e.,* to access the storage structure in a fixed order (*e.g.,* the lexicographical order). If the structure is physically contiguous in memory, sequential data access can be regarded as sequential memory access. For example, *scan_vertices()* and *scan_edges(v_{src})* can be regarded as sequential data access or memory access on arrays.

Random Data Access. Random data access refers to accessing the storage structure (or part) in a random order. Since the access is unpredicted, performing random data access on either contiguous or non-contiguous memory can be considered as random memory access. Correspondingly, *scan_vertices(cond)* can be regarded as random data access because *cond* is unpredictable. *read_vertex(v)* and *read_edge(v_{src},v_{dst})* also access data in a random order.

2.2 Hardware Prefetching in Graph Processing

Hardware prefetching is a predefined mechanism in the processor that, based on the data stream requested by the running program, identifies and prefetches the subsequent elements that the program might need, loading them into the processor's cache beforehand [6]. However, its effectiveness is affected by different data structures. Dynamic graph storage generally incorporates contiguous and non-contiguous memory structures to support efficient graph computation and updates. Specifically, Array-like structures [32, 41, 47] use contiguous memory, while Linked List-like [18, 22, 30] and Tree-like structures [16, 22, 36] use non-contiguous memory.

Hardware prefetching in Array-like structures. The data in an Array-like structure is stored contiguously in the main memory. As shown in Figure 2 ❶, when *scan_edges(v_{src})* is performed on an array, the hardware prefetchers prefetch the next several segments adjacent to the requested segment into the cache from the main memory. Here, the size of a segment matches that of a cache line [6, 20, 44]. Therefore, when performing sequential memory access on array-like structures (*e.g., scan_edges(v_{src})* in CSR), most of the required data will be hit in the CPU cache. However, if the operation randomly accesses data, most of the data will be missed.

Hardware prefetching in Linked List-like structures. The data stored in the Linked List-like structure is discontiguous, and pointers in memory connect the data. Under the sequential data access pattern, the data to be accessed next can be known in advance via the fixed access paths. Modern hardware prefetchers prefetch data in the sequential data access pattern by *jump-pointer mechanism* [12, 20, 33, 39, 40]. In Figure 2 ❷, the jump-pointer mechanism prefetches Node 2 when the processor reads Node 1, preventing a cache miss when the processor reads Node 2. However, such a jump-pointer mechanism does not work for the head of a linked list. For instance, upon initially entering the linked list in Figure 2 ❷, Node 1 can not be prefetched, leading to a cache miss. We offer the following explanation regarding the occurrence of cache misses: i) The first node is not located in the caches with a high possibility while accessing a linked list. Therefore, it results in a cache miss (in fact, the front few nodes might trigger cache misses, depending on how many nodes the jump-pointer mechanism prefetches ahead). ii) Even if the memory address of the next node is far from the current block, the jump-pointer mechanism can load the afterward nodes in advance while traversing on a linked list as shown in Figure 2. Therefore, fewer cache misses occur in the subsequent process.

Hardware prefetching in Tree-like structures. Several previous works, like Aspen [16] and Terrace [36], utilize tree-like structures to store the high-degree vertices. In these structures, there is little inherent spatial locality between accessed nodes since they are dynamically allocated from the heap and can have arbitrary addresses [33]. The hardware prefetching no longer works for such structures [20, 25] as blocks may contain more than one pointer, and hardware prefetchers cannot predict the path that will be selected. Therefore, both *scan_edges(v_{src})* and *read_edge(v_{src},v_{dst})* (sequential and random data access in a tree) result in a large number of cache misses.

In summary, hardware prefetching mechanisms provide more direction for our data structure design. However, the unexpected performance of hardware prefetching inspires us to employ other modern techniques to eliminate some of the cache misses from data fetching, such that a fragmented data structure for dynamic graph computing could also achieve better performance.

2.3 Software Prefetching via Coroutines

Although hardware prefetching relieves the access time for contiguous data, it performs badly on graphs. In response, recent efforts utilize coroutines [15, 28] to mitigate cache misses. For example, prior works [11, 25, 28, 37] propose optimizing the overhead of cache misses caused by random memory access, by introducing software prefetching via coroutines. Software prefetching techniques [11, 28, 29, 37] leverage workload semantics to issue prefetch instructions [26] to explicitly bring data into CPU caches [25].

This solution enables the overlapping of data loading and computation. Data can be explicitly loaded using prefetch instructions as needed. However, given the delay involved in moving data from memory to cache, the CPU switches to another task to compute to overlap with data loading time. We need a tool to finish the switching between tasks. While the multithreading approach is

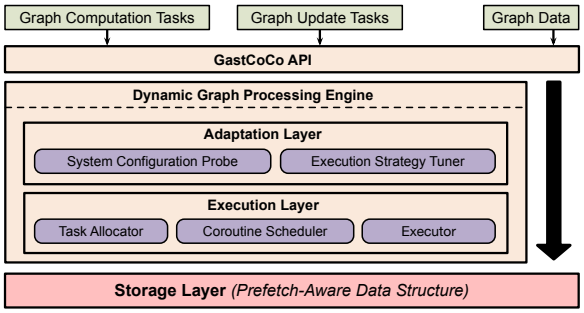


Figure 3: Overview of GastCoCo.

straightforward, it has much heavy overhead on thread switching relative to the cost of LLC cache miss. To this end, we need new techniques to switch between tasks.

C++20 Stackless Coroutine are special functions that can be suspended voluntarily and resumed later at low cost during the execution [14, 15, 25]. The efficient suspension and resumption of stackless coroutines make it possible to implement the interleaving execution mode. Specifically, when issuing a software prefetching instruction, we can suspend the coroutine, switch to another coroutine for computing, and later resume the previous coroutine to continue the corresponding task. Stackless coroutines are essentially state machines, *i.e.,* coroutine switching simply involves changing the instruction pointer register. All coroutines within a physical thread share the system stack, and it is unnecessary to save the states of registers explicitly. The stackless coroutines in C++20 [27] standard exhibit low overhead in terms of construction and context switching [25], and GastCoCo is constructed based on C++20 stackless coroutines.

Hardware prefetching is an automatic prefetching technique executed by the processors but shows inefficiency in many cases (as discussed in subsection 2.2). We apply software prefetching to complement hardware prefetching, achieving a more effective performance. Switching between coroutines can cause considerable cost, and our work focuses on how to implement effective software prefetching for dynamic graphs.

3 OVERVIEW OF GASTCOCO

This paper proposes GastCoCo, an in-memory system for dynamic graph processing that employs an interleaved execution mode combining coroutines and software prefetching. GastCoCo focuses on graph storage and coroutine-based prefetch co-design, proposing a novel dynamic graph data structure friendly to prefetching techniques. Besides, GastCoCo comprises various task allocating, prefetching, and scheduling strategies to mitigate the overhead of cache misses, thereby delivering robust performance for different graph processing tasks across various hardware environments.

As shown in Figure 3, GastCoCo is primarily composed of three layers: storage layer, execution layer, and adaptation layer. Given a graph processing task (*i.e.,* graph computation or graph update), user execution begins by calling GastCoCo API, as shown in Table 1.

GastCoCo API activates the dynamic graph processing engine, at which point the system configuration probe in the adaptation layer detects the hardware environment to configure the system

Table 1: GastCoCo APIs.

Name	Parameters	Description
LoadGraph	(file_path)	Load graph data files.
UpdateVertex	(obj_v, update_mode)	Add, delete, and modify vertices.
UpdateEdge	(obj_e, update_mode)	Add, delete, and modify edges.
BatchUpdate	(update_content)	Handle batch update tasks.
ProcessVertex	(f, active)	Process vertices task.
ProcessEdge	(dense_f, sparse_f, active)	Process edges task.

parameters of GastCoCo, *e.g.,* the numbers of coroutines per thread and hybrid prefetching strategy. To process tasks using software prefetching, graph data should be partitioned into several parts and allocated to a coroutine within a coroutine pool. All coroutines collaborate according to the scheduling strategy to produce the final results. The execution strategy tuner tailors the partition and scheduling strategy to suit the specific task to achieve better performance. The task allocator partitions the graph data according to the partition strategy and constructs the coroutine pool, while the coroutine scheduler schedules coroutines according to the scheduling strategy. Finally, the executor performs all read and write operations on the storage layer.

Storage Layer (§4). This layer is our specially designed data structure CBList, which supports efficient graph computation and frequent graph updates. The design of CBList is made to align as closely as possible with hardware prefetching mechanisms (as discussed in subsection 2.2) for reducing cache misses.

Execution Layer (§5). The execution layer includes the task allocator which breaks down a task into multiple subtasks and allocates them to coroutines, the coroutine scheduler which controls the interleaving execution and prefetching, and the executor which executes the computation tasks according to the scheduling strategy.

Adaptation Layer (§6). The adaptation layer includes two components: the system configuration probe and the execution strategy tuner. They utilize prefabricated programs to probe the optimal coroutine parameters (*e.g.,* number of coroutines per thread) and the most suitable execution and prefetching strategy for the current tasks and runtime environment.

4 PREFETCH-AWARE STRUCTURE CBLIST

4.1 Overview

For dynamic graph processing tasks, we have the following expectations for an ideal data structure. It should: (1) maintain cache locality as possible to produce efficient graph computation; (2) be easy to dynamically adjust for frequent graph updates. To enhance the performance from both perspectives, a basic idea is to tailor as closely as possible to fit the hardware prefetching mechanisms and prepare for the implementation of software prefetching.

Figure 4 presents the proposed data structure CBList and we elaborate on its details as follows. For *storing vertices*, we employ an ID-map table and a vertex table (array of vertex structures), as shown in Figure 4 ❶ and ❷, to map the vertex's original ID to the logical ID (unique number identifier used in graph computation). A record contains multiple fields as presented in Figure 4 ❸, which includes two pointers to the neighbors (*i.e.,* traversal pointer and update/query pointer), size (number of adjacent edges), delete flag,

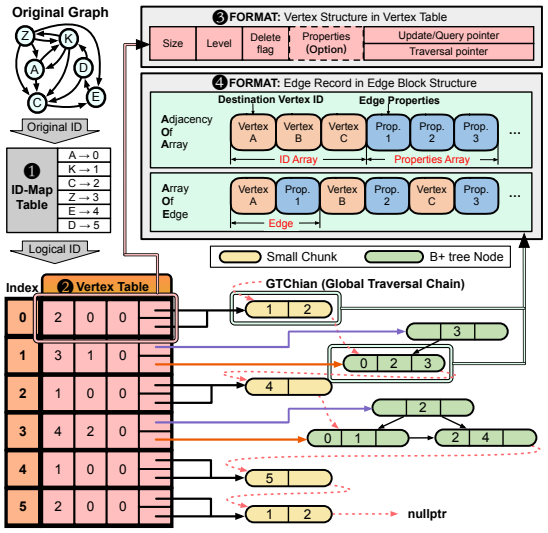


Figure 4: Prefetch-aware structure `CBList`. (The capacities of small chunks and B+ tree nodes are set as 2 and 3 edges.)

and the level. Here, we use a level variable to distinguish the structure of the vertex's neighborhood. A value 0 represents using small chunks for edge storage, while other numbers indicate using B+ trees and the value represents the number of leaf nodes in the tree.

For *storing edges*, we design the update-read balanced edge storage with hierarchical structures, as shown in the lower right of Figure 4. The details will be introduced in subsection 4.2. The edges of each edge block structure are organized as shown in Figure 4 ❹. Two pointers, namely the traversal pointer and update/query pointer, are utilized to link each edge storage and the vertex table. We provide two edge property storage modes: AOE (Array Of Edge) and AOA (Adjacency Of Array), as depicted in Figure 4 ❹.

Finally, we also design a prefetch-friendly global traversal chain to fully take advantage of hardware prefetching for graph traversal, the details of which can be found in subsection 4.3.

4.2 Update-Read Balanced Edge Storage

Array-like structure, *e.g.,* CSR, suffers from poor graph update performance due to data movements. However, we found that for small data volumes, the data movement cost is negligible since only a few data moves. Meanwhile, the continuous memory Array takes full advantage of CPU cache performance to significantly improve graph computing efficiency. For larger data volumes, incorporating non-contiguous memory designs should be considered to mitigate the cost of data movements. This means we can construct a hierarchical structure to achieve the update-read balance. In real-world graph scenarios, the number of edges per vertex varies, thus a refined design for edge storage is necessary.

Basic Idea. We use small chunks (*i.e.,* yellow capsule-shaped structures in Figure 4) to store low-degree vertices, and employ B+ trees [13] (*i.e.,* green capsule-shaped structures in Figure 4) to store high-degree vertices. The capacities of small chunks and B+ tree nodes are set as integer multiples (usually 1-4) of the cache line size. Specifically, we set the multiples as 4 and the destination nodes are inserted into the chunk with the update of the graph. When the

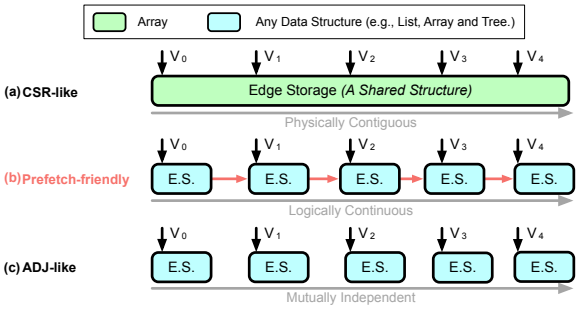


Figure 5: (a) CSR-like physically contiguous store; (b) prefetch-friendly logically contiguous store via pointers; (c) ADJ-like mutually independent store (E.S.: Edge Storage).

data exceeds the capacities of small chunks, we reorganize it in a B+ tree with tree node size also multiples of cache line size. For better readability, we set the capacities of small chunks and B+ tree nodes to 2 and 3 edges, respectively, in Figure 4. While in practice, we make the size of the chunks and tree nodes aligned with the cache line, such that the structure can provide better cache locality and enhance cache utilization for graph computation.

We enumerate the advantages of our Edge Storage design as follows. Firstly, we use independent edge structures (refer to ADJ-like structures in Figure 5) for each vertex to store its neighborhood to promote parallel updates of edges and facilitate different implementations like chunks and B+ trees for different nodes. In contrast, in CSR-like structures (*e.g.,* PCSR [47] and Teseo [32]), nodes are stored in identical and shared edge structure (refer to the CSR-like structures in Figure 5), and to update neighbors of one vertex may affect the vertex records and memory locations of other edges. Secondly, *scan_edges*(v_{src}) accessed from Traversal pointer in a B+ tree can be considered as sequential data access on a linked list, which can effectively utilize hardware prefetching as discussed in subsection 2.2. Thirdly, *read_edge*(v_{src},v_{dst}) accessed from update/query pointer completes in $O(log(n))$.

4.3 Prefecth-Friendly Global Traversal Chain

In graph computation, a significant portion of tasks (*e.g.,* PageRank [35]) require all vertices and edges in the graph to be involved in the computation. Their data access patterns can be represented as a combination of *scan_vertices()* and *scan_edges*(v_{src}). This combination equates to sequential data access over the entire graph.

From a global graph perspective, although CSR-like structures with shared edge structures impact update performance, they store all graph data in contiguous memory as shown in Figure 5. In contrast, ADJ-like structures, despite their independently stored vertex neighborhoods being update-friendly, result in complete fragmentation of the entire graph data in memory.

Basic Idea. We consider a transitional form: based on the data structure, we connect the neighborhoods of vertices with adjacent logical IDs to form a linked list (the chain formed by red dashed lines in Figure 4) that represents the entire graph named *Global Traversal Chain* (GTChain). Consequently, it can naturally leverage hardware prefetching to significantly improve the performance during sequential access (composed of *scan_vertices()* and *scan_edges*(v_{src}))

to entire graph data. Additionally, it provides assistance in load balancing for coroutines, as will be detailed in subsection 5.2.

We have also considered the overhead of forming GTChain. One of the reasons we chose B+ trees is the ease of forming GTChain. Each B+ tree inherently contains a traversal chain formed by leaf nodes. Its implementation for GTChain simply involves using pointers to link the leaf node chain to the global chain. However, most tree structures (*e.g.,* balanced binary trees, B-trees, and red-black trees) require manual implementation of a traversal chain and the additional traversal chain is challenging to form as a singly linked list (each node has only one outgoing pointer) to support the jump-pointer mechanism of hardware prefetching.

5 INTERLEAVED EXECUTION WITH COROUTINE

5.1 Coroutine with Software Prefetching

Sequential execution and interleaved execution. This paper focuses on reducing the cache miss overhead based on the interleaving execution mode [37]. Figure 6 illustrates the difference between sequential execution and interleaving execution of traversal on a graph in Figure 6 ❶. We first demonstrate an example of *scan_edges(v_{src})* on AL (adjacency list) in Figure 6 ❷ for clarity: In the sequential execution mode, when the CPU accesses Node 1, the first neighbor of vertex A, a cache miss is likely to occur in the non-contiguous memory linked by pointers, making the CPU stalled, *i.e.,* waiting until the data is fetched. Such stalling [25] occurs whenever cache misses happen due to non-contiguous memory access and pointer chasing. Alternatively, in interleaved execution mode, the CPU will issue a prefetch for Node 1 and switch to another task (*i.e.,* accessing Node 3, the first neighbor of vertex B). Each coroutine handles a linked list and is in charge of computing the corresponding task. The algorithm starts from an arbitrary computation task. It switches to another task whenever it encounters a software prefetching instruction to fetch data (the instruction is set before accessing the next node in the linked list). When switching back to the previous task, the CPU does not need to wait for the needed data as they have been loaded into the cache. Thus, the process of fetching data and computations can be overlapped and the time of cache misses due to pointer chasing can be avoided. Note that switching tasks also causes time costs [37]. We choose C++20 stackless coroutines and design scheduling strategies, such that the switching time is minimal, thereby achieving performance enhancements.

Coroutine in Graph Computation. Next, we present the coroutine pool constructor and scheduler in graph computation. As we previously mentioned, C++20 stackless coroutines are a special kind of function and it requires manual intervention to schedule these functions. ①In algorithm 1 (line 1 - line 3), we present the basic logic of the coroutine pool constructor. The time complexity of the constructor is $O(m)$, where m is the number of coroutines in the coroutine pool. The coroutine pool relies on the for loop (line 2) to initialize each coroutine as *Func*. *Func* defines the graph access operations (as proposed in subsection 2.2) which will be detailed in algorithm 2. ②In algorithm 1 (line 5 - line 11), we present a polling scheduler implementation. The overhead generated by the scheduler itself is also a cost of using coroutines. Therefore, we

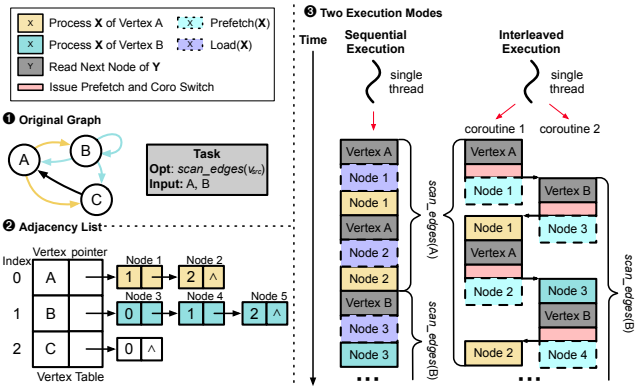


Figure 6: Sequential execution vs. interleaved execution.

design it as simply as possible to reduce the cost of using coroutines. Its core logic includes checking if the coroutine pool has finished through a while loop (line 6) and polling each coroutine via a for loop (line 7). If a coroutine is not completed, the polling scheduler resumes execution from the suspension point (line 9). If a coroutine is completed, the polling scheduler will destroy it (line 11). Assuming the number of suspensions for the i-th coroutine in the polling scheduler is k_i, the time complexity for the entire polling scheduler is $O(\sum_{i=1}^{m} k_i)$. If the tasks are evenly distributed and each coroutine is suspended approximately k times, the overall time complexity is $O(mk)$. ③We detail the logic of edge access operations in algorithm 2. `GetNeighbors(`*vertex*`)` and `GetNeighbors(`*chain*`)` correspond to the *scan_edges(v_{src})* operation for accessing edges and their usage and difference will be elaborated in subsection 6.1. The time complexities of the two operations are $O(D)$ and $O(E_{chain})$ respectively, where D is the degree of the vertex, and E_{chain} is the number of edges in the chain. `FindNeighbor(`*edge*`)` corresponds to *read_edge(v_{src},v_{dst})*. Based on `CBList`, regardless of whether using a binary search on small chunks or searching within a B+ tree, the time complexity for `FindNeighbor(`*edge*`)` is $O(\log D)$. The user can specify the search algorithm either on small chunks or B+ tree leaf nodes (refer to line 25, line 31). Here, all edge access operations are constructed using coroutines and `prefetch(`*ptr*`)` denotes prefetching the content pointed to by the pointer in line 8, line 16, and line 28. This is in the form of software prefetching. In line 9, line 17, and line 29, *co_await* is a keyword in C++20 and its invocation of **suspend_always** means that this coroutine actively entering an always suspended state. Coroutines in the suspended state will be resumed by the scheduler (line 9 in algorithm 1).

Coroutines in Graph Update. `GastCoCo` primarily aims to enhance performance in the context of *batched edge updates* by utilizing coroutines. For other updates, we have the following discussions. (1) For individual updates, as previously mentioned, there is no overlapping execution with other tasks to diminish the data loading overhead. Therefore, it cannot form an interleaved execution mode and leverage coroutines to optimize performance. However, `GastCoCo` can still rely on the inherent efficiency of `CBList` for handling individual updates. (2) Furthermore, graph updates can be divided into vertex updates and edge updates. For vertex updates, the vertex table in `CBList` makes it efficient to query a vertex

Algorithm 1: Constructor & Polling Scheduler

```
1  Function Constructor(coroutine_num, Func):
2      for i ← 1 to coroutine_num do
3          coroutine_pool.append(Func)
4
5  Function Scheduler(coroutine_pool):
6      while not coroutine_pool.finish() do
7          for i ← 1 to coroutine_pool.size do
8              if not coroutine_pool[i].done() then
9                  coroutine_pool[i].resume()
10             else
11                 coroutine_pool[i].destroy()
```

Algorithm 2: Edge Access Operation

```
1  // scan_edges(v_src)
2  Function GetNeighbors(vertex):
3      n ← 1
4      if VertexTable[vertex].level ≠ 0 :
5          n ← VertexTable[vertex].level
6      ptr ← VertexTable[vertex].Traversal_pointer
7      for i ← 0 to n do
8          prefetch(ptr)
9          co_await suspend_always
10         compute(ptr)
11         ptr ← ptr.next
12 // scan_edges(v_src)
13 Function GetNeighbors(chain):
14     ptr ← chain.start
15     while ptr ≠ chain.end do
16         prefetch(ptr)
17         co_await suspend_always
18         compute(ptr)
19         ptr ← ptr.next
20 // read_edge(v_src,v_dst)
21 Function FindNeighbor(edge):
22     SmallChunk_level ← 0
23     ptr ← VertexTable[edge.src].Query_pointer
24     if VertexTable[edge.src].level = SmallChunk_level :
25         Return search(ptr, edge.dst)
26     else
27         while ptr ≠ nullptr do
28             prefetch(ptr)
29             co_await suspend_always
30             if ptr.Type = LeafNode :
31                 Return search(ptr, edge.dst)
32             else
33                 ptr ← locate(ptr, edge.dst)
34     Return False
```

by the operation *read_vertex(v)*. The vertex insertion operation in the vertex table can be achieved by an append operation because we can align the new vertex to the maximum logical ID based on the ID-map table. In this way, we do not use coroutines for vertex updates in the vertex table either.

Specifically, in GastCoCo, no matter whether an update is an insertion, deletion, or modification, the process is divided into two steps: **locating** and **operating**. We first locate the position to be updated through a query operation, and then proceed with the specific update operation. Thus, we formulate the query operation *read_edge*(v_{src},v_{dst}) as a coroutine, as shown in algorithm 2 (line 21). During the queries in CBList, whenever we meet a pointer for data loading, we issue a software prefetch (line 28) and switch back later to hide the overhead of loading data into the cache.

5.2 Load Balancing of Coroutines

As we have described in the example of interleaving execution mode in Figure 6, each coroutine in a graph processing task represents a unit of concurrent execution that processes the subgraph from the overall task. We can reduce the overhead of cache misses by overlapping the time of fetching data through the switching of different coroutines. However, switching coroutines also causes an overhead. Suppose a coroutine requires much more switch times than other coroutines. Under the scheduling by polling scheduler (line 5 in algorithm 1), it will occupy the most time of the algorithm's execution, with other coroutines having no overlapping with it. In such case, it will lead to an unbalanced interleaving execution mode, as illustrated in "Vertex Table Partition" in Figure 7 ❸.

To this end, two solutions can solve the issues. The first is to avoid the unbalanced partition by ensuring that each coroutine has approximately the same switch times, and another is to optimize the polling scheduler to avoid the repeated suspension and resumption of the remaining single coroutine after all other coroutines have been destroyed. Correspondingly, we propose two techniques to address the issue: a graph partition strategy to ensure partition evenness and a trimmed-polling scheduler with fine-grained checkpoints. "Vertex Table Partition" is the basic graph partition strategy by dividing the vertex table into continuous chunks, which could maintain the natural locality of the graph data [51]. This approach may also lead to unbalance in some extreme cases, as partitioning on the vertex table means that the smallest granularity of the partition is the edge block of a single vertex. For example, if a vertex is a super vertex (*e.g.,* having 95% of the edges), the approach may fail.

A Fine-grained Graph Partition Strategy, on the other hand, controls the balance by partitioning GTChain into continuous sub-chains, as shown in Figure 7 ❷. Suppose there are N coroutines and CBList contains X edge structure blocks, then each coroutine will process a sub-chain (cut from GTChain) containing X/N edge blocks. While this strategy achieves nearly perfect balance, it requires the participation of all the vertices and edges in the entire graph computation, just like the conditions for using GTChain. However, even though we evenly divide the data before computation, if certain edge blocks are not involved in the computation, it still cannot form a balanced interleaving execution mode at run time.

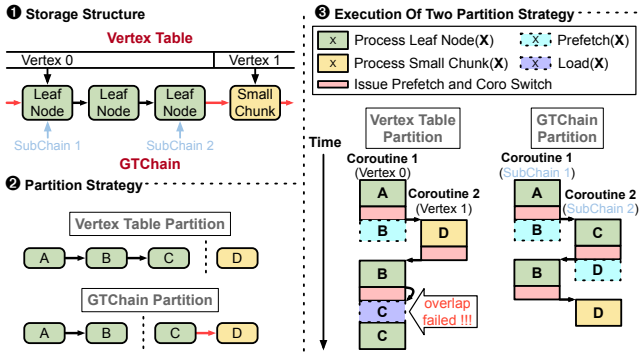


Figure 7: Load balancing of coroutines.

As a result, "GTChain Partition" can only be used for *scan_vertices()* with *scan_edges(v_{src})*. In summary, we design and employ two graph partition strategies in GastCoCo, and their specific usage will be detailed in subsection 6.1.

A Trimmed-Polling Scheduler. Compared with the polling scheduler, a more proper approach would be to pass the number of uncompleted coroutines to the resumed coroutine such that it can decide when to switch. If the coroutine is the only one that remains working, it would not issue a suspension operator, as shown in algorithm 3. On the contrary, this may introduce extra overhead because of the additional checks in practice.

Algorithm 3: Trimmed-Polling Scheduler

```
1  remain_num ← coroutine_pool.size
2
3  Function Scheduler(coroutine_pool):
4      while remian_num ≠ 0 do
5          for i ← 1 to coroutine_pool.size do
6              coroutine_pool[i].resume()
7              if coroutine_pool[i].finish() then
8                  coroutine_pool[i].destroy()
9                  remain_num ← remain_num − 1
10
11 Function Coroutine_Func(...):
12     // code segment 1
13     if remain_num ≠ 1 then
14         co_await suspend_always
15     // code segment 2
```

6 ADAPTATION LAYER

6.1 Execution Strategy Tuner

In this subsection, we explain how the execution strategy tuner tailors partition and scheduling strategies based on graph processing tasks and how the task allocator allocates tasks to coroutines. It is noteworthy that modeling each task as a coroutine and creating the interleaved execution mode among multiple tasks is feasible and easy. Thus, we focus on building interleaved execution patterns among multiple coroutines within a single task in this subsection. We present the logic of the execution strategy tuner in Figure 8 for a better understanding.

Graph Computation Tasks. For a specific graph computation task, it is necessary to analyze the patterns of vertex and edge access to determine the most suitable strategy. As we have described in the example of interleaving execution mode in Figure 6, each coroutine in a graph processing task represents a unit of concurrent execution that processes the subgraph within the overall task.

Graph Update Tasks. In GastCoCo, we support both individual updates and batch updates, depending on the specific requirements of the application scenario. However, it is important to note that the coroutine optimization scheme used in GastCoCo can only optimize batch edge updates. Specifically, for individual updates, we apply vertex-level locks on the CBList to ensure accuracy. For batch updates, similar to systems that support batch updates (*e.g.,* Aspen [16] and Terrace [36]), we classify update tasks by source vertex before updating to avoid the overhead of locks caused by data conflicts. Each vertex that is waiting to be updated will have a collection of update tasks. The task allocator models each vertex's task collection as a coroutine, which is similar to FindNeighbor(*edge*) in algorithm 2. Since insertions may modify the structure making it difficult to estimate the number of suspension points before the update, we opt for the trimmed-polling scheduler for batch updates.

6.2 Hybrid Prefetching

Hardware prefetching fetches data silently to users. However, if hardware prefetching succeeds, it will lead to additional overhead to repeatedly operate software prefetching. Accordingly, the ideal scenario is to use software prefetching when the hardware prefetching fails to fetch data and to avoid using software prefetching when hardware prefetching is effective.

Assume that the cost of a cache miss is C_m and the probability of a cache miss being resolved by hardware prefetching is P_h. This probability varies based on the memory layout of the data. The software prefetching solution via coroutines incurs a certain overhead, C_{coro}. If the cost $C_m \times (1 - P_h) < C_{coro}$, then the software prefetching becomes redundant. To avoid this overhead of redundant data fetching, we aim to avoid employing software prefetching on data that is highly likely to be prefetched by the hardware prefetchers. We also need to consider the phenomenon caused by the skewed graph structures from three perspectives discussed in subsection 2.2.

Therefore, we aim to use software prefetching to complement hardware prefetching for better performance. However, implementing software prefetching via coroutines also has overhead, as shown in section 5. To better utilize hardware and software prefetching, we design four prefetching strategies, as shown in Figure 9.

All Hard. This strategy uses hardware prefetching exclusively. The performance of applying *All Hard* depends on how much the data structure supports hardware prefetching. In CBList, designs like cache alignment, B+ trees, and GTChain aim to support hardware prefetching better.

All Soft. This strategy uses software prefetching exclusively. Although it may conflict with hardware prefetching in various tasks, subsection 7.7 proves it is an effective strategy.

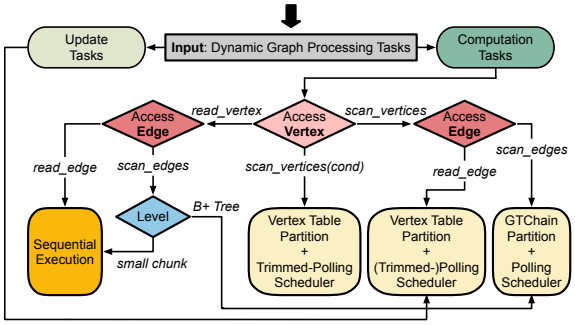


Figure 8: The flow of execution strategy tuner.

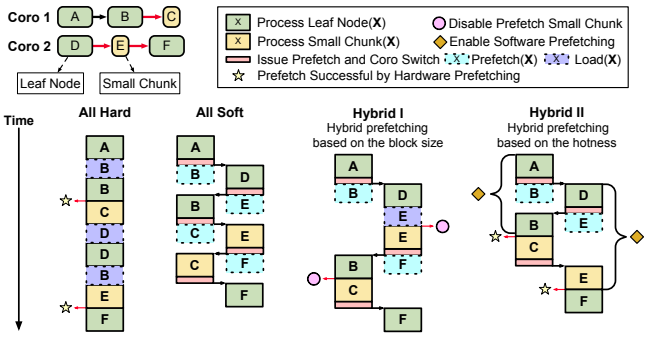


Figure 9: Hybrid prefetching.

Hybrid strategy I (Hybrid prefetching based on the block size). In the implementation of `CBList`, all small chunks are arranged contiguously in memory, resulting in a high probability of being prefetched by the hardware prefetcher. In this case, we skip software prefetching for small chunks and instead rely on the hardware prefetcher's inherent capability to prefetch these small chunks. In most cases, this strategy is more effective than the previous "All Soft" strategy. This strategy is more suitable for scenarios where small chunks have a high cache hit rate.

Hybrid strategy II (Hybrid prefetching based on the hotness). To further prevent redundant data fetching, we consider the cold start issue of the jump-pointer mechanism described in Section 2.2. Based on Hybrid I, we use software prefetching at the beginning of the linked list. For the latter part of the linked list, we solely rely on the hardware prefetcher's jump-pointer mechanism. In summary, this strategy utilizes software or hardware prefetching based on the cache's "cold" and "hot" data status. Under ideal conditions, this strategy achieves optimal results. However, it requires iterative tuning and adjustments to determine the optimal threshold parameters of using hardware or software prefetching based on the cache's state. Thus, the strategy is particularly suitable for scenarios where repeated fine-tuning of computations is possible.

7 EVALUATION

This section introduces the experimental setup, and then validates the efficiency of the proposed system through various evaluations.

7.1 Experimental Setup

We run the experiments on a server equipped with one Intel Xeon Platinum 8269CY CPU whose clock speed is 2.5GHz. The server contains 26 cores (52 hyper-threads) and 371GB of main memory.

Table 2: Description of graph datasets.

Graph	#V	#E	$\overline{D}$
Livejournal [5]	4,846,609	68,475,391	14.12
UK-2002 [8]	18,484,117	298,113,762	16.13
Com-friendster [48]	65,608,366	1,806,067,135	27.53
Orkut [48]	3,072,441	117,185,083	38.14
Hollywood [8]	1,139,905	116,050,145	101.81
SNB-sf1000 [3]	3,144,492	202,282,791	64.33
Ogbl-citation2 [46]	2,927,963	30,561,187	10.44

The sizes of the L1-cache, L2-cache, and L3-cache of the CPUs are 832KB(data cache)+832KB(instruction cache), 26MB, and 35.8MB, respectively. We compiled all systems with the O3 optimization flag using GCC v10.3. All experiments are conducted with 52 worker threads by default unless otherwise stated, with each test running five times to report the average outcome.

Datasets. We used five real-life graphs in our evaluation (see Table 2), including social networks Livejournal [5], Hollywood [8] and Com-friendster [48], and web graph UK-2002 [8] and Orkut [48]. The detailed information of the datasets is shown in Table 2, which includes the name of the dataset, the number of vertices, the number of edges, and average degrees. To simulate the realism of dynamic graph processing scenarios in practice, the datasets are shuffled. This is because other structures that contain pointers may be penalized during loading due to the inherent order of the data. To ensure fairness, all systems load datasets using the weighted graph mode, and random weights are generated for those unweighted datasets. Besides, we used the SNB-sf1000 dataset [3] to test the system's update performance in real-world scenarios. The SNB-sf1000 dataset includes timestamped person nodes and their relationships, along with 7,285 vertex deletions and 39,877,751 edge insertions. We also used the Ogbl-citation2 [46] dataset, which includes 128-dimensional word2vec features, to test graph attribute updates.

Workloads. We conduct three types of workloads. (i) Graph Query: we randomly query 5% edges of the data set. (ii) Graph Algorithms: we use five typical graph analysis algorithms, *i.e.,* BFS, Single Source Shortest Path (SSSP) [10], PageRank (PR) [35], Connected Components (CC) and Label Propagation (LP). (iii) Graph Update: we synthetically generate 10,000 random graph updates in the form of edge insertions and edge deletions.

Competitors. We compare `GastCoCo` with state-of-the-art solutions including LLAMA [34], PCSR [47], LiveGraph [52], GraphOne [30], RisGraph [21], Teseo [32], Sortledton [22] and Terrace [36].

LLAMA, Teseo, and PCSR utilize structures similar to CSR to enhance data locality. LiveGraph employs a log-structured approach for data insertion to achieve faster insertion performance. GraphOne and RisGraph adopt block-based adjacency lists and adjacency arrays, respectively, to support rapid data insertion. Sortledton and Terrace employ different structures for storing neighbor data based on vertex degree, striking a balance between insertion performance and graph analysis capabilities.

7.2 Performance of Graph Query

We first evaluate the performance of edge queries by `GastCoCo` on all datasets in Table 2, which plays a crucial role in multiple

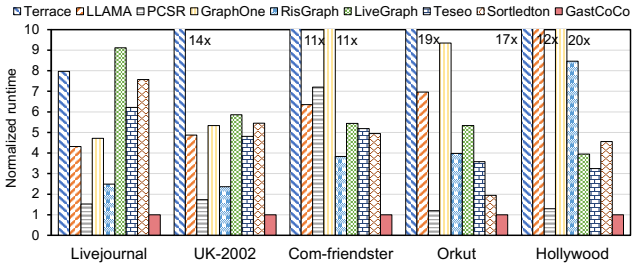


Figure 10: Query execution time.

domains such as social network analysis, recommendation systems, and transportation networks. For example, in social networks, edge queries can be utilized to check whether there is a direct friendship between two users or not.

Figure 10 shows the normalized time for each system to execute queries on different datasets. Here the response time of GastCoCo is treated as the baseline, *i.e.,* finishes in unit time. We can observe that GastCoCo outperforms others in all the cases. Specifically, GastCoCo achieves an average 13.7× (up to 16.8×) speedup over Terrace, 7.4× (up to 12.0×) speedup over LLAMA, 2.8× (up to 7.2×) speedup over PCSR, 10.9× (up to 20.4×) speedup over GraphOne, 4.7× (up to 8.5×) speedup over RisGraph, 6.0× (up to 9.1×) speedup over LiveGraph, 4.6× (up to 6.2×) speedup over Teseo, and 5.6× (up to 7.6×) speedup over Sortledton. For Terrace, LLAMA, and GraphOne, they experienced a slowdown exceeding 10× on the Hollywood dataset. This can be attributed to the dataset's vertices having the highest average degree (*i.e.,* 101.81), which leads to significant query overhead for vertices with high degrees due to the storage structures of these three systems during retrieval. Specifically, Terrace adopts a hierarchical structure. When the degree of a vertex is large, its neighbors will be distributed in multiple layers of data structures, which results in the need to search across multiple storage structures when querying. For GraphOne, it uses a block-based adjacency list to store the neighbors of each vertex. Therefore, for vertices with a high degree, this necessitates traversing a longer chain, thereby extending the query time. Compared with other systems, GastCoCo also achieves better performance, mainly because GastCoCo adopts more "stubby" structures, *i.e.,* B+ Tree, as part of our edge storage. The B+ tree, due to its strict rebalance rules, has fewer levels, meaning that we encounter fewer pointers and experience fewer cache misses during queries. We can offset the benefits with software prefetching via coroutines to amortize the rebalance overhead during updates.

7.3 Performance of Graph Analysis

We next evaluate the performance of graph analysis, including the execution time of five graph analytics algorithms, by comparing GastCoCo with competitors. Figure 11 shows the normalized time for each system to execute graph analysis algorithms on different datasets. Here the response time of GastCoCo is treated as the baseline, *i.e.,* finishes in unit time. We can see that GastCoCo outperforms others in most cases. Specifically, GastCoCo achieves an average 1.4× (up to 3.9×) speedup over Terrace, 41.1× (up to 243.8×) speedup over LLAMA, 11.0× (up to 39.7×) speedup over

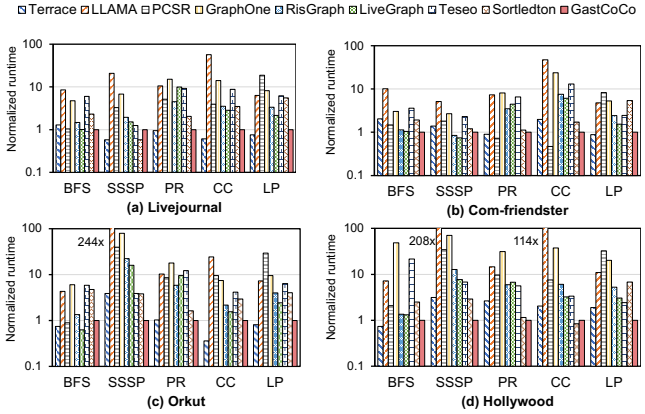


Figure 11: The execution time of graph algorithms on different datasets.

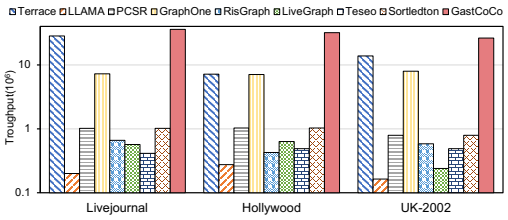


Figure 12: The throughput of graph updates.

PCSR, 21.7× (up to 79.4×) speedup over GraphOne, 4.9× (up to 22.3×) speedup over RisGraph, 4.2× (up to 16.0×) speedup over LiveGraph, 5.8× (up to 21.7×) speedup over Teseo, and 2.8× (up to 6.8×) speedup over Sortledton. As shown in Figure 11, LLAMA and GraphOne exhibit the most significant slowdown. LLAMA employs a multi-version array to store the neighboring edges of vertices. As data is inserted, the edges of a vertex may be distributed across multiple arrays, which is disadvantageous for graph analysis algorithms since they always require access to all neighbors of a vertex. LLAMA connects the edges of a vertex distributed in different arrays by pointers, thereby reducing search time. However, this still results in extensive pointer chasing. GraphOne faces similar issues with expensive pointer chasing due to its use of block-based adjacency lists. Furthermore, Figure 11 shows that Terrace, PCSR, and Sortledton achieved better graph analysis performance compared to other competitors. This is attributed to Terrace and Sortledton using arrays to store the neighbors of vertices with low degrees, and PCSR using arrays to store the neighbors of all vertices, which have a better spatial locality. However, this design also results in higher update costs due to data movements caused by insertions/deletions, which will be detailed in Section 7.4. Additionally, Terrace and Sortledton employ B-Trees or skip lists to store the neighbors of high-degree vertices, still leading to extensive pointer chasing. In contrast, GastCoCo utilizes software prefetching and coroutines to accelerate data retrieval in graph analysis and reduce cache misses, thereby achieving superior graph analysis performance.

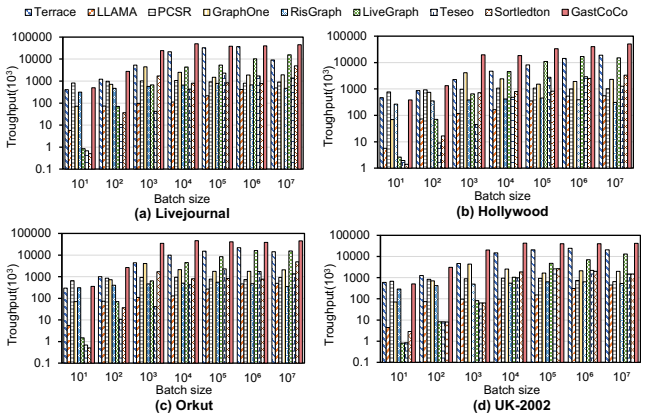


Figure 13: The throughput of graph updates with varying batch sizes on different datasets.

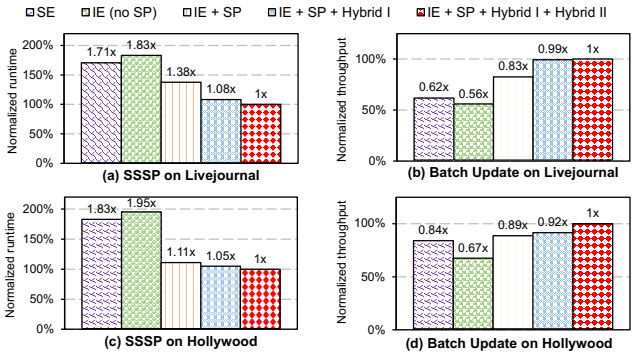


Figure 14: Performance analysis. (SE represents Sequential Execution, IE represents Interleaved Execution, and SP represents Software Prefetching)

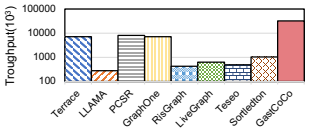


Figure 15: Update properties on ogbl-citation2.

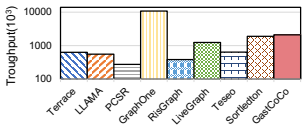


Figure 16: Update on SNB-sf1000.

7.4 Performance of Graph Updates

We evaluate the graph update performance of different systems on LiveJournal, Hollywood and UK-2002 for continuous edge insertions/deletions of 10K edges. Figure 12 shows the throughput (edges per second) for all systems under different data sets. `GastCoCo` outperforms all competitors in all tests. Specifically, `GastCoCo` achieves an average 2.5× (up to 4.5×) speedup over Terrace, 152.4× (up to 180.2×) speedup over LLAMA, 33.3× (up to 35.4×) speedup over PCSR, 4.3× (up to 5.0×) speedup over GraphOne, 58.2× (up to 75.0×) speedup over RisGraph, 74.4× (up to 109.4×) speedup over LiveGraph, 68.8× (up to 87.2×) speedup over Teseo, and 33.3× (up to 35.4×) speedup over Sortledton. It can be observed that Terrace, GraphOne, and `GastCoCo` have higher throughput. GraphOne utilizes a block-based adjacency list storage structure, which is particularly advantageous for insertions. However, as demonstrated in the previous experiment, it performs poorly in graph queries and graph analysis. Both Terrace and `GastCoCo` adopt a batch updates design, *i.e.,* classify update tasks by source vertices before updating, to avoid lock overhead caused by data conflicts. Moreover, since Terrace maintains the updated data in an ordered manner, a search operation is required to determine the position of the new data, leading to a significant number of cache misses. In contrast, although `GastCoCo` also maintains ordered updates, our coroutine-related design effectively reduces cache misses, thereby achieving better performance than Terrace.

7.5 Varying Batch Size of Graph Updates

We also conduct experiments to evaluate the impact of batch size of graph updates on throughput performance. Figure 13 reports the throughput of the systems on graph updates with different batch sizes on LiveJournal, Hollywood, Orkut, and UK-2002. For each batch size from 10 to 10^7, we report the throughput (edges per second) for all systems. Figure 13 shows that `GastCoCo` outperforms other systems with batch sizes ranging from 10^3 to 10^7, the reason for which is primarily consistent with the analysis provided in Section 7.4. Moreover, when the batch size exceeds 10^3, the throughput of `GastCoCo` remains relatively stable, indicating that `GastCoCo` possesses excellent scalability. On the other hand, when the batch size is 10 and 100, Terrace, PCSR, and GraphOne exceed `GastCoCo`. This is because the construction and scheduling overheads in `GastCoCo`'s `CBList` are difficult to balance against the benefits brought by coroutines for smaller batch sizes.

7.6 Real-World Graph Updates

Firstly, we conduct a properties update experiment using the OGB dataset. The results are shown in Figure 15. Since updating properties does not cause structural changes, many systems have shown good performance in this task.

Secondly, we conduct a real-world graph update experiment on the SNB-sf1000 dataset to test the system's update performance in real-world scenarios. Unlike the previous update experiments, this dataset includes vertex deletions. The SNB-sf1000 dataset includes timestamped person nodes and their relationships, along with 7,285 vertex deletions and 39,877,751 edge deletions. When deleting a vertex, we need to delete all its edges at the same time. We test the throughput of graph updates on different systems using the SNB-sf1000 dataset. As shown in Figure 16, GastCoCo outperforms most systems except for GraphOne. GraphOne uses a block-based adjacency list storage structure and does not actually delete content during updates, but rather appends deletion records, which is particularly advantageous for insertions. However, as demonstrated by the experiments in subsection 7.2 and subsection 7.3, it performs poorly in graph queries and analysis. In contrast, our system shows good performance in both graph updates and analysis.

Table 3: Cache miss and cache stall percentage for different workloads and execution modes.

Workload	Execution mode	Livejournal		Hollywood	
		Cache miss	Percentage	Cache miss	Percentage
PageRank	SE	2.09×10^9	50.4%	2.70×10^9	69.5%
	IE + SP	2.01×10^9	42.9%	2.26×10^9	49.4%
SSSP	SE	1.75×10^9	31.7%	2.64×10^9	45.4%
	IE + SP	1.31×10^9	24.4%	2.00×10^9	29.4%
Batch Update	SE	1.20×10^9	58.3%	2.95×10^9	81.1%
	IE + SP	1.18×10^9	58.2%	2.92×10^9	80.5%

7.7 Performance Analysis

We conduct performance analysis on SSSP and batch insertions for our designs to understand the performance gains and losses. SSSP includes `GetVertices(`*cond*`)`, making the probability of hardware prefetching failures increase and software prefetching more effective. Batch insertions utilize update/query pointers in `CBList` for locating update positions in the edge storage structure. Hence, the hybrid strategy I cannot be applied as it is only implementable on GTChain. In particular, *Sequential Execution* (SE) means executing tasks in sequential execution mode (refer to subsection 5.1); *Interleaved Execution* (IE) means using coroutines to form an interleaved execution mode (refer to subsection 5.1); *Software Prefetching* (SP) means incorporating software prefetching instructions during the interleaved execution process; *Hybrid I* and *Hybrid II* represent employing hybrid strategy I with hardware failure time and hybrid strategy II with the content to be prefetched respectively in section 6. From Figure 14, we could observe that the interleaved execution mode (IE no SP) spends longer running time compared with the sequential execution mode (SE) due to the additional switching cost of coroutines. However, if we further employ software prefetching and hybrid prefetching strategy, `GastCoCo` achieves at most 1.83× (graph computation) and 1.62× (graph update) speedup compared with sequential execution on `CBList`.

Additionally, we record the cache miss rates for software prefetching with interleaved execution and sequential execution. As shown in Table 3, the application of software prefetching reduced both cache misses and cache stall percentages. Specifically, in graph algorithms, cache misses were reduced by up to 35%.

8 RELATED WORK

Dynamic graph storage systems. There have been graph storage systems developed for dynamic graph storage and processing [16, 21, 22, 30–32, 34, 36, 47, 52]. LiveGraph [52] adopts a log-structured design, allocating a contiguous edge block to each vertex for fast neighbor scanning and enabling swift updates through append-only insertions. However, this method leads to unordered neighbor storage, impeding the efficient execution of common operations, and data must be copied to larger blocks to maintain continuity once current edge blocks are full. LLAMA [34] uses multi-version array storage to enhance spatial locality in graph analysis. However, continuous data insertion leads to numerous array versions and scatters a vertex's neighbors across them, still causing extensive CPU cache stalls during analysis. Aspen [16] uses a tree structure for edge data to enable fast insertion and maintain ordered storage of neighbors. However, this structure results in minimal inherent spatial locality among data nodes, leading to poor access performance. Terrace [36] and Sortledton [22] adopt different data structures for storing neighbors of vertices of varying degrees to enhance data locality and minimize cache misses during access. However, for vertices with a high degree, they are still stored in discontinuous structures, and neighbors of a single vertex may span multiple structures, ultimately impacting search and graph algorithm performance. Teseo [32] and PCSR [47], employing CSR-like structures, use reserved gaps to prevent severe data movement overhead. However, they still face the same challenges as CSR when facing a surge in data updates. Different from existing systems, we design a novel dynamic graph data structure to facilitate rapid graph insertions. It integrates the advantages of hardware prefetching, software prefetching, and coroutines to minimize cache misses, thereby enhancing graph computation performance.

Prefetching and Coroutines. Due to the significant disparity between CPU computation speeds and memory access latency, some existing graph systems [16, 22, 32, 34, 36] employ prefetching with the aim of reducing CPU cache stalls during graph analysis. However, these systems merely utilize the prefetching interfaces provided by the system in a straightforward manner, which does not guarantee that data is fetched into the cache prior to access. As a result, they still incur significant CPU cache stall overheads. Furthermore, some works [25, 50] have adopted coroutine-based prefetching to accelerate existing applications. For example, CoroBase [25] utilizes coroutines to expedite the transaction processing. In CoroGraph [50], coroutine-based prefetching is employed to reduce CPU cache stalls during the Gather-Apply-Scatter (GAS) process in static graph systems. Compared to these systems, our approach addresses the unique challenges brought by data insertion and graph analysis in dynamic graph scenarios. By integrating the characteristics of prefetching and coroutines, we design a novel dynamic graph structure and access pattern that supports rapid graph data insertion while enhancing graph analysis performance.

9 CONCLUSION

This paper presents `GastCoCo`, a prefetch-aware graph storage and software prefetching co-designed system by leveraging coroutines for dynamic graph processing. C++20 coroutines are introduced to the system to achieve prefetching randomly stored graph data and reduce cache misses. To benefit more from the software prefetching, several workload-balancing strategies and a prefetch-friendly data structure, `CBList`, are proposed. Extensive experiments on various datasets show the superiority of `GastCoCo` in achieving the balance of graph computation performance and graph update performance.

ACKNOWLEDGMENTS

This work was supported by the National Key R&D Program of China (2023YFB4503601), the National Natural Science Foundation of China (U2241212, 62072082, and 62202088), the 111 Project (B16009), the Joint Funds of Natural Science Foundation of Liaoning Province (2023-MSBA-078), the Fundamental Research Funds for the Central Universities (N2416011), the Distinguished Youth Foundation of Liaoning Province (2024021148-JH3/501), and a research grant from Alibaba Innovative Research (AIR) Program. Yanfeng Zhang and Shufeng Gong are the co-corresponding authors.

REFERENCES

[1] 2023. Alibaba DAU. https://news.futunn.com/en/post/33640347?level=1&data_ticket=1709021520459060.

[2] 2024. Taobao. https://www.taobao.com/.

[3] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).

[4] Saeid Azadifar, Mehrdad Rostami, Kamal Berahmand, Parham Moradi, and Mourad Oussalah. 2022. Graph-based relevancy-redundancy gene selection method for cancer diagnosis. *Comput. Biol. Medicine* 147 (2022), 105766.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*. 44–54.

[6] Jean-Loup Baer and Tien-Fu Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings Supercomputing*. ACM, 176–186.

[7] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *2015 IEEE International Symposium on Workload Characterization(IISWC)*. IEEE Computer Society, 56–65.

[8] Paolo Boldi and Sebastiano Vigna. 2004. The WebGraph Framework I: Compression Techniques. In *Proceedings of the Thirteenth International World Wide Web Conference (WWW)*. ACM Press, 595–601.

[9] David Callahan, Ken Kennedy, and Allan Porterfield. 1991. Software prefetching. *ACM SIGARCH Computer Architecture News* 19, 2 (1991), 40–52.

[10] Venkatesan T. Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. 2017. Scalable Single Source Shortest Path Algorithms for Massively Parallel Systems. *IEEE Trans. Parallel Distributed Syst.* 28, 7 (2017), 2031–2045.

[11] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17.

[12] Jamison D. Collins, Suleyman Sair, Brad Calder, and Dean M. Tullsen. 2002. Pointer cache assisted prefetching. In *Proceedings of the 35th Annual International Symposium on Microarchitecture (MICRO)*. ACM/IEEE Computer Society, 62–73.

[13] Douglas Comer. 1979. The Ubiquitous B-Tree. *ACM Comput. Surv.* 11, 2 (1979), 121–137.

[14] Melvin E. Conway. 1963. Design of a separable transition-diagram compiler. *Commun. ACM* 6, 7 (1963), 396–408.

[15] Ana Lúcia de Moura and Roberto Ierusalimschy. 2009. Revisiting coroutines. *ACM Trans. Program. Lang. Syst.* 31, 2 (2009), 6:1–6:31.

[16] 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 (PLDI)*. ACM, 918–934.

[17] Bolin Ding, Kai Zeng, and Wenyuan Yu. 2020. Alibaba Sponsor Talk at VLDB.

[18] David Ediger, Robert McColl, E. Jason Riedy, and David A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *IEEE Conference on High Performance Extreme Computing (HPEC)*. IEEE, 1–5.

[19] Assaf Eisenman, Ludmila Cherkasova, Guilherme Magalhaes, Qiong Cai, Paolo Faraboschi, and Sachin Katti. 2016. Parallel Graph Processing: Prejudice and State of the Art. In *Proceedings of the 7th ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 85–90.

[20] Babak Falsafi and Thomas F. Wenisch. 2014. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers.

[21] Guanyu Feng, Zixuan Ma, Daixuan Li, Xiaowei Zhu, Yanzheng Cai, Wentao Han, and Wenguang Chen. 2020. RisGraph: A Real-Time Streaming System for Evolving Graphs. *CoRR* abs/2004.00803 (2020).

[22] Per Fuchs, Jana Giceva, and Domagoj Margan. 2022. Sortledton: a universal, transactional graph data structure. *Proc. VLDB Endow.* 15, 6 (2022), 1173–1186.

[23] Pankaj Gupta, Venu Satuluri, Ajeet Grewal, Siva Gurumurthy, Volodymyr Zhabiuk, Quannan Li, and Jimmy Lin. 2014. Real-Time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 7, 13 (2014), 1379–1380.

[24] Tao He, Shuxian Hu, Longbin Lai, Dongze Li, Neng Li, Xue Li, Lexiao Liu, Xiaojian Luo, Binqing Lyu, Ke Meng, et al. 2023. GraphScope Flex: LEGO-like Graph Computing Stack. *arXiv preprint arXiv:2312.12107* (2023).

[25] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444.

[26] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer Manuals*. (Oct. 2016).

[27] ISO/IEC. 2017. Technical Specification — C++ Extensions for Coroutines. https://www.iso.org/standard/73008.html.

[28] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin J. Levandoski, and Gor V. Nishanov. 2018. Exploiting Coroutines to Attack the "Killer Nanoseconds". *Proc. VLDB Endow.* 11, 11 (2018), 1702–1714.

[29] Yusuf Onur Koçberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (2015), 252–263.

[30] Pradeep Kumar and H. Howie Huang. 2020. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. *ACM Trans. Storage* 15, 4 (2020), 29:1–29:40.

[31] Aapo Kyrola, Guy E. Blelloch, and Carlos Guestrin. 2012. GraphChi: Large-Scale Graph Computation on Just a PC. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 31–46.

[32] Dean De Leo and Peter A. Boncz. 2021. Teseo and the Analysis of Structural Dynamic Graphs. *Proc. VLDB Endow.* 14, 6 (2021), 1053–1066.

[33] Chi-Keung Luk and Todd C. Mowry. 1996. Compiler-Based Prefetching for Recursive Data Structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 222–233.

[34] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering (ICDE)*. IEEE Computer Society, 363–374.

[35] Larry Page, Sergey Brin, and R Motwani. 1998. Winograd., T. The pagerank citation ranking: bringing order to the web. *Unpublished manuscript* (1998).

[36] Prashant Pandey, Brian Wheatman, Helen Xu, and Aydin Buluç. 2021. Terrace: A Hierarchical Graph Container for Skewed Dynamic Graphs. In *International Conference on Management of Data (SIGMOD)*. ACM, 1372–1385.

[37] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2017. Interleaving with Coroutines: A Practical Approach for Robust Index Joins. *Proc. VLDB Endow.* 11, 2 (2017), 230–242.

[38] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. 2018. Real-time Constrained Cycle Detection in Large Dynamic Graphs. *Proc. VLDB Endow.* 11, 12 (2018), 1876–1888.

[39] Amir Roth, Andreas Moshovos, and Gurindar S. Sohi. 1998. Dependence based prefetching for linked data structures. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM Press, 115–126.

[40] Amir Roth and Gurindar S. Sohi. 1999. Effective Jump-Pointer Prefetching for Linked Data Structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Computer Society, 111–121.

[41] Yousef Saad. 2003. *Iterative methods for sparse linear systems*. SIAM.

[42] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing. *Proc. VLDB Endow.* 11, 4 (2017), 420–431.

[43] Aneesh Sharma, Jerry Jiang, Praveen Bommannavar, Brian Larson, and Jimmy Lin. 2016. GraphJet: Real-Time Content Recommendations at Twitter. *Proc. VLDB Endow.* 9, 13 (2016), 1281–1292.

[44] Alan Jay Smith. 1978. Sequential Program Prefetching in Memory Hierarchies. *Computer* 11, 12 (1978), 7–21.

[45] Ke Tu, Wei Qu, Zhengwei Wu, Zhiqiang Zhang, Zhongyi Liu, Yiming Zhao, Le Wu, Jun Zhou, and Guannan Zhang. 2023. Disentangled Interest importance aware Knowledge Graph Neural Network for Fund Recommendation. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM)*. ACM, 2482–2491.

[46] Kuansan Wang, Zhihong Shen, Chiyuan Huang, Chieh-Han Wu, Yuxiao Dong, and Anshul Kanakia. 2020. Microsoft Academic Graph: When experts are not enough. *Quant. Sci. Stud.* 1, 1 (2020), 396–413.

[47] Brian Wheatman and Helen Xu. 2018. Packed Compressed Sparse Row: A Dynamic Graph Representation. In *2018 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–7.

[48] Jaewon Yang and Jure Leskovec. 2012. Defining and Evaluating Network Communities Based on Ground-Truth. In *12th IEEE International Conference on Data Mining (ICDM)*. IEEE Computer Society, 745–754.

[49] Mohamad Zamini, Hassan Reza, and Minou Rabiei. 2022. A Review of Knowledge Graph Completion. *Inf.* 13, 8 (2022), 396.

[50] Xiangyu Zhi, Xiao Yan, Bo Tang, Ziyao Yin, Yanchao Zhu, and Minqi Zhou. 2024. CoroGraph: Bridging Cache Efficiency and Work Efficiency for Graph Algorithm Execution. *Proc. VLDB Endow.* 17, 4 (2024), 891–903.

[51] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *12th USENIX Symposium on Operating Systems Design and Implementation, (OSDI)*. USENIX Association.

[52] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034.