High-Performance Graph Processing on NVMe SSDs via Chunk-Based Representation and Access

RUI WANG, Zhejiang University, Hangzhou, China and Innovation and Management Center, School of Software Technology(Ningbo), Zhejiang University, Ningbo, China
WEIXU ZONG, Zhejiang University, Hangzhou, China and Zhejiang Lab, Hangzhou, China
SHUIBING HE*, Zhejiang University, Hangzhou, China
SONGHUA ZHOU, Zhejiang University, Hangzhou, China
XINYU CHEN, Zhejiang University, Hangzhou, China
ZHENXIN LI, Zhejiang University, Hangzhou, China
ZHENG DANG, Zhejiang University, Hangzhou, China

Existing external graph processing systems struggle with low I/O efficiency, high computational overhead, and substantial development costs on modern NVMe SSDs, due to their reliance on complex loading and computing models that transform random I/Os into sequential access. While in-memory graph systems with general-purpose memory-storage caches (like OS page cache or TriCache) offer improved support for fine-grained I/Os and simplified programming, they often fail to leverage specific graph access patterns, resulting in suboptimal performance. This paper aims to enhance the I/O efficiency of large-scale graph processing on NVMe SSDs. We first introduce a novel chunk-based graph representation model, featuring classified and hierarchical vertex storage and chunk layout optimization, to improve I/O utilization. Additionally, we present a latency-optimized access mechanism featuring user-space asynchronous I/O execution and hotness-aware chunk caching management to accelerate I/O and boost cache efficiency. Our prototype, ChunkGraph, enables flexible graph algorithm implementation and efficient execution. Experiments demonstrate ChunkGraph significantly outperforms existing external graph systems and in-memory graph systems using general-purpose cache subsystems, delivering several-fold speedups on overall performance and superior I/O efficiency.

CCS Concepts: • **Information systems** → **Hierarchical storage management**; *Data layout*; • **Hardware** → *External storage*.

Additional Key Words and Phrases: large-scale graphs processing, graph storage, NVMe SSD

1 Introduction

Graph is widely used to represent real-world relationships and is applied across diverse scenarios [13]. To support efficient graph analysis, numerous in-memory graph processing systems have been developed, including Ligra [55], GraphOne [26], and GAP [3, 4]. However, as graph sizes continue to grow, they often exceed the memory capacity

*Corresponding author.

Authors' Contact Information: Rui Wang, Zhejiang University, Hangzhou, Zhejiang, China and Innovation and Management Center, School of Software Technology(Ningbo), Zhejiang University, Ningbo, Zhejiang, China; e-mail: rwang21@zju.edu.cn; Weixu Zong, Zhejiang University, Hangzhou, Zhejiang, China and Zhejiang Lab, Hangzhou, Zhejiang, China; e-mail: zorax@zju.edu.cn; Shuibing He, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: heshuibing@zju.edu.cn; Songhua Zhou, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: sh.zhou25@zju.edu.cn; Xinyu Chen, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: xy.chen@zju.edu.cn; Zhenxin Li, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: zhenxin@zju.edu.cn; Zheng Dang, Zhejiang University, Hangzhou, Zhejiang, China; e-mail: dangzheng@zju.edu.cn.

ACM 1553-3093/2026/7-ART
https://doi.org/10.1145/3829357

of a single machine, presenting challenges for large-scale analysis. Emerging storage devices such as NVMe SSDs provide high-performance I/O at a low price [11], offering a cost-effective solution for scaling large graph processing.

Several external graph processing systems have emerged over the past decade [1, 27, 28, 54, 63, 74], with the purpose of enhancing the performance of large-scale graph analysis on earlier-generation external storage devices like HDDs and SATA SSDs. These traditional devices suffer from performance degradation when faced with intensive random access patterns [28]. To mitigate this issue, these systems adopt a *subgraph-based iterative model* that partitions the graph into subgraphs, stores them on disk, and processes them iteratively in sequential access. However, this approach incurs high computation overhead and suffers from limited I/O utilization due to synchronization and coarse-grained I/O [65]. While such trade-offs are acceptable for slower storage devices like HDDs, they are no longer cost-effective for NVMe SSDs, which provide higher bandwidth and comparable performance for both random and sequential access [30].

Recently, various techniques have been proposed to enhance graph processing performance on fast SSDs, including reducing access granularity to minimize I/O volume [8, 34] and scheduling access patterns to maximize I/O bandwidth utilization [9, 19, 20, 25]. Despite recent advancements in modern external graph processing systems, those based on the subgraph-based iterative model still face limitations. One major issue is low I/O efficiency, as accessing a single vertex's neighbors often requires loading an entire subgraph into memory. For instance, when executing the BFS algorithm on the YahooWeb graph, the average I/O utilization is observed to be lower than 2%, when using the latest external graph system Blaze [25]. Another challenge lies in the substantial computational overhead introduced by frequent synchronization between subgraphs. Experimental results show that running BFS on the YahooWeb graph using Blaze requires 154 times more CPU instructions compared to the widely used in-memory system Ligra[55]. Furthermore, these systems impose high development costs, as users must reimplement their graph algorithms within the constraints of a subgraph-centric computation model, often resulting in additional complexity and limited portability.

From another perspective, integrating a lightweight in-memory graph system with a memory-storage cache, such as the operating system (OS) page cache [45] or TriCache [11], presents an opportunity to facilitate large-scale graph processing with easy programming. For instance, the OS page cache can cache SSD-resident data in DRAM via swapping [44] or memory mapping [43], typically operating under a *page-centric caching model* with 4 KB page granularity. Users can use an in-memory graph system like Ligra to process large graphs by simply mapping graph data to SSD-backed files, without modifying the original algorithm code.

However, existing memory-storage cache systems are not tailored to efficiently support external graph processing and suffer from several limitations. One key issue is the mismatch between the fixed 4 KB page granularity and the varying sizes of graph vertices. This mismatch leads to low I/O efficiency for small vertices with few neighbors, and increased metadata overhead for large vertices with numerous neighbors. Moreover, vertices may still encounter the *vertex cut problem*, where even if a vertex fits within a single page, it can be split across two adjacent pages, resulting in redundant I/O operations. These limitations undermine the effectiveness of memory-storage cache systems in supporting external graph workloads.

To overcome the limitations of both the subgraph-based iterative model and the page-centric caching model, and enable I/O-efficient and user-friendly large-scale graph processing on modern storage devices such as NVMe SSDs, we propose ChunkGraph, featuring a novel *chunk-based graph representation model*, and a latency-optimized chunk access mechanism. Our main contributions are summarized as follows.

- We conduct a comprehensive analysis of existing single-machine solutions for large-scale graph processing, encompassing external graph systems and in-memory graph systems working with memory-storage cache subsystems. We identify and assess their limitations in supporting large graph processing on modern storage devices.

- We propose a *chunk-based graph representation model* to support efficient graph processing on modern storage devices. Central to this model is a *classified and hierarchical vertex storage strategy* that organizes vertices into aligned chunks based on structural properties, improving I/O efficiency for vertices with different sizes. This is complemented by a *chunk layout optimization method*, where vertices are strategically reordered and grouped within chunks to enhance access locality and significantly minimize internal fragmentation.
- We introduce a *latency-optimized chunk access mechanism* to accelerate I/O processing and enhance cache efficiency; this comprises *user-space asynchronous I/O execution* (converting kernel operations into asynchronous user-space requests to bypass kernel overhead and boost throughput) and *hotness-aware chunk caching* (dynamically identifying high-priority chunks using graph topology and access patterns, then applying differentiated cache replacement policies to maximize hit rates and minimize redundant I/O operations).
- We implement the prototype of ChunkGraph and conduct extensive experiments to demonstrate its efficiency. Experiments demonstrate that ChunkGraph outperforms both existing external-memory graph systems and in-memory systems relying on general cache solutions by several times attributed to substantially reduced I/O and computation overhead.

2 Background and Motivation

In this subsection, we first introduce the subgraph-based iterative model commonly adopted by existing out-of-core graph processing systems, and explore its limitations when used with modern storage devices. We also discuss the design of memory-storage cache subsystems and analyze their inefficiencies in supporting large graph processing.

2.1 Out-of-Core Graph Processing Systems

Subgraph based iterative model. To reduce random accesses on older external storage devices like HDDs and SATA SSDs, many external graph processing systems have adopted a *subgraph-based iterative loading and computing model* to convert numerous random I/O operations into a smaller number of sequential I/O operations. In this model, all vertices are divided into disjoint intervals, and each interval is associated with a subgraph, which stores all edges whose source vertices fall within this interval. Graph computing is performed in an iteration-based manner. In each iteration, the subgraphs are sequentially loaded from disk into memory, and the computations related to the loaded subgraph are executed. This process continues for multiple rounds until all computation is completed. As shown in Figure 1, the subgraph-based iterative loading and computing model is exemplified using the case graph. The case graph is divided into three subgraphs, namely g_0, g_1, and g_2, which are stored on disk. When the graph application requires accessing the neighbors of vertices v_1 and v_3, we load the whole subgraphs g_0 and g_1 sequentially into memory for computation.

To improve graph processing performance on faster SSDs, semi-external graph systems such as FlashGraph [8] and Graphene [34] keep vertex states in memory and only store edge data on SSDs, which reduces frequent disk accesses for querying vertex states. They also optimize I/O efficiency by employing fine-grained disk I/Os, such as loading graph data in a series of 2 MB blocks, to minimize the total I/O amount. Additionally, systems like GraFBoost [20], VPart [9], Blaze [25], and RealGraph+ [19] schedule the graph access pattern to maximize IO bandwidth utilization, through techniques such as vertex sorting, graph partitioning, value propagation using in-memory concurrent bins, and workload allocation. RealGraph+ also emphasizes increasing IO bandwidth by implementing SPDK-based optimization strategies to reduce the time costs of issuing IO requests, idle state waiting, and block processing. Note that these out-of-core graph systems still utilize the subgraph-based iterative model for external graph analysis.

Limitations when using modern storage devices. The subgraph-based iterative loading and computing model effectively reduces random disk I/Os but comes with drawbacks such as low I/O efficiency, expensive computation overhead and high development cost. While these costs are justified for slow external memory devices like HDDs

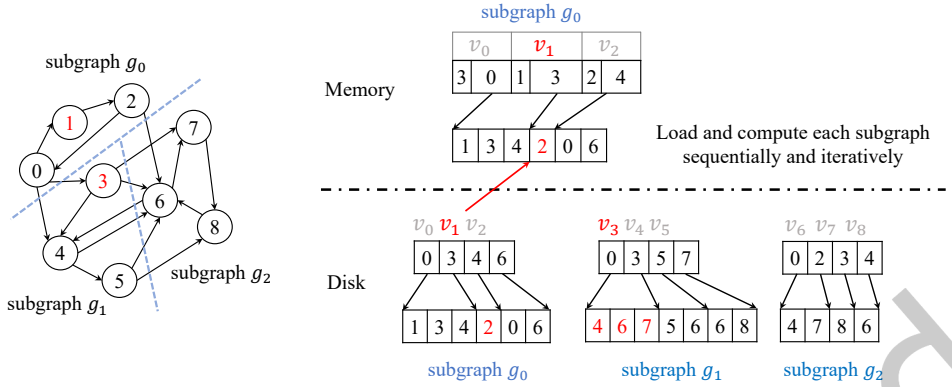


Fig. 1. Graph accessing in existing out-of-core graph processing systems.

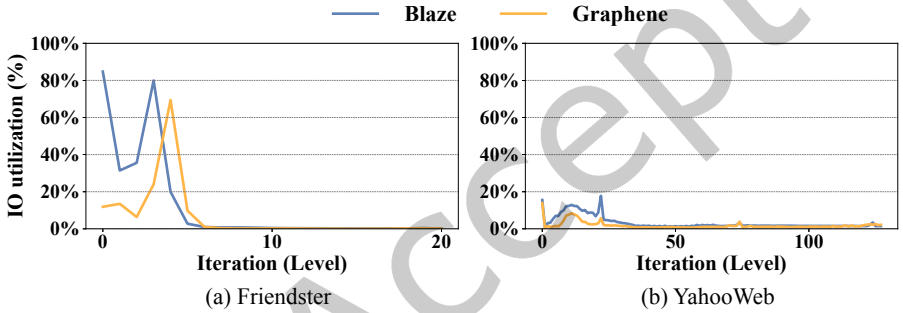


Fig. 2. I/O utilization during each BFS computation level.

that suffer from significant performance degradation due to random accesses [28], they become performance bottlenecks on emerging storage devices like NVMe SSDs, which have similar performance for random and sequential access [30].

Firstly, the subgraph-based iterative model suffers from *low I/O efficiency*. It requires loading entire subgraphs, even if only a small portion is needed, such as when visiting neighbors of a single vertex. To demonstrate this, we conducted experiments using Graphene [34] and Blaze [25] on the Friendster [12] and YahooWeb [69] datasets. We measure the I/O utilization during each BFS level, and show the results in Figure 2. In the initial levels of BFS, where more vertices are accessed, the I/O utilization is relatively higher. However, in subsequent iterations, the I/O utilization is very low, as few vertices are actually in need. On the Friendster graph, the average I/O utilization is only 6.32% for Graphene and 12.41% for Blaze, respectively. On the YahooWeb graph, the average I/O utilization is even lower, below 2%, for both systems. RealGraph+ [19] attempts to improve I/O efficiency by emphasizing data locality within subgraphs, thereby reducing the number of subgraphs accessed. However, it overlooks the intra-page fragments, leading to excessive read amplification for specific vertices. Additionally, RealGraph+ predominantly optimizes out-graph considerations, neglecting in-graph access for directed graph algorithms like betweenness centrality and PageRank, which leads to read amplification issues arising from random in-graph access.

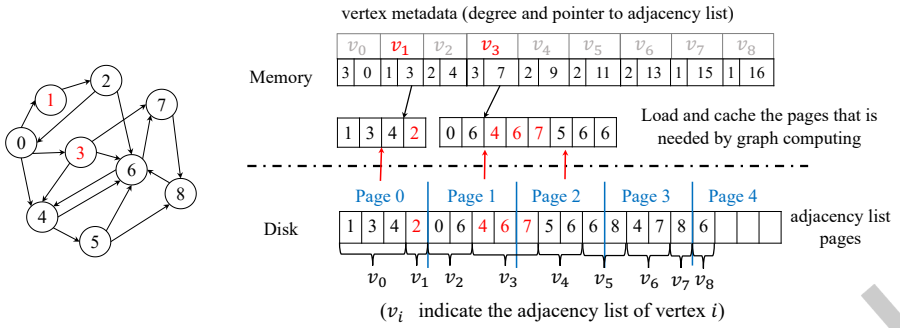


Fig. 3. Graph accessing in existing memory-storage cache systems.

Secondly, the subgraph-based iterative model also introduces *extra computing overhead* due to synchronization between subgraphs. To demonstrate this, we execute BFS algorithm on the YahooWeb graph using Blaze and Ligra-mmap, i.e., in-memory graph system Ligra working with OS's page cache by mapping graph data to SSD files. Ligra-mmap completed the computation in just 5.33 seconds, whereas Blaze took 46.18 seconds. We further counted the number of CPU instructions during these executions and found that Blaze required 154 times more CPU instructions than Ligra-mmap. Similar results were also observed for other graphs and algorithms (see §5.2). This high computing overhead significantly contributed to the poor performance of Blaze.

Furthermore, using these external graph systems incurs *expensive algorithm development costs*. User applications are required to implement their graph algorithms based on the subgraph-centric model and are responsible for managing data interaction between computation and I/O. Even experienced programmers need to invest significant effort in writing new algorithms after understanding the working model and interface of the respective systems. For example, implementing the BFS algorithm in the in-memory system Ligra only requires 34 lines of C++ code, while Blaze requires 75 lines, and Graphene even necessitates a total of 763 lines. This significantly increases the cost for user to utilize these systems, and similar situations arise for other graph algorithms.

Additionally, modern external graph processing systems *face significantly increased I/O issuance overhead*. Unlike earlier approaches like Graphene, which coalesce multiple user requests before dispatching them, contemporary systems like Blaze frequently generate numerous random I/O operations. This proliferation of fine-grained requests severely escalates handling costs and system overhead, illustrated by a 10.6 times increase in I/O operations when executing BFS on the YahooWeb dataset with the state-of-the-art system Blaze compared to Graphene. Greater I/O traffic imposes higher handling costs and elevates the overall system overhead. To further quantify the impact of these bottlenecks, we conduct a detailed performance breakdown of kernel-level I/O in § 5.3. As illustrated in Figure 17, the software-level I/O stack overhead, comprising kernel-mode coordination and buffer/page management can be substantial. For instance, during BFS on YahooWeb, software-related costs account for up to 52.6% of the total I/O time, hindering the system from saturating NVMe bandwidth. This significant orchestration overhead necessitates bypassing the kernel to minimize software-layer costs and enable high performance processing.

2.2 Memory-Storage Cache Subsystems

Page-centric memory caching scheme. Page cache [45] is a cache mechanism in the Linux operating system that caches pages read from storage devices into memory. It is implemented in system kernels and transparent to user applications through the swapping [44] or memory mapping techniques. With page cache, we can leverage a concise in-memory graph system like Ligra to perform out-of-core graph processing without any code modification.

During the computation process, page cache loads the corresponding adjacency list into DRAM when accessing a vertex, and caches recently accessed graph data in DRAM using the LRU cache replacement strategy [41], typically at the granularity of 4 KB pages. Figure 3 shows this page-centric memory caching model for the case graph. We store the vertex metadata, including vertex degree and adjacency list pointer, in memory, while maintaining all adjacency lists in disk through *file mmap*. Then, the page cache will be utilized to facilitate the transfer of adjacency list data between memory and disk. Considering the same example of accessing the neighbors of v_1 and v_3, we totally need to load three pages into memory for computation.

TriCache [11] is a recently introduced cache mechanism that operates as a user-space block cache utilizing a virtual memory interface and constructed on SPDK. One of its key benefits is the reduction of substantial kernel overhead associated with cache misses. TriCache also implements the page-centric memory caching model to effectively handle user data. Through the integration of TriCache with in-memory graph systems, it becomes feasible to conduct out-of-core processing of large graphs without algorithm code rewriting.

Limitations for large graph processing. The page-centric memory caching scheme used in existing memory-storage cache systems is not well-suited for graph processing, since it overlooks the unique characteristics of graphs. Graph algorithms typically access graph data at the vertex level, whose sizes vary greatly for different vertices in real world power law graphs [6, 14, 64]. These vertex accesses do not align well with the management granularity of 4KB pages. This mismatch results in issues such as low I/O utilization for small vertices with few neighbors and redundant metadata management costs for large vertices with massive neighbors. For example, when profiling the real-world graph Yahoo [69], it was found that 51.17% of non-sink vertices have only one or two in-neighbors, occupying a small amount of storage space (4 or 8 bytes). However, accessing these small vertices requires reading a full 4KB page from SSD, resulting in low I/O utilization. On the other hand, a small percentage (0.09%) of non-sink vertices have over 1024 in-neighbors, accounting for 58.44% of the total graph edges. The largest vertex even has over seven million in-neighbors, occupying 7459 4KB-pages that are consistently accessed together, but may be loaded or evicted separately by page cache, resulting in complex and redundant metadata management. Quantitatively, standard page-centric replacement policies like LRU and FIFO exhibit significantly lower hit rates (up to 21% lower) and higher read amplification compared to our hotness-aware approach. We empirically evaluate this in § 5.3. This underscores the limitations of general-purpose caching in effectively prioritizing graph data.

Besides, even for vertices whose sizes fit in pages, they may also suffer the *vertex cut problem*, in which a vertex smaller than a page is placed across two adjacent pages, such as the v_3 and v_5 shown in Figure 3. It also incurs twice the I/O cost for these cut vertices. We further profile the YahooWeb graph and find that among the vertices with in-degree in the range of $[3, 1024]$, 9.73% of them are cut and placed across two pages, and these cut vertices are present in 74.69% of the total pages. This vertex cut problem further reduces I/O efficiency and graph processing performance.

Summary. The elaboration provided above clearly demonstrate that existing solutions fall short of fully leveraging the capabilities of modern NVMe SSDs to facilitate efficient and easy-to-use large-scale processing. This motivates us to design a new large graph processing model, aiming to enhance the graph I/O and computation efficiency without altering the computation mode of existing in-memory graph systems, thereby enabling better performance on modern storage infrastructure.

3 Chunk-Based Graph Representation Model

In this section, we aim at enhancing the I/O efficiency of graph data and enabling effective and user-friendly large-scale graph processing on modern storage devices such as NVMe SSDs. However, designing an effective graph storage structure remains challenging due to the following issues:

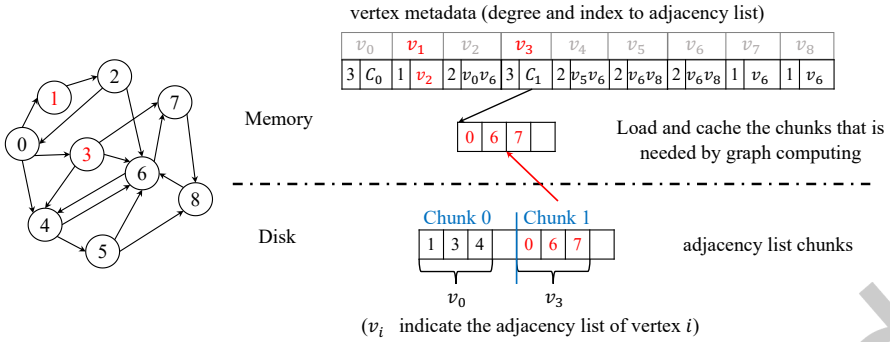


Fig. 4. Chunk based graph representation model.

- Graph data layout should align with the access granularity of the operating system and storage hardware, as misaligned partitioning can introduce additional I/O overhead.
- Real-world graph workloads suffer from access mismatch caused by highly skewed degree distribution: low-degree vertices exhibit poor I/O efficiency, while high-degree vertices incur excessive metadata overhead. These disparities make it challenging to design a storage strategy that balances efficiency and compactness.
- Diverse graph access patterns, including sparse accesses to small subsets and dense accesses to large portions, pose challenges in designing storage model that accommodates such variability.

To address these challenges, we propose a *chunk-based graph representation model*. In contrast to the fixed-size page-centric model, which uniformly manages different vertex data in 4KB pages, our chunk-centric model meticulously organizes diverse vertex data of varying degrees into aligned chunks of different sizes. To further illustrate the above idea and analyze its benefits, we still consider the aforementioned example graph and show the data organization and I/O process of the chunk-centric model in Figure 4. For very small vertices, e.g., v_1, v_2, we directly encode them in the metadata which is stored in memory, thus completely avoiding the I/O overhead of these vertices. For other vertices with different degrees in different ranges, e.g., v_0, v_3, we organize them in different chunks of different appropriate sizes, and ensure that one vertex does not straddle two chunks through a whitespace based alignment chunk design. Take the same example mentioned above which needs to access v_1 and v_3, v_1 has only one neighbor which can be read from DRAM vertex metadata, and v_3's neighbors are stored in chunk 1, which may need to be read from SSD. So we only need one page I/O in total. While the fixed-size page-centric model requires three pages to be loaded (refer to Figure 3). This approach aims to enhance I/O efficiency for small vertices while minimizing redundant metadata management costs for large vertices, addressing the mismatch issues encountered with power-law graphs in the real world. Subsequently, we delve into the details of its key design techniques, including classified and hierarchical vertex storage and chunk layout optimization.

3.1 Classified Hierarchical Vertex Storage

In this subsection, we introduce the classified and hierarchical storage formats , as shown in Figure 5. To identify and index the vertex neighbor data for each vertex, we allocate 12 bytes for each vertex to store its metadata information. This consists of 4 bytes for its degree and 8 bytes for an index into its adjacency list, which aligns with the approach used in existing graph processing systems [55]. Additionally, we store all vertices' metadata in memory, as this metadata is frequently accessed and typically fits in memory in most scenarios [8, 25, 34]. We classify all vertices into three categories according to their degrees, namely (1) *mini vertex* with degree equals one or two, (2) *medium vertex* with degree in the range of $[3, d_\theta]$, (3) *super vertex* with degree larger than d_θ, where

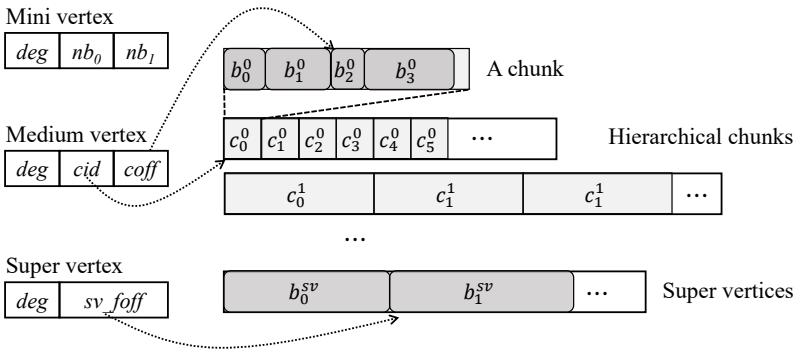


Fig. 5. Differentiated data structures for different vertices.

d_θ is set to adjust the size of a medium vertex. We then utilize different data structures to store different types of vertices. Next, we introduce the data structure and access flow of each vertex type, respectively.

In-index mini vertex storing. For mini vertices with at most two neighbors, we directly store them in the 8-byte vertex index. It's worth noting that this 8-byte vertex index is stored in memory and is previously used to store the pointer to the vertex adjacency list in existing works [8, 25, 34, 55]. However, we utilize these 8-byte vertex metadata fields in different ways for different vertices in our chunk based graph representation model. Specifically for mini-vertices, we use these 8-byte vertex metadata fields to directly store their neighbors. This simple but effective approach offers several benefits. First, it allows us to save on the storage cost of these mini vertices without incurring any additional overhead. Second, it helps in avoiding the I/O overhead when accessing neighbors of these mini vertices, as these metadata fields are always kept in memory. Finally, we can retrieve the neighbor information of these mini vertices in a single memory access, whereas all existing graph systems require at least two memory accesses (one for reading the vertex index pointers and one for looking up neighbors based on the vertex index pointers), thereby improving CPU cache performance. We also study its impact on CPU cache load misses in §5.3.

Chunk based medium vertex storing. For medium vertices with degrees limited in the range of $[3, d_\theta]$, we organize them into aligned chunks of appropriate sizes. We classify the storage space into multiple segments, and each segment stores a series of chunks of the same size, and each chunk consists of the adjacency lists of multiple vertices. If the remaining space of current chunk is too small to hold all neighbors of next vertex, then we will leave this whitespace unused and store the whole adjacency list of next vertex to next chunk. In this way, we can ensure each medium vertex to be completely stored in only one chunk, and avoid extra chunk I/O caused by the vertex cutting issues. We also study its impact on number of accessed chunks in §5.3. To find the neighbors of a medium vertex v, we use the 8-byte vertex index field to store the 4-byte cid and 4-byte $coff$, which denote the ID of the chunk to which v belongs and the offset within this chunk where v's adjacency list resides, respectively.

Hierarchical chunk implementation. Based on the chunk-based storing strategy, a key challenge in our strategy is determining the optimal number of layers. ChunkGraph adaptively tunes these parameters based on the graph's degree distribution: graphs with highly skewed or wide-ranging distributions benefit from more layers to ensure chunk sizes closely match adjacency list lengths. By default, we employ a four-layer configuration, which we found offers a practical balance between adaptability and implementation simplicity. In particular, four layers allow vertices of varying sizes to be distributed uniformly across different layers, which simplifies buffer management and reduces internal fragmentation within chunks. Another question is how to set the chunk size, as smaller chunks improve I/O efficiency for small vertices, while larger chunks accommodate large ones. To accommodate vertices of varying sizes, we propose the hierarchical chunk implementation by putting medium vertices with different degrees to chunks of different sizes. Figure 6 shows a four-layer hierarchical chunk implementation, where the

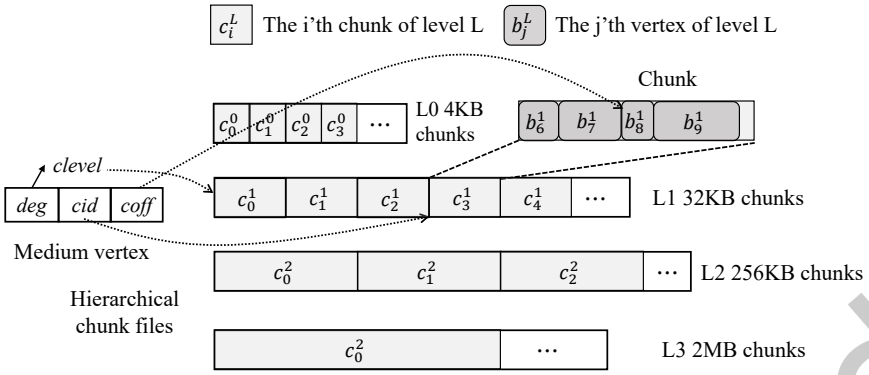


Fig. 6. A four-layer implementation of hierarchical chunks based medium vertex storing.

chunk sizes of each layer are 4KB, 32KB, 256KB and 2MB, respectively. We place a vertex in a lower-level chunk if its adjacency list size is smaller than the chunk size. For instance, the vertices with degrees in the range of $[3, 1021]$ would be stored in L0 4KB-chunks, the vertices with degrees in the range of $[1022, 7168]$ would be stored in L1 32KB-chunks, the vertices with degrees in the range of $[7169, 58365]$ would be stored in L2 256KB-chunks, and the vertices with degrees in the range of $[58366, 465920]$ would be stored in L3 2MB-chunks. Finally, when we query a medium vertex v, we can first get its level according to degree, then we get the corresponding chunk according to its cid in that level, next we can get this v's adjacency list from its $coff$ within the chunk.

Threshold parameter settings. We have set a default threshold of 2 for mini-vertices, considering that each vertex's metadata reserves 8B for its neighbor pointer on a 64-bit machine, and each neighbor occupies 4B. If a vertex's degree does not exceed two, we store the neighbors within the field of the 8B-pointer to minimize indirect access and storage overhead. Regarding the value of d_θ, our default setting is determined by the chunk size and neighbor size. Assume that the degree range for the l-th layer is denoted as $[d_\theta^{l-1}, d_\theta^l]$, we set the value of d_θ^l as $\frac{sizeof(chunk^l)}{sizeof(vertex)} - (d_\theta^{l-1} + 1)$. By recursively applying this expression, we can derive the degree threshold for each layer chunk. In the case of our four-layer chunk implementation with sizes 4KB, 32KB, 256KB, and 2MB, we compute d_θ^l values accordingly: $4KB/4B - (2 + 1) = 1021$ for the 4KB layer, $32KB/4B - (1021 + 1) = 7168$ for the 32KB layer, $256KB/4B - (7168 + 1) = 58365$ for the 256KB layer, and $2MB/4B - (58365 + 1) = 465920$ for the 2MB layer. Our empirical testing in §5 has shown stable performance with these default settings, and users have the flexibility to adjust these parameters for optimal performance. Our implementation also supports flexible configurations for the number of layers, and we evaluate its impact on graph processing performance in §5.5.

Hierarchical chunk buffer size setting. Based on the four-layer chunk implementation, we need to manage four chunk buffers in memory. An important problem is how to set the chunk buffer sizes. To realize fairness, we adopt a chunk file size proportional buffer size allocation strategy, which proportionally divides the buffer space into four parts, according to each layer chunk file size, and use them as the corresponding chunk buffers. Specifically, suppose we have M bytes memory for all chunk buffers, and the chunk file size of each layer is S_0, S_1, S_2 and S_3, respectively. Then we allocate $\frac{S_i \times M}{S_0+S_1+S_2+S_3}$ bytes memory for the chunk buffer of layer L_i. By default, we set the total chunk buffer sizes M to be the total memory space minus the memory size used to store vertex metadata and algorithm information. We also evaluate the impact of total buffer sizes on performance in §5.5.

Huge page based super vertex storing. For super vertices, their degrees are larger than d_θ, which equals 465920 based on the four-layer chunk implementation mentioned above. We manage them using the Direct HugePage

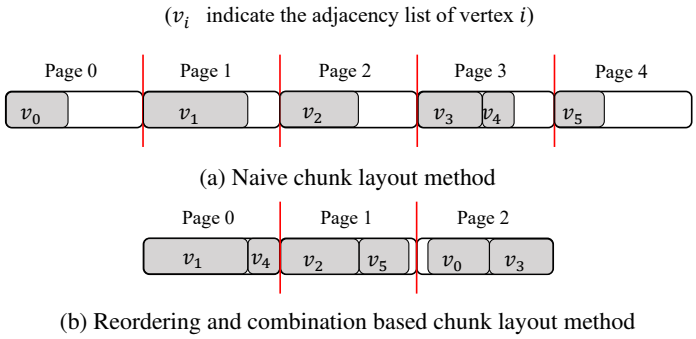


Fig. 7. Chunk layout optimization.

(DHP) technique to reduce the TLB miss ratio since the sizes of these super vertices are close to or larger than 2 MB. We use the 8-byte vertex indexes to store the pointers to the adjacency lists of these super vertices for querying them.

3.2 Chunk Layout Optimization

Problems of data locality and chunk fragmentation. Based on the above-mentioned chunk-based vertex storing strategy, a naive vertex layout method is to sequentially put each vertex into the current chunk in ascending order of vertex ID. In this way, vertices with consecutive IDs will fall in same chunks. When loading a chunk from disk into memory, these vertices in the same chunk may not be accessed simultaneously, leading to low intra-chunk data access locality and low I/O utilization issues. Besides, in order to avoid the vertex cut problem, if the remaining space of the current chunk is too small to hold the entire adjacency list of the current vertex, then we will leave this whitespace of the current chunk unused and store the entire adjacency list of the current vertex to the next chunk. Hence, there may be some whitespace fragments at the end of these chunks, as shown in Figure 7(a). Taking our four-layer chunk implementation on the real-world graph Yahoo Web [69] as an example, we observe that 95.24% of the chunks have intra-chunk fragments, and these fragments occupy 10.06% of the storage space of all chunks. Please refer to §5.3 for the detailed experiment setup and more experiment results for each level of chunks. These fragments waste valuable memory space and degrade cache performance.

Reordering based chunk layout optimization. To enhance the intra-chunk data access locality during graph processing, our goal is to place vertices that are likely to be accessed simultaneously into the same chunk. In graph algorithms, when we access a vertex v, it is highly probable that we will also access its neighbors or brother vertices, i.e., vertices linked to the same neighbor. Thus, we prioritize placing a vertex and its neighbors and brother vertices in the same chunk. Drawing inspiration from [29, 67], we conduct a reordering-based chunk layout optimization. This involves reordering vertices and then placing consecutive vertices in the new order into the same chunk. We store the out-graph and in-graph separately. For illustration, we will focus on the out-graph. The specific steps are as follows. (1) Sort all vertices in descending order of their in-degrees to create a list P. This ordering is motivated by the observation that vertices with higher in-degrees generally have a higher access probability of their out-neighbors. (2) Select a vertex from the head of list P and use it as a root to conduct a two-level BFS. Add the BFS-traversed vertices into another list Q and meanwhile remove them from P. (3) Check the total number of vertices in Q. If it is smaller than a predefined threshold R, repeat step (2) by selecting another root vertex for BFS traversing. If the total number of vertices in Q is equal to or larger than R, add list P to the tail of list Q to form the new order of all vertices. (4) Place consecutive vertices in the new order into the same chunk. We set $R = 95\% \times |V|$ by default to make most vertices reordered.

Vertex-combination based fragment optimization. Memory fragmentation is a common problem, and there are many classic system-level memory optimization strategies that can alleviate the fragmentation problem, such as the buddy system [40], slab allocation [46] and knapsack problem based approach [2, 47], etc. However, in order to deal with various general situations, these strategies often involve complex logic and implementation, and their optimization effectiveness in the case of graph computing is also limited. Fortunately, we can exploit the characteristics of the graph structure to optimize the intra-chunk fragmentation problem. Specifically, in real-world power-law graphs, the number of small vertices is usually much larger than the number of large vertices [6, 14, 64]. In order to minimize intra-chunk fragmentation and enhance cache performance, we propose a reordering and combination-based in-chunk vertex layout strategy, as depicted in Figure 7(b). This strategy involves reordering all vertices, as previously illustrated, and then prioritizing the selection of vertices from the head of list Q to be placed in the current chunk. Additionally, small vertices picked from P are used to fill the remaining fragment space in the current chunk, thereby effectively reducing fragmentation and improving space utilization across all chunks.

We point out that this vertex-combination based fragmentation optimization method is very useful in reducing the number and space occupation of intra-chunk fragments. Because the intra-chunk fragments generated by large vertices can always be filled up by small vertices, and we have enough small vertices to fill these fragments. Also taking the above-mentioned Yahoo Web graph as an example, by adopting this vertex-combination based fragmentation optimization method, only 52.77% of the chunks have intra-chunk fragments, and the space occupation rate of all intra-chunk fragments is reduced to only 6.96%. For the remaining fragments, if all whitespace fragments always appear at the end of chunks, these whitespace fragments may always fall in the same cache line group according to the multi-way set associative mapping in the last level cache, resulting in cache line wastes. Therefore, we interleave the white space in different chunks to alleviate this problem. Note that this reordering and combination based chunk layout optimization is conducted in preprocess. This reordering and combination based chunk layout method can improve graph computing performance by up to 2.54 times (refer to §5.3).

Differentiated chunk access optimization. Graph algorithms commonly involve top-down and bottom-up access patterns, corresponding to sparse access and dense access, respectively. In the top-down sparse access pattern, only a small portion of vertices is accessed. Our chunk-based design is beneficial for grouping these accessed vertices in a small portion of chunks, greatly improving graph processing performance, such as BFS performance. However, in the bottom-up dense access pattern, where most vertices are accessed in each iteration, the access order generally follows the vertex ID order. With our reordering and combination-based chunk layout design, vertices with consecutive IDs are distributed to different chunks, leading to decreased performance in these situations. To address this issue, we implement a differentiated chunk access optimization. During preprocessing, for each vertex, we generate a key-value pair $< idx, vid >$, indicating that the idx position in the chunk is associated with the neighbors of vertex vid. For bottom-up dense access pattern situations, we traverse all vertices according to the vertex idx order, and then look up the corresponding vid based on the above key-value pair and use vid for graph algorithm computing. This alignment ensures that data access is in line with the chunk order, improving I/O efficiency during graph computing.

4 Latency-Optimized Chunk Access Mechanism

Building upon the chunk-based graph representation model, we develop a *latency-optimized chunk access mechanism*, aiming at accelerating I/O processing and improving cache management efficiency. This mechanism incorporates two essential techniques: user space asynchronous I/O execution and hotness-aware chunk caching management. In the following subsections, we present their design in detail. By incorporating the above designs, we implement a prototype system ChunkGraph with user-friendly APIs, which is capable of efficiently processing large-scale graphs on modern NVMe SSDs. The overall design of ChunkGraph is illustrated in Figure 8.

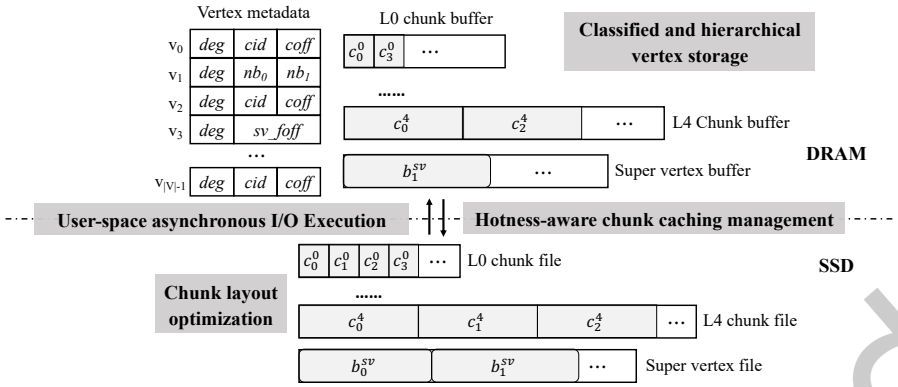


Fig. 8. Overview of chunk-based graph representation model.

4.1 User-space Asynchronous I/O Execution

Modern systems frequently process high volume of I/O requests, resulting in significant system overhead including system calls, context switching, memory copying, and scheduling. In this subsection, we introduce a user-space asynchronous I/O execution mechanism specifically designed to efficiently handle high-throughput I/O workloads during graph processing while effectively mitigating kernel-level overhead and improving overall system responsiveness.

Amplified overhead of I/O requests. The I/O workflow in graph processing systems typically consists of two primary stages: I/O request handling and data transfer execution. During the request handling stage, the operating system initiates, constructs, and schedules I/O operations, which involves context switching, building I/O request structures, scheduling, interrupt handling and file system coordination. The data transfer stage performs the actual movement of data, including device access, transmission, and memory copying. As storage devices deliver higher throughput and lower latency, the relative overhead of managing I/O requests has become increasingly significant. While individual operations complete more quickly, the elevated request rate amplifies total processing costs. For instance, in systems built on a chunk-based graph representation, we observe 87% improvement in I/O bandwidth during BFS traversal compared to Graphene, primarily due to improved I/O efficiency and more intensive requests. However, this improvement is accompanied by a 13.7 times increase in the number of I/O requests. Therefore, minimizing request-handling overhead is essential for scalable I/O efficiency in graph processing.

User-space asynchronous I/O interface based on SPDK. Recent advancements in storage technology have motivated numerous studies on user-space I/O handling [24, 42, 71]. SPDK (Storage Performance Development Kit) relocates the NVMe driver from the kernel to user space, allowing direct access to SSDs through the kernel bypass mechanism. SPDK employs a polling mechanism instead of interrupts to handle I/O requests, which enhances overall throughput. Compared to other solutions, SPDK leverages DPDK-backed hugepage memory management, providing better cache efficiency when handling large datasets, which is particularly advantageous for large-scale graph.

Building on SPDK, we design and implement a user-space asynchronous I/O interface for our system. Figure 9 illustrates the I/O management procedure leveraging SPDK. Prior to processing graph data, the system binds the NVMe device to user space via the uio_pci_generic driver. We then employ multiple queue pairs, each consisting of a submission queue and a completion queue to manage unprocessed I/O requests in a parallel and scalable manner. User I/O requests are submitted to SPDK's submission queue and processed by user-space reactor threads.

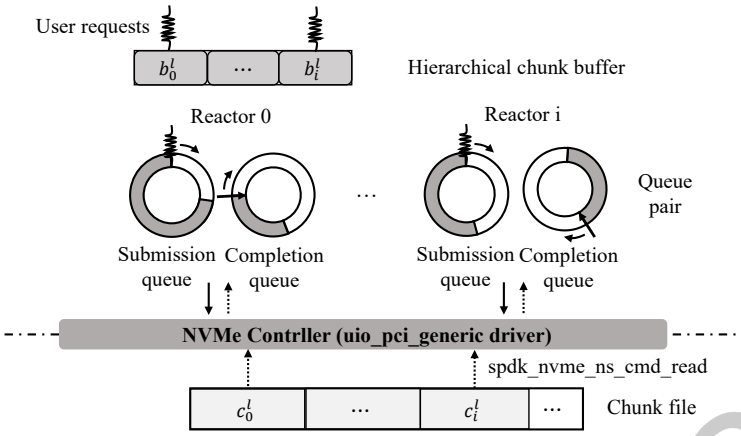


Fig. 9. User-space Asynchronous I/O Execution.

These threads forward requests to the block device driver. Upon completion of the read/write operations by the block device, SPDK captures the completion events in the completion queue, enabling subsequent data transfer to user space. Reactor threads continuously poll for submission and completion events to promptly update request status, thereby reducing overhead and allowing CPU threads to perform other tasks while I/O operations are in progress. This approach bypasses the kernel I/O path, eliminating overhead from context switching and interrupt handling. Furthermore, while ChunkGraph is optimized for the high-concurrency NVMe-SPDK path, it maintains architectural flexibility to support traditional storage. When SPDK is unavailable or when running on legacy hardware (e.g., SATA SSDs or HDDs), the system can seamlessly fall back to standard kernel-space I/O interfaces, while retaining the benefits of our chunk-level data organization and buffering strategies.

Chunk-binding based buffer management. Our system loads chunk data from layer l into the corresponding layer of hierarchical buffer, as mentioned in § 3.1. Building on the above design, we leverage SPDK to manage buffers entirely in user space. Unlike traditional I/O processes that utilize the page cache and involve memory copy to user space, SPDK pre-allocates hugepage memory as a user-space buffer and manages it via DPDK, thereby eliminating memory copy overhead. SPDK also supports direct specification of logical block addresses (LBAs) and access lengths, which allows us to load variable-sized chunks flexibly. However, the challenge lies in efficiently mapping data chunks to different buffer regions under high concurrency. To mitigate contention, we implement a chunk-binding mechanism that associates buffer partitions with SPDK reactor threads. Given i threads in the system, the buffer is divided into i partitions, and chunks are assigned to partitions based on specific hashing scheme. Meanwhile, each partition is exclusively managed by its designated reactor thread and corresponding queues. Specifically, if chunk i maps to the i-th buffer partition and a buffer miss occurs, the i-th reactor thread processes the request, loads the data into the corresponding partition, and updates the buffer page table. On buffer hits, the chunk can be accessed directly from the corresponding partition. We target to reduce contention and maintain compatibility with the single-threaded access mode of each queue pair. We also design a hotness-aware buffer managing policy to handle cases where a buffer partition reaches capacity, as detailed in § 4.2.

Engineering Overhead and Design Choice. While user-space asynchronous I/O provides considerable throughput gains, it introduces specific engineering trade-offs. From development perspective, our implementation adds approximately 2,000 lines of C++ code to manage the I/O lifecycle, covering DMA-compliant memory alignment, lockless queue pair management, and reactor-based polling. Importantly, this complexity is transparent to users

and requires no modifications to existing graph algorithm code. From deployment perspective, however, SPDK necessitates manual environment configuration, such as unbinding NVMe devices from the kernel driver and allocating dedicated hugepages.Despite these overheads, we prioritized SPDK over alternatives like io_uring or multi-threaded AIO. Unlike these kernel-based interfaces, which still incur interrupt-handling and context-switching overhead, SPDK's pure user-space polling is essential for saturating NVMe bandwidth and minimizing latency during the high-frequency, fine-grained accesses typical of large-scale graph processing. Moreover, SPDK manages I/O buffers through hugepage-backed user-space memory, further reducing software overhead compared with io_uring. We study its impact on overall performance and I/O throughput in § 5.3.

4.2 Hotness-Aware Chunk Caching Management

Given limited memory space, a key approach to enhancing the efficiency of large-scale out-of-core graph processing is to increase the cache hit rate of graph chunks stored on external devices. In this subsection, we first introduce a chunk hotness metric based on in-degree and real-time access frequency, and then introduce a hotness-aware chunk caching strategy to improve access performance.

Chunk hotness measurement. We aim to cache hot chunks with minimal space overhead to maximize buffer utilization. As graph data is typically accessed at chunk granularity, with each chunk comprising of multiple vertices, we estimate chunk hotness based on the vertices it contains. Specifically, vertex hotness is characterized by two aspects. The first relates to the graph's topology, which we define as *structural hotness*. Vertices connected to more vertices tend to be accessed more frequently, resulting in correspondingly increased storage demands. The second aspect captures real-time access frequency, which we define as *dynamic access hotness*. Given the substantial variation in access patterns across different graph computing algorithms, it is essential to incorporate runtime behavior in our estimation. Based on these two aspects, we define the hotness of a vertex v as

$$H(v) = \alpha \times H_s(v) + (1 - \alpha) \times H_d(v)$$

where $H_s(v)$ denotes the structural hotness, $H_d(v)$ denotes the dynamic access hotness, and α is a tunable parameter that balances their contributions.

The structural hotness $H_s(v)$ is related to the in-degree $indeg(v)$ and out-degree $outdeg(v)$ of the vertex, and is defined as

$$H_s(v) = (\beta_1 \times indeg(v))/(\beta_2 \times outdeg(v))$$

where β_1 and β_2 control the respective weight of in-degree and out-degree. Elevated in-degree reflects increased likelihood of being accessed, while lower out-degree implies reduced storage overhead, such that vertices with higher $H_s(v)$ values correspond to greater structural hotness. On the other hand, the definition of $H_d(v)$ must adapt to real-time workloads, i.e., the access frequency $f(v)$ of vertex v. $H_d(v)$ can be periodically refined to accommodate evolving application scenarios. Finally, based on vertex hotness measurement, we define the hotness of a chunk C as the sum of the hotness values of all vertices contained within it, namely

$$H(C) = \sum_{v \in C} H(v) = \alpha \times \sum_{v \in C} H_s(v) + (1 - \alpha) \times \sum_{v \in C} H_d(v).$$

ChunkGraph initializes the data layout according to graph topology, using structural hotness to place vertices before queries begin. During query processing, it further adjusts hotness estimates online based on observed access frequencies, allowing data layout to adapt to the evolving workload. By default, we set $\alpha = 0.5$ and $\beta_1 = \beta_2 = 1$, to provide a balanced estimation of hotness across diverse graph types. Specifically, $\alpha = 0.5$ assigns equal importance to structural characteristics and dynamic access frequency, whereas $\beta_1 = \beta_2 = 1$ balances the benefit of aggregating hot vertices within a chunk against the associated spatial overhead. Meanwhile, these parameters are tunable for

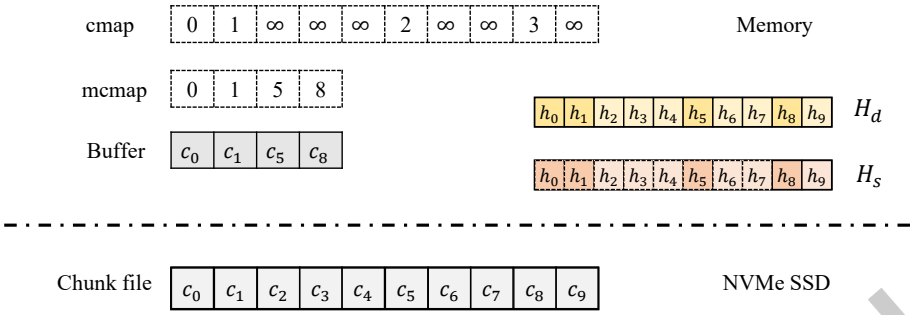


Fig. 10. Overview of Chunk buffer management architecture.

specific application needs. For workloads that exhibit strong dynamic access patterns, such as web link prediction, a smaller α and a larger β_1/β_2 ratio may enable more aggressive hotness updates.

Hotness-aware chunk storing and caching. Based on the chunk hotness measurement strategy, we can identify frequently accessed chunks across the graph data. After completing layout optimization, we obtain the structural hotness of each chunk, which enables us to preload hot chunks during system initialization to enhance startup access performance. During system execution, the influence of dynamic access hotness on chunk hotness intensifies, leading to continuous changes to the hotness information of cached chunks. Hence, it is essential to make real-time decisions on chunk retention in the buffer. Due to frequent I/O requests and constrained buffer size, we fetch graph data into the buffer based on user requests, avoiding proactive strategies that could cause additional bandwidth contention. Therefore, we prioritize considering which data to evict when the cache is full, for which we could leverage chunk hotness information. We have developed several hotness-based replacement policies, e.g., a threshold-based replacement policy. During graph computation, when loading a new chunk into memory, we select a victim chunk from the cache by traversing chunks in FIFO order. If the hotness of C is lower than the preset threshold θ, we evict C, otherwise we keep C in memory and continue searching the next cached chunk. We set the threshold θ as the average hotness of all cached chunks by default. We have also implemented alternative strategies, such as a min-heap-based policy that replaces the coldest chunk. This approach provides more precise hotness information under certain workloads, correspondingly resulting in higher maintenance overhead. Different strategies demonstrate varying benefits depending on the graph data and workload characteristics. and we evaluate their effectiveness through experiments in § 5.3.

Figure 10 illustrates the overall architecture of the cache management system. For each chunk file layer, we allocate a contiguous region of memory space based on strategy described in Section 3.2. After optimizing the chunk layout, we obtain the hotness value of all chunks, with the associated metadata stored in contiguous memory space. Structural hotness and dynamic access hotness are maintained separately to accommodate variations in graph topology and runtime workload. When the buffer is initially empty, we preload the hottest chunks, such as those exceeding predefined threshold in the threshold-based policy. To determine whether a chunk is cached and where it resides in memory, we maintain a bidirectional mapping. The chunk map (*cmap*) records the buffer position of each chunk in memory, where a value of ∞ indicates a cache miss. The memory chunk map (*mcmap*) holds the chunk ID stored in each buffer slot. During graph processing, the system looks up *cmap* to check for a hit; if found, it returns the buffer position, otherwise it loads the chunk into the buffer. When the buffer is full, the replacement strategy selects a chunk for eviction, and corresponding entry in *cmap* is invalidated using the index from *mcmap*.

Maintaining hotness information inevitably incurs extra computational and space overhead. The additional storage requirement is minimal, as the number of chunks is significantly smaller than the overall graph size. Even

when storing the entire graph dataset with the smallest chunk size, the hotness metadata accounts for less than 5% of the total dataset size. As for the computational overhead, the improvements in performance provided by hotness-aware management generally outweigh the associated costs, as shown in our evaluation in § 5.3.

4.3 Prototype System

Preprocessing. In preprocess, we create both the out-graph and in-graph of a directed graph. Specifically, we first convert the graph data from the original edge-list format to CSR format, as in existing graph systems. Next, we generate chunked data from the CSR as described above. We then create metadata for each vertex and store them in memory, and generate chunk-level hotness information based on graph topology. The preprocessing cost is also evaluated in §5.5.

Runtime vertex accessing. During the runtime of graph computing algorithms, when we need to access the adjacency list of a vertex v, we first get its degree $d(v)$ from the vertex metadata, and then perform different reading processes according to its vertex type: (1) If $d(v) \leq 2$, i.e., v is a mini vertex, then we directly read its neighbors from its $index$. (2) If $3 < d(v) \leq d_\theta$, i.e., v is a medium vertex stored in a chunk, then we further get the chunk level according to $d(v)$, and get the cid and $coff$ from its $index$. Next, we get the chunk data from the corresponding chunk buffer and read its neighbors from the chunk, and meanwhile update the chunk hotness. (3) If $d(v) > d_\theta$, i.e., v is a super vertex, then we read its neighbors from the pointer stored its $index$.

Prototype system. We take the popular in-memory graph processing system Ligra [55] as the baseline to implement our prototype system, ChunkGraph. It's important to note that we only modify the graph storage and access components of Ligra, while keeping the computing framework unchanged. Consequently, the graph algorithms already implemented in Ligra can directly run in our ChunkGraph without requiring any code modifications. For the implementation of a new graph algorithm, it only needs to call Ligra's graph interface, making it much more user-friendly than existing out-of-core graph systems, as mentioned in §2.1. Furthermore, we emphasize that our design is orthogonal to the computing models proposed in other graph systems. This implies that we can also implement our chunk-based graph representation model to other graph systems to achieve further performance improvements or to cater to more general applications.

5 Evaluation

5.1 Experiment Settings

Test bed. All experiments are performed on a server with two 2.10GHz Intel(R) Xeon(R) Gold 5318Y processors, each with 24 physical cores with hyper-threading enabled (48 logical cores). For memory, the server equips with 8× 16 GB (128 GB) DRAM which are interleaved inserted to the memory slot. For storage, the server features two 4 PCIe-attached Intel P5520 NVMe SSDs, each with a capacity of 3.84TB and capable of 1M 4KB-read IOPS and 0.2M 4KB-write IOPS in total. It also includes two 6TB HDDs for storing graph datasets. Additionally, the server is equipped with 8 × 128GB (1 TB) Intel Optane Persistent Memory 200 Series for comparison with PMEM-based graph system. The server operates on Ubuntu 20.04.5 LTS with a Linux kernel of 5.4.0.

Comparison Systems. We choose Blaze [25] and Ligra-mmap as our primary comparison baselines, as they correspond to the representations of out-of-core graph processing system and in-memory graph system working with memory-storage cache subsystem. Blaze is the state-of-the-art, open-source out-of-core graph system that introduces a pioneering scatter-gather technique known as *online binning*. It enables value propagation among graph vertices without requiring atomic synchronization, leveraging the high-speed random access capabilities of modern fast SSDs. It is important to note that these techniques are orthogonal with our designs in ChunkGraph. On the other hand, Ligra is a widely used in-memory graph processing framework designed for shared-memory architectures. It offers a suite of efficient graph algorithms capable of handling large-scale graphs on a single

machine with multiple cores. Ligra-mmap is a variant of Ligra that utilizes Linux's *mmap* mechanism to map the graph data files into the virtual memory space of the process, enabling efficient out-of-core graph processing. These two baselines were chosen to provide a comprehensive comparison across existing advanced out-of-core graph processing solutions. Additionally, we also compare with other out-of-core graph processing solutions, including the state-of-the-art external graph system MBFGraph [32], the semi-external graph system Graphene [34], the persistent memory (PM)-based graph storage system XPGraph [64], and Ligra working with the latest user-level transparent memory-storage system TriCache [11]. Note that in our experiments, most graph systems, including ChunkGraph, Ligra-swap, Ligra-mmap, Ligra-TriCache, Blaze, MBFGraph, and Graphene, were evaluated on NVMe SSDs. Only the PM-based system XPGraph utilized Optane PMs. HDDs were exclusively used for backing up and storing all graph datasets before preprocessing of these graph systems.

Graph datasets. We use four real-world graphs Twitter [60], Friendster [12], UKdomain [61] and Yahoo Web [69], as well as two synthetic Kronecker graphs, i.e., Kron-29 and Kron-30, which are generated by graph500 generator [16], for our evaluation. These graphs are all unweighted direct graphs, and widely used in graph system evaluations. Table 1 lists the graph information. CSR Size indicates the size of storing graphs in CSR format for both in-graphs and out-graphs, and Chunk Size indicates the size of storing graphs in chunk structure for both in-graphs and out-graphs. Note that, CSR is the most compact storage format for static graphs. During graph processing, additional memory space is typically needed to accommodate the dynamic growth of adjacency lists as vertices update their neighbors. For example, Ligra costs 81.0GB memory space in total to store Yahoo dataset (only 70.5GB in CSR format), as it costs extra 10.5GB memory space to store the metadata. In contrast, ChunkGraph only costs 77.8GB memory as we store all metadata in the chunk structure. Additionally, to evaluate performance on weighted graph algorithm, we randomize the weight of each edge according to its source and destination vertex ID.

Graph algorithms. We evaluate ChunkGraph and its comparison systems on eight commonly used graph computing algorithms: BFS (traverses the connected sub-graphs of random roots), SSSP (finds the shortest path from a source vertex to all other vertices in weighted graphs), BC (computes the betweenness centrality of random roots), CC (finds all the connected components in the graphs), KCores (removes all the vertices that have a degree less than K, with K set to 10), Radii (computes the radius of the graphs), MIS (finds the maximal independent set in the graphs), and PageRank (computes the PageRank value of each vertex, running for ten iterations using the delta variation).

5.2 Comparison with Primary Baselines

Overall performance. We first evaluate the overall graph analytic performance of Blaze, Ligra-mmap and ChunkGraph. We maintain default parameter settings for all comparison systems. We utilize all available threads on one socket of our server, totaling 48 threads, and bind every thread to one core in the first socket by numactl utility, to avoid the NUMA effects. We also study the impact of the number of computing threads in §5.5. Figure 11

Table 1. Statistics of datasets.

Dataset	$\|V\|$	$\|E\|$	CSR Size	Chunk Size
Twitter (TT)	61.6M	1.5B	11.9GB	13.3GB
Friendster (FS)	68.3M	2.6B	20.3GB	21.1GB
UKdomain (UK)	101.7M	3.1B	26.2GB	27.2GB
YahooWeb (YW)	1.4B	6.6B	70.5GB	77.2GB
Kron29 (K29)	512M	8B	72GB	77.5GB
Kron30 (K30)	1B	16B	144GB	154.8GB

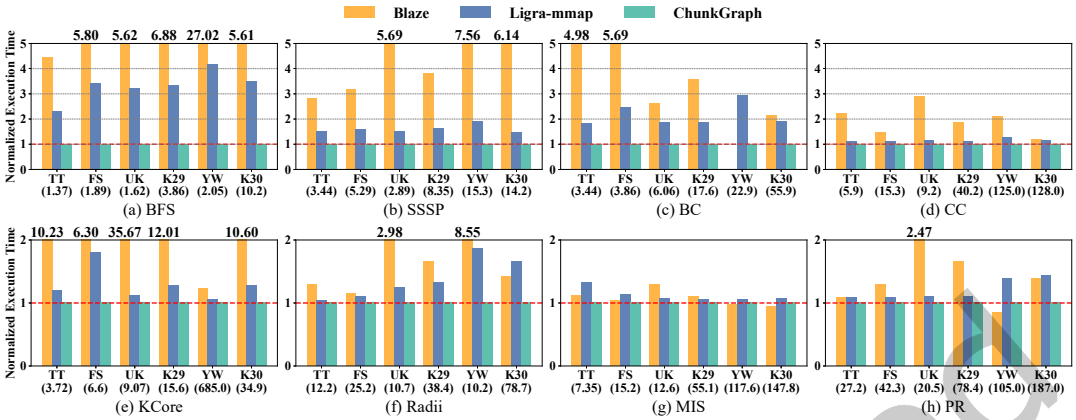


Fig. 11. Overall graph analytic performance, shown normalized to ChunkGraph, with the absolute execution time (in seconds) of ChunkGraph in parentheses below each dataset.

depicts the performance of eight algorithms. The x-axis represents different graph datasets, and the y-axis shows the performance normalized to ChunkGraph, with the absolute execution time (in seconds) of ChunkGraph shown in parentheses below the x-axis. It is important to note that Blaze fails to complete BC on Yahoo due to an out-of-memory (OOM) error. Firstly, we can see that the off-the-shelf memory-mapped-based Ligra and the storage-optimized Blaze exhibit respective advantages in specific graph processing scenarios. In scenarios where graph algorithms sparsely access a limited number of the edges, such as BFS, SSSP, and BC, memory-mapped-based Ligra excels by loading less data and incurring lower computational overhead. On the other hand, Blaze requires loading a larger subgraph, resulting in increased computational and subgraph synchronization overheads compared to in-memory graph systems. Conversely, for densely accessed graph algorithms, particularly those requiring significant edge data access like PageRank, Blaze's IO optimizations become crucial, showcasing its superior performance. For instance, when running the PageRank on Yahoo, Blaze's strengths become apparent, demonstrating its efficiency and outperforming memory-mapped systems.

In most cases, ChunkGraph outperforms Blaze and Ligra-mmap, particularly for sparsely accessed graph algorithms such as BFS, SSSP, and BC. Specifically, ChunkGraph achieves a speedup ranging from 2.14× to 23.57× when compared with Blaze, and a speedup ranging from 1.45× to 3.65× compared to Ligra-mmap. However, for densely accessed graph algorithms like PageRank, our performance improvement gets smaller, and sometimes even slightly slower than Ligra-mmap and Blaze. This is because PageRank sequentially and iteratively accesses all graph data, with low memory access locality, and thus receives minimal benefit from ChunkGraph's design. However, ChunkGraph still demonstrates performance advantages in some scenarios.

Our performance improvement can be attributed to three key factors. Firstly, both Blaze and Ligra-mmap exhibit low I/O utilization for graph accessing. Blaze is required to load an entire large subgraph even if only a small subset of vertices within it is needed, while Ligra-mmap encounters issues such as read amplification for small vertices, the vertex cut problem for medium-sized vertices, and access amplification due to poor neighborhood locality. In contrast, ChunkGraph leverages classified storage formats to appropriately allocate vertices of different sizes into chunks, along with compact chunk layout optimization, resulting in more efficient I/O of graph data. Secondly, external systems like Blaze partition all vertices into multiple subgraphs and execute graph computing in an iteration-based manner, leading to significant synchronization overhead between subgraphs and causing high computation overhead on the CPU side. On the other hand, ChunkGraph circumvents synchronization

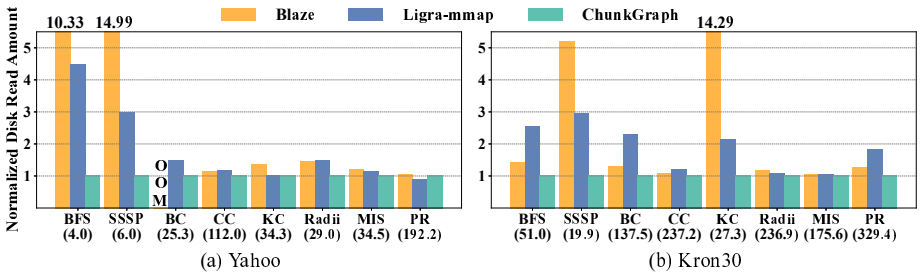


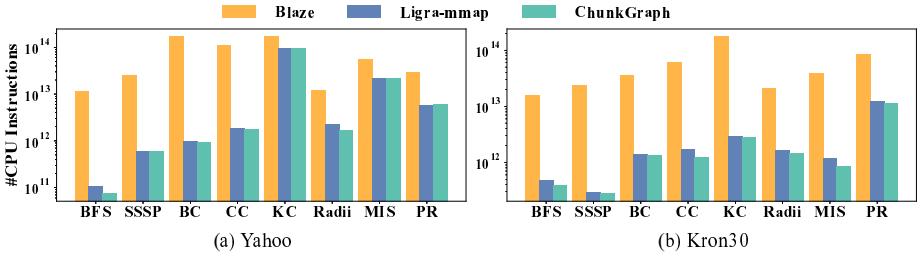
Fig. 12. Disk data read amount is shown normalized to ChunkGraph, with the absolute read amount (in GBs) of ChunkGraph displayed in parentheses below each dataset.

overhead by loading the graph in a unified memory space similar to an in-memory graph processing system. This approach reduces computation overhead and enhances overall performance. Thirdly, ChunkGraph adopts the latency-optimized chunk access mechanism to accelerate the I/O request processing by user-space asynchronous I/O execution, and improve the I/O efficiency by the hotness-aware chunk caching management. For future demonstration and analysis, we will also evaluate the I/O overhead and computation overhead during graph algorithm execution.

Takeaway #1. *ChunkGraph consistently outperforms Blaze and Ligra-mmap across most algorithms, achieving speedups of up to 23.57× and 3.65×, respectively. The performance gains are most pronounced in sparse-access algorithms like BFS, while ChunkGraph maintains competitive performance even in dense workloads like PageRank where access locality is inherently low.*

I/O overhead. We measure the I/O overhead by the disk read data amount. For a specific algorithm and dataset, the total number of accessed edges is the same for all systems, so a lower disk read data amount indicates higher I/O utilization. Figure 12 shows the comparison results on the largest two tested graph datasets, i.e., YahooWeb and Kron30. Similar results can also be observed in other test graphs. In most cases, ChunkGraph brings the least disk read data amount, and for other test cases, the read amount difference is limited to 12%. Compared with Blaze, for different graph algorithms, ChunkGraph reduces the disk read data amount by 14% to 95% on YahooWeb, and 5% to 93% on Kron30, with 73% on average. This is because Blaze uses the subgraph-based loading and computing model, causing low I/O utilization (refer to §2.1), while ChunkGraph uses the fine-grained chunk-based I/O model to reduce the loading of useless graph data. Compared with Ligra-mmap, ChunkGraph reduces the disk read data amount by 7% to 74% for different graph algorithms in two graphs, with 48% on average. This is because the PageCache manages all I/O data in 4KB chunks and does not consider data access patterns of graph applications (refer to §2.2). ChunkGraph manages graph data in vertex-centric chunks and uses chunk layout optimization to make full use of the loaded data to reduce the read times of the same pages.

Computation overhead. We quantify computation overhead based on the number of CPU instructions executed during graph algorithm execution, as recorded by the Linux *perf* tool. A higher number of CPU instructions typically indicates more complex computation. The comparison results in Figure 13 show the performance of Blaze, Ligra-mmap, and ChunkGraph on YahooWeb and Kron30 datasets. Across all algorithms and datasets, Blaze consistently incurs the highest computation cost, requiring up to 185.56× more CPU instructions than ChunkGraph. This is due to the additional computation overhead introduced by Blaze's synchronization between subgraphs. In contrast, ChunkGraph achieves comparable computation costs to Ligra-mmap, an in-memory graph system based implementation, with a difference limit of 39%. This is attributed to ChunkGraph's focus on optimizing graph data storing and loading, rather than altering the computation model of existing in-memory graph systems.



(a) Yahoo (b) Kron30

Fig. 13. Number of CPU instructions (shown in log scale).

Takeaway #2. *The performance advantage of ChunkGraph is driven by superior I/O efficiency and near-optimal computation costs. It reduces disk read volume by up to 95% compared to Blaze and 74% compared to Ligra-mmap. Simultaneously, ChunkGraph incurs computation overhead comparable to in-memory systems (within 39% of Ligra-mmap), whereas Blaze requires up to 185.56× more CPU instructions due to heavy subgraph synchronization.*

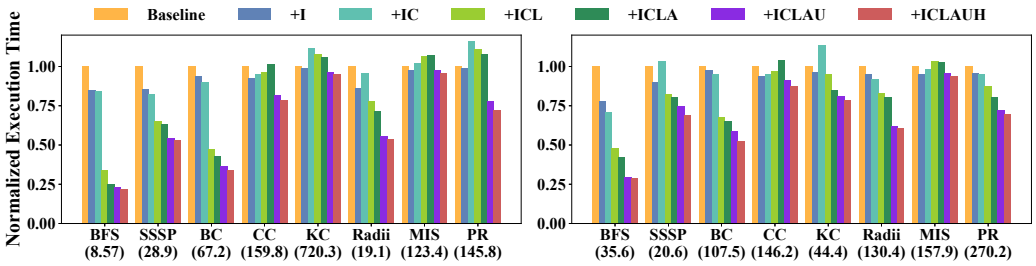
5.3 Discussion of Design Choices

Ablation study. In this subsection, we first evaluate the performance enhancements resulting from each technique introduced in §3 and §4. These techniques include in-**I**ndex mini vertex storing, **C**hunk based hierarchical storing, reorder and combination based chunk **L**ayout optimization, differentiated chunk **A**ccess optimization, **U**ser-space asynchronous I/O execution, and **H**otness-Aware Chunk Caching Management, abbreviated as I, C, L, A, U and H respectively. We use Ligra-mmap as the baseline implementation and implement five system versions by progressively adding these techniques: Baseline (Ligra-mmap), +I, +IC, +ICL, +ICLA, +ICLAU, +ICLAUH (ChunkGraph). Figure 14 depicts the graph analytic performance of these seven system versions on two large-scale datasets, Kron30 and Yahoo. The results are normalized to the Baseline, with the absolute execution time (in seconds) of the Baseline provided in parentheses below each dataset. Overall, these techniques demonstrate a step-by-step improvement in performance, collectively contributing to the overall benefits. In some rare cases of densely accessed graph algorithms, the chunk-based hierarchical vertex storing technique may lead to a performance drop due to the additional fragments in chunks, resulting in increased I/O overhead. However, these performance drops can be mitigated by other techniques. Next, we will delve into the effectiveness of each individual technique.
Takeaway #3. *The design choices in ChunkGraph provide cumulative performance improvements. While individual techniques like hierarchical storing may occasionally introduce fragments in dense workloads, subsequent optimizations, particularly chunk layout and hotness-aware caching effectively mitigate these drops, ensuring robust end-to-end gains across diverse graph topologies.*

Efficiency of in-index mini vertex storing. The in-index mini vertex storing technique involves encoding the neighbors of mini vertices into the in-memory vertex index, which was previously used to store the adjacent list pointers. This approach reduces the number of cache misses and completely eliminates the I/O cost associated with all mini vertices, since the neighbor information of these mini vertices can be retrieved in just one memory

Table 2. Average cache load misses on Yahoo dataset.

	BFS	SSSP	BC	CC	KC	Radii	MIS	PR
Baseline	420M	916M	3.68B	9.23B	19.5B	8.82B	12.4B	2.67B
+I	307M	818M	3.47B	8.95B	17.1B	4.57B	10.3B	2.53B



(a) Yahoo (b) Kron30

Fig. 14. Graph analytic performance for different design choices, shown normalized to Baseline, with the absolute execution time (in seconds) of Baseline in parentheses below each dataset.

Table 3. Number of accessed chunks on Yahoo dataset.

	BFS	SSSP	BC	CC	KC	Radii	MIS	PR
Baseline	2.94M	3.83M	8.29M	6.75M	6.28M	3.43M	4.90M	6.48M
+I	2.75M	3.53M	8.19M	6.56M	5.47M	2.74M	4.72M	6.42M
+IC	2.82M	3.52M	7.68M	6.68M	6.83M	2.91M	5.03M	6.70M
+ICL	2.78M	3.46M	7.59M	6.54M	6.33M	2.75M	4.58M	6.31M

access. To quantify the impact, we measured the average cache load misses of Baseline and *+I* on the Yahoo dataset. Table 2 illustrates that in-index mini vertex storing reduces the average cache misses up to 93%. Additionally, we evaluated the number of accessed chunks in Table 3, which revealed that in-index mini vertex storing reduces chunk accessing by 8% on average.

Efficiency of chunk based hierarchical storing. The chunk based hierarchical vertex storing technique organizes graph data into a series of aligned chunks to mitigate the vertex cut problem and minimize the number of cross-chunk vertex accesses. Our measurements indicated that 19.6% of chunk accesses were due to the vertex-cut problem in *+I*, whereas only 0.5% were observed in *+IC*. Furthermore, Table 3 illustrates that the chunk based hierarchical vertex storing approach reduces the number of chunk accesses by up to 18% when executing the Radii algorithm on the Yahoo dataset. However, it is important to note that this approach exhibits lower performance in specific test cases, such as PageRank on Kron30 and Yahoo. This performance discrepancy is attributed to the additional fragments caused by chunks, which increase the space required to store the entire graph compared to the CSR format, ultimately resulting in more chunk accesses. Fortunately, this performance drop can be mitigated through subsequent chunk layout optimization.

Efficiency of chunk layout optimization. The reorder and combination-based chunk layout optimization groups vertices with a higher probability of being accessed together into the same chunks and uses small vertices to fill up the fragments caused by the chunk-based design. This approach effectively reduces the number of intra-chunk fragments and minimizes I/O overhead. To quantify the impact, we measured the disk read data amount of *+IC* and *+ICL* on the Kron30 and Yahoo datasets. Figure 15 demonstrates that chunk layout optimization further reduces the disk read data amount by 3% to 52%. Additionally, Table 3 reveals that chunk layout optimization further reduces the number of chunk accesses by 1.2% to 9.8%.

Efficiency of fragment optimizations. We also count the number of the intra-chunk fragments on YahooWeb with naive vertex layout and our reordering and vertex-combination based fragmentation optimization. We show the results in Tables 4 and 5, where CN indicates the number of chunks of the layer, TS indicates the total size

Table 4. Fragmentation of Yahoo with naive vertex layout.

Layer	L0 (4KB)	L1 (32KB)	L2 (256KB)	L3 (2MB)	Total
CN	8943247	218152	25554	2561	9189514
TS	34.12GB	6.66GB	6.24GB	5.00GB	52.02GB
FS	2.04GB	1.23GB	1.21GB	1.66GB	6.14GB
FR	5.99%	18.47%	19.39%	33.20%	11.80%

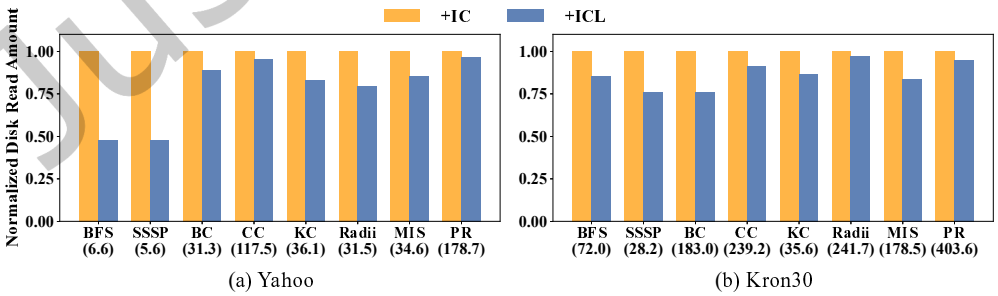
Table 5. Fragmentation of Yahoo with reordering and vertex-combination based optimization.

Layer	L0 (4KB)	L1 (32KB)	L2 (256KB)	L3 (2MB)	Total
CN	8686084	190711	24702	2213	8903710
TS	33.13GB	5.82GB	6.03GB	4.32GB	49.30GB
FS	1.05GB	0.39GB	1.01GB	0.98GB	3.43GB
FR	3.17%	6.70%	16.75%	22.68%	6.96%

of all chunks of the layer, FS indicates the total size of the fragment in the layer, and FR indicates the fragment ratio, i.e., the ratio of fragment size to the total size of the layer. These two tables show that compared to naive vertex layout, our reordering and vertex-combination based fragmentation optimization reduces the fragment ratio of YahooWeb from 11.80% to 6.96%, reducing the number of chunk accesses and improving I/O efficiency.

Takeaway #4. *ChunkGraph's storage-centric optimizations significantly enhance data locality: in-index mini vertex storing reduces average cache misses by up to 93%, and hierarchical chunking slashes vertex-cut accesses from 19.6% to 0.5%. Furthermore, reordering-based layout optimization reduces disk read volume by 3%–52% and lowers the fragment ratio from 11.80% to 6.96%.*

Efficiency of user-space asynchronous I/O. The user-space asynchronous I/O technique enables efficient I/O request handling and enhances the overall throughput of ChunkGraph. Furthermore, SPDK provides efficient and flexible cache management, improving buffer efficiency. To quantify the impact, we measured the average disk read bandwidth of *+ICLA* and *+ICLAU* on three algorithms. We conduct experiments across all graph datasets to demonstrate its effectiveness of scalability. Figure 16 depicted that user-space asynchronous I/O improves disk read bandwidth by 1.07× to 1.19×, with 1.12× on average. We also compare the execution time breakdown of ChunkGraph using kernel I/O versus SPDK-based I/O. The overall time cost is categorized into computation, software-level I/O overhead, hardware I/O latency, and others (e.g., OpenMP scheduling). The software-level I/O overhead comprises kernel interrupts/VFS for kernel I/O, SPDK polling for user-space I/O, and buffer/page



(a) Yahoo (b) Kron30

Fig. 15. Disk data read amount, normalized to *+IC*, and the numbers in parenthesis below each dataset are the absolute read amount (in GBs) of *+IC*.

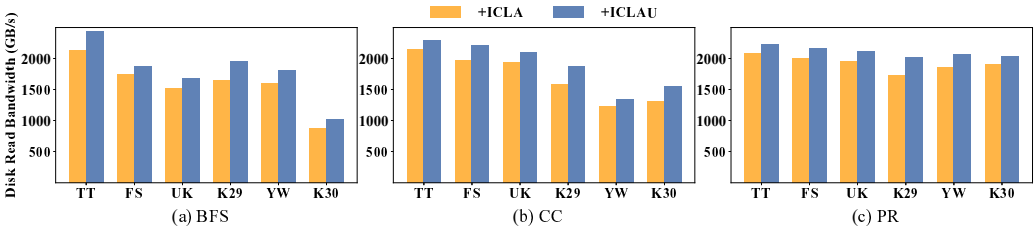


Fig. 16. Average disk read bandwidth.

management for both kernel and user-space I/O. As shown in Figure 17, software-level I/O overhead in standard kernel I/O consumes up to 52.6% of the total I/O duration, and 9.47% of the total runtime. By utilizing user-space polling and zero-copy hugepage buffers, SPDK significantly compresses these orchestration costs below 18.59% of the total I/O duration, and 6.35% of the total runtime. This efficiency allows the system to devote more CPU cycles to effective computation, yielding an average performance improvement of 1.17× across all tested algorithms.

Efficiency of chunk cache management. The hotness-aware chunk caching management strategy identifies and prioritizes retaining frequently accessed chunks, thereby improving buffer hit rates and enhancing overall graph access performance. To evaluate its effectiveness, we compare this strategy with alternative cache management policies across three algorithms and all datasets. The baselines include two widely adopted policies, FIFO and LRU, neither of which utilizes previous hotness information. Our approach incorporates two variants for different replacement strategies: a heap-based replacement policy that evicts the coldest chunk from the top of a min-heap, and a threshold-based policy that removes chunks with hotness values below a dynamically adjusted threshold θ, set to the average hotness across all chunks in our experiment.

Figure 18 presents overall performance and cache hit rates under different caching strategies. In these experiments, the hotness-aware metrics are configured with the default values ($\alpha = 0.5$ and $\beta_1 = \beta_2 = 1$) as defined in § 4.2. To eliminate the impact of compulsory misses, we conduct a uniform warm-up phase for all benchmark algorithms by executing them before measurement. The overhead introduced by this warm-up is negligible compared to the system's total runtime. Experimental results show that for locality-sensitive algorithms such as BFS, the hotness-aware strategy achieves superior performance, delivering a 1.08×-1.18× speedup over the baselines and a 1.09×-1.21× improvement in cache hit rate. For global-access algorithms such as CC and PR, the performance advantage diminishes; in some cases, its performance slightly lags behind the baselines, with a maximum degradation within 6%. In workloads where access hotness is less distinguishable, the reduced improvement in cache hit rate, coupled

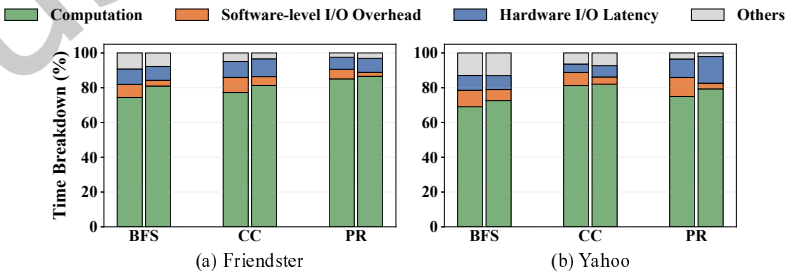


Fig. 17. Execution time breakdown of different graph algorithms on Friendster and Yahoo datasets. Left and right bars for each algorithm represent ChunkGraph using kernel-level I/O and SPDK-based user-space I/O, respectively.

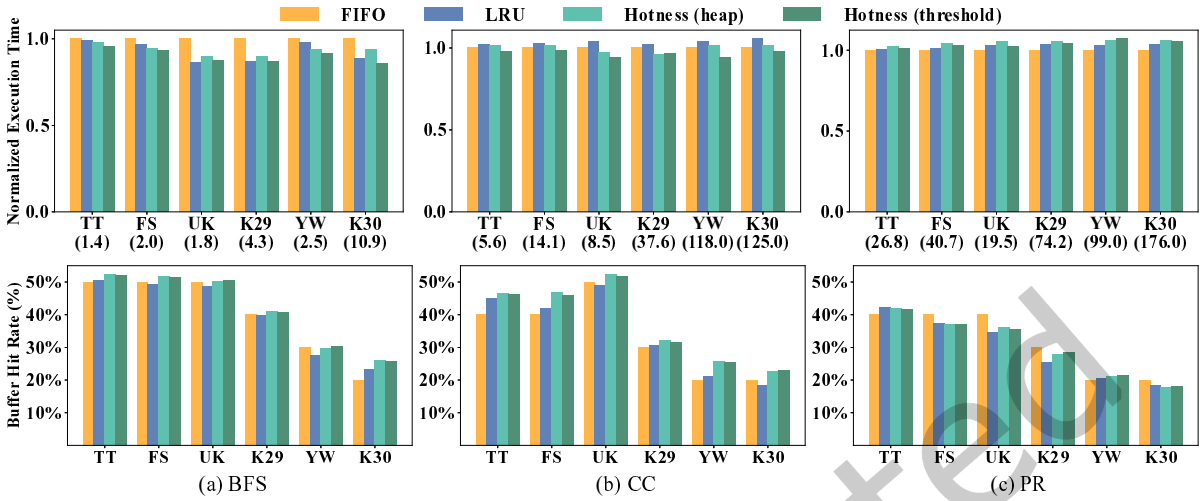


Fig. 18. Graph analytic performance and buffer hit rate for different chunk management strategies. The execution time is shown normalized to FIFO, with the absolute execution time (in seconds) of Baseline in parentheses below each dataset.

with the additional management overhead, limits the benefits of the hotness-aware approach. Between the two variants, the heap-based variant yields marginally higher hit rates in most cases, but incurs greater overhead due to frequent heap maintenance. To balance performance and efficiency, we dynamically select appropriate replacement strategy based on workload characteristics.

Takeaway #5. *Runtime mechanisms in ChunkGraph further convert storage efficiency into throughput: user-space asynchronous I/O improves disk bandwidth by up to 1.19× and yields 1.17× speedup by bypassing kernel overhead. Additionally, hotness-aware caching provides up to 1.21× higher hit rates and 1.18× speedups for locality-sensitive algorithms like BFS.*

5.4 Comparison with Other Systems

Comparison with other graph systems. We also conducted a comparison between ChunkGraph and three other graph processing systems: XPGraph, Graphene and MBFGraph. We specifically focus on the performance of the BFS, CC and PageRank algorithms, as they are implemented in all systems. The results are depicted in Figure 19. Compared with XPGraph, despite the fact that persistent memory offers higher performance compared to SSD,

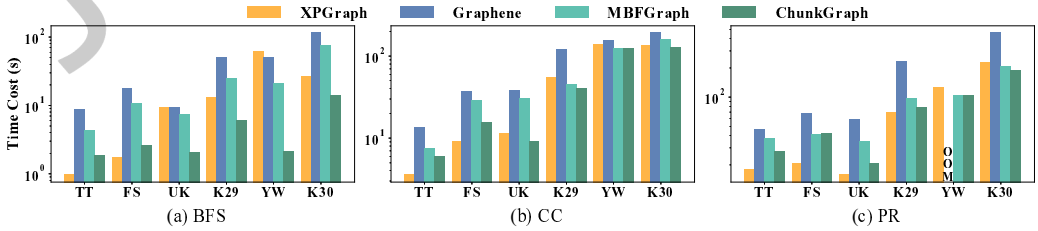


Fig. 19. Comparison with other graph systems (shown in log scale).

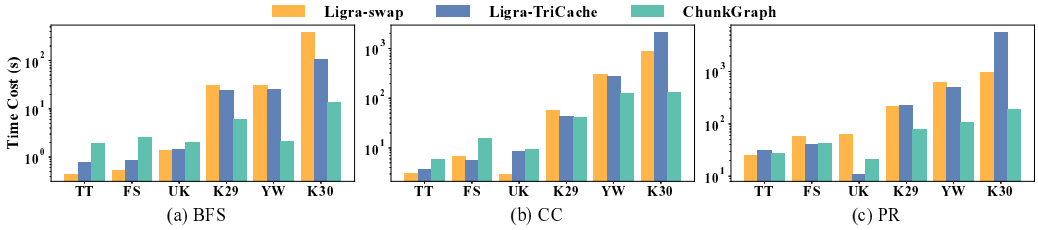


Fig. 20. Comparison with general cache systems (shown in log scale).

ChunkGraph still demonstrated faster performance compared to XPGraph in most cases, especially for large-scale graph datasets, with speedup up to 28.89×. This is due to the fact that XPGraph prioritizes optimizing graph update performance and primarily stores data on persistent memory, resulting in most graph queries requiring access to persistent memory when its buffer run out of space for storing large graph. On the other hand, ChunkGraph focuses on optimizing query performance for static graphs, aiming to reduce graph access to external storage. In the case of Graphene, it failed to successfully complete the PR algorithm on YW due to Out of Memory (OOM) fault. ChunkGraph consistently outperformed Graphene across all test cases. Specifically, ChunkGraph achieved speedups ranging from 4.74× to 23.89× for BFS, 1.24× to 4.14× for CC, and 1.61× to 2.97× for PageRank compared to Graphene. This is because the subgraph based iterative model used in Graphene leads to significant synchronization overhead between subgraphs and low I/O utilization. In contrast, ChunkGraph avoids synchronization overhead and uses classified storage formats for effective I/O. Compared with MBFGraph, ChunkGraph also demonstrates performance advantages in most scenarios, with an average overall speedup of 2.85×. MBFGraph is the state-of-the-art external graph system designed for dynamic edge updates, which leverages Bloom filters to identify the data blocks containing queried edges. While this design improves flexibility, it introduces additional data management complexity and access overhead. Compared to ChunkGraph with chunk layout optimization, MBFGraph incurs extra query overhead and more frequent block accesses, resulting in lower I/O efficiency.

Comparison with other general cache solutions. We also conducted an evaluation of ChunkGraph against Ligra, working with two other general cache solutions: Ligra-swap and Ligra-TriCache, utilizing Linux's default swap mechanism and the latest user-level memory-storage cache system TriCache [11] for memory expansion, respectively. Figure 20 illustrates evaluation results on three algorithms. Both solutions keep all graph data in memory and swap out data when memory is insufficient. In smaller graphs that fit entirely in memory, Ligra-swap and Ligra-TriCache run faster than ChunkGraph, as they do not need to read data from disk. However, as the graphs become larger, ChunkGraph outperforms Ligra-swap and Ligra-TriCache. For example, when running BFS

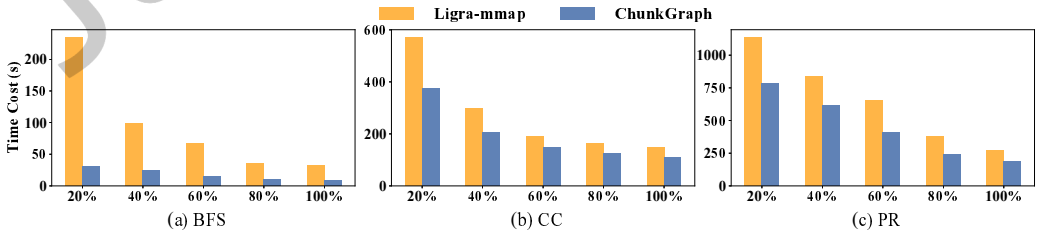


Fig. 21. Impact of memory budgets.

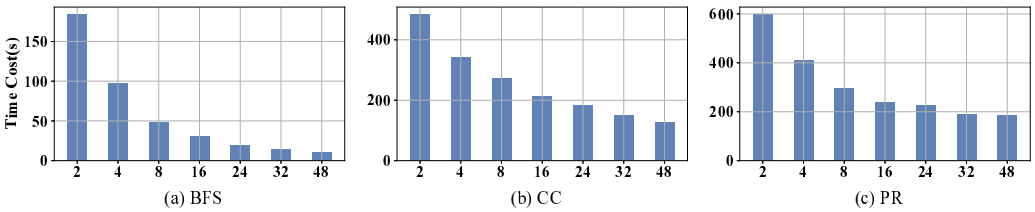


Fig. 22. Impact of the number of computing threads.

on the Kron30 dataset, ChunkGraph achieves a 28.13× and 7.68× speedup against Ligra-swap and Ligra-TriCache, respectively. The superior performance of ChunkGraph in larger graphs is attributed to the increased frequency of data swapping between memory and disk. While general cache solutions lack awareness of the graph data access pattern, they may swap out data that will be imminently accessed and the metadata for the graph data. This results in dirty data write back and additional I/O overhead.

Takeaway #6. *ChunkGraph consistently outperforms state-of-the-art graph systems (e.g., XPGraph, Graphene, MBFGraph) and general-purpose cache solutions (e.g., TriCache), with speedups reaching 28.89× and 7.68×, respectively. This superiority stems from ChunkGraph's ability to minimize I/O amplification and synchronization overhead, which typically plague both subgraph-based and page-centric caching models.*

5.5 Impact of System Configurations

Impact of memory budgets. ChunkGraph targets to support large-scale graph processing with limited memory resources. To evaluate the scalability of ChunkGraph across varying memory budgets, we run ChunkGraph on Kron30 dataset and restrict the available buffer size. Figure 21 compares the performance of BFS, CC and PageRank algorithms between Ligra-mmap and ChunkGraph, with the x-axis indicating memory budgets ranging from 20% to 100%, (capped at a maximum buffer size of 80GB). The results demonstrate that ChunkGraph consistently outperforms Ligra-mmap under identical memory constraints, and exhibits superior scalability as the memory budget increases. Notably, when the memory budget decreases from 100% to 20% for the BFS algorithm, Ligra-mmap's performance diminishes by 7.44×, whereas ChunkGraph only drops 3.43×. This advantage stems from ChunkGraph's hierarchical storage design and chunk layout optimization techniques to improve the I/O efficiency, thus reducing the impact of limited memory resources on performance. Moreover, ChunkGraph employs DPDK-based buffer implementation and hotness management strategy, further enhancing cache access efficiency. In conclusion, the system's minimum memory requirement is dictated by the vertex metadata (approximately 20 bytes per vertex); beyond this, users can configure the chunk cache based on system capacity. Our empirical evaluation suggests that a cache size of 20% - 40% of the total edge data is sufficient to achieve high performance on modern NVMe SSDs.

Impact of the number of computing threads. The number of computing threads impacts the efficiency of ChunkGraph. Figure 22 shows the scalability of BFS and KC algorithms on Kron30 dataset, with the x-axis representing the number of computing threads. We can see that the query performance of ChunkGraph scales linearly with the thread count. ChunkGraph gains performance improvement sufficiently as the thread count increases, particularly when the number of threads is not too small that the computing resources are not saturated with I/O resources.

Impact of the hierarchical layer. The implementation of hierarchical chunks for medium vertex storage can influence data access efficiency. We evaluated the effect of the number of hierarchical layer on overall performance on two datasets and four algorithms. The tested configurations consisted of 2-layer (4 KB and 2 MB), three-layer (4 KB, 64 KB, and 2 MB), and four-layer (4 KB, 32 KB, 256 KB, and 4 MB) implementations. Figure 23 reveals

notable performance gaps across different implementations, reaching up to 20.7% on the Yahoo dataset, and 17.3% on the Kron30 dataset. Furthermore, the performance benefits of the hierarchical strategy correlate with the vertex degree distribution of the dataset. E.g., the 3-layer implementation consistently achieves better performance on the Yahoo dataset, while the 4-layer for Kron30. Leveraging SPDK's flexible access granularity, ChunkGraph facilitates adaptive hierarchical implementation for different workloads.

Preprocessing cost. The graph data obtained from websites is typically in the edge list format, requiring conversion to the corresponding format for graph processing. For example, Ligra-mmap converts to the CSR format, while Blaze and Graphene convert to the partitioned CSR format, MBFGraph generate Bloom Fliters for graph data blocks. ChunkGraph is responsible for converting to the chunk format. We compared the preprocessing time cost of ChunkGraph, Graphene, Blaze, MBFGraph and Ligra-mmap, with Figure 24 showing Ligra-mmap as the fastest. Blaze and Graphene incur extra costs due to subgraph partitioning, whereas MBFGraph requires extra computation for MBF-construction algorithms, and ChunkGraph incurs additional computation cost for chunk layout optimization. Despite this, the performance gap is limited to 50% for ChunkGraph, making the preprocessing time acceptable as it only needs to be executed once for a graph dataset to support all graph applications. Meanwhile, for graphs of comparable scale, preprocessing overhead is largely insensitive to degree distribution, while skewed real-world graphs benefit more from reordering because hub-aware chunk grouping improves locality and I/O efficiency.

Takeaway #7. *ChunkGraph exhibits robust scalability across hardware and dataset constraints: it maintains 2.17× better performance than Ligra-mmap under extreme memory pressure (20% budget), scales linearly with thread count, and provides adaptive hierarchical layers for varying degree distributions. ChunkGraph's one-time preprocessing overhead remains within a 50% margin of the fastest baseline.*

6 Related Work

Large-scale graph processing. To support large-scale graphs that can not reside on a single machine's memory, distributed graph processing systems have been proposed to process graph computation on a cluster of machines [5, 6, 14, 15, 22, 31, 35, 38, 58, 59, 68, 73]. They divide the entire graph into multiple subgraphs and place them into different machines connected by network. However, these systems usually require efficient graph partitioning and caching strategies, low-cost inter-machine communication and synchronization mechanisms. Other efforts leverage large shared memory [17, 37, 39, 55], persist memory [18, 57, 64, 72], and GPUs [7, 23, 33, 52, 66] to accelerate large-scale graph processing. In contrast to those systems, ChunkGraph uses cost-effective NVMe SSDs to efficiently expand large-scale graph processing, providing a more resource-friendly alternative.

Out-of-core graph processing systems. Disk-resident single machine graph processing also receives a lot of attention by storing graphs on external storage devices. GraphChi [28] is the pioneering work, which splits all vertices into disjoint intervals and associates each interval with a subgraph, and uses a parallel sliding window

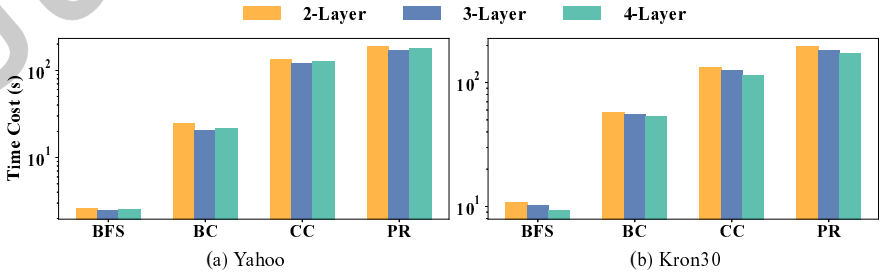


Fig. 23. Graph analytic performance for different number of hierarchical layer, with the execution time is shown in log scale.

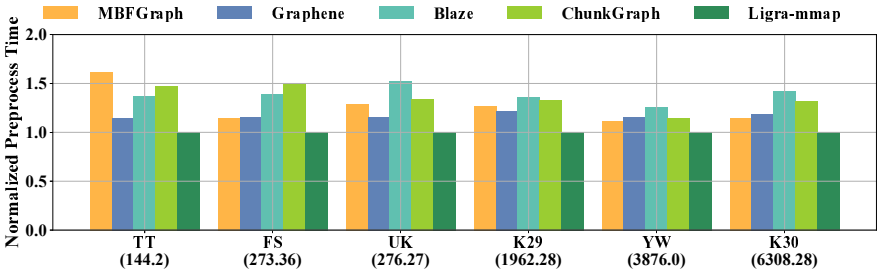


Fig. 24. Preprocessing performance of various systems, with the time cost normalized to that of Ligra-mmap. The absolute processing times (in seconds) of Ligra-mmap for each dataset are provided in parentheses below.

(PSW) to perform graph analysis, by loading the subgraphs into memory for computing in a round-robin order. Since then, a lot of works has followed this approach to optimize the performance of the out-of-core graph processing system from various dimensions [1, 28, 36, 54, 62, 63, 74]. However, they mainly focus on reducing random I/Os on HDDs or slow SSDs, which is no longer a severe problem on modern NVMe SSDs. Recent works adopt fine-grained graph loading to minimize I/O amount [8, 34] or different graph access pattern to maximize IO bandwidth [9, 19–21, 25, 32, 51, 70] on modern NVMe SSDs, but they still suffer from low I/O utilization and expensive computation overhead due to subgraph-based iterative loading and computing model. ChunkGraph adopts a chunk-based graph representation model for efficient and user-friendly large-scale graph processing on NVMe SSDs.

Memory-storage cache subsystems. Memory-storage cache systems (e.g., Linux OS's page cache) are often used to enable in-memory programs to process out-of-core datasets [10, 11, 48–50, 53, 56]. However, these general-purpose efforts are unaware of characteristics of graph access patterns, leading to a sub-optimal caching performance for graph processing. Instead, ChunkGraph proposes a novel reordering based data layout and differentiated access optimization to improve access efficiency of graph data.

7 Conclusion

This paper proposes ChunkGraph, an I/O efficient graph processing system for storing large-scale graph data on modern SSDs. ChunkGraph introduces a chunk-based graph representation model to enhance I/O efficiency during data access through classified and hierarchical vertex storage, and reordering and vertex combination based chunk layout strategy. Additionally, ChunkGraph develops a latency-optimized chunk access mechanism that accelerate I/O processing and improve cache efficiency, including user-space asynchronous I/O execution and hotness-aware chunk caching management. We implement a prototype system based on computation mode in modern graph processing framework, supporting graph algorithm to flexible implementation and efficient execution. Our experimental results show that ChunkGraph can run several times faster than in-memory graph system that works with existing general-purpose cache subsystems and existing external graph processing systems.

Acknowledgments

We would like to thank the Transactions on Storage editors and reviewers for all their valuable feedback and suggestions. We also thank all the ATC reviewers for their help in improving this work. The work of R. Wang was supported in part by the Young Scientists Fund of the National Natural Science Foundation of China (No.62502438), Ningbo Natural Science Foundation (No.2025J033), Ningbo Key Research and Development Program (No. 2025Z190), and Zhejiang University Education Foundation Qizhen Scholar Foundation. The work of S. He was

supported in part by the National Science Foundation of China (U25A20423), Major Projects of Zhejiang Province (LD24F020012), Hangzhou Chengxi Sci-Tech Innovation Corridor Special Development Fund (K20241957), Key Research Project of Zhejiang Lab (2025SSYS0005), and the Science and Technology Program of Zhejiang Province (2026SDXT012).

References

[1] Zhiyuan Ai, Mingxing Zhang, Yongwei Wu, Xuehai Qian, Kang Chen, and Weimin Zheng. 2017. Squeezing Out All the Value of Loaded Data: An Out-of-core Graph Processing System with Reduced Disk I/O. In *USENIX ATC*. 125–137.

[2] Dominic Asamoah, Evans Baidoo, and Stephen Opoku Oppong. 2017. Optimizing memory using knapsack algorithm. *International Journal of Modern Education and Computer Science* 9, 5 (2017), 34.

[3] Scott Beamer, Krste Asanović, and David Patterson. 2015. The GAP benchmark suite. *arXiv preprint arXiv:1508.03619* (2015).

[4] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *IEEE IISWC*. 56–65.

[5] Hongzhi Chen, Miao Liu, Yunjian Zhao, Xiao Yan, Da Yan, and James Cheng. 2018. G-Miner: an Efficient Task-Oriented Graph Mining System. In *ACM EuroSys*. 1–12.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, and Haibo Chen. 2015. Powerlyra: Differentiated Graph Computation and Partitioning on Skewed Graphs. In *ACM EuroSys*. 1–39.

[7] Pengjie Cui, Haotian Liu, Bo Tang, and Ye Yuan. 2024. CGgraph: An Ultra-Fast Graph Processing System on Modern Commodity CPU-GPU Co-processor. 17, 6 (2024), 1405–1417.

[8] Disa Mhembere Da Zheng, Randal Burns, Joshua Vogelstein, Carey E Priebe, and Alexander S Szalay. 2015. FlashGraph: Processing Billion-node Graphs on an Array of Commodity SSDs. In *USENIX FAST*. 45–58.

[9] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-Scale Graph Processing on Emerging Storage Devices. In *USENIX FAST*. 309–316.

[10] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. 2015. DI-MMAP–a Scalable Memory-Map Runtime for out-of-Core Data-Intensive Applications. *Cluster Computing* 18, 1 (mar 2015), 15–28. doi:10.1007/s10586-013-0309-0

[11] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. 2022. TriCache: A User-Transparent Block Cache Enabling High-Performance Out-of-Core Processing with In-Memory Programs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 395–411. https://www.usenix.org/conference/osdi22/presentation/feng

[12] Friendster Graph [n. d.]. Friendster. http://konect.uni-koblenz.de/networks/friendster.

[13] Gartner summit [n. d.]. Gartner's top 10 data and analytics technology trends for 2021. https://www.gartner.com/en/newsroom/press-releases/2021-03-16-gartner-identifies-top-10-data-\and-analytics-technologies-trends-for-2021.

[14] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX OSDI*. 17–30.

[15] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. GraphX: Graph Processing in a Distributed Dataflow Framework. In *USENIX OSDI*. 599–613.

[16] Graph500 Generator [n. d.]. Graph500. https://graph500.org/.

[17] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. 2012. Green-Marl: a DSL for Easy and Efficient Graph Analysis. *ACM SIGPLAN Notices* 47, 4 (2012), 349–362.

[18] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *ACM SC*. 1–13.

[19] Myung-Hwan Jang, Jeong-Min Park, Ikhyeon Jo, Duck-Ho Bae, and Sang-Wook Kim. 2023. RealGraph+: A High-Performance Single-Machine-Based Graph Engine that Utilizes IO Bandwidth Effectively. In *ACM WWW*. 276–279.

[20] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *IEEE ISCA*. 411–424.

[21] Soheil Khadirsharbiyani, Nima Elyasi, Armin Haj Aboutalebi, Chun-Yi Liu, Changho Choi, and Mahmut Taylan Kandemir. 2024. SmartGraph: A Framework for Graph Processing in Computational Storage. In *ACM SoCC*. 737–754.

[22] Arijit Khan, Gustavo Segovia, and Donald Kossmann. 2018. On Smart Query Routing: for Distributed Graph Querying with Decoupled Storage. In *USENIX ATC*. 401–412.

[23] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N Bhuyan. 2014. CuSha: vertex-centric graph processing on GPUs. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM.

[24] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. 2016. {NVMeDirect}: A User-space {I/O} Framework for Application-specific Optimization on {NVMe}{SSDs}. In *USENIX HotStorage*.

[25] Juno Kim and Steven Swanson. 2022. Blaze: fast graph processing on fast SSDs. In *ACM SC*. 1–15.

[26] Pradeep Kumar and H Howie Huang. 2019. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *USENIX FAST*. 249–263.
[27] Aapo Kyrola. 2013. Drunkardmob: Billions of Random Walks on Just a PC. In *ACM RecSys*.
[28] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. 2012. Graphchi: Large-scale Graph Computation on Just a PC. In *USENIX OSDI*. 31–46.
[29] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. 2019. {Pre-Select} Static Caching and Neighborhood Ordering for {BFS-like} Algorithms on Disk-based Graph Engines. In *USENIX ATC*. 459–474.
[30] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2019. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 447–461.
[31] Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, et al. 2022. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. *PVLDB* 15, 12 (2022), 3306–3318.
[32] Chun-Yi Liu, Wonil Choi, Soheil Khadirsharbiyani, and Mahmut Kandemir. 2023. MBFGraph: An SSD-based External Graph System for Evolving Graphs. In *ACM SC*. 1–13.
[33] Hang Liu and H Howie Huang. 2015. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.
[34] Hang Liu and H Howie Huang. 2017. Graphene: Fine-Grained IO Management for Graph Computing. In *USENIX FAST*. 285–300.
[35] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *PVLDB* 5, 8 (2012), 716–727.
[36] 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 *ACM EuroSys*. 527–543.
[37] Peter Macko, Virendra J Marathe, Daniel W Margo, and Margo I Seltzer. 2015. Llama: Efficient Graph Analytics Using Large Multiversioned Arrays. In *IEEE ICDE*. 363–374.
[38] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *ACM SIGMOD*. 135–146.
[39] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM SOSP*. 456–471.
[40] Online. accessed 30-June-2025. Buddy system form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Buddy_system.
[41] Online. accessed 30-June-2025. Cache replacement policies form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Cache_replacement_policies.
[42] Online. accessed 30-June-2025. io uring. https://man7.org/linux/man-pages/man7/io_uring.7.html.
[43] Online. accessed 30-June-2025. Memory-mapped file form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory-mapped_file.
[44] Online. accessed 30-June-2025. Memory paging form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory_paging.
[45] Online. accessed 30-June-2025. Page cache form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Page_cache.
[46] Online. accessed 30-June-2025. Slab allocation form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Slab_allocation.
[47] Emmanuel Ofori Oppong, Stephen Opoku Oppong, Dominic Asamoah, and Nuku Atta Kordzo Abiew. 2019. Meta-heuristics approach to knapsack problem in memory management. *Asian Journal of Research in Computer Science* 3, 2 (2019), 1–10.
[48] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. 2021. Memory-Mapped I/O on Steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 277–293. doi:10.1145/3447786.3456242
[49] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. 2021. Kreon: An Efficient Memory-Mapped Key-Value Store for Flash Storage. *ACM Trans. Storage* 17, 1, Article 7 (jan 2021), 32 pages. doi:10.1145/3418414
[50] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2020. Optimizing Memory-mapped I/O for Fast Storage Devices. In *USENIX ATC 20*. USENIX Association, 813–827. https://www.usenix.org/conference/atc20/presentation/papagiannis
[51] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *IEEE ICDE*. IEEE, 5644–5648.
[52] Jeong-Min Park, Myung-Hwan Jang, and Sang-Wook Kim. 2024. RealGraphGPUWeb: A Convenient and Efficient GPU-Based Graph Analysis Platform on the Web. In *ACM WWW*. 1011–1014.
[53] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. 2019. UMap: Enabling Application-driven Optimizations for Page Management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*. 71–78. doi:10.1109/MCHPC49590.2019.00017
[54] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. 2013. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *ACM SOSP*. 472–488.

[55] Julian Shun and Guy E Blelloch. 2013. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN*. 135–146.
[56] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. 2016. Efficient Memory-Mapped I/O on Fast Storage Device. *ACM Trans. Storage* 12, 4, Article 19 (may 2016), 27 pages. doi:10.1145/2846100
[57] Gan Sun, Jiang Zhou, Bo Li, Xiaoyan Gu, Weiping Wang, and Shuibing He. 2024. FTGraph: A Flexible Tree-Based Graph Store on Persistent Memory for Large-Scale Dynamic Graphs. In *IEEE CLUSTER*. 39–50.
[58] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. 2015. Arabesque: A System for Distributed Graph Mining. In *ACM SOSP*. 425–440.
[59] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, et al. 2024. Galaxybase: A high performance native distributed graph database for HTAP. *PVLDB* 17, 12 (2024), 3893–3905.
[60] Twitter Graph [n. d.]. Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.
[61] UKdomain Graph 2018. UK domain (2007) network dataset – KONECT. http://konect.cc/networks/dimacs10-uk-2007-05
[62] Keval Vora. 2019. LUMOS: Dependency-Driven Disk-based Graph Processing. In *USENIX ATC*. 429–442.
[63] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. 2016. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*. 507–522.
[64] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. 2022. XPGraph: XPline-Friendly Persistent Memory Graph Stores for Large-Scale Evolving Graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1308–1325. doi:10.1109/MICRO56248.2022.00091
[65] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. 2020. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *USENIX ATC*. 559–571.
[66] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. 2016. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN*.
[67] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering. In *ACM SIGMOD*. 1813–1828.
[68] Zhu Xiaowei, Feng Guanyu, Serafini Marco, Ma Xiaosong, Yu Jiping, Xie Lei, Aboulnaga Ashraf, and Chen Wenguang. 2020. LiveGraph: a Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *PVLDB* 13, 7 (2020), 1020–1034.
[69] Yahoo Graph [n. d.]. Yahoo Webscope. Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. http://webscope.sandbox.yahoo.com.
[70] Tsun-Yu Yang, Yizou Chen, Yuhong Liang, and Ming-Chang Yang. 2024. Seraph: Towards Scalable and Efficient Fully-external Graph Computation via On-demand Processing. In *USENIX FAST*. 373–387.
[71] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. 2017. SPDK: A development kit to build high performance storage applications. In *IEEE CloudCom*. IEEE, 154–161.
[72] Yunmo Zhang, Jiacheng Huang, Xizhe Yin, Junqiao Qiu, Hong Xu, and Chun Jason Xue. 2025. PIE: Enabling Fast and Scalable Incremental Evolving Graph Analytics on Persistent Memory. In *ACM ISC*.
[73] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. 2016. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX OSDI*. 301–316.
[74] Xiaowei Zhu, Wentao Han, and Wenguang Chen. 2015. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*. 375–386.

Received 28 July 2025; revised 3 April 2026; accepted 28 June 2026