BIG DATA MINING AND ANALYTICS
ISSN 2096-0654 11/15 pp156–170
DOI: 10.26599/BDMA.2023.9020015
Volume 7, Number 1, March 2024

Building a High-Performance Graph Storage on Top of Tree-Structured Key-Value Stores

Heng Lin, Zhiyong Wang, Shipeng Qi, Xiaowei Zhu, Chuntao Hong*, Wenguang Chen, and Yingwei Luo

Abstract: Graph databases have gained widespread adoption in various industries and have been utilized in a range of applications, including financial risk assessment, commodity recommendation, and data lineage tracking. While the principles and design of these databases have been the subject of some investigation, there remains a lack of comprehensive examination of aspects such as storage layout, query language, and deployment. The present study focuses on the design and implementation of graph storage layout, with a particular emphasis on tree-structured key-value stores. We also examine different design choices in the graph storage layer and present our findings through the development of TuGraph, a highly efficient single-machine graph database that significantly outperforms well-known Graph DataBase Management System (GDBMS). Additionally, TuGraph demonstrates superior performance in the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) interactive benchmark.

Key words: graph database; high-performance; graph storage

1 Introduction

Graph Database Management System (GDBMS) has been widely used in the industry. Typical scenarios include financial risk assessment, Anti-Money-Laundering (AML), social network analysis, commodity recommendation, and epidemic spreading analysis.

Although GDBMSs share many features with relational DBMSs, the workloads they serve are largely different. For example, typical graph queries involve multiple hops, requiring operations on multiple vertex types and multiple edge types, while relational database queries rarely touch more than two tables. Multi-hop graph queries usually read much more data than relational queries. For example, on a graph with an average degree of ten, a 3-hop query will read 1000 edges on average. Besides, the irregular access patterns of graph queries bring extra challenges (detailed in Section 2.2). The distinction of workloads requires different system designs. More specifically, the storage layer of GDBMSs needs to be carefully designed to suit its unique access pattern.

Key-value stores have been used as the underlying storage engine for most database systems. As a layer of abstraction, they typically provide concurrent read/scan/write/update operations on key-values, with certain transaction guarantees. Building on top of the key-value layer, database system designers can then focus more on providing the right abstraction of data, without worrying about the persistence of the data.

• Heng Lin and Yingwei Luo are with the School of Computer Science, Peking University, Beijing 100871, China. E-mail: linheng@peking.edu.cn; lyw@peking.edu.cn.
• Heng Lin, Zhiyong Wang, Shipeng Qi, Xiaowei Zhu, Chuntao Hong, and Wenguang Chen are with Ant Group, Beijing 100020, China. E-mail: botu.wzy@antgroup.com; qishipeng.qsp@antgroup.com; robert.zxw@antgroup.com; chuntao.hct@antgroup.com; cwg@tsinghua.edu.cn.
• Wenguang Chen is also with the Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China.
* To whom correspondence should be addressed.
Manuscript received: 2023-01-07; revised: 2023-06-08; accepted: 2023-06-19

Tree-structured key-value stores are most commonly used in DBMSs due to their maturity and the availability of a sorting order of the keys.

Many GDBMSs have been built, especially in the past two decades. However, the design of GDBMS storage has not been thoroughly investigated in the literature. This paper discusses the design choices of GDBMS storage, with special focus on building on top of tree-structured key-value stores. We start by analyzing the common access patterns in graph queries and the requirements for the underlying key-value store, then we compare two commonly used tree-based key-value stores, namely RocksDB and LMDB, to see which one fits better for our GDBMS, and finally, we discuss the design choices when building a graph storage layer on top of the key-value store. We implement our ideas on a graph database called TuGraph. The experimental results show that our design achieves great performance on micro-benchmarks, and state-of-the-art performance on the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB) interactive benchmark.

The main contribution of this paper involves:

- Analysis of common access patterns of graph queries and the requirements of the storage layer.
- Investigation of key design choices of a graph storage layer on top of tree-structured key-value stores.
- Implementation of our ideas on LMDB that shows good performance on both micro- and macro-benchmarks.

The rest of the paper is organized as follows. Section 2 introduces the background of our work and the common access patterns of graph databases. Section 3 discusses important design choices of the graph storage layer. Then, Section 4 evaluates the performance of our graph storage. In Section 5 we discuss related works. Finally, Section 6 concludes this research and identifies future research directions.

2 Data Model and Access Patterns

In this section, we first introduce the data model we used, namely the property graph model. Then, we summarize the common access patterns of graph queries and analyze the challenges they present for graph databases.

2.1 Property graph model

There are two main data models for graph databases: the Resource Description Framework (RDF) model and the property graph model. The RDF model represents data as subject-predicate-object triples. The property graph model, on the other hand, follows an object-oriented approach and represents entities as vertices and relationships between entities as edges, with both vertices and edges able to have properties. Through our survey, finding that RDF model is used in semantic scenario, while property graph model is used across many areas. Many GDBMSs use the property graph model, such as Neo4j, TigerGraph, and GeaBase[1−3]. Therefore, we choose property graph model as our data model.

An example of a property graph is shown in Fig. 1, which represents a money transfer between two accounts as vertices labeled “Account” and an edge labeled “Transfer” connecting the two vertices. The schema for each edge can also have optional source/destination label constraints, specifying the types of vertices that the edge can connect. Each account has two properties ID and Name, while the transfer has properties Timestamp and Amount.

2.2 Graph query

Graph databases are widely used in various fields, including financial risk management, anti-money laundering, and data lineage tracking at Ant Group, where they are used by over 100 scenarios. In this section, we discuss the most commonly used graph queries and examine the typical access patterns of these queries.

2.2.1 Access pattern

A typical query in AML is shown in Fig. 2. In this application, an account is represented as a vertex in the graph and a transfer of money is represented as an edge connecting the source and destination accounts, as shown in Fig. 1. When a new transfer is made (represented as a new edge between the sender and receiver), a cycle-detection algorithm is run to determine if a cycle (meeting predetermined criteria) will be formed if the new edge is added to the graph. If a cycle is detected, the transfer is rejected, otherwise it is accepted and the edge is added to the graph. The

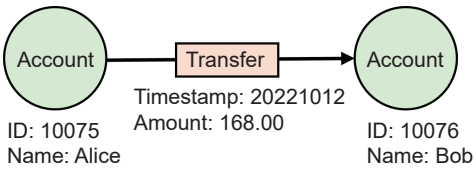


Fig. 1 Property graph example.

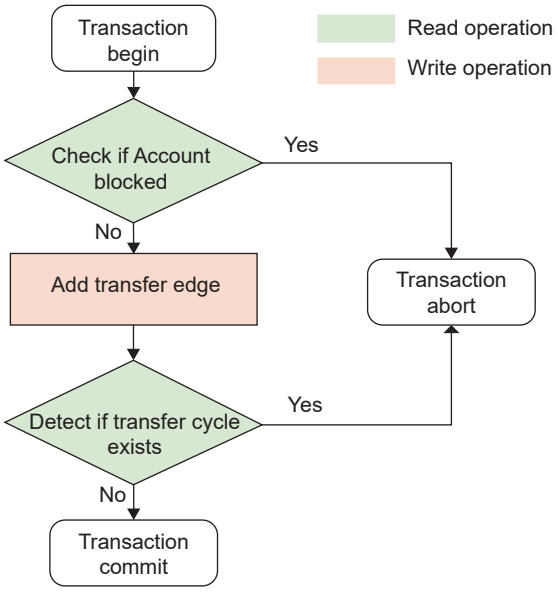


Fig. 2 Example of a query in AML, which involves both read and write.

cycle-detection algorithm only detects cycles that meet certain rules, such as having transfers of a significant amount and having downstream transfers occurring later than their predecessors. Some common access patterns can be identified in this query:

- **Multi-hop traversal:** multiple hops are required to detect cycles in the query.
- **Filtering with properties:** filtering based on the properties of edges and vertices is necessary during the traversal.
- **Transaction:** the entire query should be completed as a single transaction for atomicity.

Many graph application scenarios have similar data access patterns. For example, in post-loan risk control, we search for many-to-one patterns with recursive path filtering to find potential avatar frauds. Online gambling can be detected by multiple money transfers within a short time range. Equity penetration checks the share-holding relationships of entities recursively.

2.2.2 Read/write ratio

The read-to-write ratio of graph workloads in five online financial product systems was found to be around 20∶1 in an online GDBMS[3]. This indicates that read workloads have a greater impact on overall performance, while write workloads should also have strong performance. The unique characteristics of these access patterns pose significant challenges for GDBMS, particularly in terms of graph storage. There are various design approaches for graph storage among popular GDBMSs.

2.2.3 Observation

From the discussion in the introduction and the analysis of graph workloads, we have identified the following characteristics of these workloads:

- **Observation 1.** *K*-hop-like traversal in graph topology is a common operation, which is quite different from workload in a Relational Database Management System (RDBMS).
- **Observation 2.** There exists locality in graph workloads, with the out-edges of a certain vertex often visited together, especially edges with the same label.
- **Observation 3.** During traversal, one or more properties of vertices or edges are accessed.
- **Observation 4.** In a temporal graph workload, edges are accessed within a temporal window.
- **Observation 5.** A single query can involve both read and write operations.

These observations lead to specific system challenges and affect our design principles.

2.2.4 Challenges and design principles

The following characteristics of graph data in storage present challenges for graph storage:

- **Data dependence.** *K*-hop traversal involves accessing the destination vertex along an edge, using the destination vertex as a new source vertex, and then accessing the new destination vertex again in a repetitive process. This access pattern is highly dependent on the graph topology data.
- **Massive random read.** Due to data dependence, the source vertex, destination vertex, and relevant edges cannot always be stored together for different graph queries. Accessing edge data are similar to "JOIN" operations between tables in an RDBMS, which is always a challenge under modern CPU architectures.
- **Heterogeneous data.** Graph storage must handle data of different labels, with the number of edges connecting a single vertex ranging from one to billions.
- **Concurrent read and write queries**. GDBMSs must support online data analytics while also allowing for continued data updates.

To address the challenges presented by the unique characteristics of graph data and access patterns, the following design principles have been adopted:

- **Maintain Advance Cargo Information Declaration (ACID) as a basic requirement:** The correctness of concurrent queries should be the first priority.

- **Address the critical issue of random read:** Each read query is likely to access many vertices, while each write query only accesses one vertex or edge. Massive random read can be particularly severe.
- **Explore the locality in graph data:** Since memory has a significant advantage over external disk in terms of latency and throughput, the performance of accessing data not cached in memory can drop sharply. Data with locality within one page are more friendly to external disks.
- **Save storage capacity when possible:** Having more data in memory means fewer external disk accesses for read and write operations.

3 Design

In this section, we discuss the considerations and trade-offs that are taken into account when designing a practical graph storage system.

The first question addressed is how to pack the graph topology and properties into a key-value pair. We propose using a compact packing method with an adaptive mapping technique.

The second question is the selection of a key-value store. We choose to use LMDB with concurrent write enhancement.

3.1 Overall architecture

In this work, we divide the graph storage into two layers (see Fig. 3): the Property Graph storage layer (PG layer) and the key-value storage layer (KV layer). The PG layer maps graph operations, such as schema and vertex CRUD, onto key-value operations. It also arranges the source and destination vertex information of edges and determines how properties are integrated with the graph topology through a process called properties encoding. The KV layer contains a key-value store that is ACID-compliant, and has been optimized for common read and write patterns in graph workloads. It is responsible for providing a data manipulation interface for graph computation, such as the cypher query language and stored procedures.

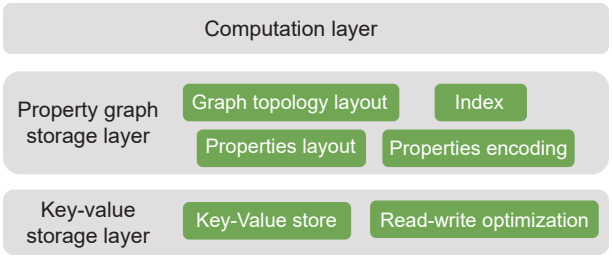


Fig. 3 Architecture of graph storage.

Not all graph storage systems follow this architecture. Some GDBMSs use native graph storage, such as Neo4j, which uses many pointers to link vertices, edges, and properties. This design allows for minimal data access, but it also means that sequential prefetch is not possible. We will compare the performance of different graph storage systems in Section 4, but will only focus on tree-structured key-value stores in the design process.

3.2 Topology packing

In the property graph model, the topology of a graph refers not only to how vertices are connected by edges, but also to the order of vertices and edges. When designing a graph storage system, it is important to consider the order of vertices and edges in order to optimize performance when accessing data.

In this work, the unique identifiers for vertices and edges, VertexUid and EdgeUid, are used to order vertices and edges. The VertexUid is an auto-incrementing integer starting from 0, while the EdgeUid consists of the source vertex VertexUid (SrcVid), the destination vertex VertexUid (DstVid), the edge's label identifier (LabelId), and a unique identifier for the edge within a given label and source/destination pair (Eid).

To take advantage of the observation that edges sharing the same label are likely to be accessed together, the order of elements in EdgeUid is chosen to prioritize the LabelId followed by the destination vertex VertexUid (DstVid) and the unique edge identifier (Eid). In addition, a temporal identifier (TemporalId) is included in EdgeUid to allow for easy access to edges within a given time frame.

To save storage space, the EdgeUid is compressed by taking advantage of the fact that the LabelId, DstVid, and Eid are auto-incrementing integers starting from 0, which means that their values are typically small. The TemporalId can also be left empty if the edge does not have a timestamp property. A length-based compression technique is used to further reduce the size of EdgeUid.

Figure 4 illustrates the graph topology packing of our design.

3.3 Property packing

Section 3.2 solves the problem of graph topology

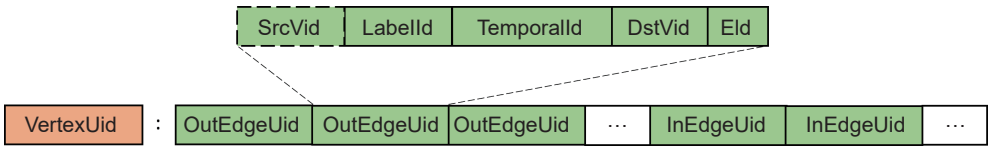


Fig. 4 CSR-like graph topology packing.

layout. Besides, the properties' layout should also be well organized to suit the graph workload.

In general, there are two methods for packing the properties. Index packing separates the properties into a new key-value pair, and leaves an index in the topology layout to locate the properties. Index packing has the advantage of quickly topology traversal without visiting properties, since properties are separately stored. Also, single property updating is fast because only the properties of one edge need to be updated. The other method is compact packing (see Fig. 5), it mixes all the properties in topology layout compact, and the properties are placed next to corresponding vertex or edge. If graph traversal visits topology and properties of edges and vertices together, compact packing only visits the mixed topology layout, while index packing needs an additional key-value lookup for every property.

Due to Observation 3 in Section 2.2.3, we prefer compact packing to suit to topology and properties together visiting pattern. Furthermore, each edge property is stored twice on both in-edge and out-edge to avoid one-side random data access. The random property problem is solved using adaptive mapping in Section 3.4.

Edges and vertices in the property graph model have the ability to accommodate an arbitrary number of properties. These properties can be of either fixed size, such as INT64, or variable size, such as String.

3.4 Adaptive mapping

In Sections 3.2 and 3.3, we have packed all the topology and properties data in one value of a vertex. The value can be very large and inefficient for random access and updating. For example, all the data should be repacked when one edge is added or a variable size property is updated. We use an adaptive mapping method to map a vertex's data to multiple key-value pairs using a threshold value size.

In Fig. 6, there are two types of mapping between graph topology and key-value, i.e., mixed mapping and split mapping. The mixed mapping has only one key-value pair, and all data are packed, whose total size is

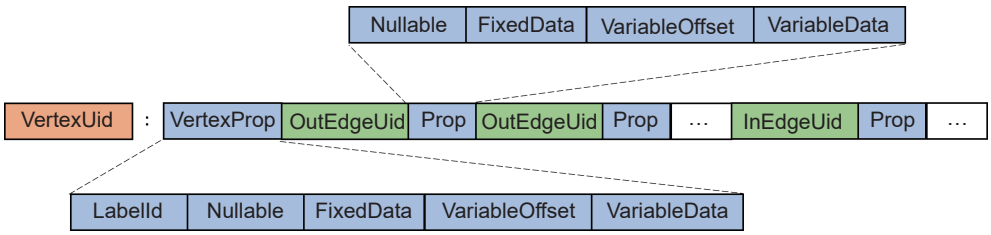


Fig. 5 Compact properties packing, the properties arranged intersections of topology data.

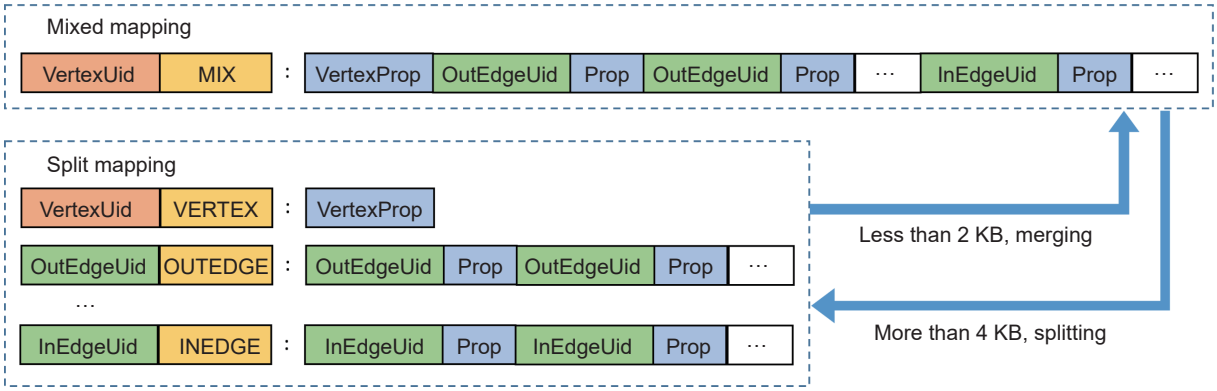


Fig. 6 Adaptive mapping among graph data and underlying key-value store.

smaller than the threshold size. The split mapping has three types of keys, namely vertex properties, out-edges with properties, and in-edges with properties.

Now let us discuss the performance of the split mapping. In terms of sequential access, all the values of the split mapping are arranged together, without performance slowdown, thanks to the same prefix of three types of keys.

In terms of random read, the order inner-value and inter-value are kept the same. Therefore, two simple binary search operations can locate the aimed data, similar to split before. In terms of random write, the value should be repacked. After splitting, the value size is limited to the threshold size.

A proper threshold is important to leverage sequential performance and repack size. In the phase of repack, the data need to be written back to disk in the same transaction. The cost of repack in the CPU cache should be comparable to latency on the external disk, i.e., 150 μs for SSD. We can limit the threshold to several KB to fit the data in L1 Cache and make it happen.

3.5 B+ tree based key-value storage

The basic idea of tree-structure is sorting data in order to support $O(\log n)$ complexity lookup and modify. Among the tree-structured key-value store, B+ tree, Log Structured Merge (LSM) tree as well as their variants are mostly used. For example, InnoDB and LMDB use B+ tree, while RocksDB uses the LSM tree.

A B+ tree uses a split and merge style in the tree node to update sorted data, while the LSM tree appends updates in log for lazy data compaction. The key advantage of LSM tree is sequential update operation in log, therefore the update operations complete immediately, which means delaying the actual compaction of data in the future. When reading happens in none-compaction data, the LSM tree needs to read through several levels of logs and results in read amplification and space amplification, causing a slowdown in read operations. In addition, periodical compaction is almost unpredictable above the PG layer.

After analyzing and evaluating the two representative key-value stores, LMDB for B+ tree and RocksDB for the LSM tree, LMDB has shown much better read performance, as well as matching sequential write performance, in spite of worse random write performance (see Section 4). In another aspect, LMDB has stable and predictable performance for further graph storage optimization. We prefer LMDB to RocksDB, as prefer reading performance to writing performance.

In terms of other B+ tree stores such as InnoDB and BerkeleyDB, benchmark[4] has shown that LMDB gets better read and write latency because of its lock-free design and simplified implementation.

3.6 Concurrent writer

LMDB has a significant shortcoming, i.e., single writer, which cannot even match the 20∶1 read-to-write performance requirement. We are going to enhance the single writer to a concurrent writer above key-value store by characteristics of graph workloads.

A single writer means each write transaction is processed one after another, and write transactions cannot be processed concurrently in a multi-thread modern CPU. Furthermore, only one thread can be used to write data to disk, which can hardly exhaust disk IO bandwidth. We reform single writer to have concurrent ability in two ways. One is optimistic query-level concurrent control, and the other is Write-Ahead-Log (WAL) based data durability.

According to Observation 5, we propose an optimistic concurrent writer. When graph storage receives many read-write mixed queries, as Fig. 7, and each query is composed of op and sync. The op indicates the operations in memory, and outputs the KVs should be written back to disk, while sync is the write back process. In read-write mixed queries, op costs much more than sync.

By default, the queries are executed sequentially, while queries can be executed in a concurrent way after optimization. In the below part of Fig. 7, the op part of each query is processed immediately when the query

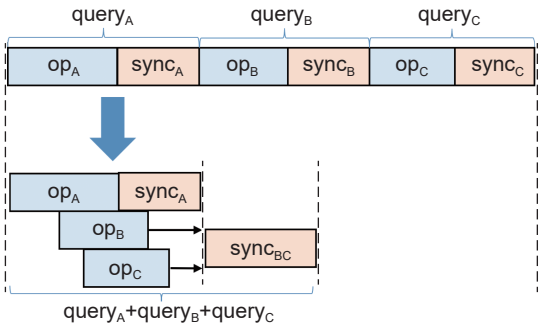


Fig. 7 Concurrent writer example. Queries can be processed concurrently, leaving synchronization processed sequentially.

arrives. After op is done, an extra check is triggered to check if the current data version is equal to the version transaction begins, to ensure no other contention. If no contention happens, this query goes to the next stage and waits for sync. Otherwise, this query fails and the transaction aborts. There is a background thread that continuously checks if there is sync.

All of the sync should be processed sequentially but get the opportunity to be processed in batches. In this example, sync_A starts when op_A finishes. When op_B and op_C end, sync_B and sync_C are blocked by ongoing sync_A. After sync_A finishes, sync_B and sync_C are both ready and can be processed in batches.

The concurrent writer is able to process op part concurrently and potentially process sync in batch, efficiently improving the performance. The side effect is that a few writers may fail because of writing contention. It can be solved by redo or just left alone due to the very low possibility and no harm to the ACID principle.

The writing performance bottleneck comes to single writer on external disks, especially random writing. We are going to improve writing performance without sacrificing reading performance. It is a trade-off between LMDB's simplicity currently control and multi-writer's functionality completeness. In micro-benchmark[4], InnoDB of MySQL using multi-writer has much worse read performance compared to LMDB using single-writer. Therefore, we keep the single-writer design of LMDB at this work, and we use a WAL to speed up single writer (see Fig. 8). Firstly, incoming write transactions are sequentially appended to a log file, and then do compaction every one minute. In LMDB case, the update is immediately applied to graph storage in memory, which means the data of B+ tree has been updated, and all the read operations do not need extra check in log, which is very different from compaction in RocksDB.

Optimistic concurrent control solves the unnecessary contention of read operations, while WAL solves the write bandwidth of external disks. These two techniques enhance the concurrent writer ability of LMDB, and make LMDB obvious shortcoming and qualified key-value store for graph storage.

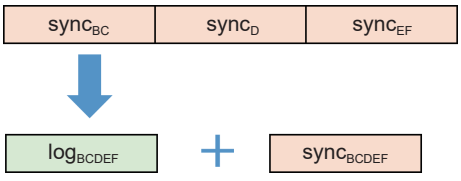


Fig. 8 WAL example. B+ tree in memory is always updated, while B+ tree on disk is updated asynchronously using WAL to ensure durability.

3.7 Other design

Along with the key techniques enabling graph storage's efficiency and functionality, it also adopts some existing techniques that are used in common databases, but requires extension and optimization to match graph models.

Batched data import. It is observed that inserting sorted data are about 10 times faster than discrete data. To accelerate the import stage when storage is empty, we use an external sorting based importing method. The assumption of this technique is that the primary keys of vertices can be loaded into memory, while edges are not necessary. During the import, firstly, each vertex is assigned a unique VertexUid and the mapping is held in memory. Secondly, replace the primary key to VertexUid in edges' original data, and store them as SrcVid and DstVid correspondingly. Finally, edges are batch-loaded as buckets, and each bucket is loaded to memory and sequentially writes data to graph storage. By this technique, the throughput of data import is able to achieve as much as 70 MB/s on a modern computer.

Index. Each property of a vertex or edge can be indexed as unique or non-unique. The B+ tree design is naturally applied to build compare-based index, e.g., INT64, String. In terms of full text index, the graph storage is integrated with Lucene[5] using a JNI interface.

Profiler. A profiler is integrated into graph storage to enhance stability. The profiler detects runtime transaction behavior and monitors memory leakage in system development and pre-release running.

4 Evaluation

4.1 Setup

We evaluated our graph storage design in Aliyun[6]; the parameters are listed in Table 1. All the tests are

Table 1 Environment of evaluation.

Item	Description
Instance	ecs.i3g.4xlarge
CPU	Intel 8269CY, 8 cores, 16 threads
Memory	128 GB
Disk	NVMe SSD

completed on a single machine, and ulimit is used in some experiments to simulate the out-of-memory case.

Different operations are designed for different purposes of evaluation.

● **Key-value operations.** The operations can be read or write, sequential or random of different value sizes.

● **Graph queries.** The targets of graph queries are vertices or edges. Writing related graph queries are vertex singular insert, vertex batch insert, edge singular insert and edge batch insert. Update and delete are not included to make it simple. Read-related graph queries are *K*-hop based, e.g., neighborhood lookup and pairwise shortest path.

● **Comprehensive graph workloads.** We follow the official audit process of LDBC SNB Interactive[7], which is similar to TPC-H for relational database. SNB Interactive is composed of seven short queries, eleven complex queries, and eight insert queries.

Synthetic and real-world data are used during evaluation. The former contains uniformly distributed data and SNB defined free-scale data, while the latter contains Twitter2010[8], which has around 41.65 million vertices and 1.47 billion edges. The Twitter2010 graph has a power-law distribution, e.g., the maximal out-degree is 3.0 million.

4.2 Scalability

Table 2 shows the throughput of six graph queries when the volume of data in graph storage changes, namely, strong scalability. The edge factor is 10 and one INT64 property and one String property are assigned to vertex, while the edge has no property.

The performance of the six queries may experience a slight decrease in throughput when the data can fit in memory. This is due to the fact that the B+ tree structure used to store the graph data has more levels, which can slightly slow down both reading and writing processes. However, when the data exceed the available memory, the throughput experiences a significant decline.

Vertex insert is sequential write in graph storage, since every vertex is assigned an auto-increment integer. Vertex batch insert has about 30 times higher throughput than vertex singular insert, meaning that synchronizing small pieces to disk costs much more than sequential data structure update in memory.

Edge insert randomly selects two existing vertices and inserts an edge between them. This means that edge insert is a random write in graph storage. Both edge singular insert and vertex singular insert have similar throughput, as the main overhead comes from synchronization rather than in-memory operations. The significant difference in throughput between edge batch insert and vertex batch insert is due to the difference between sequential and random writes.

Neighborhood lookup accesses the out-edges and in-edges of a randomly selected vertex. The topology of the graph is adaptively packed into one value or more. Because the edges are evenly distributed with 20 neighborhoods per vertex, the split situation merely appears. The access pattern of neighborhood lookup is 1-hop like reading one key-value. The throughput of neighborhood lookup exceeds one million Query Per Second (QPS), but sharply drops to 71 000 QPS when accessing data on disk.

Pairwise shortest path computes the shortest path between two vertices at the longest length of three. Pairwise shortest path behaves as 3-hop random read in

Table 2 QPS of graph workloads in different database sizes. Memory is limited to 10 GB. -S-Ins. and -B-Ins. stand for singular and batch insert, respectively.

$\|V\|$	Vertex-B-Ins.	Edge-B-Ins.	Vertex-S-Ins.	Edge-S-Ins.	Neighborhood lookup	Shortest path	DB size (MB)
100 000	330 590	66 460	11 318	12 848	1 310 460	131 118	25
200 000	328 543	56 419	10 896	12 302	1 381 920	146 216	61
400 000	330 042	48 897	10 798	12 502	1 356 900	139 376	113
800 000	324 460	43 758	10 203	11 092	1 336 650	139 396	227
1 600 000	316 510	39 652	9841	10 197	1 299 270	137 984	466
3 200 000	307 936	36 843	9757	10 636	1 275 310	137 096	955
6 400 000	298 254	34 147	9078	10 258	1 314 730	136 333	1909
12 800 000	301 011	31 794	9300	10 016	1 230 460	137 507	3824
25 600 000	298 293	29 326	9110	9598	1 188 270	133 115	8056
51 200 000	301 298	15 302	8826	4954	129 393	17 898	16 589
102 400 000	299 288	5715	8727	3253	71 053	10 801	33 467

graph data access pattern. The throughput is around 130 000 QPS, which is an important metric for the following optimization evaluation.

4.3 Comparison of tree-structured key-value stores

In this experiment, we are going to discuss the trade-off between B+ tree and LSM tree, i.e., LMDB and RocksDB. The experiment starts with operations in underlying key-value, then graph queries used in Section 4.2.

RocksDB is configured using TransactionDB, while LMDB has the default configuration. In Table 3, sequential and random read are compared with different value sizes. Within key-value store, zero-copy technique is used in reading, and we record the latency of getting the reading pointers for comparison. LMDB has nearly constant latency on sequential and random read. This is likely due to the mmap memory management of LMDB, which depends on 4 KB OS page size, resulting in lower latency for sequential reads. In contrast, the multi-level structure of LSM tree in RocksDB requires multiple comparisons, leading to higher latency as the database size increases. When the value size reaches 64 KB, the total data can no longer fit in memory, causing a dramatic increase in latency for RocksDB. Overall, LMDB exhibits significantly better sequential reading latency and is more stable when the database size increases.

In Table 4, RocksDB has a random write bandwidth of about 26 times more than LMDB when value size is small, thanks to WAL feature, while the sequential write gap is about two times. When the value size goes above 4 KB, LMDB catches up with RocksDB in sequential write. LMDB uses mmap as the data access method, resulting in a 4 KB minimal access block and being inefficient below 4 KB. RocksDB is not suggested for larger value sizes; its writing bandwidth becomes even lower when value size increases, which should be configured using the BlobDB engine[9].

To confirm it, the above evaluation is repeated on a slower disk, with the maximal Input/output Operations Per Second (IOPS) dropping from 200 000 to 10 000. Under this condition, the random write query of RocksDB outperforms LMDB by two to six times. Therefore, we can conclude that the advantages of RocksDB are much fewer when the ability of disks rapidly increases.

As a conclusion of Tables 3 and 4, LMDB has better and stable read latency over RocksDB, and a

Table 3 Reading latency of key-value queries in LMDB and RocksDB in different value sizes, lower is better. The tested database has per-inserted one million key-value pairs of desired size, and it reads one million values of ten threads.

Value size	Reading latency (ms)			
	Read-Seq-LMDB	Read-Seq-RocksDB	Read-Rand-LMDB	Read-Rand-RocksDB
16 B	4	487	185	33
64 B	6	523	197	36
256 B	9	584	205	43
1024 B	17	728	201	78
4 KB	5	1958	185	210
16 KB	5	5153	184	234
64 KB	5	38 314	185	32 407

Table 4 Write bandwidth of key-value write queries in LMDB and RocksDB in different value sizes using ten threads, larger is better.

Value size	Write bandwidth (MB/s)			
	Write-Seq-LMDB	Write-Seq-RocksDB	Write-Rand-LMDB	Write-Rand-RocksDB
16 B	12.97	24.50	0.30	7.79
64 B	47.55	93.70	1.10	1.10
256 B	143.34	345.01	4.27	104.66
1024 B	360.18	721.13	15.17	312.39
4 KB	686.67	533.75	53.38	516.00
16 KB	1258.95	421.39	158.80	271.22
64 KB	994.04	150.37	352.67	70.00

comparable writing performance when the value size is no less than 4 KB. For a read optimized graph storage design, LMDB is a better candidate.

Synthetic graph data and workloads in Section 4.2 are used for graph workloads evaluation, and the RocksDB is configured as TransactionDB while leaving most of the configurations default. As Fig. 9 shows, LMDB has up to 4.6 times better reading performance compared with RocksDB, thanks to the quick lookup mechanism in B+ tree. Vertex insert is only 0.1 times because the vertex property is 20 B, and the gap can be much narrower when the property size becomes over 4 KB. In terms of edge batch insert, LMDB performs at 50% slowdown when the number of vertices, $|V|$, is 51 200 000, while RocksDB drops to 30%. This results in a 4.3 times difference in performance. The size of the database for LMDB is 16.6 GB, while it is 9.8 GB for RocksDB. Even RocksDB has a smaller database size, it consumes more memory for data processing and reaches an out-of-memory performance drop earlier. As the graph data becomes too large to make use of memory, the difference between the two storage engines becomes smaller as the bottleneck shifts to disk-memory data exchange.

The inefficient reading performance of RocksDB comes for several reasons. One of the most serious reasons is unpredictable background compaction. The read query performance on data before and after compaction has a two to three times' difference. We also use the official profiling tool[10] to analyse the reading performance of RocksDB, finding that RocksDB needs to build an iterator data structure to abstract the data flow, and this part of the cost occupies about 80% overall. Simplifying the principle of LMDB can avoid such kinds of cost.

The write workloads in the evaluation use a single thread, while read workloads use ten threads. RocksDB is able to get better writing results to outperform LMDB. An early implementation of multi-thread writing can speed up to three times in vertex insert and six times in edge insert. It is obvious that RocksDB does better in writing performance, especially when properties are small.

In conclusion, LMDB is read-optimized and deals with larger property data better, while RocksDB is more suitable for smaller property and gets better writing performance. We prefer LMDB due to the reading performance requirement in financial real application profiling.

4.4 Graph packing

Figure 10 compares the QPS of compact packing and indexed packing. Regardless of vertex or edge, read or write, compact packing always has better performance. The read-related queries, i.e., neighborhood lookup and pairwise shortest path, have up to 1.5 times' speedup, thanks to less key-value access of packing.

The total key number of indexed packing is twenty times more than compact packing, equal to twice of the edge factor. Also, the key size of EdgeUid is around four times larger than VertexUid, which largely inflates the storage size, up to five times. Therefore, writing related queries also have up to 1.4 times' speedup. Edge singular insert of indexed packing is supposed to write only the edge related data while compact packing

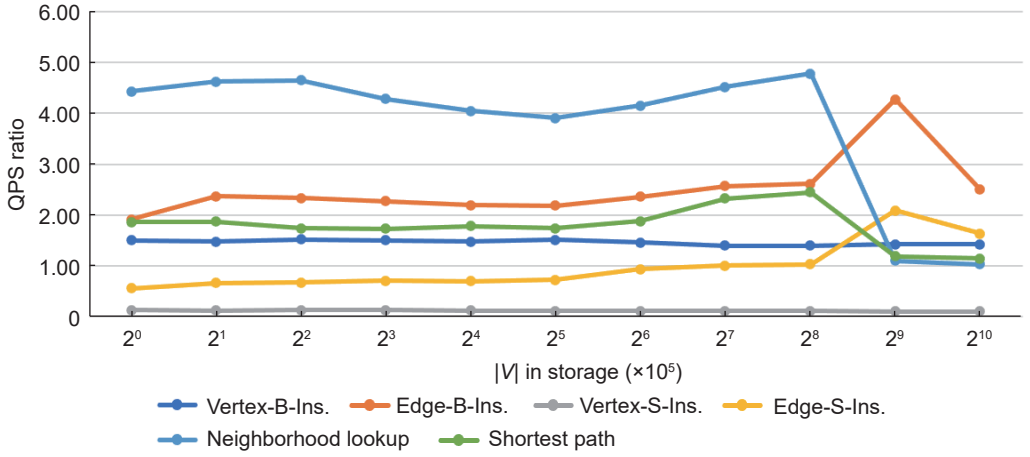


Fig. 9 QPS ratio of graph workloads of LMDB and RocksDB. The larger the number, the better the performance of LMDB. Memory usage is limited to 10 GB. -S-Ins. and -B-Ins. stand for singular and batch insert, respectively.

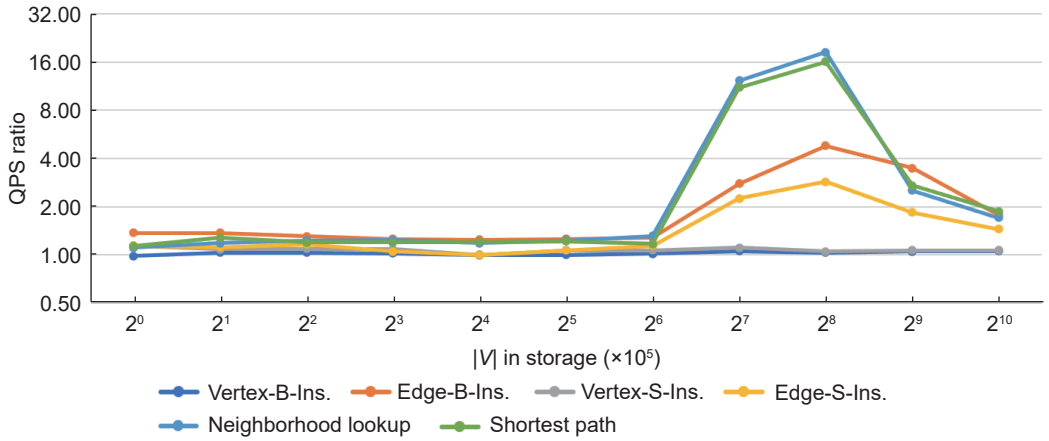


Fig. 10 QPS ratio of compacting and indexed packing using LMDB. The larger the number, the better the performance of compacting. Memory usage is limited to 10 GB. -S-Ins. and -B-Ins. stand for singular and batch insert, respectively.

needs to rewrite the whole value of the vertex. Indexed packing quickly runs out of memory, resulting in 11.07 to 18.39 times' huge gaps in read queries. When both methods read data on external disks, compact packing keeps 1.71 to 1.88 times' benefit.

However, the data that need to be updated is both within 4 KB, i.e., the basic access block of LMDB. Therefore, the performance of edge singular insert makes no difference.

4.5 Writer optimization

To solve the performance bottleneck of singer writer in LMDB, concurrent writer optimization is introduced in Section 3.6. The experimental result in Table 5 shows the random writing bandwidth compared with original direct synchronization using 128-bytes value. WAL can replace random writing in B+ tree with sequential writing in the log file, and the bandwidth improved from 7.53 MB/s to 10.73 MB/s in single writer. When the number of writers increased to ten, the bandwidth also increased to 21.19 MB/s, thanks to concurrent access to log files. In the compaction stage to replay logs, the operation on the same key can be merged to reduce the total write amount in LMDB.

4.6 GDBMS comparison

We use *K*-hop as a typical graph access workload on real world data, i.e., Twitter's social network, to compare the performance with other well-known graph databases, including Neo4j[11], JanusGraph[11], Dgraph[12], and an anonymous graph database TG.

In Table 6, TG has the shortest import cost and smallest on-disk size. TuGraph is 14% slower because of the single write limitation, but faster than the other three GDBMSs. The original Twitter data size is 24.6 GB; TuGraph merely has compression, i.e., occupies 24 GB. The in-memory design of TG trades off between immediate write to disk and extreme memory compress, while TuGraph does not implement any content-based compression, and chooses a more balanced design, focusing on high reading performance as well as ACID guarantee. Thanks to batched data import in Section 3.7, TuGraph gets first-class import throughput.

In Fig. 11, the latency of *K*-hop in various GDBMS is given. Each test should be completed within 2 h. During the runtime of Dgraph 6-hop test, it runs out of memory and crashes. In spite of being 1.8 ms slower than Dgraph in 1-hop, TuGraph has the best latency of all other tests, with even better orders of magnitudes.

This substantial advantage can be attributed to

Table 5 Bandwidth of WAL and direct synchronization, using random write of 128-bytes value.

Sync. method	Bandwidth (MB/s)	
	Single writer	Ten writers
Direct	7.53	9.19
WAL	10.73	21.19

Table 6 Import cost and on-disk size of importing Twitter data.

GDBMS	Import cost (s)	On-disk size (GB)
TuGraph	652	24
Neo4j	979	48
TG	577	6.1
Dgraph	2704	8.8
ArangoDB	8851	106
JanusGraph	2802	51

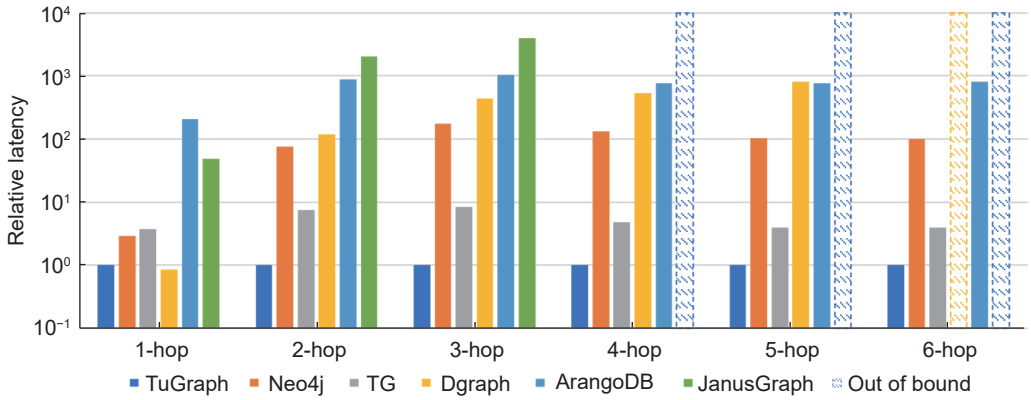


Fig. 11 ***K*-hop latency of different GDBMSs.**

several factors related to TuGraph's storage model and implementation. Firstly, TuGraph employs a compact packing design that efficiently handles read-only *K*-hop workloads with a smaller memory footprint compared to other GDBMS. Secondly, TuGraph leverages inner-query parallelism to effectively utilize CPU resources. This feature is not supported in some other GDBMSs, such as JanusGraph. Thirdly, TuGraph avoids unnecessary data copying by utilizing pointers within a single machine.

We plan to conduct a write benchmark using graph operations to evaluate the write performance of TuGraph and Neo4j, representing GDBMSs. Figure 12 illustrates the QPS ratio of TuGraph and Neo4j for singular and batch inserts of vertices and edges. It is worth noting that Neo4j adopts a linked block approach for storing graph data, which differs significantly from TuGraph's B+ tree based compact packing, resulting in divergent performance outcomes.

In terms of vertex inserts, TuGraph and Neo4j show similar performance in singular conditions, with no significant difference between them. However, when batch inserts are performed, TuGraph outperforms Neo4j by 1.55 times. This improvement is attributed to the sequential write pattern of vertex insert, which comes from the single-writer nature of TuGraph's LMDB. By batching the inserts together, the overall write QPS is significantly enhanced.

On the other hand, when it comes to edge inserts, TuGraph faces some challenges. Each randomly added edge requires repacking the value, resulting in a less favorable batch edge insert pattern. In fact, in comparison to Neo4j, TuGraph's batch edge insert performance is only 0.44 times. When examining singular edge inserts, Neo4j's performance is similar to that of TuGraph. This similarity is due to the fact that Neo4j needs to manage numerous small data blocks to ensure ACID guarantees, resulting in comparable QPS for singular edge inserts.

In Fig. 12, there are some noteworthy observations. When the data scale is small, Neo4j's implementation incurs additional overhead due to fixed internal data

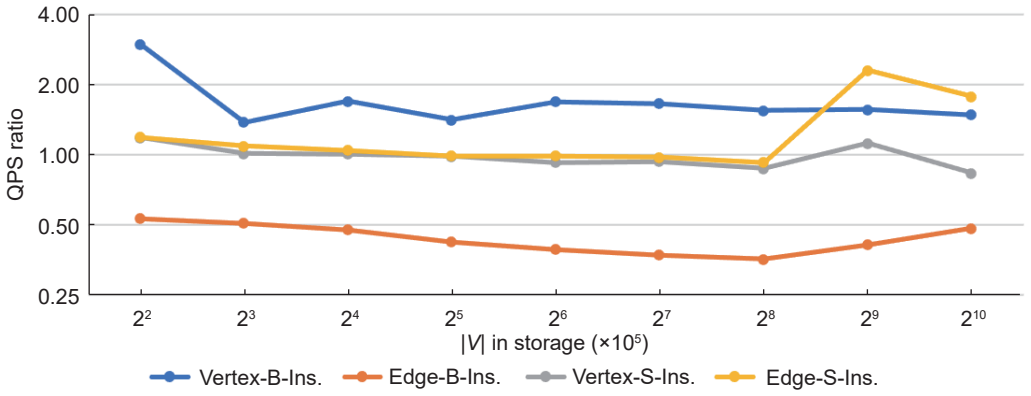


Fig. 12 QPS ratio of graph workloads of TuGraph and Neo4j. The larger the number, the better the performance of TuGraph. Neo4j uses 10 threads.

structures. As the data scale increases, Neo4j is the first to exceed memory limits and subsequently slows down. The QPS ratio stabilizes again when both TuGraph and Neo4j encounter memory limitations.

4.7 Mixed throughput

LDBC SNB interactive defines graph workload which simulates a social network scenario, to evaluate overall performances of GDBMSs. The environment is Amazon Web Services, using an r5d/12xlarge instance. Two machines are used for driver instance and server instance, to simulate the real client/server network connection.

In Table 7, a scale factor of 300 means the total data size is about 300 GB, and Time Compression Ratio (TCR) is a parameter for turning inside the benchmark. We can see that the load rate is up to 77.12 MB/s and the total throughput is up to 12 934.61 queries per second. During benchmarking, the execution time of read and write is about ten to one, which is indicated by Facebook online system profiling. The storage of TuGraph is designed to handle SNB's mixed read and write graph workload, and it achieves the first place on the board of SNB Interactive. The full result can be found on LDBC's official website[7].

5 Related Work

Graph storage refers to the organization of vertices, edges, and properties data in persistent storage, which is important for both query performance and storage capacity. There are several approaches to graph storage used by different GDBMSs.

Some GDBMSs design their own graph storage solutions. Neo4j[1] uses a linked storage block to adapt the random vertex and edge access. The vertex record and edge record are stored at a fixed size, using a pointer to link two related data, resulting in losing the opportunity to compress data and sequential access. ArangoDB[13] uses hash tables for quick vertex and edge lookup, while the edges of a vertex are put in a linked list. The shortage of hash tables includes lacking support for range scan, which may benefit from the locality of vertex.

Table 7 LDBC SNB Interactive benchmarking result of TuGraph.

Scale factor	Load (MB/s)	TCR	Throughput QPS
30	67.29	0.0028	12 252.50
100	75.70	0.0104	12 934.61
300	77.12	0.0360	12 721.24

Other GDBMSs, such as JanusGraph[11], use established NoSQL stores as their underlying graph storage. JanusGraph is able to use different storage backend as a wide-column store, including Cassandra[14], HBase[15], Google Cloud Bigtable[16], and Berkeley DB[17]. This allows for flexibility, but requires a standard interface and may not allow for deep co-design.

Document stores, such as OrientDB[18], are well-suited for storing and indexing hierarchical data, such as property graph data in this case. OrientDB has vertex documents and edge documents to record graph data and supports abundant property operations. At the same time, document structured data are not as efficient as hand-tuned data arrangement. Wide-column stores, such as JanusGraph[11], can store different numbers of columns for each row, which is useful for storing indeterminate numbers of edges for each vertex. JanusGraph packs each vertex's properties and neighbor edges in a wide-column store. Each property and edge are regarded as cells that can be further stored as key-value pairs to store properties of edges. TuGraph uses key-value store to achieve simplicity and scalability. The value can be furthest redesigned to adapt to graph-specific processing.

Some GDBMSs, such as TigerGraph[2], GraphflowDB[19], and A1[20], focus on in-memory graph workloads. TigerGraph compresses graph data and tries to load the whole graph into memory, which always reserves memory for the whole graph data even if the data is currently not required in workload.

Distributed processing is supported by some other GDBMSs, which comes with the data consistency problem. It can be handled by user-side, or only master replica is able to write[21, 22], or two-phase committed related design. For example, Cosmos DB[23] uses a last-write-win policy to resolve writing conflicts. Distributed processing is likely having order of magnitude slows down compared with single machine implementation, in the case data can be handled in a single machine.

Hybrid Transactional/Analytical Processing (HTAP) is another area of GDBMS research, with systems like LiveGraph[24] using a transactional edge log and concurrency control mechanism to support both analytical and transactional graph workloads.

This work focuses on graph storage, and could

potentially be combined with research on execution models, such as those presented in Tao[25] and Grasper[26].

6 Conclusion

Graph databases are rapidly gaining popularity as they provide efficient storage and retrieval of graph-structured data. In this paper, we have presented the design and implementation of our graph storage engine, TuGraph. We began by summarizing common access patterns for graph databases, followed by the introduction of key techniques aimed at enhancing the performance of graph storage engines, such as adaptive compact packing of graph data and property packing. Subsequently, we provide an in-depth overview of the implementation of these techniques in TuGraph.

The experimental results have showcased the outstanding performance of TuGraph, surpassing other popular graph database management system like Neo4j and JanusGraph by several orders of magnitude. Moreover, TuGraph has demonstrated good performance on standard GDBMS benchmarks, including the LDBC SNB.

While these achievements are commendable, there are still opportunities for further improvement in this field. For instance, exploring a hash table based approach for vertex lookup holds the potential to deliver faster results compared to tree-structured key-value stores. Additionally, extending the capabilities of TuGraph to support distributed graph storage is an area that we aim to explore.

In conclusion, our work on TuGraph represents notable advancements in graph storage technology. The impressive performance demonstrated by TuGraph, coupled with the identified areas for future improvement, lays a foundation for further research and development in the field of graph databases.

References

[1] Neo4j, https://neo4j.com, 2023.

[2] A. Deutsch, Y. Xu, M. Wu, and V. Lee, TigerGraph: A native MPP graph database, arXiv preprint arXiv: 1901.08248, 2019.

[3] Z. Fu, Z. Wu, H. Li, Y. Li, M. Wu, X. Chen, X. Ye, B. Yu, and X. Hu, GeaBase: A high-performance distributed graph database for industry-scale applications, *Int. J. High Perform. Comput. Netw.*, vol. 15, nos. 1&2, pp. 12–21, 2019.

[4] Symas Corp, Memcache benchmark, http://www.lmdb.tech/bench/memcache/, 2013.

[5] Lucene, https://lucene.apache.org/, 2023.

[6] Alibaba cloud, https://www.aliyun.com/, 2023.

[7] Ldbc social network benchmark, https://ldbcouncil.org/benchmarks/snb/, 2023.

[8] Twitter follower network 2010, https://snap.stanford.edu/data/twitter-2010.html, 2023.

[9] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, WiscKey: Separating keys from values in SSD-conscious storage, *ACM Trans. Storage*, vol. 13, no. 1, pp. 5, 2017.

[10] Perf context and IO stats context, https://github.com/facebook/rocksdb/wiki/perf-context-and-io-stats-context, 2022.

[11] JanusGraph, https://janusgraph.org, 2023.

[12] Dgraph, https://github.com/dgraph-io/dgraph, 2023.

[13] ArangoDB, https://www.arangodb.com/, 2023.

[14] Apache Cassandra, https://cassandra.apache.org/, 2023.

[15] Apache HBase, https://hbase.apache.org/, 2023.

[16] Google Cloud Bigtable, https://cloud.google.com/bigtable/, 2023.

[17] Oracle Berkeley DB, https://www.oracle.com/database/technologies/related/berkeleydb.html, 2023.

[18] OrientDB, https://orientdb.com, 2023.

[19] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu, Graphflow: An active graph database, in *Proc. 2017 ACM Int. Conf. Management of Data*, Chicago, IL, USA, 2017, pp. 1695–1698.

[20] C. Buragohain, K. M. Risvik, P. Brett, M. Castro, W. Cho, J. Cowhig, N. Gloy, K. Kalyanaraman, R. Khanna, J. Pao, et al., A1: A distributed in-memory graph database, in *Proc. 2020 ACM SIGMOD Int. Conf. Management of Data*, Portland, OR, USA, 2020, pp. 329–344.

[21] Aws Neptune, https://aws.amazon.com/neptune/, 2023.

[22] Alibaba GDB, https://www.aliyun.com/product/gdb/, 2023.

[23] Azure Cosmos DB documentation, https://docs.microsoft.com/en-us/azure/cosmos-db, 2023.

[24] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen, LiveGraph: A transactional graph storage system with purely sequential adjacency list scans, arXiv preprint arXiv: 1910.05773, 2019.

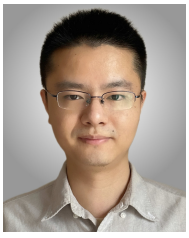
[25] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, et al., TAO: Facebook's distributed data store for the social graph, in *Proc. 2013 USENIX Conf. Annual Technical Conf.* (*USENIX ATC 13*), San Jose, CA, USA, 2013, pp. 49–60.

[26] H. Chen, C. Li, J. Fang, C. Huang, J. Cheng, J. Zhang, Y. Hou, and X. Yan, Grasper: A high performance distributed system for OLAP on property graphs, in *Proc. ACM Symp. on Cloud Computing*, Santa Cruz, CA, USA, 2019, pp. 87–100.

Heng Lin is currently a postdoctoral researcher at Peking University and a technical expert at Ant Group, China. He received the PhD degree in computer science from Tsinghua University in 2018. His interests include graph database and parallel computing.

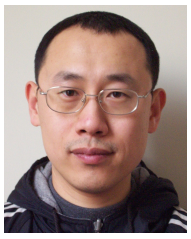
Zhiyong Wang received the bachelor degree from Hangzhou Dianzi University in 2014. He previously worked at DiDi Global Inc., and is currently a technical expert of Ant Group, China. He has many years of experience in large-scale distributed storage system development, including SQL, NOSQL, GRAPH, OLTP, and OLAP.

Shipeng Qi received the bachelor degree in computer science and technology from Huazhong University of Science and Technology, and the second bachelor degree in economics from Wuhan University in 2017. He is currently a graph database engineer at Ant Group, China, a developer relations advocate in TuGraph-DB community, and a benchmark expert in Linked Data Benchmark Council leading the LDBC Financial Benchmark. Before he joined Ant Group, he served as a co-founder in a startup building ToB AI platforms.



Xiaowei Zhu is currently a researcher at Ant Technology Research Institute, Ant Group, China. He received the bachelor and PhD degrees in computer science from Tsinghua University in 2013 and 2018, respectively. His research focuses on storage and computation aspects of graph processing systems.

Chuntao Hong received the PhD degree from Tsinghua University, China in 2011. He later joined Microsoft Research Asia and co-founded Beijing FMA Technology Co., Ltd. In 2020, he joined Ant Group, where he currently leads the development of Ant Group's graph database.



Wenguang Chen is currently a professor at Tsinghua University, and the president of Ant Technology Research Institute, Ant Group, China. He received the bachelor and PhD degrees in computer science from Tsinghua University in 1995 and 2000, respectively. His research focuses on parallel and distributed systems and programming systems.

Yingwei Luo received the PhD degree in computer science from Peking University, China in 1999. He is a full professor of computer science at the School of Computer Science, Peking University. His research interests include operating system, system virtualization, and cloud computing.