LakeMem: An Elastic Disaggregated-Memory Caching Layer for Analytical Processing Systems

Xinyi Yu
yuxinyi02@stu.xmu.edu.cn
Xiamen University &
Alibaba Cloud Computing
Xiamen, China

Yingqiang Zhang
yingqiang.zyq@alibaba-inc.com
Alibaba Cloud Computing &
Zhejiang University
Hangzhou, China

Hao Chen*
ch341982@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Zhaoxiang Huang
huangzhaoxiang@stu.xmu.edu.cn
Xiamen University &
Alibaba Cloud Computing
Xiamen, China

Xinjun Yang
xinjun.y@alibaba-inc.com
Alibaba Cloud Computing
Sunnyvale, CA, USA

Feifei Li
lifeifei@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Chuan Sun
hualuo.sc@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Jing Geng
wormhole.gl@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Jiong Xie
xiejiong.xj@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Ninglong Weng
ninglong.wnl@alibaba-inc.com
Alibaba Cloud Computing
Hangzhou, China

Yiming Zhang*
zhangyiming@cs.sjtu.edu.cn
Shanghai Jiao Tong University &
Xiamen University
Shanghai, China

Abstract

As lakehouse/lakebase architectures become more common, analytical processing workloads, often running over shared storage with open table formats, are frequently memory-bound: engines cache large base tables for throughput and maintain substantial intermediate states (e.g., hash tables, shuffle buffers). This pressure makes disaggregated memory (DM) a natural choice, offering elastic capacity without per-node overprovisioning. However, most DM caches remain byte-uniform, treating all data alike and ignoring the semantic asymmetry between shared base-table data and node-private intermediate data.

To address this inefficiency, this paper presents LakeMem, a disaggregated-memory-based elastic caching layer for analytical processing systems. LakeMem adopts a dual-path architecture that distinguishes between shared and private data types, where (i) *PrivateCache* follows a client-centric model for private intermediate data to enable low-latency access with minimal coordination, and (ii) *SharedCache* provides a server-coordinated, globally coherent cache for shared base-table data to maximize reuse. We further propose a dynamic rebalancing mechanism that adaptively allocates memory across the two paths to maintain high resource utilization. We implement LakeMem as an internal prototype integrated with DuckDB and evaluate it on a cluster using DuckLake and *smallpond*. Our experiments show that LakeMem achieves 2.0× to 5.9× speedups on memory-bound queries compared to a hybrid DRAM-SSD cache.

CCS Concepts

• **Information systems** → **Information storage systems**.

Keywords

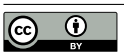
Online Analytical Processing (OLAP) systems, cache, memory disaggregation, lakehouse

ACM Reference Format:
Xinyi Yu, Yingqiang Zhang, Hao Chen, Zhaoxiang Huang, Xinjun Yang, Feifei Li, Chuan Sun, Jing Geng, Jiong Xie, Ninglong Weng, and Yiming Zhang. 2026. LakeMem: An Elastic Disaggregated-Memory Caching Layer for Analytical Processing Systems. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26), May 31-June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3788853.3803100

1 Introduction

As lakehouse/lakebase architectures gain adoption [6, 11, 35], shared storage and open table formats enable more analytical processing workloads to be executed over the same underlying data in modern systems, such as Snowflake [23], Apache Doris [4] and Amazon Redshift [15]. Consequently, modern analytical processing workloads

*Hao Chen and Yiming Zhang are the corresponding authors.



SIGMOD Companion '26, Bengaluru, India

ACM ISBN 979-8-4007-2450-3/2026/05
https://doi.org/10.1145/3788853.3803100

increasingly execute against shared object stores, while remaining highly memory-demanding: engines cache large base tables for throughput and maintain substantial intermediate states (e.g., join hash tables, sort/shuffle buffers) [47]. These two data classes have very different lifetimes and reuse patterns: base-table data is long-lived and reused across queries, whereas intermediate data is short-lived and query-private. Together, they create high, often bursty memory pressure, making elasticity and the balance between base and intermediate caches a first-order design concern.

Disaggregated memory (DM) [26, 43, 44] naturally emerges as a promising solution to address these challenges. By decoupling memory from compute into a shared, network-attached pool, DM provides elastic capacity that can be provisioned on demand to absorb workload fluctuations. Enabled by high-speed interconnects such as RDMA, DM offers a direct path to breaking rigid per-node memory boundaries, allowing memory to scale independently of compute.

However, existing DM architectures are semantically blind: they treat all data uniformly and enforce a universal access model designed for correctness under concurrent access. To provide strong consistency, these systems adopt generic coherence and synchronization mechanisms that keep data valid across multiple clients. While such guarantees are necessary for shared base-table data, they add substantial and unnecessary overhead for intermediate data, which is private to a single query task. For such accesses, the same mechanisms translate into avoidable costs such as multi-round-trip network protocols [53] for coherence or locking [29]. This additional latency falls directly on the critical path of query execution and weakens the low-latency benefits that disaggregated memory could otherwise provide for analytical processing workloads.

Motivated by this tension, we present LakeMem, a semantic-aware caching layer built on a distributed memory pool (DMP). This memory pool is a prior implementation [21, 48] of ours that provides a general-purpose memory substrate managing physical memory and exposing elastic regions through low-level APIs. LakeMem specializes the control plane above this substrate with tailored metadata, placement, and rebalancing policies, while preserving a lightweight data plane based on one-sided RDMA access. This layering avoids invasive changes to either the query engine or the underlying DM infrastructure.

LakeMem adopts a specialized dual-path architecture. Shared base-table data and query-private intermediate data therefore follow different coordination paths. For shared base-table data, which must remain accessible cluster-wide, *SharedCache* retains a server-coordinated path to provide global coherence and cross-node reuse. For private intermediate data, such coordination would lie on the critical path without improving sharing. *PrivateCache* therefore keeps metadata and indexing on the owning compute node and bypasses centralized coordination for ordinary data access. As a result, ordinary accesses on the private path do not require cluster-wide coherence or locking. Finally, a dynamic rebalancing mechanism orchestrates memory allocation between *SharedCache* and *PrivateCache* to maintain high utilization while mitigating severe performance cliffs.

In our evaluation, LakeMem achieves speedups of 2.0× to 5.9× for memory-bound queries and 1.2× to 2.6× for I/O-intensive queries, compared to a hybrid DRAM-SSD cache.

The paper makes the following contributions:

- The design and implementation of LakeMem, an OLAP-optimized caching layer that leverages disaggregated memory to provide fine-grained elasticity. LakeMem is implemented as an internal prototype integrated with a production distributed memory pool infrastructure.
- A dual-path architecture that provides distinct access paths for intermediate and base-table data, optimizing both latency and resource efficiency.
- A dynamic, workload-aware rebalancing mechanism that adaptively partitions memory resources to meet shifting workload demands.
- An evaluation showing substantial performance gains and high elasticity under analytical workloads on a controlled cluster.

The paper is organized as follows. We begin with background and motivation in §2 and §3. §4 and §5 present the overview and detailed design of LakeMem, respectively. We evaluate its performance in §6, discuss related work in §7, and conclude in §8.

2 Background

2.1 OLAP in the Lakehouse Architecture

Online Analytical Processing (OLAP) systems [4, 7, 15, 23, 39, 51] are designed to support complex analytical queries over large-scale datasets. To achieve high performance, systems employ optimizations such as columnar storage [14, 31], pipelined execution [20], and predicate pushdown [49, 50]. While these techniques are essential for query efficiency, system-level scalability and elasticity demand a more fundamental architectural innovation.

This has driven the widespread adoption of the lakehouse architecture, a paradigm shift from traditional shared-nothing models. As illustrated in Figure 1, the typical lakehouse design decouples stateless compute nodes from a central data lake residing in cloud object stores (e.g., Amazon S3 [1]). Data in the lake is typically stored in open columnar formats such as Parquet [5], enabling multiple, independent query engines to operate on a single source of truth. This decoupling is the key enabler for independent scaling of compute and storage, dramatically improving elasticity and operational simplicity. Systems such as DuckLake [11] and Mooncake [9] exemplify this approach by leveraging existing OLAP query engines and open columnar formats to build lightweight lakehouses.

2.2 Characteristics of OLAP Workloads

OLAP workloads are typically read-heavy, involving large-scale scans over massive datasets stored in persistent storage. Queries are often complex, combining multi-table joins, aggregations, sort operations, and window functions that place significant demands on both I/O and memory. These operations require access to base data while simultaneously generating substantial transient state during execution, leading to two primary categories of data that shape memory management design: *base-table data* and *intermediate data*.

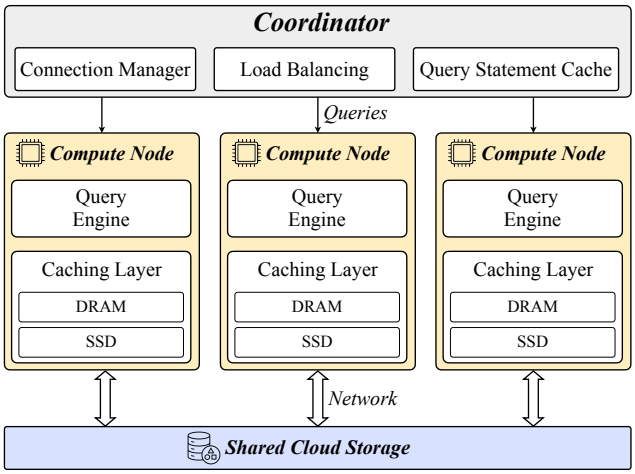


Figure 1: The architecture of shared-storage analytical lakehouse systems.

Base-table data. Base-table data denotes durable table content (columnar files in formats such as Parquet [5] or ORC [10]) stored in long-term cloud object stores. It is read repeatedly across queries and serves as the primary input for analytical workloads. This data has a long lifetime, independent of any single query, and is often accessed repeatedly across multiple workloads. In shared-storage architectures, base-table data is cached near compute nodes to reduce remote I/O, and its blocks are typically shared among concurrent queries and across different compute nodes. Access patterns are dominated by large sequential scans over data blocks ranging from several megabytes, placing high demands on storage bandwidth rather than per-access latency. Efficient management therefore prioritizes throughput and cache reuse over low-latency random access.

Intermediate data. Intermediate data is generated during the execution of complex analytical queries, particularly by operators such as hash joins, aggregations, and sorts. For example, a hash join operator builds an in-memory hash table from one input before probing with the other; this structure represents a key form of intermediate data. Such state is ephemeral and tightly bound to the lifetime of a specific query; it is not shared across queries or nodes and is discarded upon completion. Critically, its management must meet strict performance requirements: access to intermediate data is often on the critical path of query execution (e.g., probing a spilled hash table), and any delay directly blocks downstream processing. Therefore, storing this data in a low-latency, high-throughput memory layer is essential to avoid pipeline stalls and ensure predictable performance.

2.3 Caching Strategies in OLAP Systems

Modern OLAP systems in the lakehouse architecture employ caching to mitigate high-latency access to remote storage. These strategies generally fall into two architectural patterns [28]. The first architectural pattern is a node-local cache. In this model, each compute node manages its own local cache, typically organized as a two-tier hierarchy of DRAM and SSD. To facilitate data sharing, nodes may form a distributed layer using techniques such as consistent hashing [32], allowing them to fetch cached data blocks from one another. The second pattern involves external caching services such as Alluxio [38]. These systems can operate as independent, shared memory pools accessible to all compute nodes via RPCs. While this approach centralizes cache management, it introduces network latency for all cache interactions.

In both designs, these caches are responsible for storing two distinct types of data: blocks of persistent tables fetched from remote storage, and ephemeral intermediate data generated during query execution.

3 Motivation

3.1 Asymmetric Impact of Memory Capacity

To evaluate the impact of memory capacity on OLAP performance, we conduct experiments using DuckDB [41] with a DRAM-SSD hybrid cache, where all data is stored in remote object storage (see §6.1 for setup details). We vary the available memory from 20% to 100%. The 100% configuration is sized to cache all base-table data and accommodate the peak intermediate data generated during query execution.

We evaluate two representative classes of TPC-H [8] queries: I/O-intensive queries (Q1, Q5), characterized by high I/O intensity, and memory-bound queries (Q13, Q18), which generate large volumes of intermediate data. As shown in Figure 2, both query types benefit significantly from increased memory capacity. For Q1 and Q5, we observe a 2.5× speedup, primarily due to reduced remote I/O as frequently accessed table blocks are cached in memory. In contrast, Q13 and Q18 achieve substantially higher improvements—up to 9.2× and 9.8×, respectively, when sufficient memory prevents intermediate data from spilling to disk.

The performance contrast between these two query categories stems from their distinct memory behaviors. Table 1 quantifies this difference, showing that memory-bound queries produce intermediate data more than an order of magnitude larger than I/O-intensive queries. The resulting memory pressure forces the system to evict cached data when memory capacity becomes insufficient. However, base-table data and intermediate data respond very differently to memory scarcity.

Base-table data can be safely evicted because it is persistent and can be reloaded from disk or remote storage on demand without affecting correctness. Consequently, while more memory reduces I/O and improves performance, less memory leads to graceful degradation as the system performs more I/O.

In contrast, intermediate data (e.g., hash tables, aggregation buffers, and sort runs) must be explicitly spilled to durable storage during execution and reloaded before operators can proceed. These spill-and-reload operations occur on the critical path: a hash join cannot begin probing until its build side is reconstructed in memory, and a window function cannot emit results until all partitioned states are reloaded. Once such spills occur, previously memory-bound queries suffer severe performance degradation due to expensive disk I/O.

3.2 The Caching Challenge in OLAP Systems

Intermediate data in analytical processing exhibits high variability and unpredictability in both size and lifetime. Empirical analyses

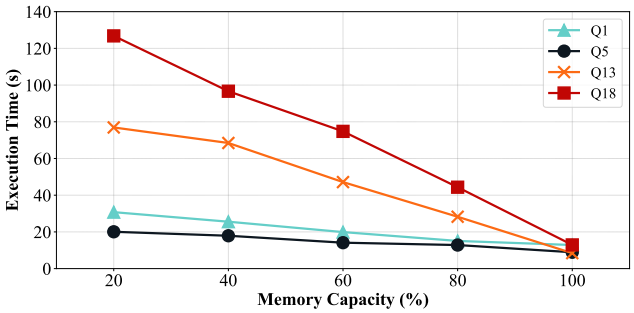


Figure 2: Performance improvement with increasing memory capacity.

Table 1: Intermediate data sizes across workload types in TPC-H.

Workload Type	Query Name	Intermediate Data Size
I/O-intensive	Q1	260 MiB
	Q3	7.5 GiB
	Q5	6.2 GiB
	Q8	4.1 GiB
Memory-bound	Q9	113.7 GiB
	Q13	42.6 GiB
	Q18	83.1 GiB
	Q21	57.2 GiB

show that the volume of intermediate results can differ by several orders of magnitude even across queries over the same dataset [47]. This variability makes it difficult to determine an appropriate cache size, as the working set changes with query patterns and execution plans.

Such unpredictability creates a fundamental challenge for cache management: the system must balance resource usage and performance under constraints. Over-provisioning memory guarantees high cache hit rates but incurs substantial cost and idle capacity; under-provisioning, in contrast, forces frequent evictions and spills to persistent storage, which offers durability guarantees that intermediate data does not need, significantly increasing latency. As shown in Figure 2, queries that spill to disk can experience slowdowns of about an order of magnitude.

Existing cloud OLAP systems largely rely on coarse-grained horizontal scaling to address this issue [15, 47]. However, this approach has two major drawbacks. First, elasticity operates at the granularity of entire compute nodes, leading to slow response times—typically on the order of tens of seconds for cluster reconfiguration [47]. Second, compute and memory resources remain tightly coupled, forcing users to allocate additional CPU cores to obtain more memory. This tight coupling leads to severe resource under-utilization and inflated operational costs. From an economic perspective, to prevent catastrophic spilling during peak intermediate data generation, users must conventionally provision expensive, large-memory compute instances.

An ideal elastic system should enable fast, fine-grained resource adaptation with minimal overhead, allowing each resource to scale independently. Unfortunately, current horizontally scaled designs fall short of this ideal because their monolithic scaling units and slow provisioning paths fundamentally limit responsiveness and efficiency in managing transient intermediate data.

3.3 Disaggregated Memory: The Path to Elasticity

Disaggregated memory (DM) [34, 37, 43, 44, 52] decouples memory from compute nodes, creating a shared, remotely accessible memory pool that serves as a unified tier between compute and storage. Integrating a DM into database systems transforms the traditional two-tier (compute-storage) architecture into a three-tier (compute-memory-storage) model [21, 48], enabling fine-grained resource management and breaking the tight coupling between CPU and memory capacity.

DM naturally aligns with the requirements of OLAP intermediate data along two key dimensions: semantics and elasticity. First, DM provides volatile, non-persistent memory that matches the transient nature of intermediate results, avoiding the overhead of unnecessary durability guarantees. Second, it enables fine-grained, on-demand allocation: the system can instantly draw memory from a shared pool to absorb demand spikes, thereby preventing spills to slow disk even when local memory capacity is exhausted. Once computation completes, the memory is promptly released, ensuring high utilization and cost efficiency.

Thus, disaggregated memory directly resolves the core tension identified earlier: by offering fine-grained, on-demand memory elasticity, it eliminates the trade-off between performance and resource efficiency.

3.4 Distributed Memory Pool

The distributed memory pool (DMP), which we have implemented in prior work [21, 48], is a cluster-wide memory infrastructure that aggregates physical memory into a scalable, remotely accessible pool. It exposes a lightweight, low-level abstraction for managing a large memory pool through three core APIs: `Allocate` to allocate a memory region, `Resize` to dynamically adjust its capacity, and `Free` to release resources back to the pool.

As shown in Figure 3, DMP consists of two types of nodes: *slab nodes (SNs)*, which contribute their local DRAM to the shared memory pool and serve data access requests, and a centralized *home node (HN)* that maintains metadata on cluster membership, node availability, and allocation status. Clients interact directly with the HN for coordination and with SNs for data placement and retrieval.

Memory Management. The DMP adopts a hierarchical memory-management model. The basic management unit is the *slab*, a fixed-size RDMA-registered region, with the size configurable via system parameters. Each slab node manages multiple slabs and registers them with the RDMA NIC, enabling remote clients to issue one-sided RDMA operations (READ/WRITE) directly to slab memory. The home node maintains a global catalog of all slabs and tracks their state (e.g., free or allocated).

Allocation is performed at the granularity of a logical memory region that is backed by multiple remote slabs. The DMP provides coarse-grained capacity by granting such regions to clients. Each region acts as a remote heap that supports fine-grained allocations

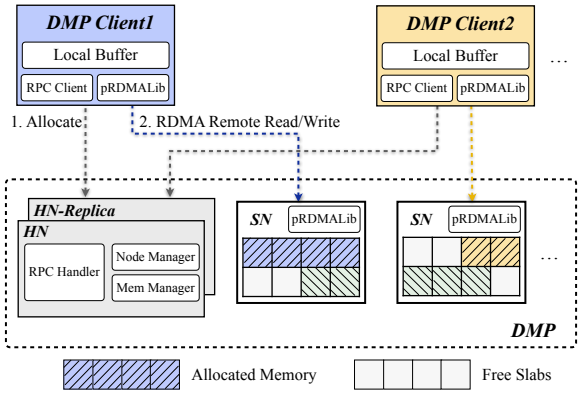


Figure 3: The architecture of DMP.

(e.g., objects or data structures) without further interaction with the DMP.

Procedures. After the DMP is initialized, each slab node registers with the home node, which monitors liveness through periodic heartbeats. As shown in Figure 3, when the home node receives an `Allocate` request, it selects free slabs, marks them as *allocated*, and returns a descriptor that lists all assigned slabs. Once the client receives this descriptor, it can directly issue one-sided RDMA operations to any slab in this region without further interaction with the home node.

The DMP provides a ready-to-use, elastic memory abstraction that simplifies building memory-disaggregated systems. As a general-purpose substrate, it provides coarse-grained elasticity and leaves workload-specific metadata management, access policies, and other application logic to upper-layer services.

3.5 The Overhead of Generality

While DM directly addresses the elasticity challenge that plagued traditional OLAP systems, repurposing a general-purpose DM system as a caching layer remains suboptimal. The core issue is that such systems are architecturally oblivious to the distinct semantics and lifecycles of data in OLAP workloads and treat ephemeral intermediate data and persistent base-table data identically.

Intermediate data is written to DM only when local memory is exhausted, read once by the query that produced it, and immediately discarded. In contrast, caching base-table data requires long-term management, including eviction policies, cross-client coherence, and crash consistency.

To support these requirements, general-purpose DM systems incorporate heavyweight mechanisms that are uniformly applied to all data. These include multi-RTT operations for global metadata lookups or eviction sampling [43], synchronization primitives such as `RDMA_CAS` and RDMA-based locks to enforce coherence [29, 53], and replication or logging protocols for crash consistency [26, 44, 46]. For intermediate data, which demands none of these guarantees, these mechanisms impose entirely avoidable overhead. Worse, they introduce substantial latency directly on the critical path of query execution, undermining the performance benefits that memory disaggregation promises.

This analysis reveals a crucial insight: an effective memory tier for OLAP must be more than elastic; it must be *semantically aware*. This calls for a specialized architecture with distinct data paths rather than a single generic path with different policy bits: one path is optimized for shared, persistent base-table data, and another provides low-overhead access to private, ephemeral intermediate data by avoiding distributed coordination on the critical path.

4 Overview

4.1 Design Principles

We now present the design of LakeMem, an elastic disaggregated-memory caching layer for OLAP workloads. Based on prior lessons, LakeMem follows three design principles.

- **Fast, Fine-Grained Elasticity.** By decoupling memory from compute, LakeMem delivers elasticity along two dimensions: (1) *fine-grained* allocation, scaling memory capacity in small units to match precise workload needs; and (2) *low-latency* adaptation, with rebalancing decisions and memory reallocation occurring in milliseconds or even microseconds to absorb bursty demands.
- **Semantic-Aware Caching.** Recognizing that not all data is semantically equal, LakeMem introduces a dual-path architecture. For node-private intermediate data, it offers a lightweight, direct-access path optimized for low latency. For shared base-table data, it provides a globally coherent cache designed to maximize cross-node reuse.
- **Demand-Driven Rebalancing.** To maximize resource utilization, LakeMem initially allocates a minimal capacity for the *PrivateCache*. As workload demands for intermediate data arise, it dynamically reclaims memory from the *SharedCache* to expand the *PrivateCache*. This reactive approach ensures that memory is allocated only when and where needed, preventing waste while mitigating severe performance cliffs.

4.2 Architecture Overview

As shown in Figure 4, LakeMem treats the DMP as a foundational memory provider and builds a dedicated OLAP caching layer above it. This layering is deliberate: DMP provides a general-purpose remote-memory abstraction shared across services, whereas LakeMem specializes the control plane for OLAP caching semantics, including coherence for shared base-table blocks, query-scoped ownership for intermediate state, and runtime rebalancing between the two. The data plane remains lightweight, as ordinary reads and writes still use one-sided RDMA to remote memory. Internally, LakeMem is structured into two distinct caching services, each specialized for a specific data category:

- ***SharedCache***: A cluster-wide cache for *base-table data*. *SharedCache* employs a centralized metadata server (MS) that manages remote memory and all cache entries, ensuring data coherence across compute nodes and enabling efficient sharing of cached data across the cluster without redundant copies.
- ***PrivateCache***: A per-compute-node cache for *intermediate data* generated during query execution. Given the transient,

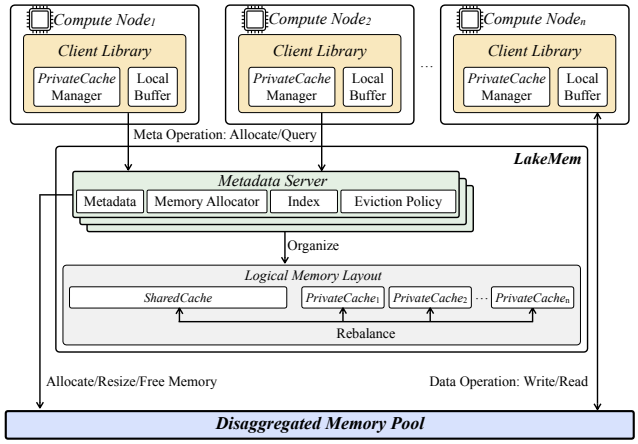


Figure 4: The architecture of LakeMem.

high-frequency, and node-local nature of such data, *PrivateCache* adopts a client-centric architecture with client-side metadata management. This design minimizes coordination overhead and provides the fine-grained, on-demand elasticity needed to absorb bursty intermediate memory workloads.

The client library, co-located with each compute node, provides a key-value interface to the query engine while masking remote-memory access details. It manages a configurable, RDMA-registered memory region (1 GiB by default) that serves as a bounded staging buffer rather than as a replacement for the query engine's own allocator. This simplifies integration, requiring only minimal modifications to the query engine's existing logic. For the upper layer, LakeMem behaves like a conventional external cache system. As shown in Figure 6, our client library provides a simple key-value interface (Put/Get/Has/Remove) to the application.

4.3 Memory Management

LakeMem adopts two-level memory management: it first acquires multiple large, coarse-grained chunks from disaggregated memory, then sub-allocates and manages them at fine granularity for its own internal use. This layering keeps the responsibilities clear: DMP handles coarse-grained region allocation and failure handling, while LakeMem sub-allocates those regions to implement OLAP-specific caching policies. As a result, workload-specific logic remains in the upper caching layer, and the underlying DMP can remain a general-purpose memory substrate for shared infrastructure.

This process begins at startup, when the MS invokes `Allocate` to obtain the initial region, which includes metadata such as remote addresses and RDMA remote keys (rkeys). This region is then partitioned internally: a small default capacity (e.g., 1 GiB) is reserved as the initial quota for each compute node's *PrivateCache*, while the remaining memory is allocated to the *SharedCache* for cluster-wide sharing.

The boundary between these partitions is fluid to enable dynamic resource allocation. As workload demands grow, a *PrivateCache* can acquire additional memory from the *SharedCache*'s managed space. Conversely, when memory is freed from a *PrivateCache*, it is returned to the shared pool of the *SharedCache*, ensuring high resource utilization. This internal rebalancing is the primary mechanism for elasticity. For coarse-grained adjustments, LakeMem can also invoke `Resize` or `Free` on the MS to change the total size of the top-level region managed by the DMP.

5 Design

LakeMem is structured into two specialized caches, the *SharedCache* and the *PrivateCache*, along with a client library co-located on each compute node. We now present the design of this architecture, detailing its core operations, elastic resource management, and workload-aware rebalancing mechanisms.

5.1 The *SharedCache*

Design Overview. The *SharedCache* provides a shared, cluster-wide cache for base-table data. To support scalable concurrent access from multiple compute nodes, it employs a metadata server (MS) and a lock-free design with sharding to avoid making metadata coordination a bottleneck.

Metadata Server. As illustrated in Figure 4, *SharedCache* employs a centralized MS to enable cluster-wide access to cached base-table data. The MS manages all memory contributed by the underlying DMP, tracks whether each chunk belongs to the shared or private cache, and maintains fine-grained mappings from *SharedCache* entries to their physical locations. When accessing data, a client first queries the metadata server to resolve the address of the target block and then performs direct data operations (e.g., RDMA reads/writes) to the corresponding slab node. Clients communicate with the MS using RDMA-based RPCs.

Memory Allocation. In contrast to traditional OLTP database buffer pools that manage uniformly sized pages (e.g., 16 KB in MySQL/InnoDB), analytical workloads cache variable-length data blocks from columnar formats like Parquet, which range from hundreds of kilobytes to multiple megabytes. This necessitates a flexible memory allocator. To this end, the *SharedCache* employs a custom allocator based on size-segregated segment classes [16, 22], where each class manages objects of similar sizes. The allocator uses configurable segments and assigns each allocation to the smallest size class that can accommodate it. For objects exceeding the maximum segment size, a dedicated large-object allocator is used to fulfill requests by merging multiple contiguous segments, thereby avoiding internal fragmentation in the size-segregated classes.

Scalability. *SharedCache* stores large base-table blocks from columnar formats such as Parquet, typically ranging from hundreds of kilobytes to multiple megabytes. As a result, the metadata lookup to the MS is short relative to the subsequent transfer of the data block itself. In addition, metadata is managed at block granularity rather than at fine object granularity, which keeps the number of metadata entries moderate. These workloads are also predominantly scan-heavy, allowing clients to batch metadata lookups for consecutive blocks before issuing the corresponding data reads.

We further reduce metadata overhead through lock-free design. Traditional lock-based implementations risk becoming bottlenecks under concurrent access [18, 19, 36], so all critical metadata paths are implemented with fine-grained atomic operations and non-blocking data structures. In particular, free lists and indexes are

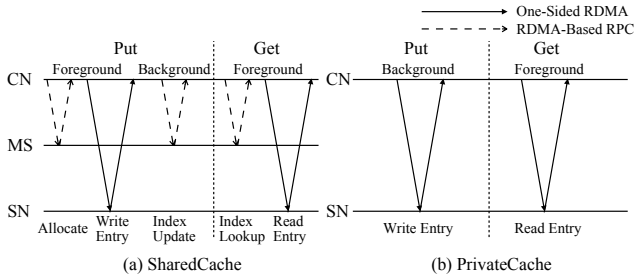


Figure 5: The Put/Get procedures of *SharedCache* and *PrivateCache*.

accessed without global locks, and eviction is implemented with a multi-versioned concurrent queue. Rather than updating shared recency metadata in place, the system appends a new version of the entry and reclaims only the most recent valid versions during eviction. Each segment class also maintains its own LRU list, further reducing cross-thread contention.

Procedures. Since modern lakehouse table formats (e.g., Delta Lake, Apache Iceberg) rely on immutable files and MVCC, updates to base tables manifest as new files rather than in-place modifications. Consequently, LakeMem does not require complex cache-coherence protocols for updates; newly written files are simply loaded into the cache on demand, while blocks from obsolete files are naturally reclaimed by the eviction policy.

As shown in Figure 5, request handling in the *SharedCache* follows a two-phase protocol: metadata resolution followed by data access. For a `Put` request, the client first sends a request to the MS, which checks whether the target entry is already cached. If so, the server returns the existing remote address to the client; otherwise, it allocates a new address. Upon receiving the address, the client issues a one-sided RDMA write directly to the corresponding slab node. To maintain consistency, the index update on the MS is deferred until after the data write and can be performed asynchronously in the background.

`Get` requests follow a similar pattern: the client first queries the MS to retrieve the remote address from the index, then performs a one-sided RDMA read directly from the corresponding slab node.

5.2 The *PrivateCache*

Design Overview. The *PrivateCache* is a per-node cache designed to store transient, query-scoped intermediate data generated during query execution. For efficiency, it adopts a client-centric architecture: metadata management and data-placement decisions are handled locally, thereby removing server-side coordination from the ordinary data path.

Client-Centric Architecture. Since intermediate data is short-lived and exclusively owned by a single query task, *PrivateCache* adopts a client-centric architecture that minimizes server-side involvement (only for rebalancing, detailed in §5.3). The client manages all metadata required for remote memory access (e.g., remote address, entry size, and rkeys) and maintains the index structures locally, thereby avoiding the distributed synchronization overhead associated with primitives such as `RDMA_CAS` or RDMA-based locks [29]. The private path avoids expensive cluster-wide coherence and distributed locking in ordinary data access. While local threads within the query engine may still concurrently access the client library, this is handled efficiently using lightweight local synchronizations (e.g., lock-free queues). With all necessary metadata and index maintained on the client side, data operations can be performed directly via one-sided RDMA reads and writes to remote memory, bypassing the MS on the critical path of query execution.

As shown in Figure 6, the LakeMem client library is co-located with each compute node. It receives requests from the upper-layer query engine and interacts with server-side components, including the MS and SNs. It does not replace the query engine's local allocator; the engine continues to manage its in-memory working set and invokes *PrivateCache* only when data must spill.

The client library manages both RDMA-registered buffers and remote memory in the *PrivateCache* through a hybrid allocator. Unlike the *SharedCache*, which handles variable-length base-table blocks, intermediate data in analytical queries often exhibits predictable, fixed-size patterns. To exploit this regularity and enable high-throughput concurrent allocation, the allocator shards frequently accessed object sizes across dedicated segment classes, reducing cross-thread contention. Once an allocation is made, the index records the mapping from the cache key to its physical location. The RDMA-registered local buffers serve only as a bounded staging area; spilled private data ultimately resides in remote memory allocated from DMP.

Query-Driven Data Lifetime. Intermediate data is tightly bound to the lifetime of its query task and must remain accessible throughout query execution. The *PrivateCache* therefore retains every entry until the query engine explicitly reads it back into local memory. Only after this read-back does the system mark the remote entry as obsolete and eligible for deallocation. Because the query engine controls the data's lifetime, *PrivateCache* does not need an eviction policy. This design avoids the overhead of tracking recency or frequency, relying instead on the query's natural access pattern to determine when data can be safely discarded.

Asynchronous Spilling. Intermediate data is spilled only when the query engine exhausts its local memory and must reclaim space to proceed. In a synchronous design, the engine would block until the remote write completes, stalling query execution on network I/O. To avoid this stall, we employ an asynchronous spilling mechanism. The query engine then writes the data into an RDMA-registered local buffer and submits a spilling request. It may resume execution without waiting for the remote write to complete as long as a staging buffer remains available. Spilling is implemented using multiple spilling queues, each operating in a multi-producer, single-consumer (MPSC) fashion. The foreground thread selects the least loaded queue to submit the request and notifies the corresponding background thread via a lightweight semaphore. If no free local buffer is available, the foreground thread waits until a buffer is reclaimed, providing backpressure rather than allowing unbounded memory growth.

Procedures. As shown in Figure 5, when the query engine spills intermediate data, it issues a `Put` request. The client stores the data in a locally allocated buffer and immediately submits an asynchronous spill request. A background thread then issues the remote

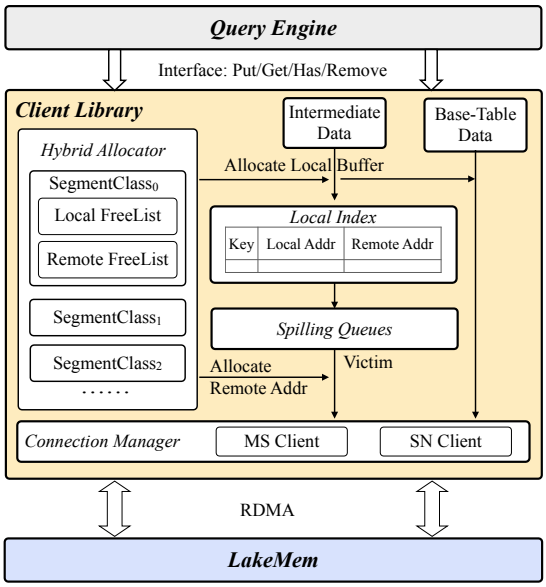


Figure 6: The architecture of the LakeMem client.

RDMA write. If the foreground thread cannot obtain a free local buffer, it waits until a buffer is reclaimed by the completion of a prior spill.

Upon a `Get` request, the client first checks whether the data is still present in local memory (i.e., the spill has not yet completed). If so, it serves the request directly from the local buffer and cancels the pending spill operation, thereby avoiding an unnecessary remote write. If the spill has already completed, the client allocates a new local buffer and issues a one-sided RDMA read to retrieve the data from the remote address recorded in its local index, without any MS involvement.

5.3 Elasticity and Rebalancing

OLAP workloads exhibit rapidly shifting memory demands: a memory-bound query may exhaust local DRAM and trigger spills to remote memory, while a subsequent I/O-intensive query benefits from a large base-table cache. A static partition between the *PrivateCache* (for short-lived intermediate data) and the *SharedCache* (for long-lived base-table data) inevitably leads to either chronic under-utilization or catastrophic performance cliffs.

While the underlying DMP provides elasticity through its `Resize` API, invoking it for every workload fluctuation is too coarse-grained for fast OLAP memory shifts. Each call introduces region-level coordination latency and cannot directly repurpose already-allocated but idle segments between *SharedCache* and *PrivateCache*. To achieve low-latency, high-utilization elasticity, LakeMem employs a two-level strategy: (1) fast, fine-grained rebalancing within an already-allocated memory pool, and (2) coarse-grained resizing as a last resort.

Intra-Pool Rebalancing. The key to fast, fine-grained rebalancing is a *priority-driven ownership model*, motivated by two properties of intermediate data. First, intermediate data is typically short-lived and fits entirely in local DRAM for most queries; allocating a large static portion to *PrivateCache* would waste capacity. Consequently, the shared memory pool is, by default, dedicated to caching base-table data, a design choice that becomes critical when spills occur due to memory pressure, as they severely degrade performance. As shown in our motivational study (§3.1), spilling intermediate data to disk stalls query execution on the critical path, causing catastrophic performance cliffs. This necessitates that *PrivateCache* acquire memory on-demand with the highest priority.

These principles directly shape our ownership model: the *SharedCache* holds default ownership of the shared memory pool, while each *PrivateCache* starts with only a minimal footprint (e.g., 1 GiB) that enables immediate, low-latency allocations for initial spill events and acquires temporary ownership of additional segments precisely when spills occur. This design enforces strictly unidirectional rebalancing: *PrivateCache* may preempt idle segments from *SharedCache*, but the reverse is never needed, as borrowed segments are automatically released once their transient intermediate data is consumed.

As the workload shifts toward memory-bound tasks, the demand for *PrivateCache* capacity increases. Expansion of *PrivateCache* is triggered based on memory utilization: when the *PrivateCache*'s local allocator detects that its free space has dropped below a threshold (e.g., 20% remaining), it asynchronously sends an expansion request to the MS via RDMA-based RPC. The client then proceeds with the allocation using any remaining available space. If all remote memory assigned to *PrivateCache* is exhausted, the allocation path blocks until the expansion request is fulfilled and new segments are made available.

Upon receiving the request, the MS attempts to satisfy it in three stages. First, it allocates segments from a global free-segment list. If the list is empty, it preempts idle segments from the *SharedCache* by evicting their contents and reassigning them to the *PrivateCache*. Only when preemption is insufficient to meet the demand does the server escalate to the underlying DMP by invoking `Resize` to provision additional memory.

When the workload shifts back from memory-bound to I/O-intensive, demand for *PrivateCache* decreases while the need for *SharedCache* grows. To reclaim resources efficiently, a background thread in the client library periodically scans free segments and reclaims those that have remained idle long enough to avoid transient reuse (e.g., >1s). Once an idle segment is identified, the background thread returns it to the LakeMem server's shared pool. The scanning interval is adaptively adjusted: it shortens when *PrivateCache* utilization drops rapidly (e.g., after a large query completes) and lengthens during stable or high utilization.

Inter-Pool Resizing. While rebalancing provides fast, fine-grained elasticity, LakeMem also supports coarse-grained, cluster-level scaling by interfacing with the underlying DMP. The MS acts as the sole coordinator for resizing the memory region.

Cluster-level expansion is a last-resort mechanism: when a *PrivateCache* expansion request cannot be satisfied even after reclaiming all available idle segments from the *SharedCache*, the MS invokes `Resize` to provision additional memory from the DMP. Newly acquired memory is added to the global free list and becomes immediately available. Conversely, for contraction, the MS periodically monitors the size of its global free list. If the fraction of idle memory in the pool exceeds a configured threshold (e.g., 20% of the total

pool capacity) for a sustained period, the MS releases the surplus capacity back to the DMP, ensuring efficient cluster-wide resource utilization.

5.4 Failure Handling

Compute Node Failure. The client library on each compute node maintains the metadata for its *PrivateCache*, such as the index. Since intermediate data is transient and query-scoped, its loss upon node failure does not compromise correctness of persistent data or other queries. Neither the data nor its metadata needs cross-query recovery after a crash.

To prevent memory leaks, cleanup is handled differently depending on the failure mode. (i) On query failure (e.g., cancellation or error), the query engine explicitly invokes a cleanup interface to release the associated *PrivateCache* segments. (ii) On compute node crash, the client cannot perform cleanup; instead, the MS automatically reclaims all segments of the failed node via liveness leases (e.g., heartbeat timeouts).

Metadata Server Failure. The MS manages metadata and allocation state for LakeMem. To avoid a single point of failure, we replicate the MS using a leader-follower architecture. For critical operations, such as inserting a *SharedCache* entry or allocating a segment to a *PrivateCache*, the leader MS synchronously replicates the update to all follower replicas before acknowledging the request. This ensures seamless takeover upon leader failure.

Disaggregated Memory Failure. Our fault tolerance model assumes the DM layer provides two key features: a highly-available metadata service and the ability to automatically replace memory resources from failed nodes. Consequently, LakeMem monitors metadata updates and handles the resulting data loss.

To achieve this, the MS periodically polls the DM layer for its latest memory topology. If the MS detects that a previously active memory node has become unavailable, it immediately invalidates all cache entries residing on that node. The consequences are handled based on data type: queries relying on volatile *PrivateCache* entries are aborted and must be re-executed, while future accesses to lost *SharedCache* data will trigger a reload from persistent storage. Subsequent memory allocations will transparently utilize the newly provisioned resources from the DM layer.

6 Evaluation

We conduct evaluations to answer the following questions:

- How does LakeMem's end-to-end query performance on TPC-H compare to traditional caching architectures? (§6.2)
- How does LakeMem perform as local memory is progressively constrained? (§6.3)
- How effectively does LakeMem's memory rebalancing mechanism adapt to shifting workload demands by dynamically reallocating memory between *PrivateCache* and *SharedCache*? (§6.4)
- How do read and write operations perform in *PrivateCache* and *SharedCache* in terms of latency and component-level overhead? (§6.5)
- How effectively does LakeMem accelerate out-of-core data preprocessing workloads with heavy sorting and spilling? (§6.6)

6.1 Setup

Test Platform. Our evaluation is conducted on a controlled public-cloud cluster. Each machine is equipped with two Intel Xeon Platinum 8369B CPUs (2.90 GHz), 1 TB of DDR4 DRAM, a 4 TB Intel DC P4510 SSD, and runs CentOS 7. These physical machines are interconnected via a 100 Gbps Mellanox ConnectX-6 NIC. All data is persisted on dedicated storage nodes and accessed through MinIO [3], providing an S3-compatible interface for remote access.

We conduct the experiments in a DuckLake [11] environment serving as the lakehouse catalog, with four DuckDB [41] compute nodes (CNs). LakeMem is integrated into DuckDB as an external cache, replacing the temporary block manager with a client library. The disaggregated memory comes from the DMP, configured with one home node (HN) and two slab nodes (SNs). All communication between CNs and DMP uses RDMA over Converged Ethernet (RoCE).

Baselines. We compare the performance of LakeMem with two baseline caching configurations. The first baseline is a disk-based cache following the conventional two-tier hybrid design [17, 47]: data is first cached in local memory and spills over to local disk once the memory tier is full. The second baseline is a local-memory cache configured with a capacity equal to the total cache size of LakeMem. Although such a configuration is typically infeasible in real-world deployments, it is an idealized upper bound to quantify the overhead introduced by our disaggregated remote-memory-based cache. Our evaluation focuses on conventional local caching baselines rather than the full design space of disaggregated-memory caches.

Workload. We evaluate LakeMem using the following benchmarks:

- **TPC-H** [8] is an industry-standard benchmark for decision-support workloads, featuring 22 complex analytical queries with multi-table joins and aggregations. We generate the dataset at Scale Factor 300 using the official TPC-H data generator [13]; all tables are stored in Parquet format, one file per table.
- **GraySort** [2] is a benchmark for large-scale external sorting. We use it as a representative spill-heavy data preprocessing workload because sorting, repartitioning, and materialization are common building blocks in ETL and ML data preparation pipelines before model training. Our test measures the time to globally sort a 100 GiB dataset, converted from the original binary format to Parquet to align with our other experiments.

6.2 Overall Performance

We evaluate LakeMem using the TPC-H benchmark. To measure steady-state performance, we first warm up the system with a series of random queries, then execute each of the 22 TPC-H queries once and report execution times. The end-to-end results are summarized in Figure 7.

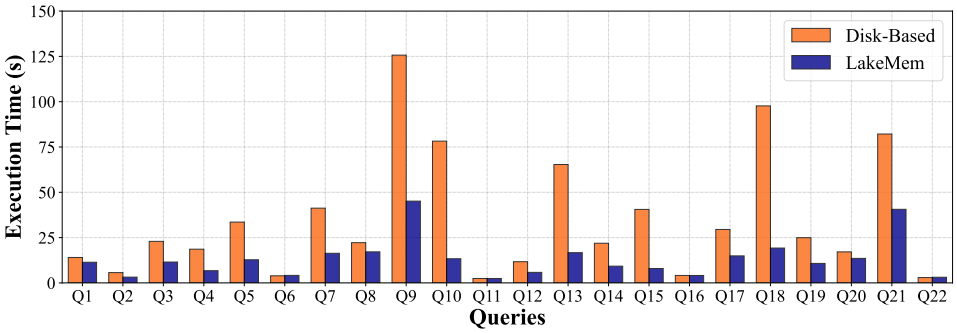


Figure 7: Performance across all 22 TPC-H queries compared to the disk-based baseline.

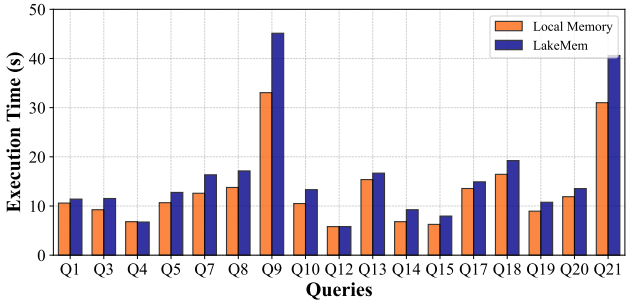


Figure 8: The selected TPC-H query performance compared to a fully local memory cache baseline.

For I/O-intensive workloads (e.g., Q1, Q3, Q4, Q5, Q7, Q8, Q17), involving sequential access to large volumes of base-table data, LakeMem outperforms the disk-based cache by 1.2× to 2.6×. This gain stems from *SharedCache* caching frequently accessed base tables in remote memory. By serving scans over high-bandwidth RDMA links instead of SSD-backed storage, LakeMem eliminates the latency and I/O amplification of disk-based retrieval.

In contrast, for memory-bound queries that generate large intermediate states (e.g., Q9, Q10, Q13, Q18, Q21), LakeMem efficiently manages ephemeral data using low-latency, one-sided RDMA operations, thereby avoiding costly disk spilling. The disk-based cache performs poorly due to high disk I/O overhead, with LakeMem achieving speedups of 2.0× to 5.9×.

For queries such as Q6, Q11, Q16, and Q22, LakeMem and the disk-based cache exhibit similar performance. These queries have small working sets that fit entirely in local DRAM, so both base-table and intermediate data are served from memory, eliminating the performance gap.

To evaluate the performance overhead of our disaggregated architecture, we selected I/O-intensive and memory-bound queries for direct comparison between LakeMem and a fully local memory cache. As shown in Figure 8, LakeMem achieves performance highly comparable to the baseline across these demanding workloads. The largest performance overhead occurs in Q8 (20%) and Q9 (27%), representing I/O-intensive and memory-bound queries respectively, primarily due to the additional network round-trips required for metadata resolution and remote data transfer.

6.3 Sensitivity to Local Memory Size

We then evaluate the impact of local memory size on query performance. This experiment measures how each system behaves as the available local memory decreases from 100% to 10% or 20%. Here, the memory percentage denotes the ratio between the available local DRAM and the total dataset size required by the queries. We include two I/O-intensive queries (Q3, Q4) and two memory-bound workloads (Q13, Q18).

As shown in Figure 9, the disk-based baseline exhibits high sensitivity to local memory size: as the cache shrinks, more data spills to SSD, causing execution time to increase sharply. For Q3 and Q4, performance degrades by 2.9× to 3.2× at 20% capacity due to growing disk I/O overheads. The memory-bound workloads suffer even more severely, showing 6.7× to 9.8× slowdowns when intermediate data spills to disk.

In contrast, LakeMem maintains nearly constant performance across all memory configurations. Because base-table data and intermediate results are served directly from the low-latency remote memory tiers (*SharedCache* and *PrivateCache*), LakeMem is largely decoupled from the size of local DRAM and remains close to peak performance even at 20% local memory.

Overall, these results show that LakeMem delivers near-local-memory performance even with severely limited DRAM, as its remote memory tiers effectively compensate for reduced local capacity. This enables LakeMem to maintain performance comparable to having the full dataset in local memory, while disk-based designs suffer dramatic performance cliffs.

6.4 Elasticity

LakeMem dynamically balances memory between *PrivateCache* and *SharedCache*. We evaluate this rebalancing strategy by comparing four configurations:

- *Private-only*: all memory is allocated to *PrivateCache* for intermediate data.
- *Shared-only*: all memory is dedicated to *SharedCache* for base-table data.
- *Static*: splits memory between *PrivateCache* and *SharedCache* in a fixed 1:1 ratio.
- *LakeMem*: dynamically reallocates memory between the two caches based on workload demand.

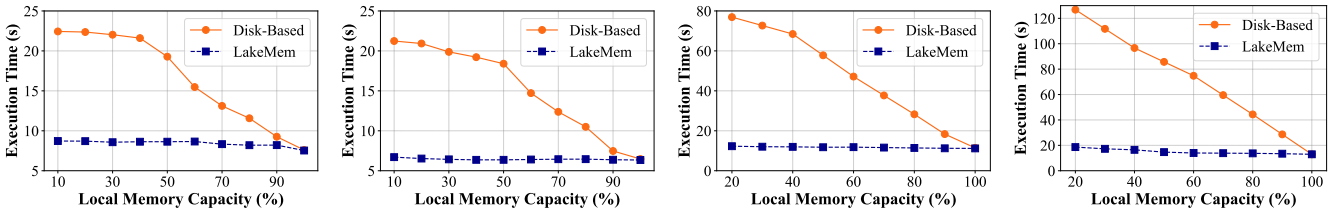


Figure 9: The sensitivity of LakeMem to local memory size, compared with disk-based baseline.

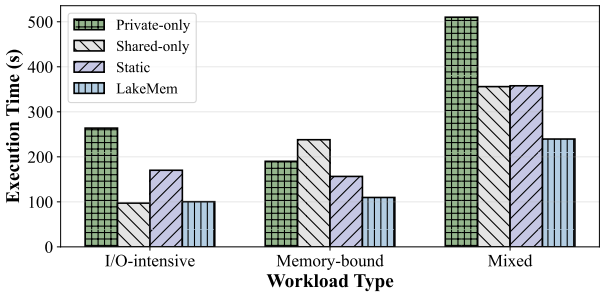


Figure 10: Performance of the four memory allocation configurations across I/O-intensive, memory-bound, and mixed TPC-H workloads.

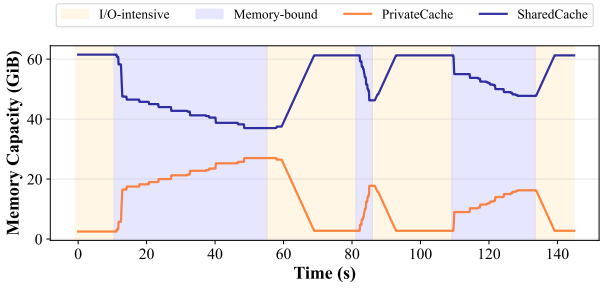


Figure 11: The memory size of *PrivateCache* and *SharedCache* under workload variation.

We construct two representative workloads using TPC-H queries: *I/O-intensive* (e.g., Q1, Q3, Q4, Q5) and *memory-bound* (e.g., Q9, Q10, Q13, Q18).

As shown in Figure 10, under I/O-intensive workloads, the *Shared-only* and *LakeMem* configurations achieve similar performance, as both maintain a large *SharedCache* with high hit rates. The *Static* configuration performs worse because half the memory is reserved for *PrivateCache* but remains unused, effectively halving base-table cache capacity. The *Private-only* configuration yields the lowest performance, as all base-table data must be loaded from remote storage.

For memory-bound workloads, LakeMem achieves the best performance by allocating sufficient memory to *PrivateCache* to keep intermediate data in memory. The *Static* configuration performs worse because many memory-bound queries also scan large base tables and its fixed split cannot accommodate some large intermediates, forcing partial spilling to disk. The *Shared-only* configuration degrades most severely because all intermediate data spills to disk.

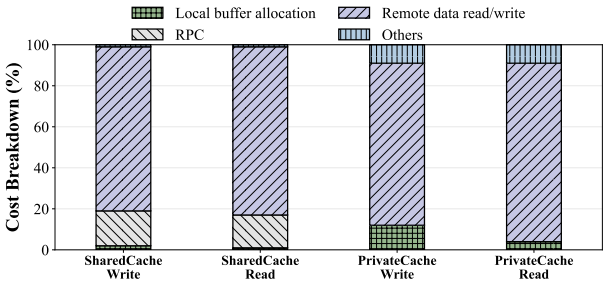


Figure 12: The cost breakdown of LakeMem.

Although *Private-only* avoids such spills, it still performs poorly because substantial base-table data must be fetched from remote storage. For mixed workloads, LakeMem incurs minor overhead from transient cache misses after workload shifts as the rebalancing mechanism adapts, yet still outperforms all other configurations.

We also evaluate LakeMem's memory allocation under varying workloads. As shown in Figure 11, *PrivateCache* starts with minimal capacity while most memory is assigned to *SharedCache*. When the workload shifts to memory-bound queries, *PrivateCache* expands rapidly to absorb growing intermediate data; when it shifts back to I/O-intensive queries, memory is returned to *SharedCache* to improve base-table caching. Later workload changes exhibit the same pattern, showing that LakeMem can quickly reassign memory as demand changes.

6.5 Cost Breakdown

We evaluate the cost breakdown by continually issuing read/write requests with sufficient concurrency to saturate the available RDMA bandwidth of the 100 Gbps interconnect. We use 64 KiB for the *PrivateCache*, reflecting the typical intermediate data size, and 1 MiB for the *SharedCache*, matching the average base-table block size.

As shown in Figure 12, we break down the time spent in each phase of read and write requests. For *SharedCache*, the majority of latency is dominated by data transfer to the SN, while RPC communication with the MS accounts for 16% (reads) and 17% (writes). Buffer allocation and other overhead (e.g., index updates) are negligible. For *PrivateCache*, both read and write requests are primarily dominated by data transfer. Write requests incur additional buffer allocation cost, as they must acquire a local registered buffer before proceeding. Notably, the "remote write" time for writes reflects the foreground thread's wait for a free buffer, since the actual RDMA write is performed asynchronously by background threads. Other overhead (e.g., index updates) remains minimal.

Table 2: Performance of the 100 GiB GraySort benchmark.

Spill Target	Sort Phase (s)	Output Phase (s)	Total (s)
Local Disk	122.83	31.97	154.8
LakeMem	73.81	30.79	104.6

6.6 ML Data Preprocessing

To evaluate LakeMem on spill-heavy data preprocessing stages common in ETL and ML data preparation, we integrated it into *smallpond* [12], a lightweight data processing framework for AI workloads. Modern ML pipelines typically include substantial data preparation before training, such as sorting, repartitioning, and feature-table materialization. These stages often generate large intermediate states and trigger external-memory behavior even though they are not model-training kernels themselves. The integration was achieved by modifying *smallpond*'s underlying DuckDB execution to use the LakeMem client library.

We use the GraySort benchmark [2] to quantify the performance benefits of LakeMem as a high-performance spill target. GraySort isolates the spill-heavy external sorting path common in data preprocessing, making it a useful stress case for LakeMem's remote spill path. Our experiment sorts a 100 GiB dataset using *smallpond*, configured with four parallel DuckDB instances as the sort engines. To isolate the impact of the spill device, the measurements focus on the sorting time, excluding the initial data generation and partition shuffling phases. As shown in Table 2, redirecting spills to LakeMem's remote memory instead of a local disk yields a 1.7× speedup in the core sort phase and a 1.5× speedup in total end-to-end time.

7 Related Work

Disaggregated Memory Systems. High-speed interconnects like RDMA have spurred the development of disaggregated memory (DM) systems [30, 43, 44, 52]. These platforms decouple memory from compute nodes into a shared, network-attached pool, improving elasticity and utilization. Systems such as FaRM [26] and FUSEE [44] provide general-purpose key-value stores over disaggregated memory, focusing on strong consistency and fault tolerance. Redy [52] and CliqueMap [46] explore large-scale distributed caches using RDMA, while Ditto [43] studies elastic and adaptive memory-disaggregated caching.

These systems largely expose a unified cache path and treat data as semantically uniform blocks. For query-private intermediate data in OLAP workloads, such generality adds avoidable coordination to the critical path. LakeMem differentiates data types at the architectural level: it retains coordinated access only for shared base-table blocks and provides a separate low-overhead path for private intermediate state.

Caching Systems for Analytical Workloads. Existing analytical systems, including Snowflake [47], Redshift [15], and Alluxio [38], typically collocate cache with compute or use a monolithic caching tier. For OLAP, this means long-lived base-table data and short-lived intermediate results compete for space, while tight cache-compute coupling restricts elasticity and fragments memory across the cluster.

Large analytical services such as Dremel/BigQuery [39] and systems such as Magnet [45] also highlight the importance of managing remote intermediate state and shuffle efficiently. These systems primarily optimize either the shared scan path or the ephemeral shuffle path. LakeMem differs in unifying both shared base-table caching and query-private intermediate spilling within one disaggregated memory pool, with runtime rebalancing between the two paths.

Ephemeral Data Management. Recent work focuses on managing ephemeral (or intermediate) data in cloud-native and serverless environments. Systems like Pocket [33] and Locus [40] propose disaggregated ephemeral storage for serverless analytics, allowing shuffle outputs to outlive short-lived compute functions.

They treat ephemeral data in isolation and do not integrate with persistent data caching. In OLAP workloads, where both intermediate results and base-table caches coexist, this separation forces the deployment of two disjoint systems and prevents dynamic resource sharing between them. LakeMem instead combines both paths in a single disaggregated memory pool and rebalances capacity between them at runtime.

Semantic Caching. Semantic caching [24, 25, 27, 42] reuses query results across semantically related queries by caching the logical predicates or views defining what data has been computed. It enables answering new queries whose results are logically contained in previously cached outputs, even when the query text differs. This requires maintaining rich query-level metadata and performing containment checks at runtime—overhead that is often prohibitive in high-throughput or memory-constrained environments.

In contrast, our system does not perform semantic reasoning or result containment. Instead, LakeMem operates at the block level and treats cached data as opaque payloads. Its key distinction lies not in query understanding, but in recognizing that intermediate data and base-table data exhibit fundamentally different access patterns and performance sensitivities. Accordingly, LakeMem partitions the shared memory pool into *PrivateCache* and *SharedCache* to provide differentiated elasticity and protection policies without any reliance on query semantics.

8 Conclusion

We present LakeMem, an OLAP-aware caching layer for analytical systems that leverages disaggregated memory through a dual-path architecture for shared base-table data and private intermediate data, reducing coordination on the private path while preserving coherent sharing on the shared path. A dynamic rebalancing mechanism adapts memory allocation to workload demands. In our evaluation, LakeMem substantially outperforms a hybrid DRAM-SSD cache in both memory-bound and I/O-intensive scenarios while remaining close to fully in-memory execution in our setting.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work is sponsored by the National Natural Science Foundation of China (62441220) and CCF-ApsaraDB Research Fund (CCF-Aliyun2024002). Yiming Zhang and Hao Chen are corresponding authors.

References

[1] 2006. Amazon S3. https://aws.amazon.com/s3/. Accessed: 2025-11-17.
[2] 2007. GraySort Benchmark. https://sortbenchmark.org/. Accessed: 2025-11-17.
[3] 2014. MinIO. https://www.min.io/. Accessed: 2025-11-17.
[4] 2019. Apache Doris. https://doris.apache.org/. Accessed: 2025-11-17.
[5] 2019. Apache Parquet. https://parquet.apache.org. Accessed: 2025-11-17.
[6] 2019. DeltaLake. https://delta.io/. Accessed: 2025-11-17.
[7] 2020. ClickHouse. https://clickhouse.com/. Accessed: 2025-11-17.
[8] 2020. TPC-H Benchmark. https://www.tpc.org/tpch/. Accessed: 2025-11-17.
[9] 2024. Mooncake. https://www.mooncake.dev/. Accessed: 2025-11-17.
[10] 2025. Apache ORC. https://orc.apache.org. Accessed: 2025-11-17.
[11] 2025. DuckLake. https://ducklake.select/. Accessed: 2025-11-17.
[12] 2025. Smallpond. https://github.com/deepseek-ai/smallpond. Accessed: 2025-11-17.
[13] 2025. TPC-H Data Generator. https://github.com/clflushopt/tpchgen-rs. Accessed: 2025-11-17.
[14] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (Vancouver, Canada) *(SIGMOD '08)*. Association for Computing Machinery, New York, NY, USA, 967–980. https://doi.org/10.1145/1376616.1376712
[15] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 2205–2217. https://doi.org/10.1145/3514221.3526045
[16] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association, Renton, WA, 389–403. https://www.usenix.org/conference/nsdi18/presentation/beckmann
[17] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 753–768. https://www.usenix.org/conference/osdi20/presentation/berg
[18] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 483–498. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/berger
[19] Aaron Blankstein, Siddhartha Sen, and Michael J. Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 499–511. https://www.usenix.org/conference/atc17/technical-sessions/presentation/blankstein
[20] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Conference on Innovative Data Systems Research*. https://api.semanticscholar.org/CorpusID:1379707
[21] Wei Cao, Yingqiang Zhang, Xinjun Yang, Feifei Li, Sheng Wang, Qingda Hu, Xuntao Cheng, Zongzhi Chen, Zhenjun Liu, Jing Fang, Bo Wang, Yuhui Wang, Haiqing Sun, Ze Yang, Zhushi Cheng, Sen Chen, Jian Wu, Wei Hu, Jianwei Zhao, Yusong Gao, Songlu Cai, Yunyang Zhang, and Jiawang Tong. 2021. PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2477–2489. https://doi.org/10.1145/3448016.3457560
[22] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. USENIX Association, Santa Clara, CA, 379–392. https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/cidon
[23] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741
[24] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. 1996. Semantic Data Caching and Replacement. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 330–341.
[25] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. 1998. Caching multidimensional queries using chunks. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data* (Seattle, Washington, USA) *(SIGMOD '98)*. Association for Computing Machinery, New York, NY, USA, 259–270. https://doi.org/10.1145/276304.276328
[26] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No compromises: distributed transactions with consistency, availability, and performance. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 54–70. https://doi.org/10.1145/2815400.2815425
[27] Dominik Durner, Badrish Chandramouli, and Yinan Li. 2021. Crystal: a unified cache storage system for analytical databases. *Proc. VLDB Endow.* 14, 11 (July 2021), 2432–2444. https://doi.org/10.14778/3476249.3476292
[28] Kira Duwe, Angelos Anadiotis, Andrew Lamb, Lucas Lersch, Boaz Leskes, Daniel Ritter, and Pınar Tözün. 2025. The Five-Minute Rule for the Cloud: Caching in Analytics Systems. In *Conference on Innovative Data Systems Research*. Conference on Innovative Data Systems Research.
[29] Jian Gao, Qing Wang, and Jiwu Shu. 2025. ShiftLock: Mitigate One-sided RDMA Lock Contention via Handover. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 355–372. https://www.usenix.org/conference/fast25/presentation/gao
[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, Boston, MA, 649–667. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu
[31] Brian Hentschel, Michael S. Kester, and Stratos Idreos. 2018. Column Sketches: A Scan Accelerator for Rapid and Robust Predicate Evaluation. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 857–872. https://doi.org/10.1145/3183713.3196911
[32] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. 1997. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing* (El Paso, Texas, USA) *(STOC '97)*. Association for Computing Machinery, New York, NY, USA, 654–663. https://doi.org/10.1145/258533.258660
[33] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 427–444. https://www.usenix.org/conference/osdi18/presentation/klimovic
[34] Seung-seob Lee, Yanpeng Yu, Yupeng Tang, Anurag Khandelwal, Lin Zhong, and Abhishek Bhattacharjee. 2021. MIND: In-Network Memory Management for Disaggregated Data Centers. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 488–504. https://doi.org/10.1145/3477132.3483561
[35] Justin Levandoski, Garrett Casto, Mingge Deng, Rushabh Desai, Pavan Edara, Thibaud Hottelier, Amir Hormati, Anoop Johnson, Jeff Johnson, Dawid Kurzyniec, Sam McVeety, Prem Ramanathan, Gaurav Saxena, Vidya Shanmugan, and Yuri Volobuev. 2024. BigLake: BigQuery's Evolution toward a Multi-Cloud Lakehouse. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 334–346. https://doi.org/10.1145/3626246.3653388
[36] Conglong Li and Alan L. Cox. 2015. GD-Wheel: a cost-aware replacement policy for key-value stores. In *Proceedings of the Tenth European Conference on Computer Systems* (Bordeaux, France) *(EuroSys '15)*. Association for Computing Machinery, New York, NY, USA, Article 5, 15 pages. https://doi.org/10.1145/2741948.2741956
[37] Feng Li, Sudipto Das, Manoj Syamala, and Vivek R. Narasayya. 2016. Accelerating Relational Databases by Leveraging Remote Memory and RDMA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 355–370. https://doi.org/10.1145/2882903.2882949
[38] Haoyuan Li. 2018. *Alluxio: A Virtual Distributed File System*. Ph. D. Dissertation. UC Berkeley. https://escholarship.org/uc/item/4n80320w ProQuest ID: Li_berkeley_0028E_17794. Merritt ID: ark:/13030/m5vf1w16.
[39] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: a decade of interactive SQL analysis at web scale. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3461–3472. https://doi.org/10.14778/3415478.3415568
[40] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. 2019. Shuffling, fast and slow: scalable analytics on serverless infrastructure. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA, USA) *(NSDI'19)*. USENIX Association, USA, 193–206.

[41] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1981–1984. https://doi.org/10.1145/3299869.3320212

[42] Peter Scheuermann, Junho Shim, and Radek Vingralek. 1996. WATCHMAN: A Data Warehouse Intelligent Cache Manager. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB '96)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 51–62.

[43] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Yuxin Su, Jiazhen Gu, Hao Feng, Yangfan Zhou, and Michael R. Lyu. 2023. Ditto: An Elastic and Adaptive Memory-Disaggregated Caching System. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 675–691. https://doi.org/10.1145/3600006.3613144

[44] Jiacheng Shen, Pengfei Zuo, Xuchuan Luo, Tianyi Yang, Yuxin Su, Yangfan Zhou, and Michael R. Lyu. 2023. FUSEE: A Fully Memory-Disaggregated Key-Value Store. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 81–98. https://www.usenix.org/conference/fast23/presentation/shen

[45] Min Shen, Ye Zhou, and Chandni Singh. 2020. Magnet: push-based shuffle service for large-scale data processing. *Proc. VLDB Endow.* 13, 12 (Aug. 2020), 3382–3395. https://doi.org/10.14778/3415478.3415558

[46] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo M. K. Martin, Amanda Strominger, Thomas F. Wenisch, and Amin Vahdat. 2021. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference* (Virtual Event, USA) *(SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 93–105. https://doi.org/10.1145/3452296.3472934

[47] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[48] Xinjun Yang, Yingqiang Zhang, Hao Chen, Feifei Li, Bo Wang, Jing Fang, Chuan Sun, and Yuhui Wang. 2024. PolarDB-MP: A Multi-Primary Cloud-Native Database via Disaggregated Shared Memory. In *Companion of the 2024 International Conference on Management of Data* (Santiago AA, Chile) *(SIGMOD '24)*. Association for Computing Machinery, New York, NY, USA, 295–308. https://doi.org/10.1145/3626246.3653377

[49] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (July 2024), 1643–1670. https://doi.org/10.1007/s00778-024-00867-8

[50] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1802–1805. https://doi.org/10.1109/ICDE48307.2020.00174

[51] Chaoqun Zhan, Maomeng Su, Chuangxian Wei, Xiaoqiang Peng, Liang Lin, Sheng Wang, Zhe Chen, Feifei Li, Yue Pan, Fang Zheng, and Chengliang Chai. 2019. AnalyticDB: real-time OLAP database system at Alibaba cloud. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2059–2070. https://doi.org/10.14778/3352063.3352124

[52] Qizhen Zhang, Philip A. Bernstein, Daniel S. Berger, and Badrish Chandramouli. 2021. Redy: remote dynamic memory cache. *Proc. VLDB Endow.* 15, 4 (Dec. 2021), 766–779. https://doi.org/10.14778/3503585.3503587

[53] Pengfei Zuo, Jiazhao Sun, Liu Yang, Shuangwu Zhang, and Yu Hua. 2021. One-sided RDMA-Conscious Extendible Hashing for Disaggregated Memory. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 15–29. https://www.usenix.org/conference/atc21/presentation/zuo