ByteGraph: A High-Performance Distributed Graph Database in ByteDance

Changji Li[1,2,†], Hongzhi Chen[2,*], Shuai Zhang[2], Yingqian Hu[2], Chao Chen[2], Zhenjie Zhang[2], Meng Li[2], Xiangchen Li[2], Dongqing Han[2], Xiaohui Chen[2], Xudong Wang[2], Huiming Zhu[2], Xuwei Fu[2], Tingwei Wu[2], Hongfei Tan[2], Hengtian Ding[2], Mengjin Liu[2], Kangcheng Wang[2], Ting Ye[2], Lei Li[2], Xin Li[2], Yu Wang[2], Chenguang Zheng[1,2,†], Hao Yang[2], James Cheng[1]

[1]{cjli, cgzheng, jcheng}@cse.cuhk.edu.hk, [2]{lichangji, chenhongzhi, zhangshuai.root, huyingqian, chenchao.chen, zhangzhenjie.zz, limeng.1, lixiangchen, handongqing, chenxiaohui.ai, wangxudong.zsy, zhuhuiming.eureka, fuxuwei, wutingwei, tanhongfei, dinghengtian, liumengjin, wangkangcheng.qwq, yeting.dev, lilei.rd, lixin.andy, wangyu.cole, zhengchenguang, yanghao.2019}@bytedance.com

[1]The Chinese University of Hong Kong, [2]ByteDance Inc

ABSTRACT

Most products at ByteDance, e.g., TikTok, Douyin, and Toutiao, naturally generate massive amounts of graph data. To efficiently store, query and update massive graph data is challenging for the broad range of products at ByteDance with various performance requirements. We categorize graph workloads at ByteDance into three types: online analytical, transaction, and serving processing, where each workload has its own characteristics. Existing graph databases have different performance bottlenecks in handling these workloads and none can efficiently handle the scale of graphs at ByteDance. We developed ByteGraph to process these graph workloads with high throughput, low latency and high scalability. There are several key designs in ByteGraph that make it efficient for processing our workloads, including edge-trees to store adjacency lists for high parallelism and low memory usage, adaptive optimizations on thread pools and indexes, and geographic replications to achieve fault tolerance and availability. ByteGraph has been in production use for several years and its performance has shown to be robust for processing a wide range of graph workloads at ByteDance.

PVLDB Reference Format:
Changji Li, Hongzhi Chen, Shuai Zhang, Yingqian Hu, Chao Chen, Zhenjie Zhang, Meng Li, Xiangchen Li, Dongqing Han, Xiaohui Chen, Xudong Wang, Huiming Zhu, Xuwei Fu, Tingwei Wu, Hongfei Tan, Hengtian Ding, Mengjin Liu, Kangcheng Wang, Ting Ye, Lei Li, Xin Li, Yu Wang, Chenguang Zheng, Hao Yang, James Cheng. ByteGraph: A High-Performance Distributed Graph Database in ByteDance. PVLDB, 15(12): 3306 - 3318, 2022.
doi:10.14778/3554821.3554824

* Hongzhi Chen is the Corresponding Author.
† This work was done when the authors were in ByteDance.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554824

1 INTRODUCTION

Graph data exists ubiquitously in ByteDance's products, e.g., TikTok, Douyin, and Toutiao, where the sizes of graphs are in the scale of tens of billions of vertices and trillions of edges (and still growing). Graph workloads in ByteDance can be categorized into three types: online *analytical*, *transaction*, and *serving* processing (i.e., OLAP, OLTP, and OLSP). OLAP workloads on graph data usually contain multi-hop graph traversal queries which generally have large intermediate results due to the existence of super-vertices (i.e., vertices with a large number of neighbors). OLTP workloads require transactional guarantee for modifications on multiple graph objects (i.e., vertices, edges, and their properties). OLSP workloads serve applications in real time and data freshness is critical [25]. In addition, OLSP workloads usually have high concurrent writes (with higher throughput requirement than OLTP), while read queries need to fetch the latest data to serve applications (e.g., recommendation service, risk management). We describe the OLAP, OLTP and OLSP workloads in details in Section 2.

Existing graph databases suffer from various performance problems in handling these workloads. Graph databases offered by cloud vendors such as AWS Neptune [8] and Alibaba GDB [6] only use one (master) machine to handle write operations and thus cannot scale to handle high concurrent writes in our OLSP and OLTP workloads, while Azure CosmosDB [9] stores graph data in a document store where super-vertices are managed as large JSON documents which leads to high latency in data access. Open source graph databases such as ArangoDB [4], AgensGraph [3], Neo4j [10] and JanusGraph [5] generally have poor scalability and cannot satisfy the high throughput and low latency required in handling ByteDance's workloads. A1 [15] and TigerGraph [20] focus on in-memory architectures to provide low query latency, but in-memory systems are hard to be scaled to handle large graphs at ByteDance, while storing the entire graph data in memory is also a waste of the resource as not all graph data are needed for query processing at all times. There are also other graph databases proposed in recent years [16–18, 23, 26, 33], but these systems are more research prototypes and do not provide fault tolerance and availability guarantee required by ByteDance. We will discuss the limitations of existing graph databases in details in Section 7.

Table 1: The amplification rate of OLAP workloads

Cluster	Mean	P99	Max
A	989.6	6732.2	81735.7
B	1625.7	82066.9	188620

We developed ByteGraph, a distributed graph database, to efficiently store, query and update large scale graph data at ByteDance. ByteGraph adopts a two-tier architecture that consists of a durable storage layer to ensure data durability and a cache layer for low latency data accessing. To optimize the cache layer, we propose a btree-like structure, *edge-tree*, to store the adjacency lists of vertices, which not only provides high parallelism for accessing super-vertices' large neighborhood, but also reduces the overall data loaded into memory for queries such as edge searching. The edge-tree can be configured to fit different workloads with various read-write requirements to balance the read and write amplifications. For the durable storage layer, since most data do not have a fix size (i.e., the number of neighbors, length of vertex/edge properties), we choose to use a persistent Key-Value store to provide fine-grained storage management for the edge-tree.

As a large scale production system, ByteGraph should be robust to sudden workload changes and machine failure. First, to promptly react to sudden situations (e.g., rapid workload increases) and protect the system from crash, we propose two adaptive optimizations on thread pools and indexes to adapt to such situations. Second, to provide continuous services during machine failure or even whole cluster shut-down, we provide several techniques to guarantee the availability of ByteGraph, including weighted consistent hashing [31] and geographic replications. Geographically replicating our data can also reduce the latency of queries from different regions. We demonstrate the performance, scalability and availability of ByteGraph with both simulated workloads and our real production workloads.

2 WORKLOAD ANALYSIS

ByteGraph has been extensively used in ByteDance for all kinds of application scenarios such as social media, knowledge graph search and analysis, risk management, e-commerce, and recommendation services. Each business unit in ByteDance may run multiple ByteGraph clusters to process different types of workloads for its applications. Currently there are over 600 ByteGraph clusters being deployed on more than 13 thousands machines, and the numbers are steadily growing as our businesses grow. We first describe the workloads handled by the ByteGraph clusters (Section 2.1) and then discuss some design principles of ByteGraph based on the characteristics of these workloads (Section 2.2).

2.1 The Workloads

We categorize the workloads into three types: *online analytical processing (OLAP)*, *online serving processing (OLSP)*, and *online transaction processing (OLTP)*. We describe the characteristics of each type of workloads as follows.

Online Analytical Processing (OLAP). In ByteDance, graph analytical processing exists widely in all kinds of risk management and knowledge graph applications. OLAP queries are relatively more complex compared with OLSP and OLTP queries, and they also access a larger portion of the graph data. For example, subgraph pattern matching is required by risk management applications that aim to detect abnormal patterns in a transaction graph. The filters on the intermediate results vary from simple property filtering to complex subquery matching. OLAP queries often traverse multiple hops from a starting vertex. Due to the nature of ByteDance's applications (e.g., Douyin, TikTok, Toutiao), a graph in ByteDance usually has a relatively high number of *super-vertices* (i.e., vertices with very high degree). Thus, OLAP queries can easily access a massive number of vertices through some super-vertices in just 2 hops. We show the amplification rate for the OLAP workloads of a typical day in two ByteGraph clusters in Table 1, where the amplification rate is defined as the average number of accessed vertices per query per minute. The amplification rate can be three orders of magnitude on average and five to six orders of magnitude at peak, which indicates the large volume of intermediate results generated by the analytical queries. In addition, OLAP workloads in ByteDance also process periodical graph updates, which may be generated in two ways: (1) The updates are invoked by user actions, where the overall throughput is not high (i.e., up to tens of thousands writes per second). (2) The updates are aggregated by applications and scheduled to be executed in batches periodically with high write throughput.

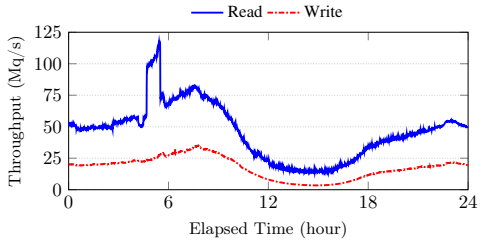


Figure 1: The throughput (million queries per second) of read and write queries of an OLSP cluster

Latency and error rate are two crucial metrics that ByteDance uses to measure the performance of its OLAP workloads. On the one hand, we need to provide high parallelism to process a query (especially a complex query) to reduce its latency as most of our applications require quick response. On the other hand, when the computing resources are saturated, we should also provide high scalability to avoid frequent errors that would causes poor service level objective (SLO) and expensive retries.

Online Serving Processing (OLSP). The concept of serving processing is introduced in [1], which describes the scenarios that real-time features are fetched or further processed (e.g., aggregated) to serve continuously updated applications or models. ByteDance has numerous scenarios with such workloads in which data are graphs. For example, to support real-time training for recommendation services in short video applications, real-time user actions (e.g., clicks, views, likes) are aggregated to provide samples for model training. These user actions are recorded in the form of edge insertion between users and video vertices, or by modifying the attributes of existed edges. Since the updates come mostly from frequent actions in users' daily life, there can be a large volume of

Table 2: The characteristics of OLAP, OLSP and OLTP workloads in ByteDance (Thpt, Lat, ER, and AR stand for throughput, latency, error rate, and abort rate)

Workload	Applications	Read Ratio (%)	Overall Thpt (QPS)	Traversal Hops	Write Frequency	Performance Concern
OLAP	Knowledge Graph, Risk Management	100%, dropping to 60% during data ingestion	tens of thousands (10^4)	3 to 5	Periodically	Lat, ER
OLSP	Recommendation, GNN Sampling, Feature Serving	75% to 90%	hundreds of millions (10^8)	1 to 2	Real-Time	Thpt, Lat, ER
OLTP	E-commerce, Content Record	90% to 99.9%	tens of millions (10^7)	1	Real-Time	Thpt, Lat, AR

updated data. Figure 1 shows the read and write query throughput for a typical OLSP cluster. There are tens of millions of write queries per second, where each write query inserts an edge, and the number of read queries at least doubles that of write queries.

A read query for an OLSP workload is usually used to select samples or aggregate features for model training. In most cases, the query only accesses the neighboring vertices/edges within two hops of a given vertex. Compared to OLAP workloads, OLSP queries are simpler and easier to be processed. However, usually a massive number of read queries are processed concurrently for an OLSP workload. As shown in Figure 1, the throughput of read queries can reach more than 70 millions queries per second, where most of them (99.9%) are retrieving one-hop neighbors. In addition, the throughput can burst within a short time. As shown in Figure 1, the number of read queries is nearly doubled within a short period. This explosive increase in the read throughput can result in high latency and error rate if the computing resources are saturated. Therefore, ByteGraph needs to provide high elasticity and scalability to promptly react to this situation.

Moreover, there are also predicates on the vertex/edge properties that need to be verified during graph traversals in read queries. However, the key of the most frequently accessed property changes dynamically, which may invalidate the existing index and accordingly incur high CPU consumption for full scan. Thus, ByteGraph also needs to decide whether and which index should be built.

Online Transaction Processing (OLTP). Transactional requirements in the graph context consist of mainly two aspects: (1) To ensure the consistency of index and original data, ByteGraph should provide atomic update inside the system to ensure data freshness. (2) Some applications require atomicity when updating multiple vertices and edges. For example, if a user posts an article with a tag, three edges, namely $(user, article)$, $(user, tag)$ and $(article, tag)$ should be atomically inserted. Otherwise, this triangle relationship can be lost if failure happens during the insertion, which can further lead to other cascading problems in our business and hurt user experience.

Both read and write transactions are lightweight in our OLTP workloads. Specifically, read transactions mostly only contain one-hop traversal, while writes are usually manipulating a handful of vertices/edges. The overall throughput of an OLTP workload can reach from tens of millions to hundreds of millions of transactions per second. Meanwhile, read operations can dominate the overall throughput, where the percentage of reads varies from 90% to 99.9%.

2.2 System Design Principle

Table 2 summarizes the key characteristics of the three types of workloads. We discuss the limitations of existing systems in handling these workloads in Section 6 and Section 7. To the best of our knowledge, there is no existing system that can efficiently handle OLAP, OLSP and OLTP graph workloads. While we may use a dedicated system for each type of workloads, maintaining multiple systems significantly increases the costs of system development, operation and maintenance. To this end, we set out to design a unified system to handle OLAP, OLSP and OLTP workloads in ByteDance. We summarize our design principles as follows.

- **Decoupling computation and storage.** Different types of workloads present different requirements on CPU and storage utilization. To fully utilize both resources, ByteGraph can adopt the design of separating computation and storage, which provides independent scalability.
- **Read-write amplification.** Our storage design should consider both read and write amplifications from the persistent storage. The read amplification incurs overhead on read operations and wastes the limited memory resource, which further leads to a low cache hit rate. The write amplification results in high latency on write operations and also occupies more I/O bandwidth. There is sometimes a trade-off between reducing read and write amplifications. Our design should balance the read and write amplifications on different types of workloads for the best overall performance.
- **Cache hit rate.** Frequent disk access is detrimental to performance due to the significant speed difference between SSD and memory. The cache in memory should be fully utilized to accelerate data extraction. The system should provide fine-grained data management to reduce the volume of accessed data during execution to increase cache hit rate.
- **High scalability.** The system should have high scalability when the throughput requirement of our workloads is increased both normally and explosively. In particular, the system should promptly react to explosive throughput increases as they are common in ByteDance's workloads.

3 SYSTEM ARCHITECTURE

The architecture of ByteGraph is depicted in Figure 2. A ByteGraph cluster consists of three layers: an execution layer (BGE), a cache layer in memory (BGS), and a durable storage layer based on a persistent KV store. BGE mainly handles computation-intensive

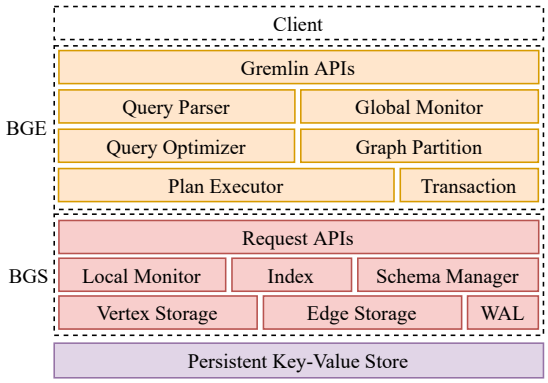


Figure 2: The architecture of ByteGraph

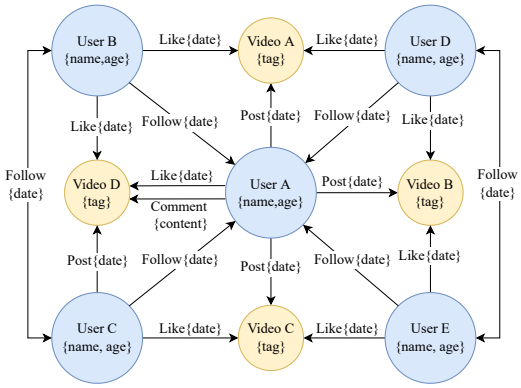


Figure 3: An example of property graph

operations (e.g., sorting, aggregation), while BGS focuses on graph-native cache data management and log management. Both layers can be independently scaled out according to the workloads and available resource. The durable storage layer is responsible for persisting all KV pairs generated by BGS (i.e., graph data, logs, and metadata). Existing KV stores (e.g., RocksDB [12], TerarkDB [29]) can be used in this layer, which is treated as a black box in ByteGraph.

ByteGraph uses Gremlin [7] as the query language as most of the applications built on ByteGraph are based on navigational queries. Currently, ByteGraph supports 54 Gremlin steps, where the unsupported Gremlin steps are mostly graph algorithms (e.g., PageRank, Single-Source Shortest Paths, and Peer Pressure) for which we have a dedicated graph system (i.e., a Pregel-like system) to process them, or rarely-used steps in ByteDance (i.e., Skip, None, and Coin). A Gremlin query is translated into a physical execution plan by the parser and optimizer in BGE. We provide both rule-based and cost-based optimizer. To increase the cache hit rate, we logically partition the graph data with consistent hashing algorithm [27] where a partition is mapped to a BGS instance. Thus, to access the data, vertices located at the same partition are grouped and shipped to the associated BGS instances through RPCs for further processing. BGE maintains a global view of BGS instances by monitoring their heartbeats. Moreover, BGE coordinates the distributed transactions using the 2PC protocol, which we discuss in Section 5.1. For fast predicate evaluation and data compaction, the schemas of vertices and edges should be defined before loading into ByteGraph. To allow schema modification during runtime, the schema manager assigns versions to schemas such that vertices/edges can be deserialized based on the version number.

BGS receives the requests from BGE and caches the accessed data in memory for fast accessing. In addition to the graph data, BGS also maintains system data that are used to accelerate the processing or to ensure the consistency and atomicity (i.e., indexes and logs). We enforce schema on vertex and edge properties for fast lookup and data integrity. The schema is multi-versioned to provide asynchronous modification. Besides, we introduce two adaptive optimizations to address workload changing issues in Section 4.3.

Moreover, as a large-scale industry-level graph database, ByteGraph provides availability by replicating data cross data centers and geographical regions. The synchronization among different clusters is completed via binlog [13]. We introduce how ByteGraph guarantees fault tolerance and availability in Section 5.2.

4 SYSTEM DESIGN

ByteGraph adopts the property graph model where both vertices and edges have associated types and properties. An example of a property graph is shown in Figure 3. There are two vertex types (*User, Video*) and four edge types (*Like, Post, Comment,* and *Follow*), where the schema is various with the type of vertex/edge (e.g., *User {name, age}*, and *Video {tag}*). We first discuss how graph data are stored in a KV store (Section 4.1) and how queries are executed (Sections 4.2). Then we present two adaptive optimizations to handle workload changing situations (Section 4.3).

4.1 Data Storage

As shown in Figure 4, BGS caches vertices and edges in memory with *Vertex Storage* and *Edge Storage*, respectively. Each vertex and its properties are stored as a KV pair, where the key is encoded by a unique ID and the vertex type, and the value is a list of properties of the vertex. For example, the key of vertex *User A* in Figure 3 is encoded as $\langle A, User \rangle$. To access the properties of a vertex, BGS invokes a *get()* request to the underlying KV store and caches this KV pair in *Vertex Storage*. Once there is a modification on any property of a vertex, its associated KV pair will be instantly flushed to disk by a *set()* request.

To efficiently execute graph traversal queries, edges are organized as adjacency lists. To reduce the possibility of generating massive intermediate results after visiting super-vertices in a graph traversal, adjacency lists are further divided according to edge types and directions. Thus, the key to identify an adjacency list is encoded as $\langle vID, vType, eType, dir \rangle$, where $eType$ denotes the edge type and dir is the edge direction. To handle super-vertices and frequent updates, the storage should be able to (1) reduce the write amplification brought by edge insertion and (2) incur less disk I/Os during a whole list scanning. To satisfy these requirements, each adjacency list is stored as a tree structure, called *edge-tree*. As shown in Figure 4, an edge-tree is composed of three types of nodes, i.e., Root Node, Meta Node, and Edge Node, each of which is stored as a KV pair. Edge-tree works like a B-Tree. That is, both root node

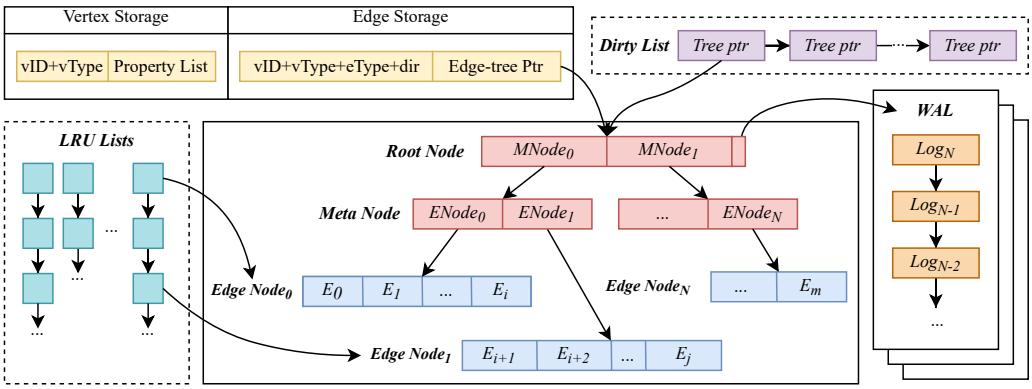


Figure 4: The memory layout of BGS

and meta nodes play the roles of indexes, while only edge nodes store the physical edge data. All the three types of nodes have a lower/upper bound on their size to balance the read/write amplification. Initially, an edge-tree has only two layers (i.e., root node and edge nodes). But with the increase in the number of edges managed by an edge-tree (i.e., the size of the edge nodes indexed by the root node exceeds the upper bound), we will create meta nodes as the middle-layer to index the edge nodes. Similarly, one edge node will be split into two nodes if its size is larger than the upper bound, or two edge nodes will be merged into one node if their sizes are smaller than the lower bound. Empirically, we set the lower/upper bound to be 1000/2000, for which the capacity of a three-layer edge-tree is 8 billions, which is enough to store an adjacency list from even a massive real-world graph. In practice, we will adjust this bound dynamically based on the read/write workloads of a cluster. For example, if read queries dominate, we set a relatively larger upper bound for the edge-tree, and consequently it will reduce the number of disk I/Os and also improve data locality in the durable storage layer. Each edge instance is formed by the ID and type of the destination vertex, as well as the list of its edge properties. Then, we compress the property data based on its schema to reduce storage footprint. Edges in an edge-tree are sorted by a specific sort key, determined by the schema of the edge properties. ByteGraph uses "tsUs" as the default sort key if no particular property has been given, where "tsUs" is the timestamp when this edge is inserted. Moreover, this sort key is configurable or dynamically adjusted according to real-time workloads (Section 4.3.2).

We follow an asynchronous workflow to process updates on an edge-tree using Write-Ahead-Log (WAL). During an update, modifications are persisted by WAL first and then applied to the edge-tree in memory, where a new node is marked as a dirty node. BGS periodically flushes dirty nodes to disk. As updates may lead to node split/merge that modifies more than one node, a child node should be flushed before the parent node to ensure consistency. Consequently, we use edge-tree as the granularity of flushing. As shown in Figure 4, BGS maintains a global *Dirty List* representing the edge-trees that include dirty nodes. We only insert a dirty edge-tree to the list when (1) the number of WALs after the last checkpoint exceeds a threshold, or (2) a least used node in the LRU list is selected to be evicted for more available memory. The WALs for an edge-tree are indexed with monotonically increasing IDs. Meanwhile, the root node records the log ID of the last *persist* operator as the checkpoint. Therefore, Condition (1) above reflects that BGS restricts the number of WALs to avoid incurring a heavy recovery load. For Condition (2), if a dirty node is selected to be swapped out in the LRU list, we also insert the associated edge-tree into the dirty list to release memory as soon as possible. Similar to flush, swap is also periodically conducted to evict the least used nodes from memory. Due to the limited CPU resources, the frequency of flushes and swaps should depend on the memory consumption. Specifically, we linearly reduce the wait time between two flushes or swaps with the increase in the overall memory consumption.

4.2 Query Processing

When a query is sent to ByteGraph, it is routed to a random BGE instance. A gremlin query is parsed and translated into an execution plan by the parser and optimizer. We provide both rule-based optimizer (RBO) and cost-based optimizer (CBO). The RBO contains a rich set of rules (29 rules), involving step fusion, predicate push-down, limit forward, data prefetch, and subquery elimination. And we also provide a CBO that can select the optimal execution order for queries that have multiple execution plans. For example, a query tries to check the existence of an edge between two given vertices. We can locate the target edge in the adjacency lists of both vertices. To reduce the data accessed, this query will start from the vertex with fewer neighbors. As we record the number of edges in the root node for each edge-tree, the overhead of the CBO is negligible. This optimization reduces the accessed data size of the current query and further balances the workload by navigating the requests to avoid starting from a super-vertex when possible. To avoid redundent processing on the same query, every BGE instance can cache the results of frequent queries. We sacrifice the consistency between the data and query cache by periodically updating the query cache. Therefore, BGE only caches the results of query with extreme high frequency and low requirement on data freshness.

For load balancing and to increase the cache hit rate, BGE logically partitions a graph by consistent hashing [27], where each partition is assigned to one BGS instance. Both vertices and adjacency lists are partitioned by their keys. Consequently, a vertex or

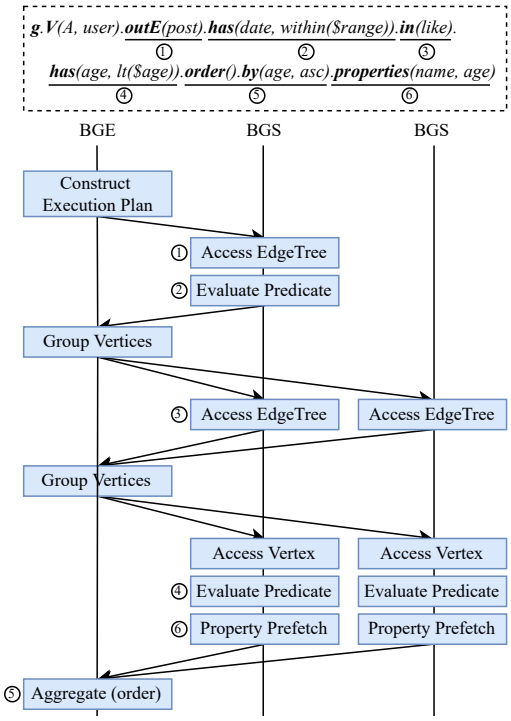


Figure 5: The workflow of processing an example query

an edge-tree only appears in a single BGS instance, which prevents write conflicts happening in the underlying KV store. However, if BGS is scaled in or out, a vertex or edge-tree may still be written by more than one BGS instance at the same time. If the underlying KV store provides atomicity on writing a single KV pair, ByteGraph can directly rely on this feature. Alternatively, we can wrap a CAS set() function to ensure atomicity based on normal KV store APIs.

To execute a query, BGE sends read/write requests to BGS based on the graph partition. Specifically, according to the inputs of each step, BGE generates a set of requests, each of which contains the fetch/update operations on target objects (i.e., vertices/adjacency lists) located on the same partition. BGS will execute the request and ship the results back to BGE for subsequent execution. We illustrate the procedure in Figure 5 with an example query, where the query attempts to obtain the two-hop neighbors of a given vertex with the filters on both edge and vertex properties. The actual execution location for each step is marked in Figure 5. BGE first sends a request to the BGS instance that handles the adjacency list with the first edge label (i.e., *post*) of the start vertex for accessing the neighbors (step ①). The predicate evaluation (step ②) is pushed down to BGS by RBO to save the network cost and the number of RPCs. After receiving the results from the first traversal, BGE groups the intermediate vertices with the edge type of the second traversal (i.e., *like*) and dispenses them to the corresponding BGS instances. As BGS is not aware of how a graph is partitioned, the result vertices of step ③ have to be sent back to BGE for further routing, which can also de-duplicate the intermediate vertices. The evaluation on the property of the vertices (step ④) and extracting the properties (step ⑥) are executed together on the corresponding BGS instances. BGE receives the results from the BGS instances and applies the aggregation (step ⑤). After receiving read/write requests from the BGE, BGS splits a request into a set of tasks where each task is responsible for accessing one KV pair. Tasks generated from one request can be executed in parallel to maximize the parallelism. The generated tasks are dispensed to task queues in a round-robin manner for load balancing.

When there are multiple BGS instances participating in one step, we provide two modes (i.e., *barrier mode* and *eager mode*) to decide when to start the execution of the next step. The barrier mode does not start the next step until all BGS instances return the results. Consequently, BGE can send all intermediate vertices located on the same BGS instance with a single RPC. Alternatively, BGE can also immediately start the next step once the results are received under the eager mode. Although the eager mode incurs more RPCs, it can avoid a long-running request blocking the whole procedure. Note that we only apply the eager mode for applications that accept partial results.

4.3 Adaptive Optimizations

We may encounter many sudden situations (e.g., rapid workload increases, significant predicate changes) in a ByteGraph cluster and we need to ensure that ByteGraph can provide continuously services with minimal degradation in performance. We introduce two adaptive optimizations, *dynamic thread pools* and *adaptive secondary edge-trees*, to handle such situations.

4.3.1 Dynamic Thread Pools. Read/write requests may have different loads depending on how many tasks are simultaneously generated for the requests. For a request with a large number of parallel tasks, we need to restrict the available resources that will be assigned to process them to avoid starving the other requests.. In particular, super-vertices often become hotspots where massive requests are pushed into a single BGS instance. In the worst case, the BGS instance rejects all subsequent requests such that ByteGraph may have a high error rate and cannot provide normal services.

To ensure the quality of service, two thread pools are created to handle the light and heavy requests, respectively (Figure 6(a)). A request is considered heavy when the number of its tasks is larger than a threshold. All the tasks of a heavy request can only be executed in the heavy thread pool. As the number of heavy requests is much less than that of light requests in our workloads, we initiate a small number of threads for the heavy thread pool. Although we can achieve a better overall error rate by restricting the available threads for heavy requests, it is also unacceptable to sacrifice the error rate of the heavy requests due to the expensive retry overhead. Intuitively, we should scale out the BGS cluster to provide more capacity. Unfortunately, as some hotspots may quickly vanish, adding new machines not only cannot address the issue in time because of the warm-up time (Section 5.2), but also incurs more overheads on solving remote write conflicts (Section 5.1).

To fully utilize the resources and scale out when necessary, we propose *dynamic thread pools* where the numbers of available threads are adjusted according to the loads. Specifically, we monitor the average number of tasks queued in each thread pool, which indicates the pressure of a thread pool. We denote the pressures as

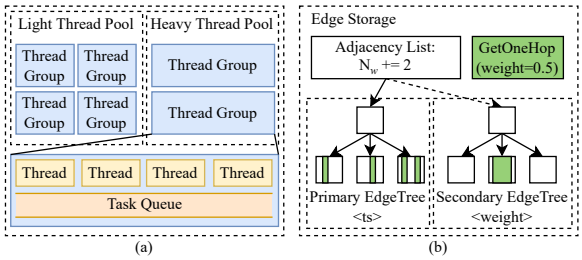


Figure 6: (a) Thread pools; (b) Secondary edge-tree

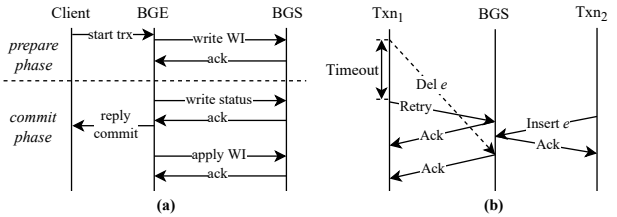


Figure 7: (a) The workflow of 2PC; (b) The inconsistency caused by retry

$\hat{P}_L$ and $\hat{P}_H$ for light and heavy thread pools, respectively. We adjust the number of threads in the heavy thread pool (T_h) as follows:

$$T'_h = (1 + \alpha * \log(\hat{P}_H / \hat{P}_L)) * T_h, \quad (1)$$

where α is a decay factor to slow down the adjustment rate ($\alpha = 0.2$ by default). With the increase in the number of heavy requests, we can assign more threads to the heavy thread pool. When the number of heavy requests reduces, we also reduce the number of threads in the heavy pool by Equation (1). Meanwhile, we still need to ensure the performance of light requests. Thus, we stop the adjustment process when the light requests start to be timeout. If the error rate of heavy requests is still increasing, scale-out is invoked.

4.3.2 Adaptive Secondary Edge-Trees. The edge-tree structure is also the essential data structure for our index on BGS to facilitate searching on the adjacency list with the sort key. However, if the search key does not match with the sort key, then a whole edge-tree scanning is still required, which leads to high CPU occupation and high possibility of cache misses. Thus, we provide a secondary edge-tree where the adjacency list is sorted by another edge property. Once a secondary edge-tree is built, a pointer pointing to the new edge-tree is added into the Edge Storage to form a forest for an adjacency list (Figure 6(b)). Each read request can select the edge-tree that matches with the search key. If all the edge-trees do not match with the search key, the read request will select the edge-tree with more edge nodes existing in memory to maximize the cache hit rate. However, if the size of an adjacency list is small or the accessing frequency is low, constructing a new edge-tree cannot improve the cache hit rate but will occupy more space. Therefore, we adopt an adaptive method to dynamically decide whether a secondary edge-tree should be built. For each property, we use the number of saved edge nodes in an edge-tree to measure the benefit brought by the secondary edge-tree. Specifically, for a request, we first record the number of accessed edge nodes in the primary edge-tree (N_{pri}) and then estimate the number of edge nodes required in the secondary edge-tree (N_{sec}). The number of saved edge nodes is calculated by subtracting the above two numbers. We accumulate the number of saved edge nodes for all the requests in a time period to measure the benefit of building a secondary edge-tree. An example is given in Figure 6(b), the request *GetOneHop(weight=0.5)* has to access all the three edge nodes in the primary edge-tree, while it only requires one edge node in the secondary edge-tree. Therefore, we record that two more edge nodes can be saved by the secondary edge-tree of property *weight*. BGS will build the secondary edge-tree once the benefit exceeds the threshold. However, as different clusters require different thresholds, we further count the number of all the edge-nodes accessed in the current BGS instance (N_{all}). Therefore, the final benefit (B_p^t) of building the secondary edge-tree for an edge property (p) in a time period (t) is calculated as:

$$B_p^t = \frac{\sum_r (N_{pri}^r - N_{sec}^r)}{N_{all}}. \quad (2)$$

Each primary edge-tree maintains a list of B_p^t for all the edge properties. For the number of estimated edge nodes required by the secondary edge-tree, we optimistically consider the best case where the smallest number of edge nodes is used. Moreover, as many edge-trees may invoke the construction of secondary edge-trees at the same time, we restrict the available threads for this process. Meanwhile, as the secondary edge-trees are persisted, BGS can directly read them from the KV store when they are needed.

5 SYSTEM IMPLEMENTATION

We discuss some implementation details of ByteGraph in this Section. We will also open source ByteGraph in due course.

5.1 Distributed Transaction Processing

ByteGraph supports ACID transactions with Read-Committed (RC) isolation level. Currently we do not consider higher isolation levels since RC can satisfy most requirements of our applications.

We leverage the two-phase commit (2PC) for distributed transaction processing. Figure 7(a) shows the workflow. Same as query processing, each transaction is received by a random BGE instance, which is designated as the coordinator. During the *prepare* phase, the modification requests are sent to the corresponding BGS instances. Similar to CockRoachDB [32], each participant stores the update in a provisional value, *write-intent*, which records the before and after image of the data and the unique identification of the corresponding transaction. However, as ByteGraph does not support MVCC, the write-intent acts as a write lock rather than a new version, where each vertex/edge can only contain one write-intent to avoid write-write conflicts. To avoid deadlock, a timestamp, *startTs*, is assigned to each transaction. The transactions with smaller *startTs* should abort during a write-write conflict.

After receiving acknowledgments from all write-intents, the coordinator starts the *commit* phase. To reduce the latency, we atomically persist the transaction status (i.e., commit or abort) and then reply to the client. The write-intents are asynchronously applied to the original data. Specifically, when encountering a write-intent,

a transaction should check the status of the transaction generating the write-intent. If the status is "commit", an asynchronous thread is invoked to apply the changes to the original data and erases the write-intent. Alternatively, if the status is "abort", the write-intent is directly erased. Moreover, if the transaction status cannot be found, we directly abort this transaction by persisting the transaction status as "abort". Therefore, the uncommitted changes of an on-going transaction would not be visible to other transactions, which guarantees the Read-Committed isolation level.

To reduce the abort rate, ByteGraph will retry the requests if they are timeout. However, such a design may cause inconsistency and break the atomicity in some circumstances. As shown in Figure 7(b), a transaction, Txn_1, tries to delete an edge e. However, due to some network issue, the deletion request cannot reach the BGS instance. After the timeout, Txn_1 retries the deletion and successfully commits. Meanwhile, another transaction, Txn_2, tries to insert the same edge e, which also succeeds. And after all these operations, the deletion request drifting in the network eventually arrives the BGS instance and deletes e again. Consequently, the insertion of e by Txn_2 is overwritten. If Txn_2 contains other operations than inserting e, then the atomicity of Txn_2 is also broken in this case. To address this problem, we maintain a table to record the transactions that touch the edge-tree in the previous five minutes. All transactions presenting in this table should be rejected.

5.2 High Availability and Fault Tolerance

As a system providing production-level services, we should ensure fault tolerance and high availability. We introduce the methods we adopt to provide high availability within a data center, within a region and crossing the regions.

Within a data center. We should ensure that the whole cluster can still serve its applications with minimal degradation in performance during machine fails. The BGS instances are monitored by all BGE instances where each BGE instance maintains the same consistent hashing ring. When a machine hosting a BGS instance is down, the heartbeat sent by this instance is stopped. BGE routes the requests belonging to the lost machine to the next machines on the hash ring. However, directly shifting all workloads of one machine to a few machines can cause workload imbalance. We adopt the weighted consistent hashing algorithm [31]. By assigning the weight indicating the load of each machine, we can keep load balancing by adjusting the weight of machines and corresponding locations on the hash ring. Meanwhile, during scale-out, the new machines without warm-up have a low cache hit rate and can further affect the overall performance. The worst case is that the instant high throughput can crash the newly added machines. The weighted consistent hashing algorithm can also avoid such a situation. We only assign small weights to the newly added machines and adjust the weights to normal according to the real and expected throughput.

Cross data centers in the same region. When the data centers are located in the same region, ByteGraph chooses to use a master-slaves replications approach to ensure high availability, as the network latency between data centers in the same region is at millisecond level. Specifically, as shown in Figure 8, both master and slave clusters can receive read and write queries as usual, while the write

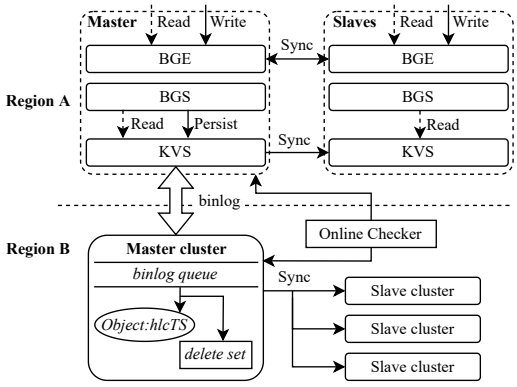


Figure 8: The cross DC and region deployment of ByteGraph

queries received from each cluster will be broadcast to the others. These updates will be applied to the BGS layer in a normal way to make them visible to the read queries immediately, but only the master cluster has the permission to flush these writes into the persistent KVS layer with WAL, in order to avoid write inconsistency among the replicas. Then, the KVS on the master cluster will synchronize the persistent writes to the other KVS replicas on the slave clusters in milliseconds. When cache missing or data expiration happens on the BGS layer of the slaves, it will read the latest data from the KVS layers. Thus, we only guarantee eventual consistency among the replicas. Note that in this setting, write queries posted to the slave clusters will only return "successful" when these writes have been flushed down to the master's KVS. According to the return flags from the master cluster, a slave cluster can monitor the status of the master. If entries in the master cluster fail due to unexpected situations (e.g., power loss), the write operators will not be successful and one of the slave clusters can be promoted to be the master. By default, we only hold one or two slave clusters to avoid a heavy *master-selection* procedure.

Cross data centers in different regions. To provide high availability and avoid high latency brought by the cross-region communication, ByteGraph provides the ability to replicate across geographical regions. Each region contains the whole master-slaves architecture and accepts both read and write requests.

We leverage the binary log(binlog) [13] to guarantee consistency among master clusters in various regions (see *region B* in Figure 8). The binlog is a set of logs that records the information of updates happened in a ByteGraph cluster. We assign a timestamp to each vertex/edge that indicates the update time. Each log contains the previous and current timestamp to resolve conflicts. Since updates can be frequently invoked from different machines in a cluster during scaling out, we cannot directly use the physical clock for the timestamp. Instead, ByteGraph adopts the Hybrid Logic Clock (HLC) [28] to avoid the inaccuracy brought by clock skew. The HLC timestamp is generated during updating or inserting vertices/edges. Consequently, the HLC timestamps for objects in one transaction can be different, which can break the atomicity of a transaction during synchronization. Therefore, the transaction coordinator selects the maximum HLC timestamp for all updates in the transaction

Table 3: Workload description

Type	Input	Description	Option
Write	N/A	Create vertices/edges/properties, Update properties, Delete edges/properties	N/A
1-Hop Query	1 vertex	Return neighboring vertices/edges of the input vertex	filter, property, order, group
	2 vertices	Find the connecting edge(s) of two input vertices	filter, property
2-Hop Query	1 vertex	Return 2-hop neighbors of the input vertex	filter, property, order, group
	2 vertices	Find the common neighbors of two input vertices	filter, property, order, group
3-Hop Query	1 vertex	Return 3-hop neighbors of the input vertex	filter, property, group
Path Query	1 vertex	Find path by *repeat().until()* until the requirement in *until()* is satisfied	filter

during the commit phase, where the new HLC timestamp should be written back to write-intents.

When generating the WAL, ByteGraph also generates the binlog and sends them to consuming queues for all data centers in other regions. We follow the *last-write-win* policy to resolve the conflicts thanks to the strict monotonicity of HLC. Moreover, when consuming the binlog, the objects in the binlog might be already deleted in the current cluster. Thus, there is no HLC timestamp that can be used to resolve the conflicts. To address this issue, we maintain a set of delete operations with their HLC timestamp for each cluster. Therefore, the binlog consumer can find the HLC timestamp of the delete operations. Besides, we also provide an online checker for each cluster to validate the data consistency by directly reading data from the clusters that generate the binlog.

6 EXPERIMENTAL EVALUATION

In the experimental evaluation, we will demonstrate that ByteGraph achieves high throughput, low latency, and high scalability for processing various types of workloads. We also demonstrate its robustness in handling spikes and its fault tolerance and availability.

Workloads. Table 3 lists the query templates we used to simulate the OLAP, OLSP, OLTP workloads in ByteDance. Here, we only briefly describes their query patterns and functionalities, while the specific query-list (more than 40 queries) will be released to public through our project website. The "*Input*" column indicates the number of input vertices to a query and the "*Description*" shows how the query does with the input vertices. The "*Option*" column represents whether a query may contain the operators being listed, where *filter* means that there can be filters on edge/vertex properties, *property* means that the query should extract the required properties, *order* and *group* indicate whether the query needs to sort or group the query result. Each query in our workload should contain at least one option. Due to the space limitation, we do not include the full query set here but release them on github[1].

6.1 Throughput and Scalability

We first demonstrate that ByteGraph achieves high throughput for various workloads. We also evaluate the scalability of ByteGraph, comparing with two well-known graph databases provided by cloud vendors, Amazon Neptune [8] (v1.0.5.1) and Alibaba GDB [6] (v1.0.27). We ran the experiments on a cluster of 10 nodes where each node is equivalent to a *db.r4x.large* in AWS.

Due to data security reasons, we could not run our production workloads on Amazon Neptune and Alibaba GDB. Thus, we simulated the workloads described in Table 3 and a smaller Douyin social graph (as Alibaba GDB cannot handler larger graphs). The OLAP workload consists of only read queries, which are composed of 70% 1-hop queries, 20% 2-hop queries, 5% 3-hop queries and 5% path queries. OLSP consists of 75% read queries and 25% write queries, where the read queries are 1-hop or 2-hop queries while the write queries create/update/delete edges. The OLTP workload consists of 99% 1-hop read queries and 1% write queries that create/update/delete edges and create/update vertices. The data graph was simulated by extracting the neighbor distribution and the features of vertex/edge properties (i.e., number, value type, and cardinality) from the Douyin social graph. There are totally 3.7 million vertices and 520 million edges in this dataset. The degree distribution follows a power law, where the P99 (i.e., 99-th percentile) degree is 116 thousand. The number of vertex and edge properties are 3 and 7, respectively. Property types include integer, float number, string, and boolean values.

To test scalability, we first compared with the single-machine Alibaba GDB by increasing the available number of vCPU cores from 4 to 16. Then, we compared ByteGraph and AWS Neptune using 2 to 10 nodes (each with 16 vCPU cores). For each system, we increased the number of clients until its throughput did not grow, and the average throughput in one hour is reported.

Figure 9 reports the throughput of the systems. For the OLAP workload, ByteGraph's throughput is up to two orders of magnitude higher than that of Neptune and Alibaba GDB. As the OLAP workload contains multi-hop queries, the intermediate result size grows exponentially with the increasing numbers of hops. To handle the heavy workload, ByteGraph maximizes CPU utilization by not only accessing the adjacency lists of multiple vertices in parallel, but also accessing a single adjacency list of any super-vertex in parallel. This high parallelism also leads to high throughput for the OLSP and OLTP workloads which have a large number of simple queries. In addition, ByteGraph does not need to load the entire adjacency list of a super-vertex into memory if the filtering property is matched with the sort key, which saves disk I/Os. Meanwhile, the sorted structure of adjacency list can also achieve fast seek with low latency. Figure 9(b) also shows that the high concurrent write queries do not stall ByteGraph thanks to its MPP architecture and asynchronous processing of write requests. In contrast, AWS Neptune does not scale out well for OLSP because it only uses one instance to accept write requests in a cluster, which becomes its bottleneck in processing massive concurrent write requests.

[1] https://github.com/Aaronchangji/ByteGraph-Paper-Query-Set

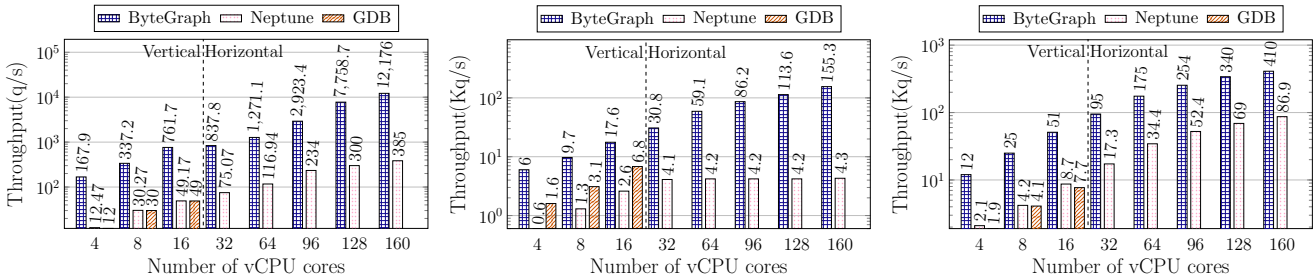


Figure 9: Scalability performance (Vertical: a single machine; Horizontal: 2 to 10 machines, each with 16 cores)

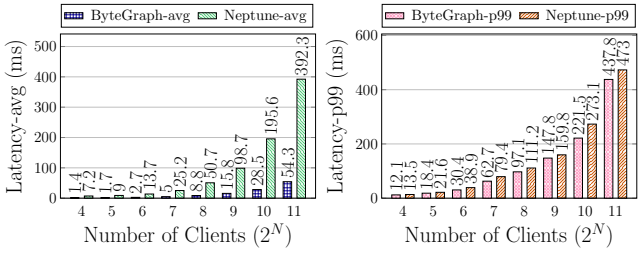


Figure 10: Average and P99 latency

6.2 Query Latency

Next we report the performance of ByteGraph on query latency. As ByteGraph focuses more on the distributed environment, we did not include Alibaba GDB in this experiment. We first compared the average and P99 latency of ByteGraph and AWS Neptune for OLTP. We used 10 nodes and increased the number of clients from 16 to 2,048 for both systems. Figure 10 shows that although the two systems have comparable P99 latency, the average latency of Neptune is 5× to 7× that of ByteGraph as ByteGraph has high parallelism for processing a large number of queries.

We also compared the single-query latency with TigerGraph [20], which is a commercial graph database reported to achieve better performance than most open-source graph databases. To execute a query in TigerGraph, its query template should be first registered into the system, which is called *query install* in TigerGraph. But the main issue is that, during *query install*, all other operations (e.g., query execution, data export, or scale-out) are not allowed, and thus it is impossible to use TigerGraph on high concurrent workloads with a large number of queries with different query templates from our users. We thus selected some queries from Table 3, where Q1-Q3 are 1-hop queries, Q4 is a 2-hop query, Q5 is a 3-hop query, and Q6 is a path query. Due to the page limit, we will release these queries in our project website. We used the Enterprise Edition of TigerGraph with v3.5.0 which was then deployed on five nodes in our cluster using the docker provided by TigerGraph.

Table 4 shows that ByteGraph has shorter query latency than TigerGraph for processing all Q1-Q6, even without counting TigerGraph's long *query install* time (23-30 seconds). For simple queries, i.e., Q1-Q3, ByteGraph enjoys the acceleration brought by sorted edge-tree. For multi-hop traversal queries, i.e., Q4-Q6, ByteGraph performs much better than TigerGraph due to the high parallelism

Table 4: Single-query latency (in *msec*)

System	Q1	Q2	Q3	Q4	Q5	Q6
BG	1.97	1.01	0.64	14.66	14.84	18,087.5
TG(install	26.1s	23.4s	26.5s	23.7s	23.9s	30.7s
+execute)	+3.6	+3.8	+2.6	+4,652.2	+3,440.1	+24,991.3

for accessing the edge-trees of visited vertices. TigerGraph executes a query within a single machine and pulls the required data located on other machines, which introduces a large overhead on distributed computing. We noticed that TigerGraph also provides another execution mode that allows a query to be concurrently executed in a cluster. However, this mode is originally designed to query with a large set of start vertices (e.g., PageRank), while a query with a few start vertices and multi-hop traversal will be severely slowed down running this mode.

6.3 Evaluation on Real Production Workload

In Sections 6.1 and 6.2, we only used a smaller graph as the systems we compared with cannot efficiently handle large graphs at ByteDance. To illustrate that ByteGraph can process large graphs efficiently, we report its performance for processing a typical OLSP workload for 24 hours on a production cluster with 130 nodes (each with 1 TB RAM) connected with 25 Gbps network, where the graph data occupies 218 TB on disk.

Figure 11(a) reports the throughput of read and write queries. The overall throughput can scale from several millions queries per second (q/s) to more than 30 millions q/s because of the high parallelism provided by BGS. When the peak throughput is reached, although the error rate inevitably increases, ByteGraph can keep less than 0.1% error rate. The average error rate is only 0.002% in the entire period of 24 hours and ByteGraph achieves a 99.99% SLA. We also present the latency of read and write queries in Figure 11(b). There is an obvious spike for the P99 latency of write queries, which is caused by the increasing throughput of write queries (i.e., the write throughput increases more than 1 Million q/s). However, the maximum P99 latency for write query is still low (i.e., less than 60ms) and the average latency is barely affected.

Figure 11(c) reports the memory usage and cache hit rate. The memory usage is stable thanks to the proper swap and flush frequency (Section 4.1). ByteGraph does not use up all the memory such that the system has adequate resources when machine failure happens, even if the cache hit rate is only 92%. Moreover, only

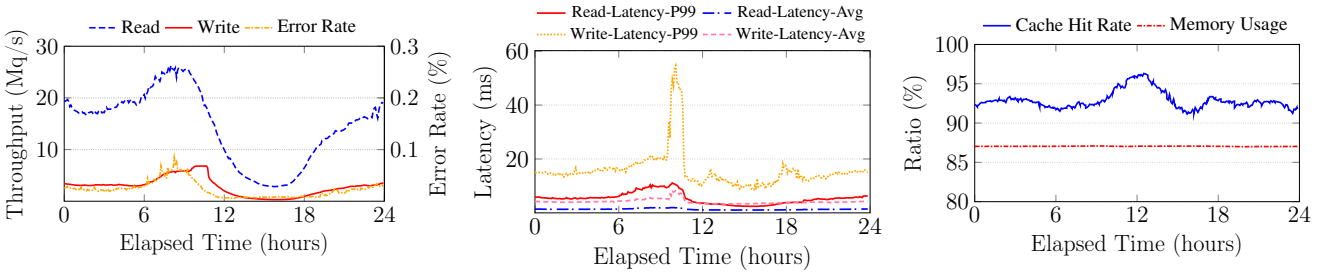


Figure 11: The Throughput, Error Rate, Latency, Cache Hit Rate, and Memory Usage on production workload

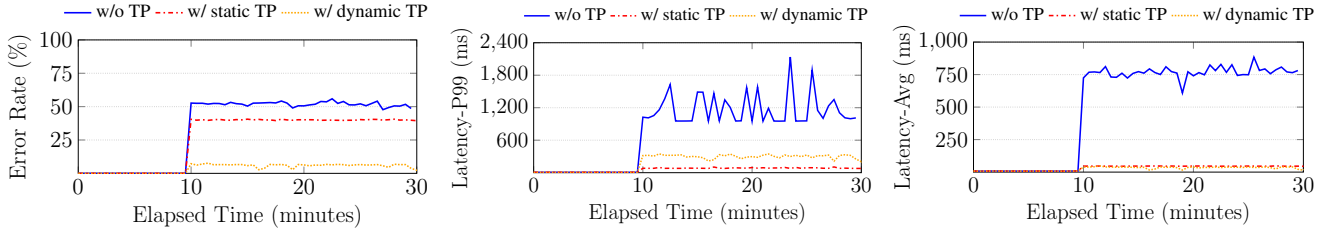
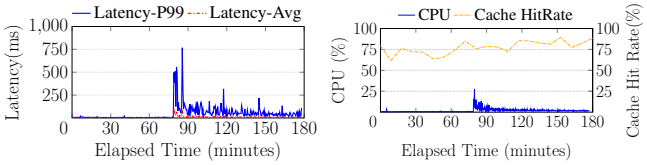
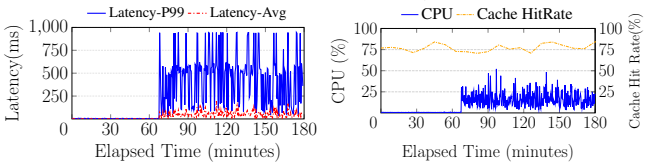


Figure 12: The error rate and latency without Thread Pools (TP), with static TP, and with dynamic TP



(a) w/ adaptive secondary edge-trees



(b) w/o adaptive secondary edge-trees

Figure 13: The latency, CPU usage, and cache hit rate with and without adaptive secondary edge-trees

130TB×87% =113TB memory is used, which indicates that not all the graph data is required to process the workload at any time and less data is loaded into memory with the use of edge-trees.

6.4 Handling Rapid Changes in the Workload

Next we evaluate the effects of two adaptive optimizations in Section 4.3.

6.4.1 Dynamic Thread Pools. We used an OLSP workload by adding more queries that require to scan adjacency lists. Figure 12 reports the error rate and the (average and P99) latency without heavy request thread pools, with static thread pools, and with dynamic thread pools. The results show that, if there is no restriction on heavy requests, the overall error rate can rapidly rise to more than 50% and the P99 latency goes higher than 1 second. If the thread pool for heavy requests are statically set, the average and P99 latency resume back to normal. However, the error rate is still unacceptable since nearly all queries with heavy requests are timeout. Consequently, dynamic thread pools increase the number of threads in the heavy request thread pool to gain more resources. We observe that the error rate of dynamic thread pools decrease to only 5% by squeezing the resources of light requests, and the P99 latency is higher than static thread pools as more heavy requests succeed.

6.4.2 Adaptive Secondary Edge-Trees. We used an OLSP workload by adding filters on edge properties for which there is no matched secondary edge-tree. Figure 13 reports the latency, CPU usage, and cache hit rate with and without adaptive secondary edge-trees. When adaptive secondary edge-trees are used, the latency first significantly increases (767ms at maximum) and then drops back to acceptable value (46ms on average) after several minutes. In this period, building the secondary edge-trees is invoked and queued to wait for the construction. And the CPU usage is not high, because we restrict the available threads for building the secondary edge-trees to reduce the affects to normal query execution. Moreover, when more secondary edge-trees are constructed (i.e., after 120 minutes), the cache hit rate also increases as less data is required to be kept in memory.

6.5 Availability

In this experiment, we demonstrate the availability of ByteGraph by inducing machine failure in the cluster. We used an OLSP workload as high availability is critical for OLSP services in ByteDance. Figure 14 reports the throughput and (average and P99) latency, where the time for machine failure and recovery are marked with two vertical dotted lines. We can observe that although the throughput instantly drops after the machine failure, the P99 latency increases for only a short period (i.e., 2 minutes) and the average latency is insignificantly affected. This is because BGS and BGE are both stateless and weighted consistent hashing can adjust the hash ring

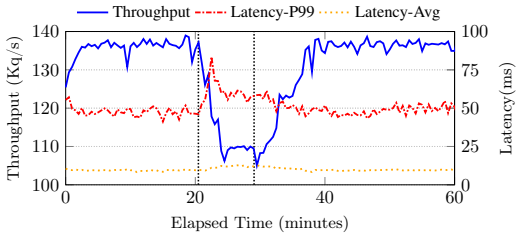


Figure 14: Throughput and latency during machine failure

to balance the workload. When the machine is recovered, we can observe a slight throughput dropping due to the cold cache in the new machine. Due to the adjustment of weighted concurrent hashing, the recovery period lasts longer than the throughput dropping of machine failure, so that the instant peak throughput would not be ingested into the new machines and crash them, which provides more reliability to our system.

7 RELATED WORK

The fast growth of graph data in recent years has attracted major cloud vendors to provide their own graph databases, e.g., AWS Neptune [8], Alibaba GDB [6], and Azure CosmosDB [9]. Alibaba GDB is a single machine system but provides availability with replications. It only allows writes on the master replication while other replications only accept read queries. AWS Neptune distributes read operators to multiple instances for execution, but write operations are executed on a single instance. Both systems cannot handle large scale workloads with high concurrent writes. Azure CosmosDB is a geographically distributed multi-model database, which provides multi-master capability by resolving conflicts with last-write-win policy. However, since CosmosDB stores graph data in a document store, super-vertices will result in large sized documents causing high latency during accessing. Facebook's TAO [14, 19] is a distributed graph store with horizontal scalability and also provides a cache layer in memory to achieve low latency. However, TAO stores vertices and edges in a logical MySQL database with relational tables and thus requires expensive joins in processing multi-hop traversal queries.

There are many open source graph databases [3–5, 10]. AgensGraph [3] uses a relational DBMS (i.e., PostgreSQL [11]), where each row represents an edge. ArangoDB [4] stores a graph in a document store where each vertex/edge is modeled as a document. Neo4j [10] stores vertices, edges, and vertex/edge properties with separate records and link them with pointers. Thus, an edge exists in two double linked-lists that represent the adjacency lists of the two end vertices, which can incur a large number of disk I/Os and random access during adjacency list accessing. JanusGraph [5] (successor of Titan [2]) uses a wide-column store where each vertex is represented by a row containing its properties and neighboring edges. Each property and edge are stored in a cell as key-value pairs. Edges are sorted with a customized sort key constructed with the edge label and sortable edge properties, which results in complicated and space-consuming key-value pairs. Among the above open source graph databases, only JanusGraph is distributed.

A1 [15] and TigerGraph [20] are distributed graph databases that fully utilize main memory. A1 is an RDMA-based in-memory graph database built upon FaRM [21, 22], which achieves serializable graph transaction with Opacity and multi-versioning. Each vertex contains two edge lists whose elements record the address of the real edge data (e.g., edge property). A1 requires RDMA atomic operator to remotely read edge properties if the edge data is not co-located with the vertex. ByteGraph does not consider remote read and locally executes read/write in a single instance. TigerGraph supports massively parallel computation of analytical queries and achieves low disk consumption with data compression. When the server of TigerGraph is created, it tries to load the whole graph into memory, which can achieve low latency and high memory utilization. If the graph cannot fit into the available memory, the excess is spilled to disk. Consequently, users should always reserve memory for the graph data even if the data is not required in query processing. In contrast, ByteGraph passively loads the required graph into memory and actively flushes and swaps data out of memory based on the cluster load.

There are also many impressive research prototypes proposed in recent years [16–18, 23, 26, 33]. GraphflowDB [26] is an in-memory graph database targeting primarily analytical subgraph query workloads equivalent to select-project-join (SPJ) queries over graph data. GraphflowDB proposes list-based processor to avoid expensive data copies on the columnar storage [24], and it also optimizes the worst-case optimal join by mixing traditional binary joins [30]. As our workloads include OLSP with high concurrent updating, pure columnar storage with read-optimized designs are not suitable to handle such workloads. LiveGraph [33] proposes a graph-aware data structure, called Transactional Edge Log (TEL), which achieves sequential scanning on the adjacency lists while supporting transactional updating. However, LiveGraph is not a distributed system, which has limited scalability to handle large graphs in industry. Grasper [16] proposes an execution model, Expert Model, with adaptive parallelism and tailored optimizations on primitive operators to accelerate query processing and achieve high resource utilization. GTran [17] extends Grasper to support distributed transaction with serializable isolation level and in-memory storage design. However, these systems generally do not provide fault tolerance and availability guarantee required at the industry level.

8 CONCLUSIONS

We present ByteGraph, a high-performance distributed graph database that is designed to process OLAP, OLSP and OLTP workloads on large scale graphs at ByteDance. We show that ByteGraph achieves competitive performance compared with Amazon Neptune, Alibaba GDB, and TigerGraph thanks to its high parallelism for query execution and data accessing. We also demonstrate the high performance of ByteGraph in terms of scalable throughput, low latency and low error rate on a massive production graph. In addition, ByteGraph achieves stable performance under rapid changes in the workload with its two adaptive optimizations. ByteGraph also provides good fault tolerance under machine failure.

ACKNOWLEDGMENTS

We thank the reviewers for their constructive comments and suggestions that have helped improve the quality of the paper.

REFERENCES

[1] 2007. *Online serving.* https://cloud.google.com/vertex-ai/docs/featurestore/serving-online.

[2] 2015. *Titan.* https://titan.thinkaurelius.com/.

[3] 2021. *AgensGraph.* https://bitnine.net/.

[4] 2021. *ArangoDB.* https://www.arangodb.com/.

[5] 2021. *JanusGraph.* https://janusgraph.org/.

[6] 2022. *Alibaba GDB.* https://www.aliyun.com/product/gdb/.

[7] 2022. *Apache TinkerPop: Gremlin Query Language.* https://tinkerpop.apache.org/gremlin.html.

[8] 2022. *AWS Neptune.* https://aws.amazon.com/neptune/.

[9] 2022. *Azure Cosmos DB.* https://docs.microsoft.com/en-us/azure/cosmos-db/graph/graph-introduction.

[10] 2022. *Neo4j.* https://neo4j.com/.

[11] 2022. *PostgreSQL.* https://www.postgresql.org/.

[12] 2022. *RocksDB.* http://rocksdb.org/.

[13] Charles Bell, Mats Kindahl, and Lars Thalmann. 2014. *MySQL High Availability - Tools for Building Robust Data Centers, 2nd Edition.* O'Reilly. http://shop.oreilly.com/product/0636920026907.do

[14] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C. Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkateshwaran Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *2013 USENIX Annual Technical Conference, San Jose, CA, USA, June 26-28, 2013.* USENIX Association, 49–60. https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson

[15] 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

[16] Hongzhi Chen, Changji Li, Juncheng Fang, Chenghuan Huang, James Cheng, Jian Zhang, Yifan Hou, and Xiao Yan. 2019. Grasper: A High Performance Distributed System for OLAP on Property Graphs. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019.* ACM, 87–100. https://doi.org/10.1145/3357223.3362715

[17] Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang. 2022. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture. *Proc. VLDB Endow.* 15 (2022).

[18] Hongzhi Chen, Bowen Wu, Shiyuan Deng, Chenghuan Huang, Changji Li, Yichao Li, and James Cheng. 2020. High Performance Distributed OLAP on Property Graphs with Grasper. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020.* ACM, 2705–2708. https://doi.org/10.1145/3318464.3384685

[19] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: Layering Atomic Transactions on Facebook's Online TAO Data Store. *Proc. VLDB Endow.* 14, 12 (2021), 3014–3027. http://www.vldb.org/pvldb/vol14/p3014-cheng.pdf

[20] 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

[21] Aleksandar Dragojevic, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014.* USENIX Association, 401–414. https://www.usenix.org/conference/nsdi14/technical-sessions/dragojevi%C4%87

[22] Aleksandar Dragojevic, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015.* ACM, 54–70. https://doi.org/10.1145/2815400.2815425

[23] Ayush Dubey, Greg D. Hill, Robert Escriva, and Emin Gün Sirer. 2016. Weaver: A High-Performance, Transactional Graph Database Based on Refinable Timestamps. *Proc. VLDB Endow.* 9, 11 (2016), 852–863. https://doi.org/10.14778/2983200.2983202

[24] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *Proc. VLDB Endow.* 14, 11 (2021), 2491–2504. http://www.vldb.org/pvldb/vol14/p2491-gupta.pdf

[25] 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. https://doi.org/10.14778/3415478.3415550

[26] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017.* ACM, 1695–1698. https://doi.org/10.1145/3035918.3056445

[27] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. 1997. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on the Theory of Computing, El Paso, Texas, USA, May 4-6, 1997.* ACM, 654–663. https://doi.org/10.1145/258533.258660

[28] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. 2014. Logical Physical Clocks. In *Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings (Lecture Notes in Computer Science)*, Vol. 8878. Springer, 17–32. https://doi.org/10.1007/978-3-319-14472-6_2

[29] Jianchuan Li, Peiquan Jin, Yuanjin Lin, Ming Zhao, Yi Wang, and Kuankuan Guo. 2021. Elastic and Stable Compaction for LSM-tree: A FaaS-Based Approach on TerarkDB. In *CIKM '21: The 30th ACM International Conference on Information and Knowledge Management, Virtual Event, Queensland, Australia, November 1 - 5, 2021.* ACM, 3906–3915. https://doi.org/10.1145/3459637.3481913

[30] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proc. VLDB Endow.* 12, 11 (2019), 1692–1704. https://doi.org/10.14778/3342263.3342643

[31] Vahab S. Mirrokni, Mikkel Thorup, and Morteza Zadimoghaddam. 2018. Consistent Hashing with Bounded Loads. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018.* SIAM, 587–604. https://doi.org/10.1137/1.9781611975031.39

[32] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, Paul Bardea, Amruta Ranade, Ben Darnell, Bram Gruneir, Justin Jaffray, Lucy Zhang, and Peter Mattis. 2020. CockroachDB: The Resilient Geo-Distributed SQL 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, 1493–1509. https://doi.org/10.1145/3318464.3386134

[33] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proc. VLDB Endow.* 13, 7 (2020), 1020–1034. https://doi.org/10.14778/3384345.3384351