Helios: A Disaggregated MySQL Database That Scales Without Replication

Masahide Fukuyama
Keio University
Fujisawa, Japan
massa5101@keio.jp

Yusuke Miyazaki
Keio University
Fujisawa, Japan
miyayu@keio.jp

Sho Nakazono
LY Corporation
Tokyo, Japan
nikezono@gmail.com

Hideyuki Kawashima
Keio University
Fujisawa, Japan
river@sfc.keio.ac.jp

***Abstract*—Helios is a disaggregated MySQL database that scales query processing without read-serving replicas. MySQL nodes execute reads and writes through a shared in-memory transaction engine while retaining unchanged query-processing cores. It validates consumed point and range reads and orders commits, preserving external consistency. To reduce per-row RPCs, Helios compiles MySQL's chosen access paths into dependent read plans. Embedded DuckDB executes consistent analytics over PAX data. At 1,024 TPC-C sessions, four query nodes deliver $3.24\times$ the throughput of one. On TPC-H SF=10, Helios is $2.20\times$ and $62.4\times$ faster than TiDB with TiFlash and MySQL with InnoDB, respectively.**

***Index Terms*—MySQL, disaggregated databases, external consistency, query processing, transaction processing**

I. Introduction

Modern in-memory transaction engines can process millions of transactions per second on a single many-core server [1]. A DBMS must also parse and optimize SQL statements and execute the resulting query plans. As transaction engines become faster, this query-processing work can limit end-to-end throughput.

Databases often scale query processing through replicas or shards. Replicas improve read capacity and availability but require update propagation and may return stale data. Shards increase storage and transaction capacity but require multi-shard coordination. They suit workloads needing those capacities.

Helios instead lets multiple read/write-capable MySQL nodes share one in-memory transaction engine without adding replicas or shards. The engine validates their transactions and orders commits, preserving external consistency across nodes. Helios derives dependent read plans from access paths chosen by MySQL's optimizer to reduce network round trips. It also embeds DuckDB in the storage server for consistent analytical reads without a separate replica.

II. Design

Figure 1 shows how Helios extends Kamo [2], a MySQL storage engine backed by LineairDB. It connects each query node to the shared server without changing MySQL's query-processing core.

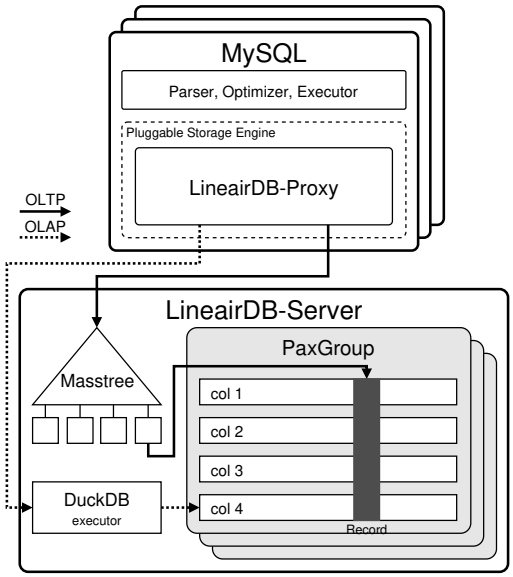


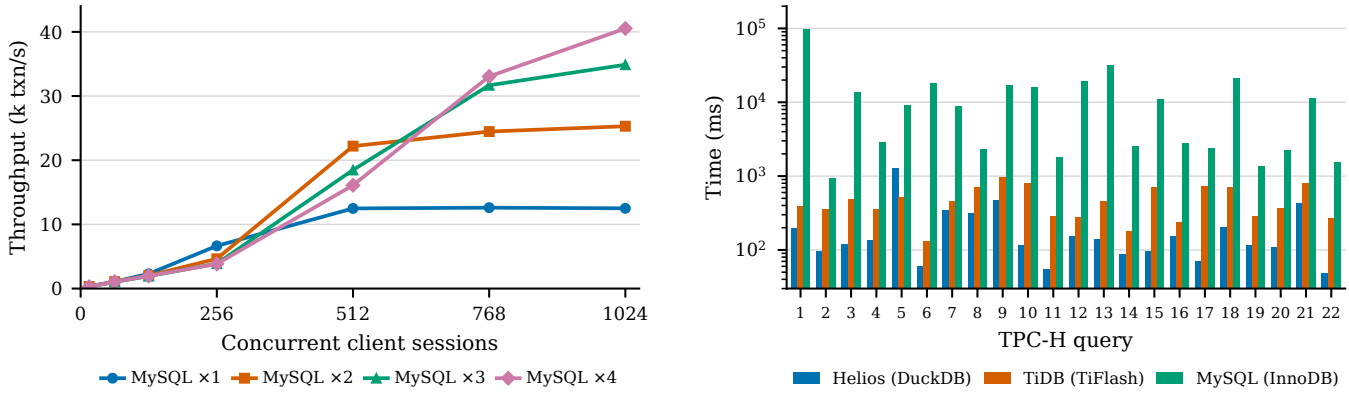
Fig. 1. MySQL nodes share a LineairDB server whose PAX layout serves OLTP through Masstree (solid) and OLAP through embedded DuckDB (dashed).

A. Shared Transaction Engine

Helios adds a PAX layout [3] shared by OLTP and OLAP to LineairDB [4], an in-memory transactional key-value store. Query nodes maintain no read-serving replicas, so replication lag cannot cause stale reads. For each OLTP transaction, they retain point- and range-read evidence consumed by MySQL. At commit, the storage server revalidates this evidence using Silo's commit protocol [5]. It establishes a serial order consistent with real time, providing external consistency across nodes.

B. Plan-derived Prefetch

Mapping MySQL's row-at-a-time handler calls directly to RPCs creates many synchronous round trips. For each supported statement, Helios instead compiles storage accesses from the optimizer-chosen query execution plan (QEP) into a read plan sent in one RPC. Later steps derive lookup keys from rows returned by earlier steps, letting the server resolve dependent accesses without further round trips. MySQL's



(a) TPC-C throughput (1,024 warehouses).

(b) TPC-H execution time (SF=10).

Fig. 2. TPC-C query-node scaling and TPC-H per-query execution time.

query executor processes the returned rows, and only evidence for consumed reads enters commit validation.

C. Consistent Analytical Reads

Embedded DuckDB [6] executes read-only queries over the OLTP-updated PAX layout, without a separate replica. Before each query, Helios activates before-image capture, takes an epoch cut, advances the commit epoch beyond it, and waits until included commits finish installing their writes. The cut includes every transaction committed before the query began and remains fixed throughout the query. Post-cut transactions save before-images before overwriting rows. DuckDB reads live data or a before-image for each row at the cut. Because one transaction's writes share an epoch, the query observes all or none of them. This provides a consistent read without blocking OLTP writers.

III. Evaluation

A. Experimental Setup

Table I summarizes the deployment, and Figure 2 reports the results. We run BenchBase's TPC-C workload [7] with all five transaction types, serializable isolation, and 1,024 warehouses. Two load generators run 16–1,024 closed-loop sessions over persistent JDBC connections, which HAProxy distributes across one to four MySQL query nodes. We measure each OLTP configuration for 60 seconds. For TPC-H, we compare Helios (DuckDB), TiDB (TiFlash with MPP), and MySQL (InnoDB) on all 22 queries at scale factor 10.

B. TPC-C Query-node Scaling

One query node plateaus near 12.6k txn/s from 512 sessions. At 1,024 sessions, it delivers 12.5k txn/s while MySQL is CPU-saturated and storage-server CPU utilization remains below 20%. Four nodes reach 40.5k txn/s, and the curve is still rising.

C. TPC-H Query Performance

Helios completes all 22 queries in 4.73 s, 2.20× faster than TiDB with TiFlash (10.4 s) and 62.4× faster than MySQL with InnoDB (295 s).

TABLE I
EC2 deployment: Ubuntu 24.04.4 LTS, 64 vCPUs, and 128 GiB (c6i) or 512 GiB (r6i).

Workload	Layer	Component	Count	Instance type
TPC-C	Client	BenchBase	2	c6i.16xlarge
	Routing	HAProxy	1	c6i.16xlarge
	Query	MySQL	1–4	c6i.16xlarge
	Storage	LineairDB	1	r6i.16xlarge
TPC-H	Query + Storage	Helios (DuckDB)	1	c6i.16xlarge
	Query + Storage	TiDB (TiFlash)	1	c6i.16xlarge
	Query + Storage	MySQL (InnoDB)	1	c6i.16xlarge

IV. Conclusion

Helios scales MySQL query processing around a shared transaction engine using dependent read plans and epoch-consistent PAX analytics. Future work will evaluate mixed workloads and add fault-tolerant storage replication.

References

[1] T. Tanabe, T. Hoshino, H. Kawashima, and O. Tatebe, "An analysis of concurrency control protocols for in-memory databases with CCBench," *Proc. VLDB Endow.*, vol. 13, no. 13, pp. 3531–3544, 2020.

[2] mitou-Kamo, "LineairDB MySQL Custom Storage Engine," GitHub repository: https://github.com/mitou-Kamo/LineairDB-storage-engine, 2026.

[3] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," in *Proc. VLDB*, 2001.

[4] S. Nakazono, Y. Bessho, and T. Nakamori, "LineairDB: Transactional key-value storage with epoch-based synchronization," *Computer Software*, vol. 41, no. 1, pp. 15–35, 2024.

[5] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *Proc. SOSP*, 2013.

[6] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," in *Proc. SIGMOD*, 2019.

[7] D. E. Difallah, A. Pavlo, C. Curino, and P. Cudré-Mauroux, "OLTP-Bench: An extensible testbed for benchmarking relational databases," *Proc. VLDB Endow.*, vol. 7, no. 4, pp. 277–288, 2013.