Galaxybase: A High Performance Native Distributed Graph Database for HTAP

Bing Tong
CreateLink & HKUST(GZ)
tongbing@createlink.com
btong799@connect.hkust-gz.edu.cn

Yan Zhou*
CreateLink
zhouyan@createlink.com

Chen Zhang
CreateLink
zhangchen@createlink.com

Jianheng Tang
HKUST(GZ)
jtangbf@connect.ust.hk

Jing Tang
HKUST(GZ)
jingtang@ust.hk

Leihong Yang
CreateLink
yangleihong@createlink.com

Qiye Li
CreateLink
liqiye@createlink.com

Manwu Lin
CreateLink
linmanwu@createlink.com

Zhongxin Bao
CreateLink
baozhongxin@createlink.com

Jia Li*
HKUST(GZ)
jialee@ust.hk

Lei Chen
HKUST(GZ)
leichen@ust.hk

ABSTRACT

We introduce Galaxybase, a native distributed graph database that addresses the increasing demands for processing large volumes of graph data in diverse industries like finance, manufacturing, and government. Designed to handle the requirements of both transactional and analytical workloads, Galaxybase stands out with its novel data storage and transaction mechanisms. At its core, Galaxybase utilizes a Log-Structured Adjacency List coupled with an Edge Page structure, optimizing read-write operations across a spectrum of tasks such as graph traversals and single edge queries. A notable aspect of Galaxybase is its execution of custom distributed transaction modes tailored for HTAP transactions, allowing for the facilitation of bidirectional and interactive transactions. It ensures data integrity and minimal latency while enabling simultaneous processing of OLTP and OLAP workloads without blocking. Experimental results show that Galaxybase achieves high throughput and low latency in both OLTP and OLAP workloads, across various graph query scenarios and resource conditions. Galaxybase has been deployed in leading banks, education, telecommunication and energy sectors in China, consistently maintaining robust performance for HTAP workloads over the years.

PVLDB Reference Format:
Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, and Lei Chen. Galaxybase: A High Performance Native Distributed Graph Database for HTAP. PVLDB, 17(12): 3893 - 3905, 2024.
doi:10.14778/3685800.3685814

* Corresponding Authors.

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685814

1 INTRODUCTION

A graph database [37] is a type of database management system specifically designed to store, manage, and query complex relationships between data entities. Unlike conventional relational databases, graph databases employ vertices, edges, and properties to model data entities and their relationships. It allows for enhanced flexibility and performance in handling structured and highly interconnected data, making them particularly well-suited for applications in fields such as social networking [8, 10, 17, 32], energy network optimization [24, 42], financial fraud detection [23, 34, 40], and knowledge graphs [7, 41].

Many graph databases encounter performance challenges in processing graph queries and transactions due to their design or functional limitations. Non-native databases often use established non-graph backends. For example, A1 [11] utilizes Key-Value stores. Titan [1] and its successor, JanusGraph [3], are based on the wide-column store, while ArangoDB [2] and OrientDB [36] employ document stores for graph representation. Although non-native graph storages rely on mature non-graph backends like HBase [19], which are well-understood operationally, they typically struggle with handling efficient graph-specific queries, particularly in graph traversal scenarios. Conversely, native graph databases with their index-free adjacency, such as Neo4j [5] and TigerGraph [16], significantly enhance traversal performance. However, Neo4j exhibits poor scalability and struggles to meet high throughput and low latency requirements on trillion-scale graphs. TigerGraph, focusing on in-memory architectures, encounters difficulties with large graphs in low-memory environments. Additionally, native graph databases also fall short of single edge queries, as they rely on traversal to locate a specific edge.

Beyond handling graph-specific queries, another vital feature of graph databases is their ability to preserve integrity and correctness during concurrent operations in various scenarios. A key capability

is the dual support for Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). Our usage statistics show that 70% of tasks involve Hybrid Transaction/Analytical Processing (HTAP) [33], 20% are dedicated to OLTP, and the remaining 10% to OLAP. Among existing systems, G-Tran [14] is notably adept at OLTP tasks and prioritizes transactional integrity, while Grasper [13] excels in managing OLAP transactions. However, using separate systems for distinct OLTP and OLAP tasks can double costs in terms of development, deployment, and maintenance.

Faced with the unique challenges of processing graph queries and transactions, we developed Galaxybase[1], a new native distributed graph database. Galaxybase features two distinct storage structures, optimized for read and write performance. The first is a Log-Structured Adjacency List, which employs adjacency lists for sequential data scanning and batch writing to reduce read/write amplifications. The second structure, Edge Page, co-locates edges for the same vertex and maintains local order within each page by type and direction while ensuring global order across all edges. This design supports efficient graph traversal in various directions and types, as well as quick and accurate single edge queries.

As a distributed graph database deployed in production-grade environments, Galaxybase is designed to handle a variety of scenarios and data scales effectively. It supports transactions using Two-Phase Commit (2PC) [26, 38] and Raft [30] protocols to ensure atomicity and durability. The system maintains isolation levels from read-committed to serializable for OLTP workloads using Two-Phase Locking (2PL) [22]. Galaxybase integrates bidirectional and interactive transactions, aligning with the unique storage structures and user demands of graph databases. For OLAP workloads, it employs Multi-Version Concurrency Control (MVCC) [35] visibility checks with lock-free mechanisms to maintain serializable snapshot isolation.

Our experiments with OLTP and OLAP workloads demonstrate that Galaxybase delivers strong performance in both single-machine and distributed setups. It achieves throughput of up to 50,000 queries per second (q/s) in single-machine mode and 85,000 q/s in distributed mode, significantly surpassing baseline graph databases. In terms of scalability, Galaxybase achieves throughput that is up to an order of magnitude higher than that of baseline graph databases. It also shows efficiency in edge queries, operating three times faster than its closest competitor. Furthermore, Galaxybase handles queries effectively in low-memory environments, enabling large graph loading and complex query execution without out-of-memory issues. Additionally, we processed a trillion-scale dataset that includes 5 billion account vertices and 5 trillion transaction edges using only 50 machines, each equipped with 12 CPUs and 128GB of memory, achieving multi-hop query results in seconds.

In tracing these endeavors, our paper consolidates the following contributions:

- We introduce Galaxybase, a high-performance, native distributed graph database designed specifically for HTAP scenarios. It provides an efficient, robust, and scalable solution for managing complex graph data.

[1]https://www.createlink.com

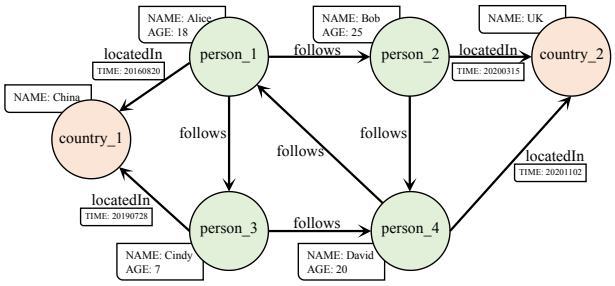

Figure 1: An example of property graph

- On the storage front, we propose the Log-Structured Adjacency List, an approach for sequential disk reads and writes that dramatically reduces read/write amplifications. Complementing this, our Edge Page design enhances graph traversal efficiency, allowing for the effective handling of edges in various directions and types, while also enabling quick and accurate single edge queries.
- On the transaction front, we implement distributed transactions for OLTP workloads using bidirectional and interactive methods. Additionally, we manage OLAP workloads with lock-free methods, allowing OLTP and OLAP workloads to run concurrently without causing blocks.
- In the distributed mode, Galaxybase achieves a throughput of up to 85,000 queries per second in OLTP workloads, and its performance in OLAP workloads exceeds competitors by an order of magnitude. This high efficiency is sustained even under restricted memory resources, enabling the execution of complex queries in environments with limited capacity.

2 BACKGROUND AND DESIGN PRINCIPLE

Reflecting on the challenges and limitations of current graph databases outlined in Section 1, this section delves into the motivation and key factors in crafting Galaxybase. Our primary objective is to build a unified system that demonstrates exceptional performance, availability, scalability, and robust transaction capabilities.

Galaxybase utilizes the property graph model [6], where vertices and edges can possess a variety of properties. Based on this model, we develop a Ming Dynasty literature knowledge graph for universities to enhance literary research and teaching, build a power grid knowledge graph for the State Grid to ensure accurate and stable power dispatch strategies, and implement a financial fraud detection graph for banks to enhance security and more effectively identify fraudulent activities. As illustrated in Figure 1, in a social network using the property graph model, each vertex/edge is assigned a type (e.g., `person`, `country`, `follows`, `locatedIn`), alongside a set of properties (e.g., `NAME:Alice` and `TIME:20201102`).

Graph databases organize data through edges, offering the significant advantage of native and efficient support for graph traversal queries. These queries navigate the graph from a specified vertex to a predetermined depth or target vertex. For example, as depicted in Figure 1, a graph traversal query starting from vertex `person_1` with a depth of 1 and a relational constraint of `follows` would identify all followers of `person_1`. Relational databases depend

on index lookups and table scans to navigate through the neighborhoods of a vertex. Graph databases typically adopt a feature called index-free adjacency, which co-locates a vertex with its edges, avoiding joins between a vertex table and an edge table. This allows for constant-time traversal of edges [5, 37], providing a substantial edge performance over relational databases.

To enhance the design of a graph database for Galaxybase customers, we summarize the following design principles:

Improving Single Edge Query Efficiency. While the introduction of index-free adjacency can speed up graph traversal, it is not a free lunch. This feature complicates the fundamental operations of single edge queries, such as locating, reading, and updating specific edges. Early graph databases, like LiveGraph [43], do not have a dedicated approach for efficiently handling single edge queries and depend on traversal to locate a specific edge. This becomes particularly inefficient when managing edges connected to a 'super vertex' in a social network, which may have millions of followers. Neo4j [5] employs index-free adjacency via a linked list method, but this often requires random disk access to read subsequent edges, thereby reducing read performance. Many recent non-native graph databases, such as JanusGraph [3] and NebulaGraph [4], rely on LSM-Tree [31] as their storage structure to support more efficient single-edge queries through KV storage. Nonetheless, the hierarchical nature of LSM-Tree storage requires merging multiple layers to reach the target data, which can result in read amplification. The efficiency of single edge queries remains a significant consideration in the design and optimization of graph database systems, driving ongoing innovations and enhancements in this domain.

Optimizing Storage for Read-heavy Scenarios. OLTP workloads emphasize real-time data retrieval and modification, including graph neighbor traversal, edge queries, and data updates. Conversely, OLAP is centered around in-depth data analysis, focusing on tasks such as k-hop neighbors and pathfinding. Our analysis shows that OLAP operations are more prevalent among our business customers, resulting in a higher frequency of read operations compared to write operations. For example, a major Chinese bank employing Galaxybase for credit card fraud prevention activities illustrates this trend. When a new credit card is issued, dozens of vertices and edges are inserted into the graph, followed by more than 10 analysis tasks. In this case, write operations account for approximately 5% of the workload, while reads constitute about 95%. Consequently, conventional storage solutions like LSM-Tree are unsuitable for this read-heavy environments, motivating us to develop a novel storage architecture tailored for scenarios with a higher demand for read operations.

Balancing Computation and Communication in OLAP. OLAP tasks, particularly k-hop and path-finding queries, often encounter significant network communication overhead due to the distributed storage of neighboring vertices. This challenge highlights the need for innovative data partitioning strategies that effectively balance computational load and communication demands in distributed graph database. Figure 2 illustrates the computation and communication times of k-hop queries in a graph, distributed with varying graph partition sizes across a three-server cluster. With a graph partition size of 1 (i.e., random storage of the neighbors), each server incurs equivalent computation and communication times. However, larger graph partition sizes lead to data 'hot-spotting',

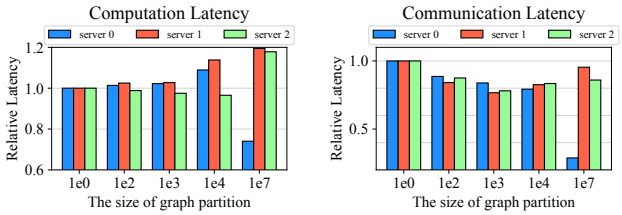

Figure 2: The computation and communication latency for k-hop queries on the Twitter social network, distributed across three servers.

as neighborhoods are more likely concentrated on the same computing server as the queried vertex. This results in imbalanced computational loads across servers, where the execution time of the slowest server becomes the bottleneck in parallel computations. To alleviate this, the development of effective graph partitioning strategies is crucial, tailored to suit specific use cases for more balanced communication and computation.

Enhancing HTAP Performance. While most distributed graph databases have adeptly implemented transaction support for OLTP workloads, support for OLAP workloads is frequently under-served. Using OLTP transaction protocols for OLAP workloads can result in increased latency. Conversely, if OLAP workloads bypass transactions, transaction consistency can be disrupted during data updates, thereby affecting the reliability of analytical outcomes. Given that OLAP workloads are predominantly read-only, it is crucial to implement non-blocking read-only transactions similar to those utilized in TAO [10, 15]. To effectively support HTAP in distributed graph databases, we also need to offer specialized support for both OLTP and OLAP workloads, ensure suitable isolation levels for real environments, and handle various transaction types based on the specific nature of each workload.

3 ARCHITECTURE OVERVIEW

The architecture of Galaxybase, as depicted in Figure 3, is designed as a distributed graph database cluster that leverages a native approach. It features a three-level storage model for data management, and two distinct transaction modules optimized for OLTP and OLAP workloads, respectively. Galaxybase adopts load balancing mechanisms to evenly distribute client requests across multiple servers. Each server comprises of an HTAP compute layer and a storage layer. The compute layer is divided into two primary modules: OLTP for transactional queries and OLAP for analytical computations. The storage layer consists of a cache module for rapid data retrieval and a disk storage module for long-term persistence. Within the disk storage module, data shard and Write-Ahead Logging (WAL) are utilized to ensure data durability.

The storage model of Galaxybase is organized into three levels: shard, partition, and page. Each shard resides within one server of the cluster. Within each shard, the system segments data into smaller, manageable units called partitions. For each partition, Galaxybase employs the Log-Structured Adjacency List for storing vertex and edge data, realizing index-free adjacency for graph traversal and facilitating sequential read and write operations from

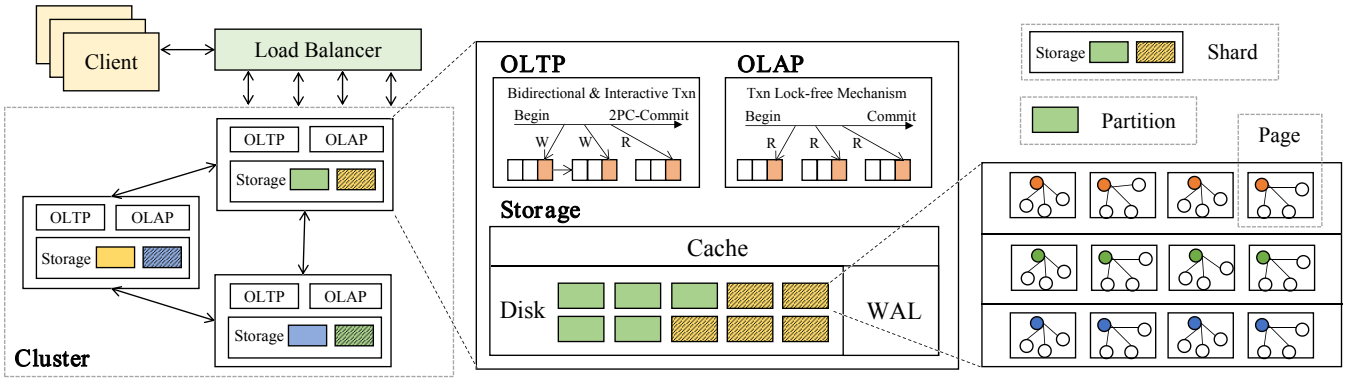

Figure 3: The architecture of Galaxybase

the disk. These partitions can be distributed and replicated across various servers, improving data availability and fault tolerance. To further optimize edge query performance, there is a paging process in each partition to create an Edge Page for each vertex. This process divides the neighboring edges of each vertex into multiple pages to address different queries and reduce conflicts for edges. The details of this storage mechanism are discussed in Section 4. Additionally, we outline two distributed storage strategies based on partitions to reduce communication in Section 5.

In the HTAP design of Galaxybase, different transaction processing modules feature different workloads: read-write transactions support OLTP, while read-only transactions are used for OLAP. We employ bidirectional and interactive transactions to accommodate graph-specific scenarios and use lock-free mechanisms to ensure OLTP and OLAP workloads do not interfere with each other. We introduce the design details of HTAP transactions for graph-specific scenarios in Section 6.

4 DATA STORAGE

4.1 Overview

Galaxybase is designed as a versatile storage structure to accommodate the diverse hardware preferences of our customers. In production environments, about 70% use SSDs, while 30% use HDDs, and data volumes can sometimes exceed 100 TB. For efficient disk operations, the data structure in Galaxybase is organized into three distinct levels: shard, partition, and page. In the partition level, Galaxybase proposes the Log-Structured Adjacency List (Section 4.2), realized through three key strategies: (1) organizing vertices and edges as adjacency lists for reading efficiency, (2) storing vertices and edges as multi-version data for consistency and system availability, and (3) writing data to disk in batches for writing performance. In the page level, Galaxybase employs the Edge Page structure (Section 4.3) to achieve high performance in fundamental operations such as graph traversal and single edge query.

As shown in Figure 4, Galaxybase employs two storage modules within each server of the cluster: a cache module in memory, designed to reduce disk I/O for read and write operations, and a durable storage module persisting on disk. The latter includes data sharding and WAL. The process for write and read operations is as follows:

Write operations. The green arrows illustrate the write process. During a write operation, data is first synchronously written to the cache (steps ①) and to the WAL (steps ②). The cache facilitates the efficiency of read-write operations, while the WAL ensures data persistence to disk, thereby preventing data loss. A write request is considered complete once the data is successfully written to both the cache and the WAL. When the written data in the cache meets the criteria for eviction, it is then batch-written in an asynchronous and sequential manner into new data partitions within the shard, effectively replacing the older partitions (step ③). In the event of a system failure or similar incidents, the data stored in the WAL is utilized for recovery purposes.

Read operations. The orange arrows represent the read process. To retrieve data efficiently, the system initially searches for it in the cache (step ①). If the data is complete in the cache, it is returned directly; if not, the system retrieves it from shard on disk (step ③), merging and returning the data from both the cache and shard. Data in the WAL is set to be synchronized to the cache by default (step ②).

To achieve a balance between performance and stability, we selectively evict vertex or edge data from the cache under three conditions: (1) the memory usage of the cache affects the system's stability, (2) the least read data is selected for eviction utilizing the Least Recently Used (LRU) strategy, or (3) the updated data in the cache reaches a threshold. The cache is designed to adjust intelligently based on the current resource situation. Condition (1) ensures system stability is maintained even under limited memory. Condition (2) employs hot data management to retain frequently queried data longer, thereby enhancing read efficiency. Condition (3) ensures that new data can be written in batches to promote efficient write operations.

To implement transactions, data in each partition is stored in multiple versions, each identified by a timestamp and ordered accordingly. When reading, a binary search locates the most suitable version with a timestamp less than or equal to the specified one, or the latest version if no timestamp is specified. Every update to a vertex or edge creates a new version. New data operations at the vertex or edge level are temporarily stored in the cache and WAL.

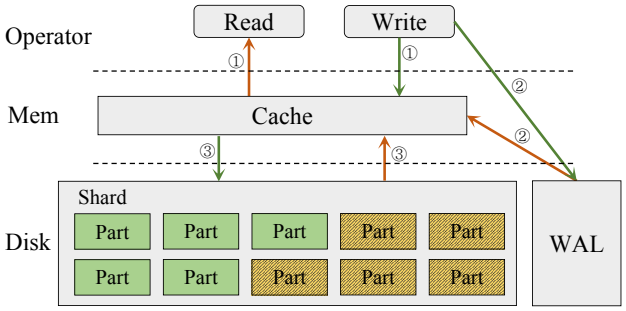

Figure 4: The storage and R/W operators in a single machine

When the update data cache reaches its threshold, the system creates a new disk copy of the entire partition, sequentially integrating both old and new data versions. A garbage collection thread removes data versions older than the timestamp of the earliest active transaction.

4.2 Log-Structured Adjacency List

Within each shard of Galaxybase, there are smaller storage units known as partitions. Each partition utilizes the Log-Structured Adjacency List to enable efficient reading and writing from the disk. The adjacency list allows efficient sequential reading, while the log structure reduces random writes. This integration addresses both read and write amplification. As shown in Figure 5, a partition is illustrated, which is dedicated to storing a single type of vertex and its edges. Within these partitions, vertices and their properties are stored together, while the neighboring edges of each vertex are grouped in an Edge Page. Every vertex is assigned a unique ID, enabling efficient retrieval of a vertex's properties or all its edges through the stored offset. For example, consider the vertex with ID 2 of type `person` in Figure 1. This partition stores the vertex with ID 2, along with its properties – `Bob` and 25. Neighboring edges related to the vertex with ID 2, such as `Alice follows Bob`, `Bob follows David`, and `Bob locatedIn UK`, are stored in Bob's Edge Page within this partition. The data of vertex with ID 2, encompassing both properties and its edges, can be accessed by querying the vertex or edge offset using ID 2.

Read optimization. To optimize data reading, we organize vertices and edges within partitions using adjacency lists, a method that greatly minimizes the need for random disk access and reduces storage space. This approach enhances the efficiency of batch data reads, such as performing graph traversal queries in Edge Pages or accessing all vertices of a specific type. However, finding a specific vertex within the VertexData or identifying its neighboring edges (contained in one Edge Page) within the EdgeData, when using adjacency lists, can present a challenge.

To address the retrieval of specific vertex properties or neighboring edges, many databases, such as JanusGraph, adopt LSM-Tree that uses ordered storage for disk data lookup. Within the LSM-Tree, data is stored in Sorted String Tables (SSTables) on the disk. This data is organized in a sorted order, so that whenever the data is read its time complexity will be `O(Log(n))` in the worst case, where n is the number of records on the disk.

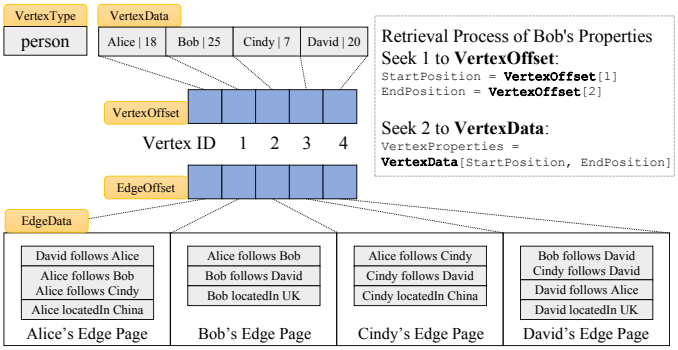

Figure 5: The storage structure and retrieval process in a single partition by Log-Structured Adjacency List

To overcome these limitations, we assign each vertex within a partition a unique ID, incrementing sequentially and incrementally. Given the variable lengths of properties in real-world data, vertex data is non-fixed-length, making computation-based direct data location by ID impractical. We instead use an offset storage mechanism based on these unique IDs - VertexOffset for vertices and EdgeOffset for Edge Page. This system enables efficient retrieval of the starting and ending positions of vertex properties or their neighboring edges from VertexOffset/EdgeOffset using the ID. This process requires only two data seeks, as shown in Figure 5, and is achieved with an `O(1)` time complexity. To reduce the storage space for offset, our approach involves storing only the relative position offsets for VertexData and EdgeData from the start of the partition, instead of their absolute positions. This method cuts storage requirements from 8 bytes Integer to 4 bytes Integer.

Write optimization. To optimize data writing, we implement a batch writer mechanism using a log-structured approach within new partitions. These partitions are activated only after the writing process is fully complete, at which point the old partitions are removed. Data is partitioned based on vertex type, with an initial allocation of 32 partitions for each type. Each partition functions independently for batch writing to prevent overload and maintain performance. The number of partitions is dynamically adjusted based on the cluster's read/write workloads.

When the cache eviction mechanism is triggered, data from write operations in the cache is transferred to partitions. Each partition evaluates its disk I/O for read and write operations since the last rewrite whenever a new write operation occurs. If the ratio of read I/O to write I/O falls below a specified, configurable threshold, the existing partition is split into two. In write-heavy workloads, the total number of partitions is increased to reduce the data volume involved in batch writing. Conversely, if this ratio between adjacent partitions exceeds the threshold, the data from these partitions is merged and written into a single new partition. In read-heavy workloads, the total number of partitions is decreased to reduce the need for accessing multiple partitions in a non-sequential manner when reading a specific type of vertex, thereby optimizing the reading process.

The partition storage units bring greater flexibility to the system, enabling it to handle large-scale graph data and high parallelism

queries. These partitions, utilized in the adjacency list and offset storage structure, are instrumental in efficiently organizing and managing data while reducing storage space. We employ a log-structured approach for sequentially writing partition batches to disk. And the number of partitions is dynamically adjusted based on the current read/write workloads. This adaptability is key to achieving optimal performance in various data processing scenarios. Therefore, we utilize the Log-Structured Adjacency List, which enables us to balance read and write performance effectively, catering to the specific demands of our users.

4.3 Edge Page

The majority of queries in a graph database involve retrieving neighboring edges, making the performance of neighbor retrieval a crucial factor. To enhance performance, edge storage strategies should have two key features: (1) reducing the number of disk seeks during queries, and (2) avoiding redundant data reads when filtering based on different query conditions. To achieve these objectives, as shown in Figure 6, we introduce a fine-grained paging strategy known as Edge Page, designed to store all neighboring edges for each vertex. We also propose various query strategies for different neighbor query scenarios.

Similar to vertex storage, within each partition, the unique ID of every vertex utilizes an Edge Offset to accurately locate the corresponding Edge Page. Within this Edge Page, adjacency list is employed to store all neighboring edges of the vertex. This means that to traverse over all the neighbors of a vertex, one simply traverses through the edge collection within the Edge Page without frequent disk seeks.

Traversing a vertex's neighbors without filtering. This involves directly traversing the edge data within the entire Edge Page for the given Vertex ID.

To mitigate the generation of numerous intermediate results across various edge types and directions, each Edge Page organizes edges based on type, direction, and size. As shown in Figure 6, the first page stores edges of the type `follows` with an `In` direction. To facilitate locating these edges, we introduce an Offset within the Edge Page and design a unique ID structure, the Edge ID, to locate specific edges by direction, type, and other parameters. This Edge ID comprises the starting Vertex ID, direction, type index, ending Vertex ID, and edge index. The parameters utilize in constructing the Edge ID are defined as follows:

- `fromId`: The unique ID of the starting vertex.
- `direction`: The direction of the edge, represented as a binary integer (e.g., 0 for `In`, 1 for `Out`).
- `typeIndex`: The index representing the type of the edge, corresponding to different edge types like `follows`, `locatedIn`, etc.
- `toId`: The unique ID of the ending vertex.
- `edgeIndex`: The sequence index of the edge, a unique number within the set of all edges with the same `fromId` and `toId`.

Within each Edge Page, we propose a Multi-Dimensional Sorted method based on this Edge ID, ensuring that edges are arranged in a specific priority order of `fromId`, `direction`, `typeIndex`, `toId`, and `edgeIndex`. Our edge storage strategy directly applies this structure, naturally grouping edges that share the same type and direction. This ensures they are ordered both locally within each type and direction and globally across the entire Edge Page.

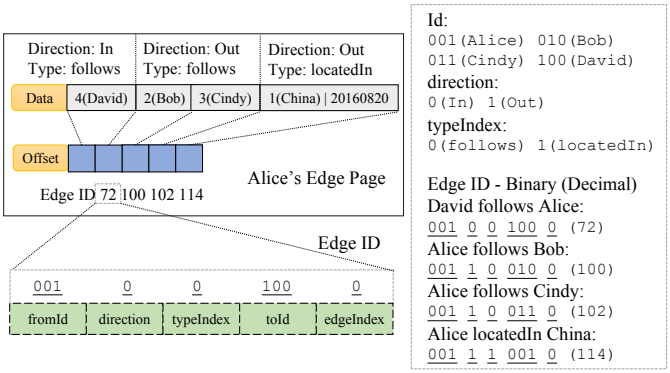

Figure 6: The storage structure within a single Edge Page and the composition of Edge ID

Traversing a vertex's neighbors in a specified direction. To find neighbors in the `In` direction for `Alice`, first locate the vertex's Edge Page and compute the boundaries for `In` and `Out`. Edges with the same direction are stored consecutively. Given `fromId` as `001`, `direction` as `1`, `typeIndex`, `toId`, and `edgeIndex` as `0`, we obtain `001100000`, which converts to 96 in decimal. Perform a binary search in the Offset to get the located position. Then, traverse the edge data from the starting position to the located position for this vertex. The same applies to traversing `Out` direction neighbors.

Traversing a vertex's neighbors in a specified direction and type. To retrieve neighbors of the type `follows` in the `Out` direction for `Alice`, first locate this vertex's Edge Page. Then compute the start and end in the Offset for `follows` with `Out`. Edges with the same direction and type are stored consecutively. Given `fromId` as `001`, the `direction` as `1`, and `typeIndex` as `0` for start and `1` for end, and other values as `0`, we get `001100000` for the start, which is 96 in decimal, and `001110000` for the end, which is 112 in decimal. Perform a binary search in the Offset to get the located positions. Then, traverse the edge data between the two located positions for this vertex. The same applies to other directions and types of neighbor traversals.

Locating a specific edge. To locate a specific edge by its Edge ID, first find the vertex's Edge Page. Then, perform a binary search using the Offset within this Edge Page to locate the edge precisely.

Our Edge Page storage strategy offers multiple advantages. Firstly, it consolidates all neighboring edges of a vertex within a single Edge Page, thereby avoiding frequent disk seeks and enhancing performance. Secondly, each Edge Page ensures that edges of the same type and direction are stored consecutively, facilitating efficient handling of various queries. Thirdly, the composition of Edge ID for orderly storage enables the quick and accurate location of specific edges.

5 DISTRIBUTED STORAGE STRATEGIES

In distributed systems, excessive inter-server communication not only diminishes network efficiency but also undermines runtime

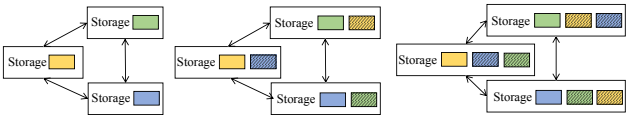

Figure 7: 1, 2, or 3 replicas in different clusters with 3 servers

performance. Effective graph data partitioning is pivotal in mitigating communication overhead. Hence, an optimal data partitioning strategy must address both communication and data hot-spotting concerns, recognizing that the ideal partitioning approach may vary across datasets and query scenarios. To address this, we introduce two strategies in this section: Data Partitioning and Data Replication.

5.1 Data Partitioning Strategies

In the realm of distributed systems within graph databases, the data partitioning strategy is crucial for system performance. Given the frequent need for multi-hop queries in graph databases, this often involves round-trip communication between different servers to retrieve data for the next hop. By effectively aggregating relevant vertices into the same partitions, a significant reduction in communication overhead in distributed systems is achieved [9, 39]. In this context, Galaxybase introduces two partitioning strategies to further improve system performance.

Built-in Graph Partitioning Strategy. Our graph database supports a variety of built-in partitioning algorithms. Users may select from: (1) hashing, which is used by default to ensure effective load balancing in data processing, (2) METIS [25], which partitions based on graph structure, and (3) METIS with property, which partitions based on graph structure and vertex/edge properties. Users can choose the most suitable partitioning algorithm based on their data and query scenarios before importing the dataset into the graph database.

Customized Graph Partitioning Strategy. If the built-in graph partitioning strategies do not meet users' requirements, they can customize their own. We offer two customization options: (1) users can specify a property as the partition identifier during data import if they wish to assign the same partition to data with the same property, and (2) even without prior knowledge of the original data, users can analyze the data within the graph database using default or custom algorithms to insert property identifiers for each vertex. Subsequently, new graphs can be generated based on these assigned property identifiers.

5.2 Data Replication Strategies

Each partition within Galaxybase supports replication across other shards, meaning that each partition can have multiple replicas stored on different servers. This enhances the system's availability and data redundancy. The partition replication strategy allows setting the number of replicas in the cluster. The relationship between the number of replicas (R), the number of cluster servers (N), and the number of allowable server failures (C) is expressed as $C < R \leq N$. This implies that the number of replicas must be not greater than the number of cluster servers, and in the worst-case scenario of maximum server failures, the number of replicas must exceed this limit to ensure data availability. As shown in Figure 7, in a cluster with three servers, we can configure various numbers of replicas depending on fault tolerance requirements and performance optimization.

It should be noted that as the number of partition replicas per shard increases, fault tolerance improves. However, this also adds complexity to write operations, as updating data requires updating all replicas in the cluster. For most of our customers, fault tolerance is a critical factor. Higher replication is particularly beneficial in scenarios with small data volumes, as the storage costs remain acceptable. Conversely, for applications with large-scale data and high parallelism requirements, it is necessary to balance the number of replicas with performance to support data growth. Therefore, Galaxybase provides users with the flexibility to choose the number of replicas based on their specific needs.

6 DISTRIBUTED TRANSACTION PROCESSING

In distributed systems, managing transactions effectively is vital for maintaining data consistency, particularly in the context of parallel operation processing. Galaxybase introduces transaction support specifically designed for graph databases, recognizing their distinct characteristics. It offers: (1) up to serializable isolation levels for HTAP workloads; (2) bidirectional transactions to maintain consistency across edge data stored on different servers within a cluster for OLTP workloads; (3) support for interactive transactions for OLTP workloads; and (4) implementation of transaction lock-free methods combined with multi-version for OLAP workloads, effectively enabling snapshot analysis.

6.1 OLTP Transaction

OLTP workloads are characterized by their ability to manage high concurrency in read and write operations, while maintaining data consistency to successfully complete transactions. These workloads, particularly in scenarios like financial transactions, exhibit a high demand for efficient transaction processing.

As an edge often spans two partitions in our graph database, creating, updating, or deleting an edge requires modifying both partitions. For instance, consider a credit card application scenario where person VertexA in PartitionA applies for a credit card represented by VertexB in PartitionB. This process would involve the insertion of new edges between VertexA and VertexB in both PartitionA and PartitionB. To optimize latency in such cross-partition edge operations during OLTP processes, we implement a bidirectional transaction method that supports parallel processing of data across both partitions.

In graph database applications, a series of operations often need to be performed within a single transaction. For example, in a credit card application, this involves creating a new credit card VertexA and linking it to existing vertices like persons, phones, or companies. Subsequently, a risk assessment is performed on the new credit card application to determine its validity, which involves reading VertexA and its k-hop neighbors. Therefore, we implement interactive transactions to ensure this.

In distributed systems, replicas play a vital role in enhancing fault tolerance capability. To ensure consistency across the system,

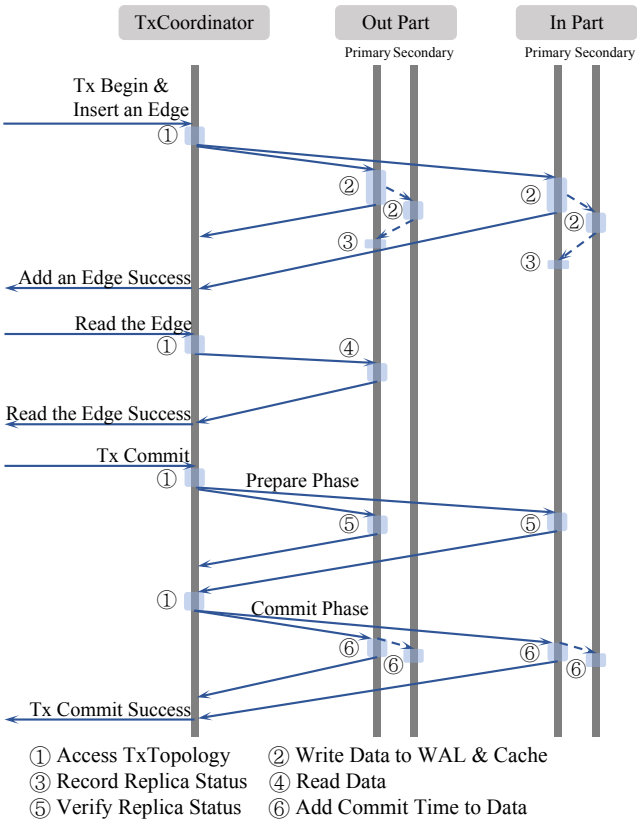

Figure 8: The OLTP transaction of Galaxybase

modifications must be replicated to both the primary replica and its secondary replicas, which can increase the latency of write operations. To deal with this, in one interactive transaction, a user can execute a sequence of operations, with immediate synchronization of write operations to the primary replica for consistency and instant visibility to the user. Replication to other replicas is managed asynchronously and finishes most of the replication when the transaction is committed. This approach guarantees eventual consistency across both primary and secondary replicas.

Galaxybase supports OLTP workloads with transaction isolation levels ranging from read committed to serializable. It utilizes Two-Phase Locking (2PL) [22] and Multi-Version Concurrency Control (MVCC) [35], along with bidirectional and interactive transactions. Figure 8 illustrates the process of a read-write transaction in an OLTP workload within Galaxybase. It involves an interactive transaction that includes edge insertion and subsequent data retrieval. The roles of various components in transaction processing are outlined below:

TxCoordinator is the transaction coordinator component available on each distributed server. Within a transaction, a set of operations is sequentially handled by the TxCoordinator. For each transaction, the TxCoordinator generates a unique transaction ID (TxId) and manages client CRUD requests, directing the relevant operations to the appropriate partitions. Every TxCoordinator maintains a list called **TxTopology** to track the partitions involved in these operations, facilitating the validation of changes during the commit phase. The TxCoordinator manages partitions in different directions to support the implementation of bidirectional transactions.

Part (Partition) forms the fundamental unit in transactional processing. Each edge is associated with Parts in both Out and In directions, encompassing PrimaryPart and SecondaryPart that share the identical data. A notable challenge arises from the fact that these partitions might be distributed across different servers, presenting difficulties in maintaining consistency across them.

In Galaxybase, an example of an interactive transaction process for OLTP is illustrated in Figure 8 as follows:

Firstly, when a user requests to insert an edge, TxCoordinator initiates a transaction and records the data modification partitions in the TxTopology. It then processes the request by sending it to the relevant PrimaryParts in both the Out and In directions, updating the WAL and cache. However, the data uses the TxId as a provisional version number and remains invisible to other transactions until the transaction commit is successful. Once the PrimaryPart completes the write operation on the WAL, it asynchronously replicates the WAL to its SecondaryPart. After the SecondaryPart finishes replicating the WAL and updating the cache, it asynchronously notifies the PrimaryPart.

Secondly, after the edge insertion, the user may request to access an edge in the Out direction. TxCoordinator checks TxTopology to ensure the involved partition is active, then forwards the query to the relevant PrimaryPart in the Out direction. The operation reads the version number for the data to determine its readability. It can read the version number equal to the TxId, ensuring "read your writes."

Thirdly, in the final phase of the transaction, the user triggers the commitment process managed by TxCoordinator using the 2PC protocol. In the prepare phase, TxCoordinator checks the states of the partitions based on the transaction's partition topology in TxTopology to evaluate the partitions involved in the operations. It establishes a minimum threshold to ensure that a majority of the SecondaryParts have completed their asynchronous WAL replication and cache updates. Once confirmation is received that most SecondaryParts have finished their tasks, TxCoordinator moves to the commit phase and marks the current time as the commit timestamp. For each affected partition, the data version is updated from the TxId to the commit timestamp, making the data visible. Finally, TxCoordinator confirms the successful completion of the transaction.

Our design introduces several features to enhance the transaction process, improving both efficiency and reliability. (1) It processes Out and In edges simultaneously, ensuring data consistency in both directions and reducing the latency typically associated with update operations. (2) The system employs asynchronous replication for SecondaryParts, which does not block the subsequent tasks in this transaction, finalizing most of these writes in the prepare phase. This approach helps achieve low latency while maintaining data consistency. (3) The use of interactive transactions allows subsequent steps of a transaction to be determined based on the outcomes of preceding read or write operations.

6.2 OLAP Transaction

The lengthy nature of OLAP workloads introduces the risk of concurrent OLTP operations during analysis. For example, in areas such as anti-money laundering, these workloads frequently involve analyzing extensive subgraphs or pathways, sometimes spanning over 10 hops to evaluate risks accurately. OLAP workloads also involve algorithms like PageRank for identifying key vertices, and Louvain for community detection, both requiring full subgraph access. To preserve the accuracy and reliability of the analysis, it is essential for OLAP workloads to rely on a data snapshot taken at the start of the workload, ensuring data consistency throughout the analysis.

OLAP workloads predominantly focus on graph analysis through read-only operations. Given this characteristic, Galaxybase handles OLAP read-only transactions using MVCC visibility checks without relying on any locks. MVCC provides a consistent view, ensuring that OLAP transactions read data with timestamps less than or equal to the transaction start time. This approach prevents interference between OLTP and OLAP workloads, maintaining serializable snapshot isolation levels for OLAP transactions.

7 EXPERIMENTAL EVALUATION

In this section, we conduct an in-depth assessment of Galaxybase, focusing on addressing the following questions: **Q1**: What is the performance level of Galaxybase compared with existing graph databases, especially in a distributed environment with multiple machines? **Q2**: How efficiently does Galaxybase handle various HTAP workloads? **Q3**: How well does Galaxybase handle a variety of graph queries, from graph traversals to single edge queries? **Q4**: What is the capability of Galaxybase in extreme scenarios, such as operating under severely limited memory?

7.1 Experiments Setup

In this subsection, we conduct comparative analyses of Galaxybase (version 3.5.0) against three well-known graph databases: Neo4j (version 4.4.27), TigerGraph (version 3.9.2), and JanusGraph (version 0.6.0). The experimental setup involves a cluster configured with 1 to 10 machines. Each machine is equipped with 20 virtual cores powered by an Intel Xeon Gold 5218R CPU and 128GB of DDR4 memory. For data transfer between servers, each machine is equipped with a 1 Gigabit pass-through network card. The entire configuration operates on the Ubuntu 18.04.6 LTS operating system.

Workloads. The usage of Galaxybase mainly encompasses two types: OLTP and OLAP. OLTP workloads in Galaxybase focus on ensuring transactional integrity for basic operations such as creating, reading, updating, and deleting vertices, edges, and properties. Conversely, OLAP workloads in graph databases are characterized by intricate k-hop graph traversal queries and analyses, such as Weakly Connected Component for community detection and PageRank for identifying key vertices. Such queries are known for producing substantial intermediate results, particularly when they come across super-vertices within the graph's framework.

Datasets. Our evaluation utilizes three distinct datasets: two synthetic property-rich graphs created by the LDBC-SNB [2] data generator and one real-world graph dataset from Twitter [3]. Comprehensive statistics for each dataset are detailed in Table 1.

Table 1: Graph Datasets

Dataset	Vertices (V)	Edges (E)
LDBC-SNB-SF10	29,987,835	176,623,445
LDBC-SNB-SF100	282,637,871	1,775,513,811
Twitter-2010	41,652,230	1,468,365,182

7.2 Single-Machine/Distributed Performance

We begin by evaluating Galaxybase under both OLTP and OLAP workloads, across single-machine and distributed configurations. OLTP queries serve as a crucial metric for evaluating a graph database's fundamental read and write capability, whereas OLAP queries access its analytical capability. Our tests are performed with 1 and 3 servers, where we execute concurrent OLTP operations on vertices and edges using the LDBC-SNB-SF10 dataset. For OLAP queries, we explore scenarios ranging from 1 to 6 hops and apply algorithms such as the Weakly Connected Component (WCC) and PageRank, utilizing the Twitter-2010 dataset for these analyses.

Figure 9 displays the OLTP performance of Galaxybase in comparison to TigerGraph, Neo4j, and JanusGraph across eight operations, with throughput measured in thousands of queries per second (Kq/s). Notably, for the vertex read operation, Galaxybase reaches a throughput of up to 50 Kq/s in a single-machine environment and 85 Kq/s when distributed, surpassing other graph databases. The y-axis scale emphasizes the substantial range of throughput, highlighting Galaxybase's dominance, particularly in distributed settings where its performance scales impressively with the increased number of machines. This suggests that Galaxybase's architecture is highly effective in both individual and collective server arrangements, ensuring rapid data handling and transaction processing in high-demand scenarios.

The OLAP results, illustrated in Figure 10, showcase a comparison of the latency for k-hop queries and graph algorithms on four graph databases under both single-machine and distributed configurations. In these tests, each database was restricted to a maximum query execution time of one hour, with durations exceeding this limit labeled as "Timeout". Notably, JanusGraph and Neo4j frequently reached these time constraints, struggling to complete deep hop queries within the timeframe. JanusGraph, in particular, lacked compatibility with the tested algorithms, indicated as "Unsupported". In contrast, Galaxybase consistently adhered to the one-hour threshold for all hops and algorithm evaluations, showcasing superior scalability and performance. Its latency was impressively low in both single and distributed settings, with the former demonstrating higher improvements.

In both OLTP and OLAP benchmarks, Galaxybase consistently outperforms other graph databases in terms of throughput and latency. This superiority can be attributed to the storage features of Galaxybase. It employs a Log-Structured Adjacency List to improve sequential data read/write operations and uses offset-based data retrieval for fast data access. The performance gap is also widened

[2] https://ldbcouncil.org/benchmarks/snb/

[3] https://snap.stanford.edu/data/twitter-2010.html

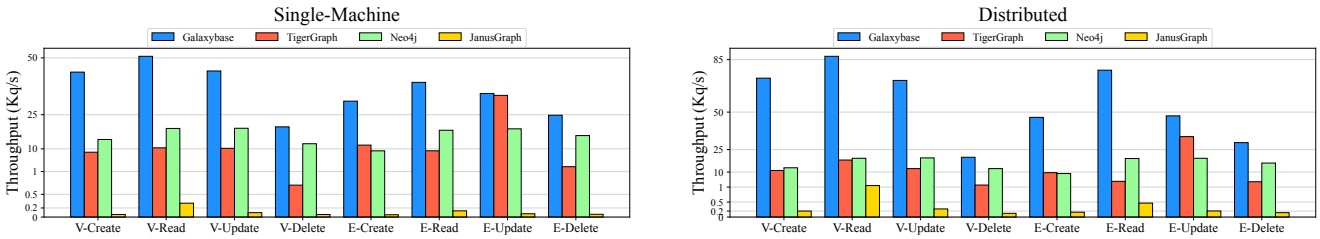

Figure 9: Comparative analysis of throughput performance in graph database OLTP operations, categorized by vertex (V) and edge (E) actions. The y-axis represents throughput, measured in thousands of operations per second (Kq/s).

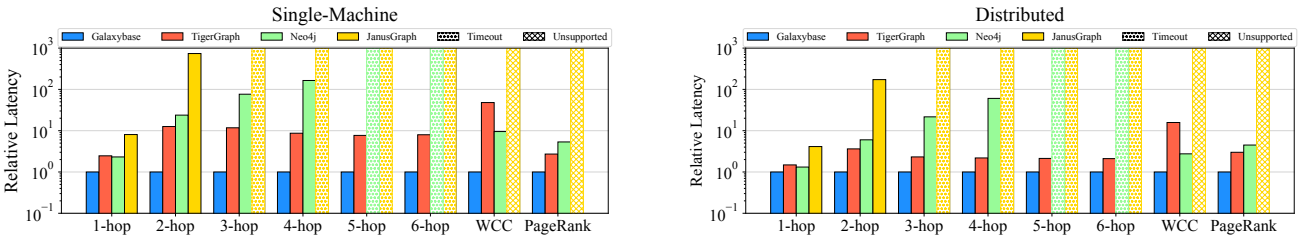

Figure 10: Comparative analysis of relative latency for OLAP queries across different graph databases on a single-machine/distributed setup. The y-axis represents logarithmic relative latency.

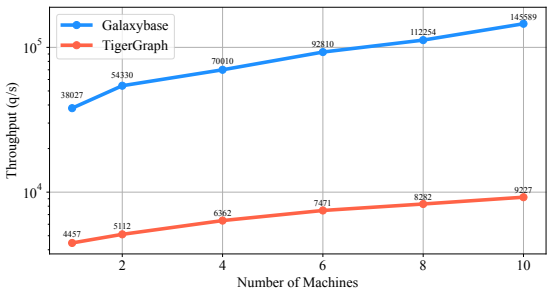

Figure 11: Scalability throughput for 1 to 10 machines

by the fact that databases like Galaxybase use specialized graph stores. In contrast, JanusGraph relies on a more generic NoSQL-based storage solution, which is not as well-suited for the specific demands of graph operations.

7.3 Scalability Analysis

We assess the scalability of Galaxybase by conducting a comparative analysis with the widely recognized distributed graph database TigerGraph. We perform 100 concurrent 1-hop neighbor queries on 1 to 10 servers using LDBC-SNB-SF100. Our objective is to measure throughput and evaluate performance across different numbers of servers.

Figure 11 shows the throughput for each system when running tests across different numbers of servers. Both systems demonstrate increased throughput with more servers. However, Galaxybase consistently outperforms TigerGraph, with at least 8.5 times higher throughput. This performance is due to Galaxybase's use of partitions to handle high parallel workloads. Meanwhile, the advantage of Galaxybase becomes increasingly evident as the number of servers grows. For instance, when comparing 10 servers to 1 server, Galaxybase achieves a 3.8 times increase in throughput, while TigerGraph only sees a 2.1 times increase. Galaxybase's load balancing strategies enable higher parallelism to process each transaction when more resources are available. These results highlight the scalability and efficiency of Galaxybase in deployments.

7.4 Edge Query Evaluation

This evaluation focuses on the impact of Edge Page technology, as detailed in Section 4.3, using the LDBC-SNB-SF10 dataset with a single server. We assess edge query performance from several angles, including traversing edges without filters, by direction, and by type, as well as locating individual edges via their Edge ID.

As shown in Table 2, Galaxybase demonstrates outstanding performance across all edge query types, especially notable in *Single Edge Query*, showcasing the effectiveness of its Edge Page design. In contrast, TigerGraph faces limitations due to its REST API's restriction to Out direction traversal and lack of support for diverse traversal directions (In, Both), impacting its overall edge query performance. Neo4j shows comparable performance to Galaxybase in the *Graph Traversal Without Filtering* and *Graph Traversal With Directed Filtering* queries but lags in *Graph Traversal With Type Filtering* and *Single Edge Query*. Neo4j's approach of storing edges

at a fixed size and multiplying ID by record size offers quick retrieval for *Single Edge Query* that do not require edge properties, but this method is less effective for more complex queries seeking detailed edge information. JanusGraph, on the other hand, exhibits weaker performance in all categories, primarily due to its non-native graph structure, a common challenge shared among the evaluated databases.

Table 2: Edge query throughput

Type	Galaxybase	TigerGraph	Neo4j	JanusGraph
Graph Traversal Without Filtering	1185	54	1155	0.67
Graph Traversal With Directed Filtering	2877	1177	2344	11
Graph Traversal With Type Filtering	10270	63	3183	121
Single Edge Query	49017	8185	19592	207

7.5 Memory Constraint Analysis

In this experiment, we aim at understanding how well a graph database can operate under memory limitations, a crucial aspect for users with constrained hardware resources. For this purpose, we conduct tests with memory caps at 16GB, 32GB, and 64GB. The Twitter dataset used in this experiment typically demands around 25GB of memory to be fully loaded, making this a challenging test of memory management capabilities in graph databases.

Our analysis includes performing k-hop queries with neighborhoods of 1 and 2 degrees. The average neighborhood size is about 1,000 for 1-hop and roughly 2 million for 2-hop. As presented in Table 3, we observe that existing graph databases struggle with limited memory. For instance, TigerGraph has difficulty loading data even with 32GB, suggesting an issue with data inflation during TigerGraph's data import process. Neo4j and JanusGraph manage to load and process 1-hop queries with restricted memory. However, they encounter out-of-memory errors during 2-hop queries, and error rates are still high even with 64GB, indicating stability issues under memory strain.

Galaxybase, on the other hand, stands out for its efficient memory usage and stability. It shows remarkable performance, effectively handling queries even when the available memory is below the dataset's full requirement. This indicates Galaxybase's superior capability to perform under memory constraints, an essential quality for graph databases in resource-limited environments.

Table 3: Performance under low memory conditions

Memory	k-hop	Galaxybase	TigerGraph	Neo4j	JanusGraph
16G	1-hop	23476 (0%)	/	11784 (0%)	94 (0%)
	2-hop	9.35 (0%)	/	OOM (100%)	OOM (100%)
32G	1-hop	24262 (0%)	/	11999 (0%)	92 (0%)
	2-hop	9.57 (0%)	/	OOM (100%)	OOM (100%)
64G	1-hop	24862 (0%)	4051 (0%)	12336 (0%)	95 (0%)
	2-hop	9.89 (0%)	6.15 (0%)	0.32 (17%)	0.18 (29%)

* The result indicates throughput (error rate). "/" indicates the graph database is unable to load the data within the specified resource constraints. "OOM" means the graph database ran out of memory when running the k-hop query within the specified resource constraints.

7.6 Evaluation on Extremely Large Graphs

In our previous evaluations, analysis is confined to smaller graphs due to the limitations of competing systems in managing larger graphs effectively. To demonstrate Galaxybase's proficiency in processing large-scale data, we present its performance on a simulated trillion-scale dataset of real financial transactions, as detailed on our official website[4]. This dataset comprises 5 billion account vertices and 5 trillion transaction edges, complete with various properties and super vertices, with degrees ranging from 10 to over 10 million.

On this dataset, Galaxybase successfully processes 6-hop queries filtered by transaction time with an average execution time of just 6.7 seconds. This level of efficiency is achieved using only 50 machines, each equipped with 12 CPUs and 128GB of memory, showcasing Galaxybase's exceptional capability to manage and process data at a truly massive scale.

8 LESSONS LEARNED

This section outlines some key insights gained from nearly a decade of developing Galaxybase.

8.1 Resource Isolation

Galaxybase's extensive experience in production environments highlights the importance of resource isolation. This is vital to prevent high-load queries from one user causing out-of-memory (OOM) errors for others. Given the unpredictable nature of graph queries, encountering a 'super vertex' can involve expansive neighbor exploration and lead to rapid data volume inflation, complicating resource management.

To address this, Galaxybase has implemented a sophisticated memory management and thread scheduling system. This system monitors the memory and thread resources usage of each query in real-time, enabling the enforcement of specific resource limits per user and query. Queries exceeding these limits are promptly terminated, preventing excessive resource consumption. Although this approach results in a performance overhead of 2-5%, it significantly boosts system stability and availability, making Galaxybase more robust in serious production environments.

8.2 Access Control

We recognize the significant differences in access control requirements across various scenarios. For instance, in banking systems, some users may be authorized to modify graph data but not the schema, while others can only view data from specific regions. In more flexible environments, like small educational organizations, users prioritize quickly sharing entire graphs or subgraphs from specific queries. These diverse needs must be considered when designing RBAC system for graph databases.

Additionally, access control in graph databases need to accommodate their unique structure, consisting of vertices and edges, which necessitates granular permission settings. Unlike relational databases, which may isolate data by region into separate tables or databases, graph databases depend on interconnected data for their functionality. This integration makes the access control needs of graph databases inherently more complex than those in relational

[4]https://www.createlink.com/news/trillion-test

systems. Such granularity and connectivity are critical considerations in designing RBAC systems for graph databases. To address this, we have implemented a customer-oriented RBAC layer that balances the need for development flexibility with strict production security requirements and graph-specific features.

8.3 Chaos Testing

Certain scenarios require exceptionally high levels of operational stability, with 24/7 uptime being crucial. For example, in the financial and energy sectors, any downtime can lead to significant economic losses, disruptions in services, and operational paralysis.

Galaxybase undergoes rigorous chaos testing prior to deployment in such critical environments. We have developed a chaos testing system tailored for graph databases that simulates potential production issues by injecting random faults into large-scale clusters. Our testing framework encompasses over 200 user cases and tens of thousands of test cases, addressing a broad spectrum of potential failures and chaotic conditions under various workloads. Using automated tools and scripts, we introduce chaos factors under various types of workloads and monitor the system to verify its operational robustness.

8.4 Query Language Choices

In the initial development phase of Galaxybase, we considered the pros and cons of designing a specialized query language. Customized languages can be optimized for specific system needs and introduce unique features, but suffer from limited compatibility with other systems and pose significant adoption barriers due to their steep learning curves. Therefore, we opted for OpenCypher[5] syntax, which closely resembles SQL and eases the learning process for users familiar with relational databases. Moreover, the launch of the GQL[6] ISO standard on April 11, 2024, marked a significant milestone. As the first standardized graph query language, GQL fosters interoperability among different graph database systems and encourages consistency in graph querying. Galaxybase is committed to supporting GQL, which incorporates most of syntactical elements from OpenCypher.

9 RELATED WORK

Graph databases are typically grouped into two main types: non-native databases, and native databases.

Non-native graph databases often use established non-graph backends such as Key-Value or Document stores. This category includes systems like JanusGraph [3], ByteGraph [27], TAO [10], A1 [11], NebulaGraph [4], RedisGraph [12], ArangoDB [2], and OrientDB [36]. JanusGraph, building on Titan [1], employs a wide-column store approach, organizing vertices and edges in a format that combines properties with Key-Value pairs. ByteGraph also adopts a Key-Value storage model, tailored to the demands of graph-based querying.

In the category of native graph databases, solutions like Neo4j [5] and LiveGraph [43] stand out for their implementation of index-free adjacency. This approach greatly improves graph traversal efficiency. However, we observe that traditional native graph databases, encounter scalability limitations. For instance, Neo4j can operate in a multi-server mode, but its setup doesn't involve dataset sharding capabilities, while LiveGraph is designed as a single-machine graph database.

Distributed native graph databases mark a significant evolution in the realm of graph database technology, advancing beyond the capabilities of traditional native databases. Examples include TigerGraph [16], Grasper [13], G-Tran [14], and our proposed Galaxybase. Some systems are designed for high-performance operations. TigerGraph, for example, tries to load the whole graph into memory, which can achieve low latency and high memory utilization when the server is created [27]. If the graph size exceeds the available memory, the excess data is stored on disk. Some systems are adept at handling OLTP or OLAP operations. For instance, G-Tran is tailored for OLTP tasks, while Grasper is better suited for OLAP tasks. Galaxybase, in contrast, offers optimized solutions for both OLTP and OLAP operations separately, showcasing its versatility.

In addition to these categories, the graph processing frameworks like Pregel [29], GraphLab [28], PowerGraph [20], GraphX [21], and GraphScope [18], is renowned for its in-memory batch processing and distributed parallel processing capabilities. Graph processing systems focus on OLAP workloads like analytics and mining, while graph databases for OLTP workloads concentrate on maintaining data consistency through transactions.

10 CONCLUSIONS

In this paper, we introduce Galaxybase, a native distributed graph database optimized for efficiently handling HTAP workloads. Our empirical evaluation demonstrates that Galaxybase significantly outperforms existing graph databases, including TigerGraph, Neo4j, and JanusGraph. This superior performance is attributed to its unique design, which includes the Log-Structured Adjacency List, Edge Page, Distributed Storage Strategies, and HTAP Distributed Transaction mechanisms. Galaxybase shows exceptional performance in both OLTP and OLAP scenarios, achieving a throughput of up to 85,000 queries per second in OLTP workloads and surpassing its competitors in OLAP workloads by a substantial margin. Galaxybase addresses the current limitations faced by existing graph databases in handling large-scale, complex queries and sets a new benchmark in processing efficiency.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. We are also grateful to the entire team at CreateLink(Chuanglin) for their dedication and support during the development of Galaxybase. Additionally, we appreciate the professional guidance and insightful feedback provided by the team at the HKUST(GZ)-Chuanglin Graph Data Joint Lab. This paper was supported by NSFC Grant No. 62206067 and HKUST(GZ)-Chuanglin Graph Data Joint Lab.

[5] https://opencypher.org/
[6] https://www.gqlstandards.org/

REFERENCES

[1] 2015. *Titan.* https://titan.thinkaurelius.com/
[2] 2024. *ArangoDB.* https://www.arangodb.com/
[3] 2024. *JanusGraph.* http://janusgraph.org/
[4] 2024. *NebulaGraph.* https://www.nebula-graph.io/
[5] 2024. *Neo4j.* https://neo4j.com/
[6] Renzo Angles. 2018. The Property Graph Database Model.. In *AMW*.
[7] Marcelo Arenas, Claudio Gutiérrez, and Juan F Sequeda. 2021. Querying in the age of graph databases and knowledge graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 2821–2828.
[8] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.
[9] Amel Awadelkarim and Johan Ugander. 2020. Prioritized restreaming algorithms for balanced graph partitioning. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 1877–1887.
[10] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, et al. 2013. {TAO}:{Facebook's} distributed data store for the social graph. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*. 49–60.
[11] Chiranjeeb Buragohain, Knut Magne Risvik, Paul Brett, Miguel Castro, Wonhee Cho, Joshua Cowhig, Nikolas Gloy, Karthik Kalyanaraman, Richendra Khanna, John Pao, et al. 2020. A1: A distributed in-memory graph database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 329–344.
[12] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. 2019. Redisgraph graphblas enabled graph database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 285–286.
[13] 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*. 87–100.
[14] 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. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2545–2558.
[15] 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. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3014–3027.
[16] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248* (2019).
[17] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.
[18] Wenfei Fan, Tao He, Longbin Lai, Xue Li, Yong Li, Zhao Li, Zhengping Qian, Chao Tian, Lei Wang, Jingbo Xu, et al. 2021. GraphScope: a unified engine for big graph processing. *Proceedings of the VLDB Endowment* 14, 12 (2021), 2879–2892.
[19] Lars George. 2011. *HBase: the definitive guide: random access to your planet-size data.* " O'Reilly Media, Inc.".
[20] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. {PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs. In *10th USENIX symposium on operating systems design and implementation (OSDI 12)*. 17–30.
[21] Joseph E Gonzalez, Reynold S Xin, Ankur Dave, Daniel Crankshaw, Michael J Franklin, and Ion Stoica. 2014. {GraphX}: Graph processing in a distributed dataflow framework. In *11th USENIX symposium on operating systems design and implementation (OSDI 14)*. 599–613.
[22] James N Gray. 2005. Notes on data base operating systems. *Operating systems: An advanced course* (2005), 393–481.
[23] Richard Henderson. 2020. Using graph databases to detect financial fraud. *Computer Fraud & Security* 2020, 7 (2020), 6–10.
[24] Bowen Kan, Wendong Zhu, Guangyi Liu, Xi Chen, Di Shi, and Weiqing Yu. 2017. Topology modeling and analysis of a power grid network using a graph database. *International Journal of Computational Intelligence Systems* 10, 1 (2017), 1355–1363.
[25] George Karypis and Vipin Kumar. 1997. METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. (1997).
[26] Butler W Lampson and Howard E Sturgis. 1979. *Crash recovery in a distributed data storage system.* Xerox Palo Alto Research Center Palo Alto, California.
[27] 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.
[28] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein. 2012. Distributed graphlab: A framework for machine learning in the cloud. *arXiv preprint arXiv:1204.6078* (2012).
[29] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.
[30] Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*. 305–319.
[31] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33 (1996), 351–385.
[32] Anil Pacaci, Alice Zhou, Jimmy Lin, and M Tamer Özsu. 2017. Do we need specialized graph databases? Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–7.
[33] Massimo Pezzini, Donald Feinberg, Nigel Rayner, and Roxane Edjlali. 2014. Hybrid transaction/analytical processing will foster opportunities for dramatic business innovation. *Gartner (2014, January 28) Available at https://www. gartner. com/doc/2657815/hybrid-transactionanalyticalprocessing-foster-opportunities* (2014), 4–20.
[34] Debachudamani Prusti, Daisy Das, and Santanu Kumar Rath. 2021. Credit card fraud detection technique by applying graph database model. *Arabian Journal for Science and Engineering* 46, 9 (2021), 1–20.
[35] David Patrick Reed. 1978. Naming and Synchornization in a Decentralized Computer System. (1978).
[36] Daniel Ritter, Luigi Dell'Aquila, Andrii Lomakin, and Emanuele Tagliaferri. 2021. OrientDB: A NoSQL, Open Source MMDMS.. In *BICOD*. 10–19.
[37] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data.* " O'Reilly Media, Inc.".
[38] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis. 1978. System level concurrency control for distributed database systems. *ACM Transactions on Database Systems (TODS)* 3, 2 (1978), 178–198.
[39] Alon Shalita, Brian Karrer, Igor Kabiljo, Arun Sharma, Alessandro Presta, Aaron Adcock, Herald Kllapi, and Michael Stumm. 2016. Social hash: an assignment framework for optimizing distributed systems operations on social networks. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. 455–468.
[40] Jianheng Tang, Jiajin Li, Ziqi Gao, and Jia Li. 2022. Rethinking graph neural networks for anomaly detection. In *International Conference on Machine Learning*. PMLR, 21076–21089.
[41] Jihong Yan, Chengyu Wang, Wenliang Cheng, Ming Gao, and Aoying Zhou. 2018. A retrospective of knowledge graphs. *Frontiers of Computer Science* 12 (2018), 55–74.
[42] TV Zhidchenko, MN Seredina, NM Udintsova, and NA Kopteva. 2021. Design of energy-loaded systems using the Neo4j graph database. In *IOP Conference Series: Earth and Environmental Science*, Vol. 659. IOP Publishing, 012108.
[43] Xiaowei Zhu, Guanyu Feng, Marco Serafini, Xiaosong Ma, Jiping Yu, Lei Xie, Ashraf Aboulnaga, and Wenguang Chen. [n.d.]. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 ([n. d.]).