BG3: A Cost Effective and I/O Efficient Graph Database in ByteDance

Wei Zhang
ByteDance Inc.
zhangwei.95@bytedance.com

Cheng Chen*
ByteDance Inc.
chencheng.sg@bytedance.com

Qiange Wang
National University of Singapore
wang.qg@nus.edu.sg

Wei Wang
ByteDance Inc.
wangwei.tab@bytedance.com

Shijiao Yang
ByteDance Inc.
yangshijiao@bytedance.com

Bingyu Zhou
ByteDance Inc.
zhoubingyu.zby@bytedance.com

Huiming Zhu
ByteDance Inc.
zhuhuiming.sy@bytedance.com

Chao Chen
ByteDance Inc.
chenchao.chen@bytedance.com

Yongjun Zhao
ByteDance Inc.
zhaoyongjun.remake@bytedance.com

Yingqian Hu
ByteDance Inc.
huyingqian@bytedance.com

Miaomiao Cheng
ByteDance Inc.
chengmiaomiao.123@bytedance.com

Meng Li
ByteDance Inc.
limeng.1@bytedance.com

Hongfei Tan
ByteDance Inc.
tanhongfei@bytedance.com

Mengjin Liu
ByteDance Inc.
liumengjin@bytedance.com

Hexiang Lin
ByteDance Inc.
linhexiang@bytedance.com

Shuai Zhang
ByteDance Inc.
zhangshuai.root@bytedance.com

Lei Zhang
ByteDance Inc.
zhanglei.michael@bytedance.com

ABSTRACT

ByteDance's products, including TikTok, Douyin, and Toutiao, generate massive amounts of graph data every day. Previously, we developed ByteGraph, a distributed graph database that manages the large-scale graph data with varying performance requirements. BG3 is deployed on the computation and storage decoupled architecture, which allows for high performance in-memory execution and independent scaling of computation and storage layers. ByteGraph has demonstrated robust performance throughout its years of service in global-scale applications. However, as the business scale expands and applications evolve, the complexity and volume of graph analysis and processing have also increased. We observe that conventional database design faces issues with high operational costs when dealing with the large-scale graph workloads in social network management.

To address this issue, we develop BG3 (ByteGraph 3.0), a cost-effective and high performance distributed graph database which provides three critical components. Firstly, a cost-effective yet query-efficient graph storage engine based on the BW-tree-based memory indices and affordable cloud storage. Secondly, a workload-aware space reclamation mechanism, which enhances storage utilization and reduces write amplifications. Thirdly, a lightweight leader-follower synchronization mechanism ensuring strong consistency for scaling out real-time graph analysis. Experimental results demonstrate that BG3 addresses the limitations of ByteGraph, offering a cost-effective, efficient, and scalable solution for processing ByteDance's large-scale graphs.

CCS CONCEPTS

• **Information systems** → **Data management systems**; **Storage management**.

KEYWORDS

Graph Database; Cloud Storage; Graph Storage

ACM Reference Format:
Wei Zhang, Cheng Chen, Qiange Wang, Wei Wang, Shijiao Yang, Bingyu Zhou, Huiming Zhu, Chao Chen, Yongjun Zhao, Yingqian Hu, Miaomiao Cheng, Meng Li, Hongfei Tan, Mengjin Liu, Hexiang Lin, Shuai Zhang, and Lei Zhang. 2024. BG3: A Cost Effective and I/O Efficient Graph Database in ByteDance. In *Companion of the 2024 International Conference on Management of Data (SIGMOD-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3626246.3653373

*Corresponding Author

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06…$15.00
https://doi.org/10.1145/3626246.3653373

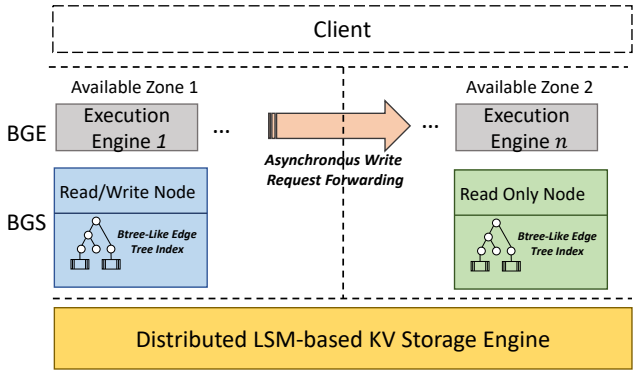

Figure 1: The Architecture of ByteGraph

1 INTRODUCTION

As the basis of ByteDance products, graph data exists ubiquitously in applications such as TikTok, Toutiao, and Douyin. The sizes of these graphs often reach of tens of billions of vertices and trillions of edges, still growing rapidly. Existing graph database products [1–3, 6, 8, 13, 15, 16, 18] in the market can hardly meet ByteGraph's performance and scalability requirements. To efficiently manage graph data at ByteDance, we developed ByteGraph [24], a distributed graph database that employs a two-tier architecture, decoupling the memory (cache) layer and persistent storage layers for separate scalability and optimization. Figure 1 shows an architecture overview. In the memory layer, ByteGraph utilizes a B-tree like structure to organize the adjacency lists of vertices, splitting each adjacency list into multiple pages. This approach enables parallel access to vertices with large neighborhoods and significantly reduces the I/O volume of queries such as edge searches. In the persistent storage layer, ByteGraph employs a distributed multi-copy key-value (KV) storage engine based on the Log Structured Merged (LSM) tree [29]. This design provides unified and fine-grained storage management for the edge-tree and vertex/edge properties with variable sizes on the graph. Over years of development and improvement, ByteGraph has become a crucial backbone for managing and processing large-scale graphs at ByteDance. The system is deployed across a massive cluster with 1K nodes spanning three data centers, comprising 1M CPU cores and 100PB of persistent storage. This setup is tasked with handling the voluminous graph management and processing tasks generated daily by ByteDance's products, including online analytical, transaction, and serving processing (OLAP, OLTP, and OLSP) [21]. Recent study [24] experimentally demonstrates its effectiveness and superiority over state-of-the-art graph database systems provided by cloud vendors [13, 15].

However, as ByteDance's user base grows and the business landscape evolves, the frequency of recording and analyzing user behavior information within applications (such as browsing, subscription, and favoriting) increases. To provide timely, accurate, and personalized services, large-scale graph analysis and learning algorithms are also being increasingly leveraged. The ongoing evolution has resulted in a steady increase in both the scale and diversity of graph workloads, posing challenges for the current design of ByteGraph in meeting ByteDance's growing needs for handling these workloads.

Firstly, inefficient graph access and high operational cost. ByteGraph uses LSM-based Key-Value (KV) storage engine as the persistent storage layer, which results in low graph access efficiency and high operating overhead. LSM-based KV indexes organize data into multiple logical layers, ordering data within each layer rather than globally. While this design transforms random disk I/O into sequential I/O, thereby enhancing write throughput [7], the weak read performance of LSM has become a primary performance bottleneck for most ByteDance applications that necessitate frequent graph accesses. To tackle these issues without modifying the system, it becomes necessary to deploy an increasing number of CPU and memory resources in the persistent storage layer to enhance the efficiency of query result merging, layer compaction, and in-memory result caching. However, this approach leads to a significant rise in operational costs.

Secondly, Workload-unaware space management. Following modern cloud database systems [5, 10, 11, 14, 34], ByteGraph's storage layer stores data on log-structured and append-only storages and utilizes out-of-place data update approaches (reading data page, modifying, and appending it to the end of storage, and deleting the old data) [14, 27, 34]. This design enables efficient data updates by converting random data updates into sequential data writing; However, it necessitates an efficient garbage collection mechanism to reclaim the space of deleted data on the append-only log storages. Currently, ByteGraph adheres to conventional garbage collection methods, which depend on generic metrics based on the Fragmentation rate (ratio of invalid data). However such an approach overlooks the spatial-temporal distribution of evolving graphs. Additionally, access pattern specific to applications, such as user-video interactions and evolving video hotness, also offer opportunities for optimizing space reclamation. Traditional designs fail to effectively optimize for these factors, resulting in wasted space and write amplification.

Thirdly, lack of support for scaling real-time graph analytics. With the rapid development of ByteDance's e-commerce and recommendation services, large-scale graph analysis and learning have been widely adopted for e-commerce risk control and content recommendation. These computation-intensive algorithms, such as Pattern Matching, demand low latency for real-time applications and typically require deployment on multiple read-only (RO) nodes to scale read-throughput. This necessitates an efficient leader-follower synchronization mechanism to ensure that RO nodes promptly receive the latest data from the read-write (RW) node, guaranteeing accuracy and real-time performance. However, the existing leader-follower synchronization solution deployed in ByteGraph, which asynchronously redirects write requests to RO nodes and replays them, can only achieve eventual consistency. The incapacity to synchronize the RW and RO nodes within a bounded time interval poses significant challenges in scaling real-time graph analytics.

To address these issues, we present BG3, a new generation of distributed graph storage engine for efficient data persistence and access in ByteGraph, which employs three critical components.

- Firstly, we develop a cost-effective storage engine leveraging affordable shared storage solutions and graph-access-optimized BW-tree indexes [23]. Instead of relying on the well-established LSM-based KV storage, our approach utilizes cheap cloud storage to minimize costs and introduces BW-tree-based forest indices for managing the adjacency lists of graphs. This design, in contrast to conventional distributed KV-based graph storage engines, significantly reduces the resource overhead at the storage layer and improves read performance via read-optimized graph indices.
- Secondly, we introduce a workload-aware garbage collection method that incorporates both the traditional fragmentation rate and the specific characteristics of workloads, including the "hotness" of evolving graphs and the data access patterns of real-world applications. This innovative approach guides space reclamation in graph management scenarios, optimizing disk utilization and concurrently reducing storage I/O operations across a diverse range of applications.
- Thirdly, we develop an I/O-efficient leader-follower synchronization mechanism tailored for our BW-tree optimized graph storage. This method employs a Write-Ahead Log (WAL) for synchronizing data between read-write (RW) and read-only (RO) nodes, alongside in-place parallel enhancements designed to enhance I/O efficiency, offering high-performance, strong consistency guarantees, meeting the demands of scaling real-time graph analytics effectively.

We demonstrate the cost-effectiveness, performance, and scalability of BG3 with both publicly available and real production workloads.

2 BYTEGRAPH

In this section, we first briefly introduce the architecture of ByteGraph [24] and then present the basic design of the graph storage engine.

2.1 Architecture of ByteGraph

As illustrated in Figure 1, a typical ByteGraph cluster comprises three layers: an execution layer (BGE) responsible for handling computation-intensive operations from the application (e.g., sorting, aggregation), a cache layer in memory (BGS) providing graph-native data management utilizing a B-tree-based graph index, and a graph data persistent layer based on distributed LSM-based KV storage engine, which is responsible for persisting all data (in KV pairs) generated by BGS. Each layer can be scaled out independently.

2.2 Graph Data Persistence in ByteGraph

ByteGraph adopts the property graph model, where both vertices and edges have associated types and properties, and the data associated with vertices and edges is managed separately. Each vertex, along with its properties, is stored as a single key-value (KV) pair. The key is encoded using a unique ID and type, and the value comprises a list of associated properties. To efficiently execute graph traversal queries, edges are organized into adjacency lists and divided into multiple groups based on the edge type. To further reduce the overhead of accessing super-vertices, each adjacency list of a vertex is split into multiple pages and indexed through a B-tree like edge tree structure. Similar to B-tree indices, an edge tree consists of multiple pages indicated by Root Node, Meta Node, and Edge Node. Root Node and Meta Node index the Edge Nodes, and Edge Nodes store the physical data. Each node in the edge tree occupies the size of a page and is stored as a KV pair.

2.3 High Availability

ByteGraph achieves high availability by deploying the memory layer and execution layer across multiple Availability Zones (AZs). ByteGraph's storage layer is based on a shared storage architecture that enables efficient access across availability zones. The nodes in the memory layer are divided into read-write (RW) nodes and read-only (RO) nodes according to the AZ it belongs to. To synchronize the memory state between RW and RO nodes, ByteGraph implements leader-follower synchronization by forwarding write requests at the execution engine layer. Write requests from the execution layer will be asynchronously forwarded to all RO nodes and replayed, thereby updating the states in RO. However, since this process is performed asynchronously, ByteGraph can only achieve eventual consistency, which limits its scalability for real-time graph analysis.

2.4 Limited Read Performance and High Operating Costs

The LSM-tree-based KV storage engine in ByteGraph is primarily designed for write-intensive workloads, sacrificing read performance to enhance write throughput. However, in ByteDance applications, graph databases are required to support workloads demanding high QPS for both read and write operations. Taking the feature recommendation function, which is frequently invoked in ByteDance's products for context pushing, as an example, user behaviors are massively generated and written to the database as users browse information and are frequently accessed by the recommendation module for real-time precise recommendations. ByteGraph's storage system is required to support high write throughput for both write and read operations. However, achieving this within ByteGraph presents several challenges. First, the underlying LSM-based storage engine exhibits high read overhead of result combination due to its multi-layer data maintenance mechanism [7]. Reading a data piece necessitates massive I/O to scan through multiple layers and extensive CPU resources to merge the results. The read operations and background data compaction operations demands a large number of CPUs. Second, the LSM tree-based KV storage, being decoupled from and unaware of the B-tree-based graph indexes in the memory layer, leads to an elongated query path. Each read operation must traverse the memory index, proxy, and LSM index before accessing the physical storage. These issues substantially increase the read overhead. A potential solution could be to allocate more CPU to enhance read combination QPS and more memory resource to improve cache hit rates, thereby shortening the query paths. However, these approaches could significantly increase operating costs.

2.5 Space Reclamation

Modern cloud database systems typically rely on log-structured append-only storage and employ an Out-Of-Place (OOP) update

approach to manage data. In this approach, the updated data is appended to the tail of the storage rather than written back to its original position, and the space of the original data is invalidated. The OOP update converts random in-place updates into sequential write operations, thereby improving write performance. However, the OOP update mechanism creates many small fragmented slices that cannot be used for storing new data. Therefore, an efficient data reclaiming mechanism is required to optimize storage utilization. Currently, space reclamation operations on traditional general-purpose storage engines have been extensively studied [14, 22, 27, 34]. Most of them rely on general metrics such as fragement rate and data hotness. In ByteDance, we observe that ByteDance's graph and application present unique opportunities due to their distinct graph data access patterns. The graph data exhibits a power-law distribution, and certain applications (such as the popularity of videos and the user preferences for videos changing over time) show spatial and temporal locality, offering opportunities for optimizing write amplification. However, due to the inability to perceive graph topology and temporal information, existing KV-based storage engines with general-purpose space reclamation mechanisms can hardly optimize these specific issues.

2.6 Leader-Follower Synchronization

ByteGraph's persistence layer is built on a distributed Key-Value (KV) engine operating on shared storage. Data synchronization across multiple instances is achieved by forwarding write requests, a method that ensures only eventual consistency. With ByteDance's business diversifying into domains such as e-commerce and risk assessment, the widespread adoption of large-scale real-time graph analysis and learning becomes crucial to delivering precise and timely services. However, the weak consistency guarantee provided by ByteGraph poses significant challenges in scaling out these tasks to multiple RO nodes to optimize performance. For example, in anti-money laundering applications, the loop detection is a critical task that benefits from the MPP (Massively Parallel Processing) on multiple machines. However, this necessitates that multiple RO nodes access the latest data promptly. In an eventually consistent system, the absence of time guarantees in data synchronization can lead to failures in accessing the latest graph snapshot, forcing a fallback to serial execution, which in turn increases the service latency. Currently, extensive research has been conducted on leader-follower synchronization solutions in distributed database systems, leading to the development of mature log-based leader-follower synchronization methods that support strong consistency in industry-leading database systems [9, 14, 34]. However, ByteGraph incorporates distinctive designs for constructing a graph storage engine on shared storage, as detailed in Section 3. Consequently, conventional solutions are not directly applicable to ByteGraph.

3 DESIGN AND IMPLEMENTATION

3.1 An Overview of BG3

The architecture of BG3 is depicted in Figure 2. The execution engine is responsible for converting query language into specific execution plans and handles computation-intensive operations such as sorting and aggregation. Unlike the previous generation of ByteGraph [24], we built the storage layer beneath the execution layer based on low-cost, append-only cloud storage. To enhance system scalability, we adopted a write-once, read-many architecture, where new data is first written to the 'read/write (RW) node' and then synchronized to 'read only (RO) nodes' to provide graph services. In our system, it's feasible to deploy multiple RW nodes, as we can distribute write requests across distinct RW nodes using hashing. To meet the stringent requirements for data freshness in applications like fraud detection, we proposed a 'I/O efficient synchronization' mechanism based on shared-storage to efficiently synchronize read and write node data, offering high-performance strong consistency guarantees (details discussed in section 4.5). We introduced 'space optimized Bw-tree forest' (details in section 3.2.1) and 'read optimized Bw-tree' (details in section 3.2.2) technologies to address the shortcomings of traditional Bw-trees in ByteDance's ultra-high concurrent read-write scenarios. Simultaneously, to design a more efficient space management method at the lower level that is aware of the upper-level graph workload characteristics, we proposed 'workload-aware space reclamation' to reduce the overhead of garbage collection (details in section 3.3).

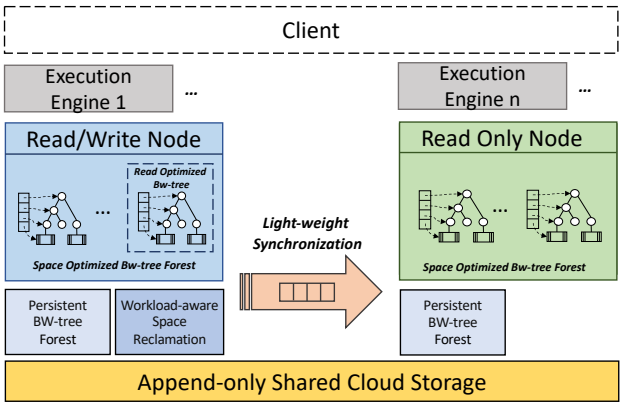

Figure 2: The architecture of BG3

3.2 Bw-tree Liked Graph Storage Engine

3.2.1 Space Optimized Bw-tree Forest. Let's consider the scenario where users like videos in Douyin as an example. In this scenario, each user and video are stored as a node in the graph within the bytegraph. Whenever a user likes a video, we establish an edge between the user and the video and store this edge as well as the edge features (for instance, the time when the like was clicked) into ByteGraph. We record all the videos liked by each user to support functionalities such as querying the user's list of liked videos or providing recommendations based on the liked videos. In conventional databases, a table is typically stored within a single Bw-tree, where the table's primary key serves as the key, and the data rows act as the values. In our case, we can adopt a similar storage approach, encoding the source and destination nodes of the edge as the key, while the edge features are stored as values. Consequently, all user-like actions are maintained within a single Bw-tree. However, in practice, we have made three key observations:

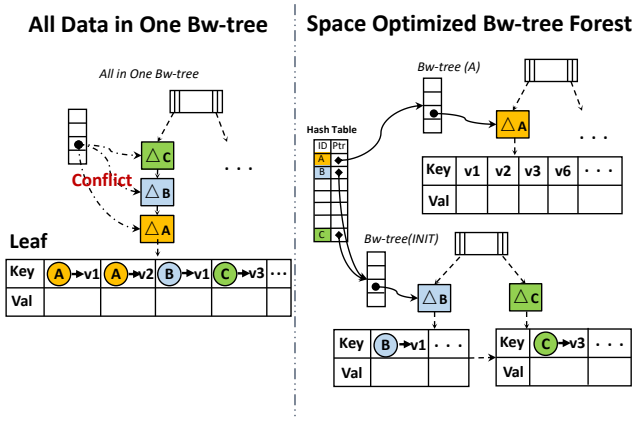

Figure 3: Space Optimized Bw-tree Forest

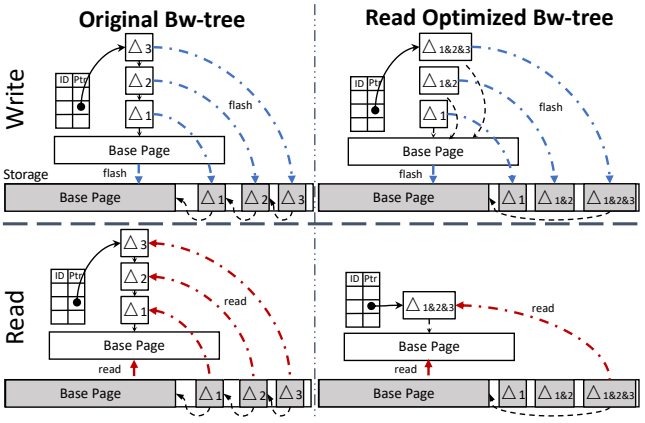

Figure 4: Read & Write Process Comparison

Observation 1: Storing all user-like actions within a single Bw-tree can potentially lead to write conflicts within the structure of Bw-tree. As shown in the left side of the Figure 3, let's assume we have three users: A, B, and C. While B and C are ordinary users, A is an active user who likes many videos every day. According to the traditional Bw-tree approach, like actions from different users would be written to the same leaf node of the Bw-tree. In the context of the Douyin app, a large number of user-like actions are generated concurrently every moment. The ultra-high concurrency scenario in ByteGraph can lead to numerous conflicts, resulting in write retries and waiting. This substantially diminishes the concurrent write throughput of the Bw-tree.

Observation 2: The read and write across various user nodes are entirely independent and do not interfere with one another. Taking the user-like workload as an example, suppose we designate a unique Bw-tree to store all the use-like edge of one user. It is obvious that a user will not like two videos at the same moment, and the liking behaviors of different users are inserted into different Bw-trees. Such a storage scheme significantly mitigates the risk of access collisions within the Bw-tree framework, thereby optimizing the overall performance of the database system.

Observation 3: Simply dividing all edges corresponding to each users into separate Bw-trees can effectively solve the problem of massive concurrent write conflict. However, we found that this approach leads to additional space wastage. The like behavior of users generally follows a power-law distribution, where some extremely active users spend hours daily browsing and liking numerous videos, while the majority of ordinary users like a very limited number of videos each day. The backend storage of Bw-trees is often block-based, and to align with these storage units, the leaf nodes of a single Bw-tree typically contain dozens or even hundreds of edges. If the use-like edges of every user are stored in separate Bw-trees, for the vast number of ordinary users, the space wastage from both the storage holes in the leaf nodes and the memory overhead in maintaining the intermediate nodes, mapping tables, and other data structures, will be substantial.

Based on the above observations, we propose to use 'Space Optimized Bw-tree Forest' as the storage engine. As shown in the right side of the Figure 3, we store all users' IDs in a hash table as keys, where the value of the hash table points to the Bw-tree storing the user's user-like edges. The like operations of each new user are centrally recorded in an initial Bw-tree (called Bw-tree (INIT)). When a user is highly active, the videos liked by this user increase rapidly, likely causing high-frequency write conflicts in the Bw-tree (INIT). Meanwhile, we observed that the more edges a user has on their Bw-tree, the more frequently it gets accessed. We allow each workload to be configured with a threshold. Once a user's number of edges surpasses this threshold, their data is divided and placed into an individual Bw-tree. As shown in the upper right of Figure 3, after User A's data is split and stored in a separate Bw-tree (A), write conflicts on the Bw-tree significantly decrease. Simultaneously, since all edges in Bw-tree (A) originate from User A, we change the key of Bw-tree (A) from user ID->video ID to video ID only, further saving space. Meanwhile, to ensure the query efficiency of Bw-tree (INIT), when the total size of Bw-tree (INIT) exceeds the threshold, we select the user with the most edges in Bw-tree (INIT) and store their data edges in a dedicated Bw-tree.

3.2.2 Read Optimized Bw-tree. In ByteDance, the writing and reading of edges often occur simultaneously. For instance, after a user likes a video, we read the list of videos liked by the user as an input of the recommendation model. When a user follows another user, the Douyin generates a user-to-user edge that is written into ByteGraph. Concurrently, the app reads the list of all users a person is following to display videos from these followed users in the app. As shown in Table 1, the proportion of read operations is very high in many cases of Bytedance. Traditional Bw-trees use a base plus delta model. As shown in the upper left of Figure 4, we perform three updates on a page of the Bw-tree at different times, creating three delta records. For data consistency, both the base page and the delta data have to be flushed to backend storage after being written into memory. The position of the data in the append-only backend storage is determined by the time of flashing. Since multiple Bw-trees are updated simultaneously, and even within a single Bw-tree, different pages might be updated at the same moment, the base page and the three delta updates are distributed at different locations in storage. To read the latest content of a page, we need to perform four random reads in storage to assemble the complete data in memory (shown in the lower left of Figure 4). In ByteDance's

scenario, with immense read/write IOPS, frequent random reads on storage significantly reduce the read bandwidth, thus failing to meet the application's demand for read IOPS.

Algorithm 1: Write Process of Read Optimized Bw-tree

```
   Input: K, V
   Output: Success
1  found ← find(K, &pos);
2  if (!found) then
3      new_base_page ← BwTreeSplit();
4      Update(new_base_page, K, V);
5      pos → new_base_page;
6      Flush(new_page);
7      return Success;
8  end
9  if IsBasePage(pos) then
10     new_delta ← DeltaAllocate(K, V);
11     old_base_page = *pos;
12     new_delta.next → old_base_page;
13     new_delta.count=1;
14     pos → new_delta;
15     Flush(new_delta);
16     return Success;
17 end
18 else
19     old_delta=*pos;
20     new_delta ← DeltaAllocate(old_delta + (K, V));
21     if old_delta.count+1>ConsolidateNum then
22         old_base_page = old_delta.next;
23         new_base_page=Consolidating(old_base_page+new_delta));
24         pos → new_page;
25         Flush(new_base_page);
26         return Success;
27     end
28     new_delta.next → old_delta.next;
29     new_delta.count=old_delta+1;
30     pos → new_delta;
31     return Success;
32 end
33 return FALSE;
```

To enhance the read performance of Bw-trees, we propose the 'Read Optimized Bw-tree'. The pseudo code of the entire procedure of handling write requests is shown in Algorithm 1. We use classic lightweight locking mechanisms [20] to guarantee safe modifications across multiple threads. Firstly, we utilize the find function to locate the page containing the key (line 1). If the page does not exist (lines 2-8), or if the page exists but this base page has not been modified before the current write (lines 9-17), our approach is the same as that of a traditional Bw-tree. We write the new key-value pair into the base page or delta, modify the mapping table and flush the corresponding base page or delta data. When we find that the page containing the key exists and has been previously modified, we merge the old delta data with the new update into a new delta (line 20). This new merged delta directly points to the base page. Simultaneously, we update the page's mapping pointer to the new merged delta, ensuring that each page only has one delta update. Compared to traditional Bw-tree, the read optimized Bw-tree increases the total size of flushed delta data (shown in the upper right of the Figure 4). However, given that the delta data size is substantially smaller compared to the base page, and considering that the flushing of merged deltas to append-only storage occurs in a sequential manner, we have observed that the additional write

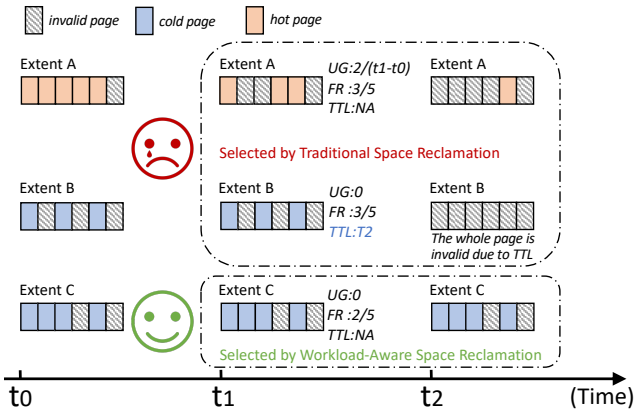

Figure 5: Spatial Changes of Different Extents (UG stands for Update Gradient; FR stands for Fragmentation Rate; TTL stands for Time to Live)

overhead is quite minimal. As the read optimized Bw-tree ensures that each page has at most one delta, when reading the latest content of any page, we only need to perform two read operations: one for the base page and one for the merged delta (shown in the lower right of the Figure 4). This significantly reduces the amount of data read and the number of random read IO operations, thereby greatly increasing the overall read bandwidth of the Bw-tree.

3.3 Workload-Aware Space Reclamation

To ensure data persistence, the base page and delta page of a Bw-tree are written to an append-only shared cloud storage. In traditional Bw-tree systems, space reclamation is managed through a First-In-First-Out (FIFO) queue. New data is added to the front of the queue. During each space reclamation cycle, the process begins by scanning from the back of the queue and rewriting any valid data to the front, thus reclaims the space occupied by invalid data.

The traditional space reclamation strategy of Bw-trees does not consider the space reclamation rates of different data segments, resulting in significant write amplification due to background data movement. From a space reclamation perspective, the write patterns of base pages and delta pages are different. Compared to base pages, delta pages have a shorter lifespan and a higher space reclamation rate. To further improve the efficiency of space reclamation and reduce write amplification, the state-of-the-art ArkDB [31] proposed writing base page and delta page data into two separate streams for individual space reclamation. Additionally, ArkDB divides each stream into extents of equal size and tracks the reclaimable space ratio of each extent. When triggering space reclamation, ArkDB first targets extents with a high ratio of reclaimable space for data movement, thereby reducing the write amplification rate.

We've incorporated ArkDB's design approach by segregating base and delta data into distinct streams and partitioning each stream's data into uniformly sized extents. Concurrently, considering the specific data access characteristics of ByteGraph, we have made the following two observations.

Observation 1: Taking the video liking scenario as an example, the power-law distribution characteristic of graph data results

in a disparity in the popularity (likes, favorites, views) of videos, presenting a clear distinction between 'hot' and 'cold' content. For any given video, the rate of increase in likes just after its release will be much higher than that a month later. This variation in the growth of likes affects how frequently the pages of each video's corresponding Bw-tree are modified. Consequently, this results in the rate of increase of invalid pages varying across each extent. As illustrated in Figure 5, an extent with data from a newly released video (Extent A) undergoes frequent updates, causing the data within it to become invalid more quickly (comparing with Extent C).

Observation 2: Given that user preferences evolve over time, we use time windows to keep track of their recent browsing history, search actions, and video preferences. This requires ByteGraph to support the functionality of expiring and deleting outdated data. This process, based on Time-To-Live (TTL), results in the lower-layer extents undergoing batch deletions when their storage duration ends. As shown in Figure 5, when ExtentB reaches the t2 point, all its data will have collectively expired, eliminating the need for any relocation.

Based on the above two observations, we propose a workload-aware space reclamation strategy. For each extent, we implement an in-memory structure 'Extent Usage Tracking', in which we record: 1. The latest update time in the extent, 2. total number of invalid page

- **Time-to-Live (TTL)** We assign the timestamp of the most recently updated piece of data in an extent as the timestamp for the entire extent. This approach is feasible because the Bw-tree operates on an append-only manner, and given the extensive scale of ByteDance's operations, the data timestamps within each extent tend to be quite similar. In scenarios where data expiration is required, we can readily determine an extent's TTL (Time-To-Live) by adding the expiration period to its timestamp.
- **Fragmentation Rate** We calculate an extent's fragmentation rate by keeping track of the count of its valid and invalid pages.
- **Update Gradient** We adopt the concept from [26]. Whenever an extent undergoes an update, we log both the time of the update and the count of invalid pages it currently contains. For instance, as depicted in Figure 5, Extent A at time t1 has three invalid pages, an increase from a single invalid page at time t0. Consequently, the update gradient for Extent A is calculated as (3-1)/(t1-t0).

Figure 5 illustrates how data becomes invalid over time in three distinct extents, assuming no space reclamation occurs. If we need to choose one extent from A, B, and C at t1 for space reclamation. Traditional reclamation strategies would select either Extent A or B, as they have the highest fragmentation rate (3/5). However, traditional space reclamation does not consider the issues of TTL and hot data being frequently updated. If Extent A is chosen at t1, it means we have to move three pages, but soon two out of these three pages will become invalid at t2, resulting in a waste of 2/3 of the write I/O. Suppose we choose Extent B for reclamation at t1, its three valid data blocks will be moved to a new extent. However, these blocks will expire soon at t2, leading to inefficient use of

Algorithm 2: Workload-Aware Space Reclamation

```
   Input: n : (Number of Extents to be reclaimed)
   Output: Success
1  for i = 1 : n do
2      list ← getExtentsWithSmallestUpdateGradient();
3      sortByFragmentationRate(list);
4      cur ← list.head;
5      while cur! = list.tail do
6          doSpaceReclamation(cur);
7          cur ← cur.next;
8          ++i;
9          if i > n then
10             break;
11         end
12     end
13 end
```

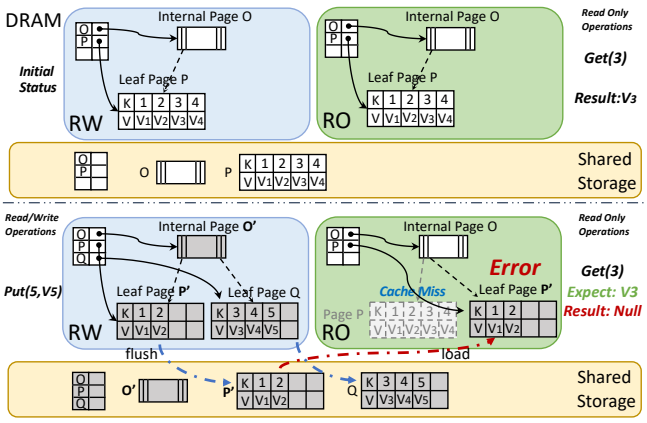

Figure 6: Data Inconsistency Issue

bandwidth. As shown in Algorithm 2, workload-aware space reclamation preferentially selects cold data with a low Update Gradient (line 2) and, among these, chooses the extent with the highest fragmentation rate for reclamation (lines 3-11). In situations where data expiration is involved, we bypass those extent and allow it to expire naturally. The experiments detailed in Section 4.4 demonstrate that our approach effectively reduces the write amplification rate in the storage system.

3.4 I/O Efficient Synchronization Mechanism

As shown in the upper part of Figure 6, BG3 utilizes a design that separates read and write operations. All new data updates are first written into the memory of the Read/Write Node (RW Node) and then flushed to the storage. Since BG3 is built on a strong-consistency shared storage architecture, once the RW node writes new data into the shared storage, the Read-Only nodes (RO) can immediately read the new data from the storage.

The shared storage layer ensures consistency after data is written. However, each RO node maintains a cache of data tailored to its read workload. Without meticulous synchronization between RW and RO nodes, this can lead to issues of data inconsistency on the RO nodes. As shown in the upper half of Figure 6, we assume that the system initially stores four key-value pairs (1, V1), (2, V2), (3, V3), and (4, V4), along with the corresponding internal node page

O. The system records the positions of Page O and Page P through the Bw-tree's mapping table. Initially, the data on the RW, RO, and shared Storage are in a completely consistent state. As depicted in the lower half of Figure 6, when we insert new data (5, V5) at the RW node, this operation causes a split in the Bw-tree, with half of the data from leaf page P being moved to the new leaf node page Q. Afterwards, RW writes the modified mapping table, the updated pages O' and P', and the new leaf page Q into the shared storage to complete the write process. Meanwhile, the RO node continuously handles read requests, and the cache on RO node dynamically evicts pages from DRAM based on the read requests. Suppose after RW has completed the insertion of (5, V5), RO node receives a request to retrieve the value for key 3. At this moment, RO node does not have page P in its cache, triggering a read operation from the shared storage. Given that page P in the shared storage has been updated to P', this update results in a consistency error during the RO node's Get(3) operation, because it cannot locate the data on the now-altered page.

The primary reason for the inconsistency between RO and RW nodes is that the corresponding data in the shared storage device has already been updated before the latest modifications in RW node's memory are synchronized to RO node's memory. Several solutions have been proposed to solve the leader-follower consistency problem. As shown in Figure 1, the previous generation of ByteGraph achieved data synchronization by asynchronously forwarding Gremlin commands of RW write operations to each RO nodes[24]. However, this approach is prone to causing disorder or packet loss during the forwarding process, requiring mechanisms like retries to achieve eventual consistency. [33] sends RW node's memory update operations to RO nodes through logs, allowing RO nodes to replay these logs to align their memory versions with RW's memory version. Meanwhile, the system ensures data consistency by requiring that RW nodes delay data flushing to the shared storage until RO nodes have finished updating the relevant data in their memory. This approach effectively maintains data consistency between RW and RO nodes. However, as described previously, the workload executing on ByteGraph display a pronounced locality. This characteristic makes the aforementioned method prone to slow log replay due to hotspots on some RO nodes, subsequently blocking RW's data flushing.

Based on the above observations, we propose the I/O Efficient Synchronization Mechanism. In summary, our approach involves synchronizing the latest updates from the RW node to the RO nodes' memory via a Write-Ahead Log. To address data consistency issues, we maintain multiple versions of data within the shared storage layer. Additionally, we've put in place various parallel enhancements to improve I/O efficiency, especially for high-pressure production environments in Bytedance. To facilitate a better understanding, we continue with the example used in Figure 6 and describe the specific process of our handling in Figure 7. As shown in Figure 7, when RW node receives a Put(5,V5) request at (1), it triggers a split in the Bw-tree nodes in RW's memory, generating dirty pages: internal page O', leaf pages P', and Q. We first records the entire Bw-tree split process through write-ahead logging (WAL), where the log sequence numbers (LSN) range from 30 to 32. The WAL is written to the shared storage immediately after the RW update at (2) and is instantly read into the RO node's memory at (3). The RO node caches the WAL in memory. Using a lazy replay mechanism, we update the cached page O in RO to the same state as RW's O' using log LSN 30 at (4). At the same time, users trigger Get(2) and Get(3) operations at (3). These operations cause cache miss actions for pages P and Q in the RO cache. At (5), the RO node looks up the old mapping in shared storage to fetch page P, and then at (6), replays the relevant log (LSN 32) on P. When the old mapping in storage does not contain the page recorded in the WAL (page Q), indicating that this page is newly generated. The RO node directly creates it in memory at (6). By this point, the data in the RO node's memory is completely consistent with the latest data in RW node. Meanwhile, the three dirty pages generated by the Bw-tree split in RW memory are asynchronously flushed to the shared storage by a background thread pool at (7). After the dirty data is flushed to the append-only data area, we update the data version on the shared storage by updating the mapping table at (8), and synchronously writes a log in the WAL, indicating that the data in shared storage has completed all modifications up to LSN 34. Once the RO reads this log item, it can discard all records in the lazy replay log with an LSN number less than 34.

Guarantee of Correctness. As shown in Figure 7, our proposed I/O Efficient Synchronization Mechanism synchronizes the latest modifications from the RW node's memory to the log area in RO's memory through WAL. This process enables the immediate updating of the outdated data in RO's memory to the latest version, aligning it with RW's memory. Meanwhile, before the dirty data on RW is asynchronously flushed to the shared storage, the mapping table in the shared storage continues to point to the old version of the data. This approach guarantees that the RO node can access precise data in its memory by merging the old version data with the WAL, at least up until point (8).

Data Freshness. Given that each WAL item is small in size and the shared storage guarantees consistency and low write latency, updates to the RW node's WAL are immediately visible to all of the RO nodes.

I/O Efficiency. To address the high-intensity read and write requests in ByteGraph scenarios, Firstly, we adopted a strategy similar to group commit[19], where accumulated dirty pages on the RW are flushed by a background thread once the accumulated dirty pages on the RW reach a specific threshold. Meanwhile, it's important to note that the data each RO node requests can differ significantly from what is written to RW. To reduce the pressure on RO nodes reading data from the shared storage, we adopted a lazy replay log mechanism. Updates from the RW are applied in RO's memory only when a page is brought into RO's memory due to an upper-layer read request. To improve the efficiency of searching the log area in RO's memory, we built an index keyed by page number. Additionally, we regularly merge multiple modifications of the same page in the log area in the background.

4 EVALUATION

In the experimental evaluation, we demonstrate that BG3 achieves high performance, good scalability in processing various types of workloads and efficient leader-follower synchronization.

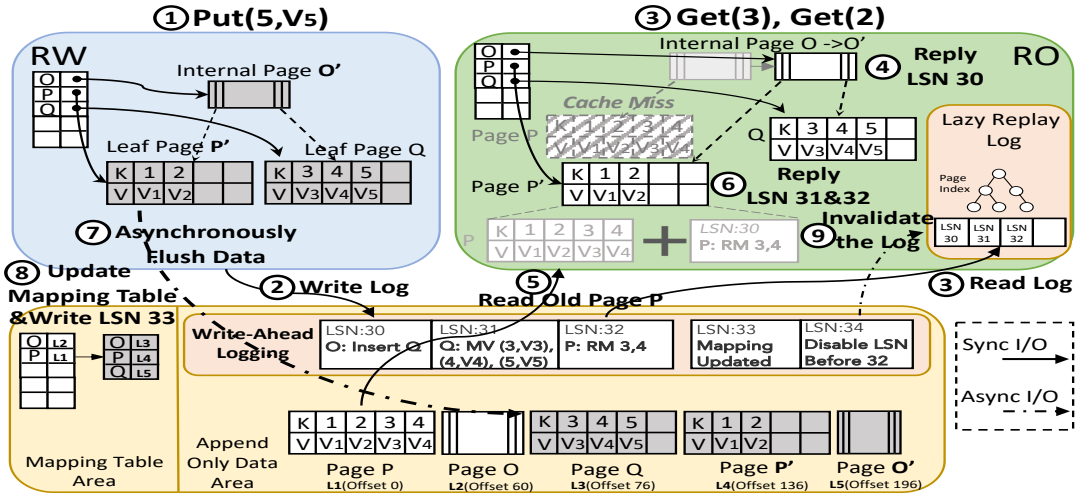

Figure 7: Workflow of I/O Efficient Synchronization

4.1 Experiment Setup

Workloads. Table 1 lists three subset of the real query workloads we used in ByteDance. The "Read/Write" column indicates the proportion of read and write queries and the "Description" column shows what operations are included in the workload. The $|V|$ and $|E|$ column indicates the number of vertices and edges on the used graph, respectively. We also list the hops accessed by each workload.

The "Douyin Follow" workloads records users' follow behavior. The throughput can reach more than tens of millions queries per second, and it contains 1% of single-edge insertions, with each insertion representing a new *Follow* record. The remaining 99% consists of one-hop neighbor queries, used to enumerate all followers of a particular user.

In "financial risk control" workloads, we perform subgraph pattern matching [32] on a graph with constantly inserted edges. Concurrently, we consistently fetch multi-hop neighbors from the RO node to check the existence of edges inserted from the RW node. This workload is used within ByteDance for detecting the security of fund transfers in payment scenarios and the throughput of this workload can reach tens of thousands. The read-write ratio is strictly fixed at 1:1, and the workload involves continuous data insertion, requiring ByteGraph to be configured with a TTL to automatically delete data once it becomes outdated.

In "Douyin recommendation" workload, we perform multi-hop neighbor query to generate subgraphs for the down stream recommendation models, and the the overall throughput can reach hundreds of millions. These three workloads cover the most typical scenarios of graph databases within ByteDance. Each of these workloads has different read-write ratios, QPS (Queries Per Second), and requirements for data timeliness. We use these three workloads as examples to validate the effectiveness of BG3.

Experiment outline & Hardware Setup. We first compare the performance of BG3 with previous version of ByteGraph [24] as well as Amazon Neptune [5] under all workloads listing in Table 1 (Section 4.2). Then we evaluate the impact of our proposed space optimized Bw-tree forest (Section 4.3), workload-aware space reclamation (Section 4.4) and I/O efficient synchronization mechanism 4.5 through a series of micro-benchmarks.

We run the overall comparision on a cluster of 10 nodes where each node is equivalent to a db.r5.4xlarge in AWS. For all the micro-benchmark tests, we utilize servers with 2.3 GHz Xeon Platinum 8336C CPU, 1007 GB DRAM, 3.5 x 2 TB NVMe SSD and 100GbE network. We use ByteDance's internal append-only cloud storage as the shared storage device, which is capable of providing cloud storage services with millisecond-level latency. This system is similar to others like Aliyun Pangu [25], Meta Tectonic Filesystem [30], and Azure Storage[17].

4.2 Overall Comparison

Our previous study [24] has already demonstrated that ByteGraph consistently outperforms Amazon Neptune [15], Alibaba GDB [4], and TigerGraph [18] under ByteDance's graph workload. In this section, we select AWS Neptune as a representative and use the workloads in Table 1 to compare the performance of ByteGraph, BG3, and Neptune. We first evaluate scalability of the ByteGraph, BG3 and Amazon Neptune on a single machine single-machine by rising the available number of vCPU cores from 4 to 16. Subsequently, we expanded the comparison to include configurations from 2 to 10 nodes, each equipped with 16 vCPU cores. For each system, we kept adding clients until a point was reached where there was no further increase in throughput. We then calculated and reported the system's average throughput over a one-hour period.

As shown in Figure 8, BG3 achieves competitive performance compared with ByteGraph and Amazon Neptune across all workloads. In particular, ByteGraph achieves up to 24.23×, 17.05× and 114.75× thoughput compared with AWS Neptune over three workloads. Meanwhile, BG3 still can consistently outperform ByteGraph. Compared to AWS Neptune, ByteGraph achieves the significant performance advantage due to the choices of different architectural designs [24]. Here, we mainly focus on exploring the reasons that BG3 outperforms the ByteGraph system. Thanks to BG3's storage

Table 1: Workload description.

Workload	Read/Write	Description	|V|	|E|	Hops
Douyin Follow	99%/1%	Managing Douyin *Follow* records, single edge insertion, one-hop neighbor query	3M	0.5B	1
Financial Risk Control	50%/50%	Pattern matching[32], single edge insertion, full graph reading, 10 hops and 100 edges	5B	100B	5 to 10
Douyin Recommendation	read-only	multi-hop neighbor query, 70% 1-hop, 20% 2-hop, and 10% 3-hop	3M	0.5B	1 to 3

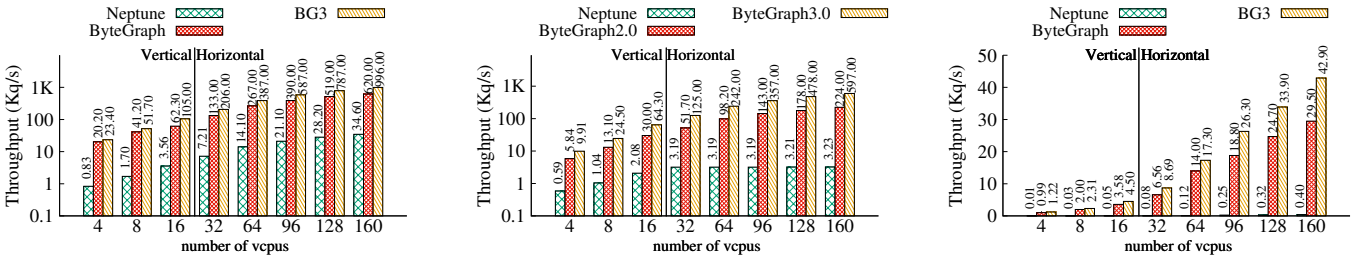

Figure 8: Overall Performance (Vertical: a single machine; Horizontal: 2 to 10 machines, each with 16 cores)

engine employing a leaner layering approach, BG3 achieves up to 1.68× and 4.06× better performance than ByteGraph across two read-dominant workloads (Figure 8(a) and Figure 8(c)). As illustrated in Figure 1, in ByteGraph, reading an edge might trigger two searches: one in the B-tree of the indexing layer and another in the underlying distributed LSM-based KV. In contrast, with BG3, we only need to perform a single search through the bw-tree to directly access the data in memory or shared storage (as shown in Figure 2). As described in Section 4.1, the performance of 'Financial risk control' depends on the data synchronization efficiency between RW and RO nodes. Thanks to our proposed I/O efficient synchronization mechanism, BG3 has achieved up to 2.68× better performance compared to ByteGraph, while both ByteGraph and BG3 show good scalability.

Storage Cost Saving. The cost of each ByteGraph machine is categorized into three parts: CPU, memory, and storage. The design of BG3 primarily focuses on optimizing storage costs. We monitored the storage costs for the three mentioned workloads on both ByteGraph and BG3, finding that BG3 can save about 80% in storage costs on average. This significant reduction in storage costs mainly stems from our proposed Space Optimized Bw-tree Forest and Workload-Aware Space Reclamation, which greatly alleviate the write amplification issue common in LSM-tree based KV engines [28] Additionally, switching from LSM-tree based KV storage to shared cloud storage further reduces the cost per bit of storage.

4.3 Evaluation on Bw-tree Liked Graph Storage Engine

4.3.1 Read Optimized Bw-tree. In this section, we choose the classic Bw-tree implementation:SLED[12] as a baseline to compare with our proposed read optimized Bw-tree strategy. For fairness, we restricted BG3 from splitting the Bw-tree and set both systems to consolidate after every 10 delta updates. All data were inserted into both a SLED and our proposed read optimized Bw-tree.

To compare IOPS amplification, we set the cache size of both Bw-trees to zero, ensuring each read results in a cache miss hitting the storage. In the experiment, we used Douyin follow data and

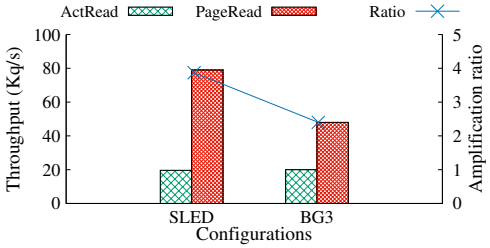

Figure 9: Read Amplification Comparision Between the Traditional Bw-tree and the Read Optimized Bw-tree.

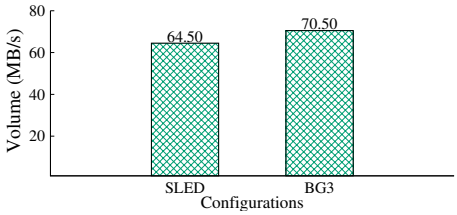

Figure 10: Write Bandwidth Comparision Between the Traditional Bw-tree and the Read Optimized Bw-tree.

simulated realistic access patterns with a power-law benchmark at 20K QPS. As shown in the Figure 9, the QPS from SLED to storage is 76k, which, compared to the entry QPS, results in an amplification ratio of 3.87 times. This roughly reflects the average number of deltas per page in SLED. As described in Section 3.2.2, BG3 retains only one delta for each base page, so the read QPS to storage is 48k. Thus, when compared to SLED, the amplification ratio to storage indicates a 36.8% decrease.

Meanwhile, we utilized a 20K QPS write-only power-law benchmark to evaluate the additional write overhead caused by merging multiple deltas of each base page when using our proposed Read Optimized Bw-tree. As shown in the Figure 10, SLED writes 64.5MB to storage, while BG3 writes 70MB. As expected, compared to the traditional Bw-tree, BG3 incur additional delta data writing due to

Table 2: Evaluation of Different Space Reclamation Policy

	Workload 1		Workload 2	
	Dirty ratio	**+Gradient**	**Dirty ratio**	**+TTL**
Write Amplification	15	12.5	8	0
Bwd Occupation (MB/s)				

delta merging. However, the extra data written by BG3 is only 9.3% more than SLED. Furthermore, all writes in BG3 are append-only sequential writes, which will further reduce the stress on the underlying system caused by the additional write data. In summary, In ByteDance's scenarios where reads are predominant, the single delta design of the read optimized Bw-tree effectively lowers random read QPS by 36.8%, while only modestly increasing sequential write demands.

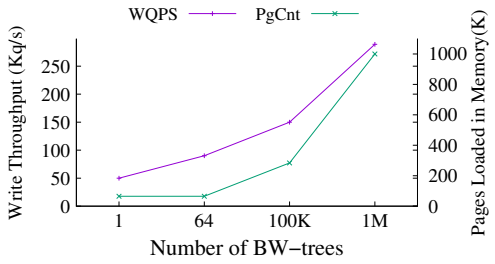

Figure 11: Scaling Performance & Space Cost with Varying Number of Bw-trees

4.3.2 Space Optimized Bw-tree Forest. As described in Section 3.2.1, we adjust the number of Bw-trees by adjusting the threshold. To eliminate the influence of irrelevant factors, we conducted full-cache stress testing and used the same write-only power-law benchmark from the previous section to test the write performance and space overhead of the Bw-tree forest. As shown in Figure 11, the write QPS increases linearly with the number of B-trees. Specifically, the write QPS rose from 50 to 90, 150, and 289 KQPS when the number of Bw-tree increase from 1 to 64, 100k and 1 million, respectively. At the same time, as the number of Bw-trees increases, memory consumption also gradually rises. In addition, we observed that when the number of Bw-trees grows from 100k to 1 million, the significant increase in memory consumption does not proportionally increase the write QPS. In particular, when the number of Bw-trees grows from 1 to 100k, QPS doubles while space consumption increases by 3.37 times. However, when the number of Bw-trees grows from 100k to 1 million, the write QPS only increases by 92%, but the memory overhead grows by 2.52 times. This indicates that beyond a certain number of Bw-trees, continuously increasing the name of the Bw-trees does not bring proportional performance improvements. In ByteDance's real-world scenarios, we adjust the number of Bw-tree splits based on whether the specific workload prioritizes performance or cost-effectiveness.

4.4 Evaluation on Workload-Awared Space Recycling

In this section, we used "Douyin Follow" and "Financial Risk Control" workload to evaluate our workload-aware space reclamation. We implemented the classic strategy of selecting extents for reclamation based on the dirty rate of extent data as a baseline [31]. Noted that whether a workload needs to set a TTL is determined by the application's own requirements. For example, the follow relationship data of "Douyin Follow" needs to be stored indefinitely after insertion and cannot be set to expire via TTL. Meanwhile, "Financial Risk Control" is primarily used for reconciliation. Once a RO node reads a piece of data written by a RW node and this data is verified, it can then be discarded by setting a TTL.

We first utilized a 40K QPS write-only power-law benchmark to simulate the "Douyin Follow" behaviors. As shown in the left part of Table 2, the amount of data that needs to be moved in the background is 15 MB/s when we employ a space reclamation strategy based on the extent dirty ratio. Due to the characteristics of hot and cold updates in 'Douyin Follow,' our proposed approach of choosing extents based on their update gradient can achieve a reduction of 16% in the volume of background writes for space reclamation.

As described in Section 4.1, 'Financial Risk Control' is used for fund proofread. Once newly inserted edges are read and verified by the RO node, the audit data automatically expires. Therefore, we need to use TTL to regularly invalidate inserted data. In this scenario, TTL can be set quite short (e.g., 10 minutes or even shorter), and as shown on the right side of the Table 2, once we enable the TTL strategy, the background process can completely forego space reclamation and wait for data to expire naturally. Workloads with TTL requirements can avoid unnecessary space transfers through our proposed workload-aware space reclamation scheme, but the current approach is not perfect. For some workloads with larger TTL settings (such as TTL set to expire in 30 days), the current strategy need to store 30 days' data in shared storage, causing significant space wastage. We might explore merging the gradient strategy with the TTL approach, which only bypass extents that have a set TTL and are close to their expiration time, but this approach requires a complex coordination mechanism and triggering strategy. We put it as our future work for further study.

4.5 Evaluation on Synchronization Mechanism

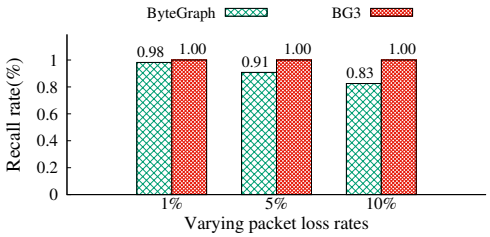

Figure 12: Evaluation of Recall Rates with Various Packet Loss Rates.

In this section, we use the "Financial Risk Control" workload to evaluate the leader-follower synchronization efficiency of ByteGraph and BG3. As described in Section 3.4, ByteGraph implements leader-follower data synchronization by forwarding RW's write Gremlin commands to each RO node. This approach can easily lose its consistency guarantee due to network fluctuations in clusters that have a high workload. To simulate packet loss phenomena

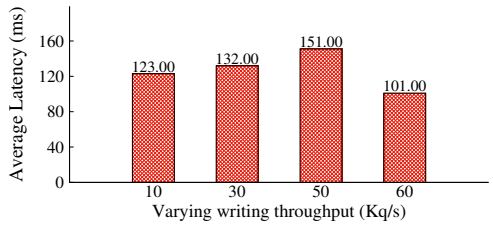

Figure 13: Leader-follower Latency with Various Writing Throughput (Kq/s).

in high-load clusters under real-world conditions, we artificially set the network's packet loss rate to range from 1% to 10%. Subsequently, we evaluate how effectively RO nodes could read the edges written by RW nodes, using the 'recall rate' as a measure, where a recall rate of 1 indicates that RO nodes can read all edges written by RW nodes. The Figure 12 illustrates that at a 1% packet loss rate, ByteGraph's RO nodes can read 98% of the data written. This recall rate diminishes to 91% and then to 83% as the packet loss rate escalates from 1% to 5% and subsequently to 10%. In contrast, since BG3 achieves data synchronization between RW and RO nodes by writing WAL to shared storage, we noted that BG3 can consistently read all data written by RW node, regardless of the network packet loss rate. We established a configuration where the number of RW and RO nodes was equal, maintaining a 1:1 ratio. We focused on the latency experienced by the RO node in reading the latest data. To evaluate the performance, we progressively increase the QPS of writes on the RW node, thereby observing how changes in write load affected read latency. Thanks to our proposed I/O efficient synchronization mechanism, we can see from the Figure 13 that as we increase the write load from 10K QPS to 60K QPS, the latency in BG3 consistently stays around the 120ms. "Given that our design has already shifted a significant portion of I/O to asynchronous operations, the latency seen in BG3's synchronization is primarily influenced by how long it takes for the RW to write the WAL to the lower-layer shared storage, and the duration it takes for RO nodes to read this log. With continuous improvements in the underlying shared storage layer, we believe our synchronization latency in BG3 can be further reduced.

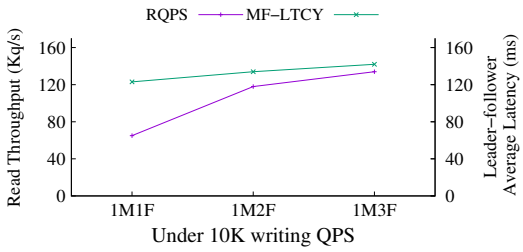

Figure 14: Scaling Performance of Leader-follower Synchronization When varying the Follower Nodes from 1 (1M1F) to 3 (1M3F), Where RQPS Indicates the Read Throughput and MF-LTCY Indicates the Leader-follower Synchronization Latency.

To observe the scalability performance of RO nodes, we broke the 1:1 read-write ratio limitation from the real "Financial Risk Control" scenario. We fixed the write traffic to RW at 10K QPS while allowing RO nodes to initiate as many read requests as possible. As shown in Figure 14, with a configuration of one RW and one RO node, the RO's read traffic can reach 65K QPS. When we expand the number of RO nodes to 2 and 4, the RO's read QPS can be increased to 118K and 134K QPS, respectively. At the same time, we observed that the data synchronization latency between RW and RO nodes consistently stays around 120ms. This demonstrates that in a one-write, multiple-read scenario, BG3 can achieve good read scalability while maintaining a stable data synchronization delay.

5 CONCLUSION

In this work, we present BG3, a high-performance distributed graph databased for the efficient management and processing large-scale graphs at ByteDance. BG3 provides three components to achieve its performance including a new storage engine based on cost-effective shared storages and BW-tree indexes that enhance read performance and reduce operation costs, a workload-aware space reclamation mechanism that reduces storage I/O operations, and a light-weight yet efficient leader-follower synchronization mechanism that provides strong consistency for scaling read throughputs. We experimentally show that BG3 achieves competitive performance compared with Amazon Neptune and ByteGraph.

REFERENCES

[1] 2021. *AgensGraph.* https://bitnine.net/.
[2] 2021. *ArangoDB.* https://www.arangodb.com/.
[3] 2021. *JanusGraph.* https://janusgraph.org/.
[4] 2022. *Alibaba GDB.* https://www.aliyun.com/product/gdb/.
[5] 2022. *AWS Neptune.* https://aws.amazon.com/neptune/.
[6] 2022. *Azure Cosmos DB.* https://docs.microsoft.com/en-us/azure/cosmos-db/graph/graph-introduction.
[7] 2022. *Google levelDB.* https://github.com/google/leveldb.
[8] 2022. *Neo4j.* https://neo4j.com/.
[9] 2022. *PostgreSQL.* https://www.postgresql.org/.
[10] 2023. *oceanbase.* https://en.oceanbase.com/.
[11] 2023. *PinCAP TiDB.* https://github.com/pingcap/tidb.
[12] 2023. *SLED.* https://github.com/spacejam/sled.
[13] Alibaba GCB 2020. Alibaba GCB. https://www.aliyun.com/product/gdb/.
[14] Alibaba PolarDB 2023. Alibaba PolarDB. https://www.alibabacloud.com/product/polardb.
[15] AWS Neutune 2020. AWS Neutune. https://aws.amazon.com/neptune/.
[16] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, Matthew Renzelmann, Alex Shamis, Timothy Tan, and Shuheng Zheng. 2020. A1: A Distributed In-Memory Graph Database. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 329–344. https://doi.org/10.1145/3318464.3386135
[17] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. 2011. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles.* 143–157.
[18] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor E. Lee. 2019. TigerGraph: A Native MPP Graph Database. *CoRR* abs/1901.08248 (2019). arXiv:1901.08248 http://arxiv.org/abs/1901.08248
[19] Hector Garcia-Molina and Kenneth Salem. 1992. Main memory database systems: An overview. *IEEE Transactions on knowledge and data engineering* 4, 6 (1992), 509–516.
[20] Goetz Graefe. 2010. A survey of B-tree locking techniques. *ACM Transactions on Database Systems (TODS)* 35, 3 (2010), 1–26.
[21] Xiaowei Jiang, Yuejun Hu, Yu Xiang, Guangran Jiang, Xiaojun Jin, Chen Xia, Weihua Jiang, Jun Yu, Haitao Wang, Yuan Jiang, Jihong Ma, Li Su, and Kai Zeng. 2020. Alibaba Hologres: A Cloud-Native Service for Hybrid Serving/Analytical Processing. *Proc. VLDB Endow.* 13, 12 (2020), 3272–3284.
[22] Sarath Lakshman, Apaar Gupta, Rohan Suri, Scott D. Lashley, John Liang, Srinath Duvuru, and Ravi Mayuram. 2022. Magma: A high data density storage engine

used in Couchbase. *Proc. VLDB Endow.* 15, 12 (2022), 3496–3508.

[23] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 302–313.

[24] 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. *Proceedings of the VLDB Endowment* 15, 12 (2022), 3306–3318.

[25] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. 2023. More than capacity: performance-oriented evolution of Pangu in Alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 331–346.

[26] David Lomet and Chen Luo. 2021. Efficiently reclaiming space in a log structured store. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 792–803.

[27] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 133–148. https://www.usenix.org/conference/fast16/technical-sessions/presentation/lu

[28] Lanyue Lu, Thanumalayan Sankaranarayana Pillai, Hariharan Gopalakrishnan, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2017. Wisckey: Separating keys from values in ssd-conscious storage. *ACM Transactions on Storage (TOS)* 13, 1 (2017), 1–28.

[29] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Informatica* 33, 4 (1996), 351–385. https://doi.org/10.1007/S002360050048

[30] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. 2021. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*. 217–231.

[31] Zhu Pang, Qingda Lu, Shuo Chen, Rui Wang, Yikang Xu, and Jiesheng Wu. 2021. ArkDB: a key-value engine for scalable cloud storage services. In *Proceedings of the 2021 International Conference on Management of Data*. 2570–2583.

[32] Shixuan Sun and Qiong Luo. 2020. In-Memory Subgraph Matching: An In-depth Study. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1083–1098.

[33] Taobao. 2021. MySQL Monthly - March 2021. http://mysql.taobao.org/monthly/2021/03/04/. Accessed: 2023-11-29.

[34] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data* (Chicago, Illinois, USA) *(SIGMOD '17)*. Association for Computing Machinery, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101