Efficient Large Graph Processing with Chunk-Based Graph Representation Model

Rui Wang, *Zhejiang University and Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security;* Weixu Zong, Shuibing He, Xinyu Chen, Zhenxin Li, and Zheng Dang, *Zhejiang University*

https://www.usenix.org/conference/atc24/presentation/wang-rui

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله للعلوم والتقنية
King Abdullah University of Science and Technology

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Efficient Large Graph Processing with Chunk-Based Graph Representation Model

Rui Wang[1,2], Weixu Zong[1], Shuibing He[1,*], Xinyu Chen[1], Zhenxin Li[1], and Zheng Dang[1]

[1]Zhejiang University, Hangzhou, China
[2]Hangzhou High-Tech Zone (Binjiang) Institute of Blockchain and Data Security, Hangzhou, China
{rwang21, zorax, heshuibing, xy.chen, zhenxin, dangzheng}@zju.edu.cn

Abstract

Existing external graph processing systems face challenges in terms of low I/O efficiency, expensive computation overhead, and high graph algorithm development costs when running on emerging NVMe SSDs, due to their reliance on complex loading and computing models that aim to convert numerous random I/Os into a few sequential I/Os. While in-memory graph systems working with memory-storage cache systems like OS page cache or TriCache, offer a promising solution for large graph processing with fine-grained I/Os and easy algorithm programming, they often overlook the specific characteristics of graph applications, resulting in inefficient graph processing. To address these challenges, we introduce ChunkGraph, an I/O-efficient graph system designed for processing large-scale graphs on NVMe SSDs. ChunkGraph introduces a novel chunk-based graph representation model, featuring classified and hierarchical vertex storage, and efficient chunk layout optimization. Evaluations show that ChunkGraph can outperform existing external graph systems, as well as in-memory graph systems relying on general cache systems, running several times faster.

1 Introduction

Graph computing has gained attention recently [12]. Various in-memory graph processing systems, including Ligra [46], GraphOne [23], and GAP [3, 4], have been proposed for efficient graph analysis. However, as graphs grow larger, they often exceed the memory capacity of a single machine, posing challenges for large-scale graph analysis. Nevertheless, emerging storage devices like NVMe SSDs offer high-performance I/O at a low price [10], making them a cost-effective option for scaling large graph computing.

Several external graph processing systems have emerged in the past decade [1, 24, 25, 45, 52, 60], with the goal of improving the performance of large-scale graph analysis on earlier-generation external storage devices like HDDs and SATA SSDs. These devices experience performance degradation due to extensive random accesses [25]. To address this issue, the systems use a *subgraph-based iterative model* to divide the graph into subgraphs, store them on disk, and process them sequentially and iteratively. However, this approach results in high computation overhead and limited I/O utilization due to synchronization and coarse-grained I/O [54]. While these trade-offs are acceptable for slower storage devices like HDDs, they are no longer cost-effective for NVMe SSDs, which offer higher bandwidth and comparable performance for both random and sequential access [27]. Recently, various techniques have been proposed to improve graph processing performance on fast SSDs, including reducing the graph access granularity to minimize the I/O amount [7, 29] and scheduling the graph access pattern to maximize IO bandwidth utilization [8, 18, 19, 22].

Despite the advancements made in modern external graph systems, those utilizing the subgraph-based iterative model still face challenges. One issue is the low I/O utilization, as visiting a vertex's neighbors often requires loading an entire subgraph. 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 [22]. Another challenge is the additional computational overhead resulting from synchronization between subgraphs. Evaluation results show that running the BFS algorithm on the YahooWeb graph with Blaze necessitates 154 times more CPU instructions than the popular in-memory graph system, Ligra [46]. Additionally, utilizing these external graph systems leads to expensive algorithm development costs, as users are compelled to reimplement their graph algorithms based on the subgraph-centric computation model.

From another aspect, combining a concise in-memory graph system with a memory-storage cache system, such as the operating system (OS) page cache [38] or TriCache [10], presents an opportunity to facilitate large-scale graph processing with easy programming. For example, the OS's page cache can cache some SSD data in DRAM through swapping [37] or memory mapping [36] techniques, typically managing user data in a *page-centric caching model* with 4 KB page granularity. Users can use an in-memory graph system like Ligra, to conduct large graph processing by mapping graph data to SSD files, without any code modification on graph algorithms.

[*]Shuibing He is the corresponding author.

However, existing memory-storage cache systems are not tailored to efficiently support external graph processing, facing issues such as the mismatch between the 4 KB page granularity and the varying vertex sizes, resulting in low I/O utilization for small vertices with few neighbors, and additional metadata management costs for large vertices with numerous neighbors. Additionally, even for vertices that fit within a single page, they may still encounter the *vertex cut problem* where a vertex is divided across two adjacent pages, resulting in duplicated I/O costs. These limitations hinder the efficiency of existing memory-storage cache systems in supporting external graph processing.

To address the limitations in both the subgraph-based iterative model and the page-centric caching model, and enable efficient large graph processing on modern storage devices like NVMe SSDs, we propose a novel *chunk-based graph representation model*. Building upon this model, we develop an I/O efficient and user-friendly graph system named ChunkGraph, with the objective of enhancing the I/O efficiency of graph data without requiring changes to the computation mode of existing in-memory graph systems. In summary, our main contributions are as follows.

- We perform a thorough analysis of existing single-machine large graph processing solutions, encompassing out-of-core 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 novel *chunk-based graph representation model* and devise a *classified and hierarchical vertex storage* strategy. This strategy elaborately organizes different vertices into aligned chunks, thereby improving I/O efficiency for small vertices and reducing redundant metadata management for large vertices. Simultaneously, it effectively mitigates the vertex cut problem and eliminates unnecessary chunk I/Os during graph accesses.

- We introduce a *chunk layout optimization* method, which utilizes a reordering and combination-based approach to arrange vertices within each chunk based on the characteristics of the real-world graph structure. This optimization aims to enhance graph access locality and minimize fragment space within each chunk. Additionally, we adopt a differentiated chunk access optimization to accommodate various graph access patterns.

- We have implemented the prototype of ChunkGraph and conduct extensive experiments to demonstrate its efficiency. Experimental results show that ChunkGraph can run several times faster than existing external graph systems, as well as in-memory graph systems relying on general cache systems, attributed to the reduced I/O overhead and computation overhead.

The rest of this paper is organized as follows. In §2, we first introduce existing out-of-core graph processing systems and memory-storage cache subsystems used for large-scale graph processing in external storage, and analyze their limitations when processing large-scale graphs using modern storage devices. In §3, we present the design and implementation details of our proposed system, ChunkGraph. In §4, we show the evaluation results of ChunkGraph compared with existing external graph systems and in-memory graph systems relying on general cache systems. Finally, §5 reviews related work and §6 concludes.

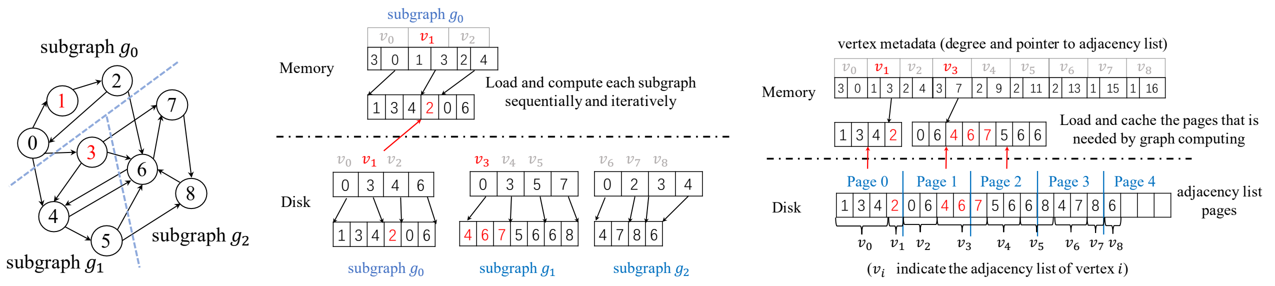
2 Background and Motivation

In this subsection, we introduce the subgraph-based iterative model used in existing out-of-core graph processing systems, and explore its limitations when used with modern storage devices. We also discuss the memory-storage cache subsystems and their limitations 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(b), the subgraph-based iterative loading and computing model is exemplified using the case graph from Figure 1(a). 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 [7] and Graphene [29] 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 [19], VPart [8], Blaze [22], and RealGraph+ [18] 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 al-



subgraph g_0
subgraph g_1
subgraph g_2
Memory
Disk
Load and compute each subgraph sequentially and iteratively
vertex metadata (degree and pointer to adjacency list)
Load and cache the pages that is needed by graph computing
Page 0
Page 1
Page 2
Page 3
Page 4
adjacency list pages
(v_i indicate the adjacency list of vertex i)

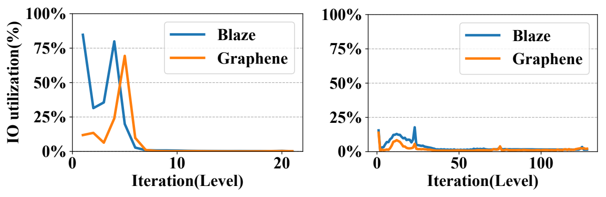
(a) Case graph (b) Subgraph based iterative model (c) Page-centric memory caching model

Figure 1: Graph accessing in existing out-of-core graph processing systems and memory-storage cache systems.

location. 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 that suffer from significant performance degradation due to random accesses [25], they become performance bottlenecks on emerging storage devices like NVMe SSDs, which have similar performance for random and sequential access [27].

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 [29] and Blaze [22] on the Friendster [11] and YahooWeb [58] 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+ [18] 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.



(a) Friendster (b) YahooWeb

Figure 2: I/O utilization during each BFS computation level.

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 §4.2). 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.

2.2 Memory-Storage Cache Subsystems

Page-centric memory caching scheme. Page cache [38] 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 [37] 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 [35], typically at the granularity of 4 KB pages. Figure 1(c) shows this page-centric memory caching model for the graph in Figure 1(a). We store the vertex metadata, including vertex degree and adjacency list pointer, in memory, while maintain 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 [10] 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, 13, 53]. 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 [58], 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.

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 1(c). 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 do not fully leverage 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 at enhancing the graph I/O and computation efficiency without changing the computation mode of existing in-memory graph systems.

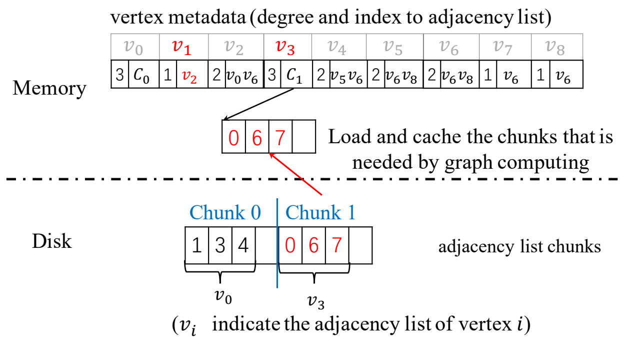
3 Design of ChunkGraph

In this section, we begin with the main idea of our proposed *chunk based graph representation model*, and provide an overview of ChunkGraph. Subsequently, we delve into the details of its key design techniques, including classified and hierarchical vertex storage and chunk layout optimization. Lastly, we introduce the prototype system implementation.

3.1 Main idea

Our aim is to enhance the I/O efficiency of graph data and enable effective and user-friendly large-scale graph processing on modern storage devices such as NVMe SSDs. To achieve this objective, we propose adopting 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. 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.

To further illustrate the above idea and analyze its benefits, we still consider the example graph in Figure 1(a), and show the data organization and I/O process of the chunk-centric



vertex metadata (degree and index to adjacency list)
v_0 v_1 v_2 v_3 v_4 v_5 v_6 v_7 v_8
3 C_0 1 v_2 2 $v_0 v_6$ 3 C_1 2 $v_5 v_6$ 2 $v_6 v_8$ 2 $v_6 v_8$ 1 v_6 1 v_6
Memory
0 6 7
Load and cache the chunks that is needed by graph computing
Disk
Chunk 0
Chunk 1
1 3 4
0 6 7
adjacency list chunks
v_0
v_3
(v_i indicate the adjacency list of vertex i)

Figure 3: Chunk based graph representation model.

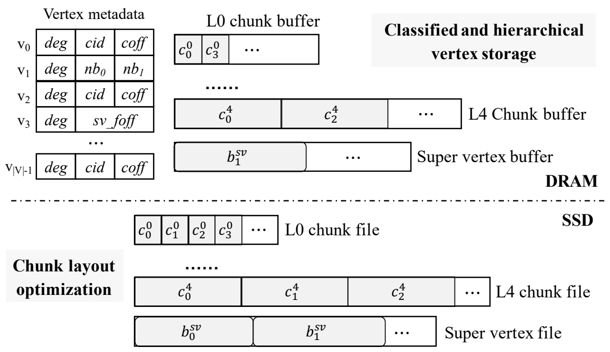


Figure 4: Overview of ChunkGraph.

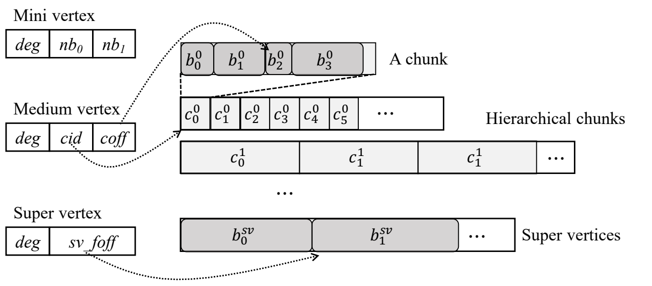


Figure 5: Differentiated data structures for different vertices.

model in Figure 3. 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 1(b)).

To realize the aforementioned chunk-based graph representation model and achieve efficient large-scale out-of-core graph processing on modern storage devices such as NVMe SSDs, we have developed ChunkGraph, a system focused on I/O effectiveness and user-friendliness. ChunkGraph primarily comprises two key components: classified and hierarchical vertex storage and chunk layout optimization. The overall design of ChunkGraph is depicted in Figure 4. In the following subsections, we present its design in detail.

3.2 Classified Hierarchical Vertex Storage

In this subsection, we introduce the classified and hierarchical storage formats used in ChunkGraph, as shown in Figure 5. To identify and index the vertex neighbors 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 [46]. Additionally, we store all vertices' metadata in memory, as this metadata is frequently accessed and typically fits in memory in most scenarios [7, 22, 29]. 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 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 [7,22,29,46]. 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 §4.3.

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 §4.3. 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.

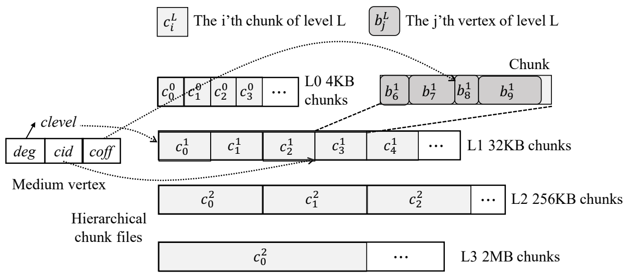


Figure 6: A four-layer implementation of hierarchical chunks based medium vertex storing.

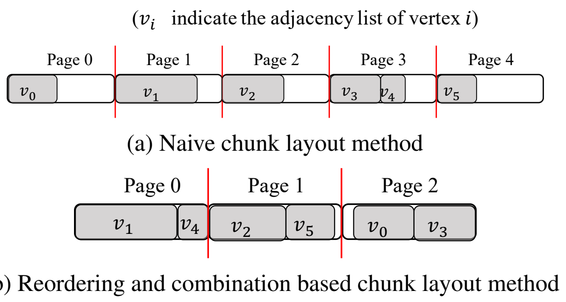

(v_i indicate the adjacency list of vertex i)
Page 0
Page 1
Page 2
Page 3
Page 4
(a) Naive chunk layout method
Page 0
Page 1
Page 2
(b) Reordering and combination based chunk layout method

Figure 7: Chunk layout optimization.

Hierarchical chunk implementation. Based on the chunk-based storing strategy, the key question is how to set the chunk size. As the smaller chunks can improve the I/O efficiency for small vertices, while the larger chunks can hold larger vertices inside a chunk. To accommodate different vertices with different 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 chunk sizes of each layer are 4KB, 32KB, 256KB and 2MB, respectively. We store a vertex to the 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 chunk level according to its 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 §4 has shown stable performance with these default settings, and users have the flexibility to adjust these parameters for optimal performance.

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.

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 (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.

3.3 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 [58] 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 §4.3 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 [26, 56], 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 is done as 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 [34], slab allocation [39] and knapsack problem based approach [2, 40], etc. However, in order to deal with various general situations, these strategies have complex logic and implementation, and the optimization effect in the case of graph computing is also limited. Fortunately, we can use 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, 13, 53]. In order to minimize intra-chunk fragments 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, effectively minimizing fragment space in 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 §4.3).

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.

3.4 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. The preprocessing cost is also evaluated in §4.5.

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 [46] 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.

4 Evaluation

4.1 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 [22] 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 semi-external graph system Graphene [29], the persistent memory (PM)-based graph storage system XPGraph [53], and Ligra working with the latest user-level transparent memory-storage system TriCache [10].

Note that in our experiments, most graph systems, including ChunkGraph, Ligra-swap, Ligra-mmap, Ligra-TriCache, Blaze, 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 [49], Friendster [11], UKdomain [50] and Yahoo Web [58], as well as two synthetic Kronecker graphs, i.e., Kron-29 and Kron-30, which are generated by graph500 generator [15], 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, more memory space is usually required to store the evolving adjacency lists for all vertices. 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

Table 1: Statistics of datasets.

Dataset	$\lvert V \rvert$	$\lvert E \rvert$	CSR Size	Chunk Size
Twitter (TT)	61.6M	1.5B	11.9GB	13.5GB
Friendster (FS)	68.3M	2.6B	20.3GB	21.2GB
UKdomain (UK)	101.7M	3.1B	26.2GB	27.5GB
YahooWeb (YW)	1.4B	6.6B	70.5GB	77.8GB
Kron29 (K29)	512M	8B	72GB	78.2GB
Kron30 (K30)	1B	16B	144GB	156.3GB

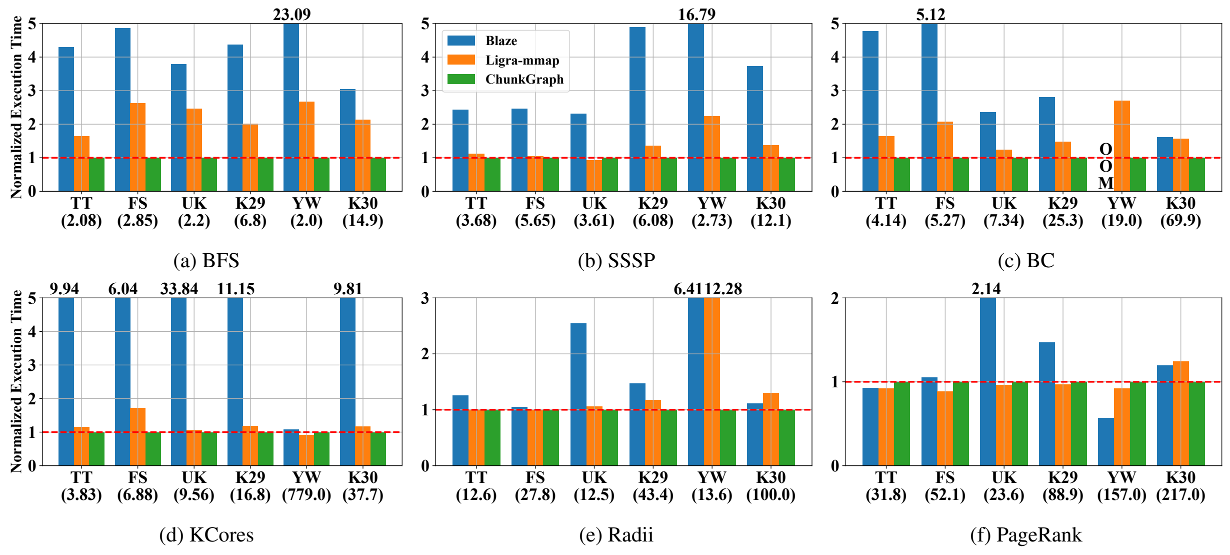


Figure 8: Overall graph analytic performance, shown normalized to ChunkGraph, with the absolute execution time (in seconds) of ChunkGraph in parentheses below each dataset.

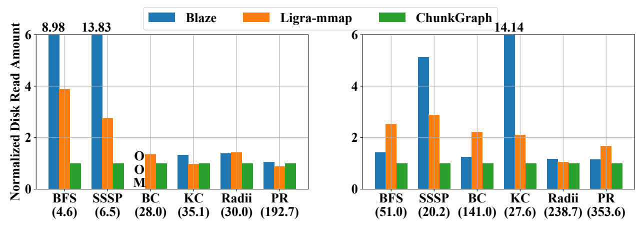
memory space to store the metadata. In contrast, ChunkGraph only costs 77.8GB memory as we store all metadata in the chunk structure. Additionally, to evaluate performance on weighted graph algorithm, we randomize the weight of each edge according to its source and destination vertex ID.

Graph algorithms. We evaluate ChunkGraph and its comparison systems on six 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), KCores (removes all the vertices that have a degree less than K, with K set to 10), Radii (computes the radius of the graphs), and PageRank (computes the PageRank value of each vertex, running for ten iterations using the delta variant algorithm).

4.2 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 §4.5. Figure 8 depicts the performance of six 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 fewer 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 1.62× to 23.09× when compared with Blaze, and a speedup ranging from 1.08× to 2.94× 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. Our performance improvement can be attributed to two 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



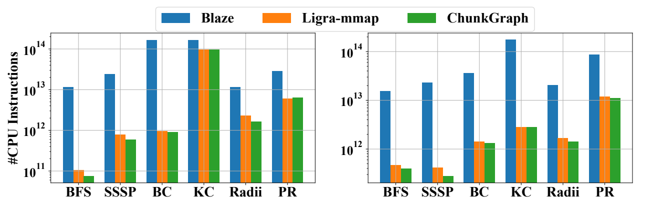
(a) Yahoo (b) Kron30

Figure 9: Disk data read amount is shown normalized to ChunkGraph, with the absolute read amount (in GBs) of ChunkGraph displayed in parentheses below each dataset.

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 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. For future demonstration and analysis, we will also evaluate the I/O overhead and computation overhead during graph algorithm execution.

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 9 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 1.06× to 13.83× on YahooWeb, and 1.16× to 14.14× on Kron30, with 4.68× 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 1.07× to 3.85× for different graph algorithms in two graphs, with 1.98× 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*



(a) Yahoo (b) Kron30

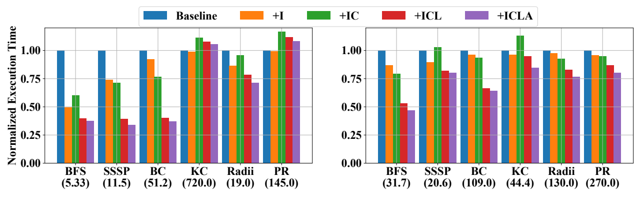
Figure 10: Number of CPU instructions (shown in log scale).

tool. A higher number of CPU instructions typically indicates more complex computation. The comparison results in Figure 10 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 185.01× 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 47%. 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.

4.3 Discussion of Design Choices

In this subsection, we evaluate the performance enhancements resulting from each technique introduced in §3. These techniques include in-**I**ndex mini vertex storing, **C**hunk based hierarchical storing, reorder and combination based chunk **L**ayout optimization, and differentiated chunk **A**ccess optimization, abbreviated as I, C, L, and A, 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 (ChunkGraph). Figure 11 depicts the graph analytic performance of these five 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 the reorder and combination-based chunk layout optimization. Next, we will delve into the effectiveness of each individual technique.

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



(a) Yahoo (b) Kron30

Figure 11: 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.

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 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 9% 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.

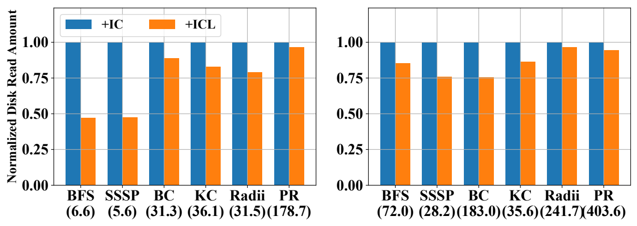
Efficiency of chunk layout optimization. The reorder and combination-based chunk layout optimization groups vertices

Table 2: Average cache load misses on Yahoo dataset.

	BFS	SSSP	BC	KC	Radii	PR
Baseline	420M	916M	3.68B	19.5B	8.82B	2.67B
+I	307M	818M	3.47B	17.1B	4.57B	2.53B

Table 3: Number of accessed chunks on Yahoo dataset.

	BFS	SSSP	BC	KC	Radii	PR
Baseline	2.94M	3.83M	8.29M	6.28M	3.43M	6.48M
+I	2.75M	3.53M	8.19M	5.47M	2.74M	6.42M
+IC	2.82M	3.52M	7.68M	6.83M	2.91M	6.70M
+ICL	2.78M	3.46M	7.59M	6.33M	2.75M	6.31M



(a) Yahoo (b) Kron30

Figure 12: Disk data read amount, normalized to *+IC*, and the numbers in parenthesis below each dataset are the absolute read amount (in GBs) of *+IC*.

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 12 demonstrates that chunk layout optimization further reduces the disk read data amount by 1.03× to 2.10×. Additionally, Table 3 reveals that chunk layout optimization further reduces the number of chunk accesses by 1.2% to 8.0%.

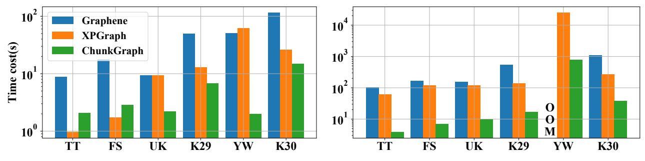
Efficiency of fragment optimizations. We also count the number of the intra-chunk fragments on YahooWeb with naive vertex layout and our reordering and vertex-combination based fragmentation optimization. We show the results in Tables 4 and 5, where CS indicates the chunk size of the layer, CN indicates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

Table 4: Fragmentation of Yahoo with naive vertex layout.

Layer	L0	L1	L2	L3	Total
CS	4KB	32KB	256KB	2MB	-
CN	8943247	218152	25554	2561	-
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%

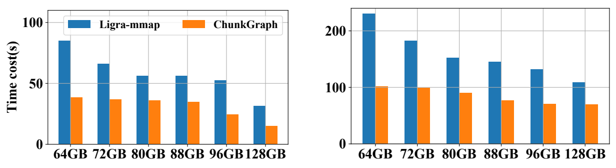
Table 5: Fragmentation of Yahoo with reordering and vertex-combination based fragmentation optimization.

Layer	L0	L1	L2	L3	Total
CS	4KB	32KB	256KB	2MB	-
CN	8686084	190711	24702	2213	-
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%



(a) BFS (b) KC

Figure 13: Comparison with other graph systems (shown in log scale).



(a) BFS (b) BC

Figure 15: Impact of memory budgets.

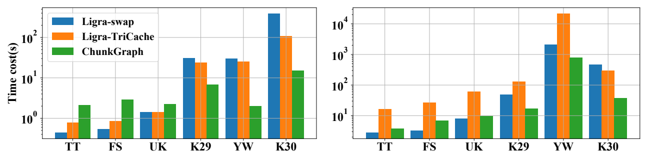
4.4 Comparison with Other Systems

Comparison with other graph systems. We also conducted a comparison between ChunkGraph and two other graph processing systems: Graphene and XPGraph. We specifically focus on the performance of the BFS and KC algorithms, as they are implemented in both Graphene and XPGraph. The results are depicted in Figure 14. In the case of Graphene, it failed to successfully complete the KC algorithm on YW due to Out of Memory (OOM) fault. Beyond that, ChunkGraph consistently outperformed Graphene across all test cases. Specifically, ChunkGraph achieved speedups ranging from 4.25× to 25.45× for BFS, and 15.89× to 32.37× for KC 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 XPGraph, despite the fact that persistent memory offers higher performance compared to SSD, ChunkGraph still demonstrated faster performance compared to XPGraph in most cases, with speedup up to 31.92×. 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. On the other hand, ChunkGraph focuses on optimizing query performance for static graphs, aiming to reducing graph access to external storage.

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 [10] for memory expansion, respectively. 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 on the Kron30 dataset, ChunkGraph achieves a 26.05× and 7.11× 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.

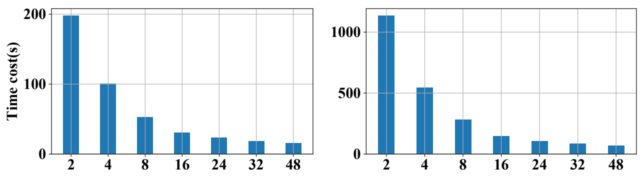
4.5 Impact of System Configurations

Impact of memory budgets. ChunkGraph targets to support large-scale graph processing with limited memory resources. To evaluate the scalability of ChunkGraph with different memory budgets, we run ChunkGraph on Kron30 dataset and use the cgroup mechanism to restrict the available memory size. Figure 15 shows the performance of BFS and BC algorithms running on Ligra-mmap and ChunkGraph, with the x-axis representing the memory budget in GBs. We can see that ChunkGraph consistently outperforms Ligra-mmap under equivalent memory budgets, and exhibits superior scalability as the memory budget increases. Notably, when the memory budget decreases from 128GB to 64GB for the BC algorithm, Ligra-mmap's performance diminishes by 2.11×, whereas ChunkGraph only drops 1.46×. This is because ChunkGraph adopts hierarchical storage and chunk layout optimization techniques to improve the I/O efficiency, thus reducing the impact of limited memory resources on performance.



(a) BFS (b) KC

Figure 14: Comparison with general cache systems (shown in log scale).



(a) BFS (b) BC

Figure 16: Impact of the number of computing threads.

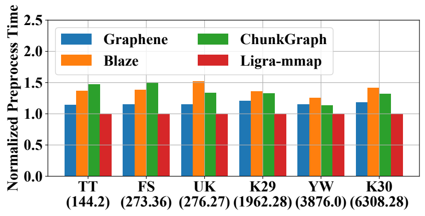


Figure 17: Preprocessing performance of various systems, with the time cost normalized to that of Ligra-mmap. The absolute processing times (in seconds) of Ligra-mmap for each dataset are provided in parentheses below.

Impact of the number of computing threads. The number of computing threads impacts the efficiency of ChunkGraph. Figure 16 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.

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. ChunkGraph is responsible for converting to the chunk format. We compared the preprocessing time cost of ChunkGraph, Graphene, Blaze, and Ligra-mmap, with Figure 17 showing Ligra-mmap as the fastest. Blaze and Graphene incur extra costs due to subgraph partitioning, and ChunkGraph incurs additional computation cost for chunk layout optimization. Despite this, the performance gap is limited to 50%, making the preprocessing time acceptable as it only needs to be executed once for a graph dataset to support all graph applications.

5 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, 13, 14, 20, 30, 32, 48, 57, 59]. 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 [16, 33, 46], persist memory [17, 53], and GPUs [21, 28, 55] 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 [25] is the pioneering work, which splits all vertices into disjoint intervals and associate each interval with a subgraph, and uses a parallel sliding window (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, 25, 31, 45, 51, 52, 60]. 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 [7, 29] or different graph access pattern to maximize IO bandwidth [8, 18, 19, 22] 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 [9, 10, 41–44, 47]. 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 proposed a novel reordering based data layout and differentiated access optimization to improve access efficiency of graph data.

6 Conclusion

In this paper, we proposed ChunkGraph, which is an I/O efficient external graph system for processing large-scale graphs, by developing a *chunk based graph representation model*, that includes classified and hierarchical vertex storage, and chunk layout optimization. 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 anonymous reviewers for their comments. This work is supported in part by the National Science Foundation of China (62172361), the Major Projects of Zhejiang Province (LD24F020012), the National Key Research and Development Program of China (2023YFB4502100, 2021ZD0110700), the Natural Science Foundation of Zhejiang Province (LHZSD24F020001), and the Science and Technology Program of Zhejiang Province (2024C01019).

A Artifact Appendix

A.1 Abstract

The artifact contains the prototype implementations of ChunkGraph, We also provide the implementations of the comparison baselines, i.e., Ligra-mmap and Blaze. All implementations are based on the Ligra framework. This artifact should enable others to reproduce a subset of our results and conduct their own studies.

A.2 Artifact check-list (meta-information)

- **Program:** Linux kernel of 5.4.0, ChunkGraph, Ligra, Blaze.
- **Compilation:** GCC 10.5.0 with OpenMP.
- **Data set:** Four real-world graphs Twitter, Friendster, UKdomain and Yahoo Web, as well as two synthetic Kronecker graphs, i.e., Kron-29 and Kron-30, which are generated by graph500 generator.
- **Run-time environment:** Ubuntu 20.04.6.
- **Hardware:** A server with two processors each with 24 physical cores with hyper-threading enabled (48 logical cores), with $8 \times$ 16GB (128 GB) DRAM and two 4 PCIe-attached Intel P5520 NVMe SSDs.
- **Execution:** Automated by shell scripts.
- **Metrics:** Graph analytic performance, I/O and computation overhead.
- **Output:** The key results would be recorded to the directory *results/*.
- **Experiments:** Graph analytic performance (Fig.8), I/O overhead for Yahoo/Kron30 datasets (Fig.9), and computation overhead for Yahoo/Kron30 datasets (Fig.10).
- **How much disk space required (approximately)?:** 5 TB.
- **How much time is needed to complete experiments (approximately)?:** 4 hours.
- **Publicly available?:** Yes.
- **Workflow framework used?:** No, but scripts are provided to automate the measurements.
- **Archived:** zenodo.org/doi/10.5281/zenodo.11181584

A.3 Description

A.3.1 How to access

The source code and scripts are host on Zenodo, `https://zenodo.org/doi/10.5281/zenodo.11181584`.

A.3.2 Hardware dependencies

This artifact runs on a server with two processors, each with 24 physical cores with hyper-threading enabled (48 logical cores). For memory, it equips with 8×16 GB (128 GB) DRAM. For storage, it equips with two 4 PCIe-attached Intel P5520 NVMe SSDs.

A.3.3 Software dependencies

This artifact runs on Ubuntu 20.04.6 LTS with Linux kernel of 5.4.0, and we use GCC 10.5.0 with -O3 optimization for evaluation. Other dependencies such as NUMA and Zlib libraries may also be necessary.

A.3.4 Datasets

We use four real-world graphs Twitter, Friendster, UKdomain and Yahoo Web, as well as two synthetic Kronecker graphs, i.e., Kron-29 and Kron-30, which are generated by graph500 generator, for our evaluation. Datasets download links:

- **Twitter:** `https://anlab-kaist.github.io/traces/WWW2010`
- **Friendster:** `http://konect.uni-koblenz.de/networks/friendster`
- **UKdomain:** `http://konect.cc/networks/dimacs10-uk-2007-05`
- **Yahoo Web:** `http://webscope.sandbox.yahoo.com`

Graph500 generator link and Kronecker graph generation commands:

- **Generator link:** `https://github.com/rwang067/graph500-3.0`
- **Make genetator:** cd graph500-3.0/src && make graph500_reference_bfs
- **Generate Kron29:** ./graph500_reference_bfs 29 16 kron29.txt
- **Generate Kron30:** ./graph500_reference_bfs 30 16 kron30.txt

A.4 Installation

Users need to download the source code and scripts from Zenodo to the server. The following is the directory structure of the source code, scripts, and instructions:

- **README.md:** This file contains a detailed step-by-step "Getting Started Instructions" guide, and "Detailed Instructions" for running the experiments.
- **src/:** This directory contains the source code of ChunkGraph and Ligra-mmap.
- **CSRGraph/:** This directory contains the source code of Chunk Layout Optimization of ChunkGraph.
- **apps/:** This directory contains contains the graph query algorithms for ChunkGraph and Ligra-mmap.

- **blaze/:** This directory contains the source code of Blaze.
- **scripts/:** This directory contains the scripts for running the experiments.
- **preprocess/:** This directory contains the scripts for converting the input data from text format to csr-based binary and chunk-format binary.

After downloading the source code and scripts, users need to compile ChunkGraph and prepare data sets. To evaluate the performance of comparison systems, users need to compile Blaze and Ligra-mmap.

A.5 Experiment workflow

The suggested workflow is organized in scripts/, which includes all the scripts for running the experiments. Users can run the experiments by 'bash scripts/run.sh'. The running progress and the expected completing time of each experiment would be printed to the file 'scripts/progress.txt' automatically. Note that you may have to change some arguments according to your environment, such as dataset path, taskset-cpu list, and the number of threads.

A.6 Evaluation and expected results

The evaluation results would be generated to the directory 'results/'. Users can reproduce the results in Figure 8, Figure 9, and Figure 10, which should roughly match the respective figures from the paper.

A.7 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

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

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

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

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

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

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

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

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

[9] Brian Essen, Henry Hsieh, Sasha Ames, Roger Pearce, and Maya Gokhale. Di-mmap–a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, mar 2015.

[10] Guanyu Feng, Huanqi Cao, Xiaowei Zhu, Bowen Yu, Yuanwei Wang, Zixuan Ma, Shengqi Chen, and Wenguang Chen. 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)*, pages 395–411, Carlsbad, CA, July 2022. USENIX Association.

[11] Friendster. http://konect.uni-koblenz.de/networks/friendster.

[12] 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.

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

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

[15] Graph500. https://graph500.org/.

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

[17] Abdullah Al Raqibul Islam and Dong Dai. Dgap: Efficient dynamic graph analysis on persistent memory. In *ACM SC*, pages 1–13, 2023.

[18] Myung-Hwan Jang, Jeong-Min Park, Ikhyeon Jo, Duck-Ho Bae, and Sang-Wook Kim. Realgraph+: A high-performance single-machine-based graph engine that utilizes io bandwidth effectively. In *ACM WWW*, pages 276–279, 2023.

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

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

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

[22] Juno Kim and Steven Swanson. Blaze: fast graph processing on fast ssds. In *ACM SC*, pages 1–15, 2022.

[23] Pradeep Kumar and H Howie Huang. GraphOne: A Data Store for Real-time Analytics on Evolving Graphs. In *USENIX FAST*, pages 249–263, 2019.

[24] Aapo Kyrola. Drunkardmob: Billions of Random Walks on Just a PC. In *ACM RecSys*, 2013.

[25] Aapo Kyrola, Guy E Blelloch, and Carlos Guestrin. Graphchi: Large-scale Graph Computation on Just a PC. In *USENIX OSDI*, pages 31–46, 2012.

[26] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam H Noh, and Jiwon Seo. {Pre-Select} static caching and neighborhood ordering for {BFS-like} algorithms on disk-based graph engines. In *USENIX ATC*, pages 459–474, 2019.

[27] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: the design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 447–461, 2019.

[28] Hang Liu and H Howie Huang. Enterprise: Breadth-first graph traversal on GPUs. In *Proceedings of the SC-International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2015.

[29] Hang Liu and H Howie Huang. Graphene: Fine-Grained IO Management for Graph Computing. In *USENIX FAST*, pages 285–300, 2017.

[30] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. Distributed GraphLab: a Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8):716–727, 2012.

[31] Steffen Maass, Changwoo Min, Sanidhya Kashyap, Woonhak Kang, Mohan Kumar, and Taesoo Kim. Mosaic: Processing a Trillion-edge Graph on a Single Machine. In *ACM EuroSys*, pages 527–543, 2017.

[32] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A System for Large-scale Graph Processing. In *ACM SIGMOD*, pages 135–146, 2010.

[33] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A Lightweight Infrastructure for Graph Analytics. In *ACM SOSP*, pages 456–471, 2013.

[34] Online. Buddy system form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Buddy_system, accessed 17-April-2023.

[35] Online. Cache replacement policies form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Cache_replacement_policies, accessed 17-April-2023.

[36] Online. Memory-mapped file form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory-mapped_file, accessed 17-April-2023.

[37] Online. Memory paging form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Memory_paging, accessed 17-April-2023.

[38] Online. Page cache form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Page_cache, accessed 17-April-2023.

[39] Online. Slab allocation form Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Slab_allocation, accessed 17-April-2023.

[40] Emmanuel Ofori Oppong, Stephen Opoku Oppong, Dominic Asamoah, and Nuku Atta Kordzo Abiew. Metaheuristics approach to knapsack problem in memory management. *Asian Journal of Research in Computer Science*, 3(2):1–10, 2019.

[41] Anastasios Papagiannis, Manolis Marazakis, and Angelos Bilas. Memory-mapped i/o on steroids. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, pages 277–293, New York, NY, USA, 2021. Association for Computing Machinery.

[42] Anastasios Papagiannis, Giorgos Saloustros, Giorgos Xanthakis, Giorgos Kalaentzis, Pilar Gonzalez-Ferez, and Angelos Bilas. Kreon: An efficient memory-mapped key-value store for flash storage. *ACM Trans. Storage*, 17(1), jan 2021.

[43] Anastasios Papagiannis, Giorgos Xanthakis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. Optimizing memory-mapped I/O for fast storage devices. In *USENIX ATC 20*, pages 813–827. USENIX Association, July 2020.

[44] Ivy Peng, Marty McFadden, Eric Green, Keita Iwabuchi, Kai Wu, Dong Li, Roger Pearce, and Maya Gokhale. Umap: Enabling application-driven optimizations for page management. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 71–78, 2019.

[45] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-stream: Edge-centric Graph Processing Using Streaming Partitions. In *ACM SOSP*, pages 472–488, 2013.

[46] Julian Shun and Guy E Blelloch. Ligra: a Lightweight Graph Processing Framework for Shared Memory. In *ACM SIGPLAN*, pages 135–146, 2013.

[47] Nae Young Song, Yongseok Son, Hyuck Han, and Heon Young Yeom. Efficient memory-mapped i/o on fast storage device. *ACM Trans. Storage*, 12(4), may 2016.

[48] Carlos HC Teixeira, Alexandre J Fonseca, Marco Serafini, Georgos Siganos, Mohammed J Zaki, and Ashraf Aboulnaga. Arabesque: A System for Distributed Graph Mining. In *ACM SOSP*, pages 425–440, 2015.

[49] Twitter. http://an.kaist.ac.kr/traces/WWW2010.html.

[50] UK domain (2007) network dataset – KONECT, January 2018.

[51] Keval Vora. LUMOS: Dependency-Driven Disk-based Graph Processing. In *USENIX ATC*, pages 429–442, 2019.

[52] Keval Vora, Guoqing (Harry) Xu, and Rajiv Gupta. Load the Edges You Need: A Generic I/O Optimization for Disk-based Graph Processing. In *USENIX ATC*, pages 507–522, 2016.

[53] Rui Wang, Shuibing He, Weixu Zong, Yongkun Li, and Yinlong Xu. Xpgraph: Xpline-friendly persistent memory graph stores for large-scale evolving graphs. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1308–1325, 2022.

[54] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. GraphWalker: An I/O-Efficient and Resource-Friendly Graph Analytic System for Fast and Scalable Random Walks. In *USENIX ATC*, pages 559–571, 2020.

[55] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN*, 2016.

[56] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *ACM SIGMOD*, pages 1813–1828, 2016.

[57] Zhu Xiaowei, Feng Guanyu, Serafini Marco, Ma Xiaosong, Yu Jiping, Xie Lei, Aboulnaga Ashraf, and Chen Wenguang. LiveGraph: a Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *PVLDB*, 13(7):1020–1034, 2020.

[58] Yahoo Webscope. Yahoo! AltaVista Web Page Hyperlink Connectivity Graph. http://webscope.sandbox.yahoo.com.

[59] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A Computation-Centric Distributed Graph Processing System. In *USENIX OSDI*, pages 301–316, 2016.

[60] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *USENIX ATC*, pages 375–386, 2015.