TierBase: A Workload-Driven Cost-Optimized Key-Value Store [Extended Version]

Zhitao Shen[†], Shiyu Yang[§], Weibo Chen[§†], Kunming Wang[§†], Yue Li[†], Jiabao Jin[†], Wei Jia[†], Junwei Chen[†], Yuan Su[†], Xiaoxia Duan[†], Wei Chen[†], Lei Wang[†], Jie Song[†], Ruoyi Ruan[†], Xuemin Lin[‡]
[†]Ant Group ; [§]Guangzhou University ; [‡] Shanghai Jiao Tong University
{zhitao.szt, chenweibo.cwb, wangkunming.wkm, ly321766, jinjiabao.jjb, jw94525, weisong.cjw, stevensu.sy, xiaoxia.dxx, cw281808, wl177541, peter.sj, ruoyi.ruanry}@antgroup.com; syyang@gzhu.edu.cn; xuemin.lin@sjtu.edu.cn

ABSTRACT

In the current era of data-intensive applications, the demand for high-performance, cost-effective storage solutions is paramount. This paper introduces a Space-Performance Cost Model for key-value store, designed to guide cost-effective storage configuration decisions. The model quantifies the trade-offs between performance and storage costs, providing a framework for optimizing resource allocation in large-scale data serving environments. Guided by this cost model, we present TierBase, a distributed key-value store developed by Ant Group that optimizes total cost by strategically synchronizing data between cache and storage tiers, maximizing resource utilization and effectively handling skewed workloads. To enhance cost-efficiency, TierBase incorporates several optimization techniques, including pre-trained data compression, elastic threading mechanisms, and the utilization of persistent memory. We detail TierBase's architecture, key components, and the implementation of cost optimization strategies. Extensive evaluations using both synthetic benchmarks and real-world workloads demonstrate TierBase's superior cost-effectiveness compared to existing solutions. Furthermore, case studies from Ant Group's production environments showcase TierBase's ability to achieve up to 62% cost reduction in primary scenarios, highlighting its practical impact in large-scale online data serving.

1 INTRODUCTION

In the current era of data-intensive applications, the demand for high-performance, cost-effective storage solutions is paramount. key-value store, with their promise of scalability, flexibility, and rapid data access, have emerged as a pivotal component in this landscape. Renowned key-value store such as Redis [54], Memcached [35], Cassandra [38], and HBase [58] have set industry benchmarks, offering solutions designed for diverse use cases ranging from in-memory caching to persistent storage.

At Ant Group, we face numerous challenges in managing our online data serving systems. These challenges stem from the intrinsic nature of our services and the evolving demands of our vast user base. We handle ***the immense volume of data*** generated by billions of users, necessitating robust, low-latency, and cost-effective storage solutions. The diverse scenarios and applications lead to ***a wide spectrum of workloads*** with varying needs for reliability, durability, and latency. Additionally, ***significant skewness in data access patterns***, with dynamically changing hot spots, complicates data access and caching strategies. Furthermore, in order to handle these workloads, we have to maintain an extremely large number of machines, which leads to a ***non-trivial configuration decision*** challenge. This challenge demands a cost model to accurately quantify the cost-performance trade-offs.

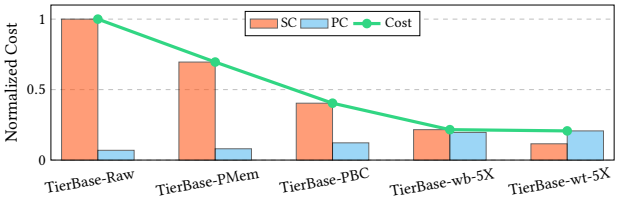


Figure 1: Cost comparison in TierBase

To address these challenges, we introduce TierBase, a distributed key-value store developed by Ant Group since 2017. Initially Redis-compatible, TierBase has evolved to support advanced functions such as CAS operations, wide-columns, and vector searching. Extensively utilized within Ant Group, TierBase maintains sub-millisecond access latency even under peak loads of hundreds of millions of queries per second (QPS), crucial for delivering seamless user experiences during events like the Double 11 shopping festival.

To optimize costs further, we developed several cost-saving strategies within TierBase. Figure 1 illustrates the cost-performance trade-offs and the impact of these optimization techniques in a real-world use case. The figure shows how enabling our cost-saving strategies leads to reductions in both space cost (*SC*) and performance cost (*PC*), which represent storage and query processing expenses, respectively. In our cost model, the overall cost is defined as the maximum of *SC* and *PC*, represented by the green line in the figure. Despite the increase in *PC*, the significant reduction in *SC* results in an overall cost decrease. Our pre-trained compression technique (TierBase-PBC) achieves up to a 62% cost reduction over the baseline (TierBase-Raw), demonstrating substantial savings in one of our primary online serving scenarios.

In developing TierBase, we focus on two key questions:

Q1: How can we develop a comprehensive cost model for large-scale online data serving systems that adapts to varying workloads? Developing a comprehensive cost model for complex online data serving systems requires identifying key metrics that accurately capture real-world costs across various workloads. Traditional models [43, 45] typically focus on overall system costs without considering specific workload characteristics. The challenge lies in developing a quantitative framework that accurately

models cost-performance trade-offs between different storage configurations while incorporating workload-specific characteristics and capturing the non-linear relationship between system configuration and cost.

Q2: How can we effectively evaluate and apply optimization techniques in key-value stores to balance performance and cost for specific workloads? The research and industry communities have developed a number of innovative techniques [37, 57] to optimize performance and storage efficiency for key-value stores. However, they are often workload-specific, and there is no universal solution. The challenge lies in developing a unified framework to evaluate and apply these diverse optimization techniques across different workloads considering both performance metrics and overall system cost. This cost model aims to enable informed decisions about which techniques to apply in different scenarios, moving beyond one-size-fits-all solutions towards a workload-aware optimization strategy for key-value stores.

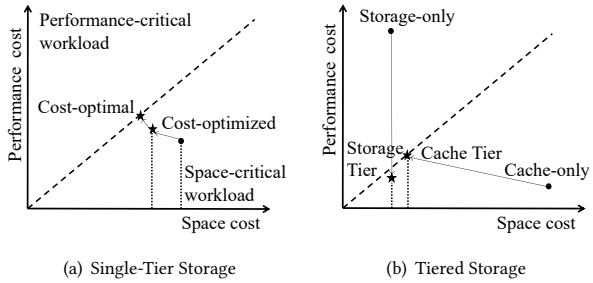
To answer these questions, in this paper, we introduce: **Space-Performance Cost Model**: A novel approach balances performance and space costs, proposing that optimal cost is achieved when these factors are equal. This model extends to tiered storage systems, incorporating cache ratios and miss ratios to determine cost-effectiveness.

TierBase: Guided by this cost model, TierBase employs a tiered storage architecture that balances performance and storage space for cost-efficiency. It incorporates innovative features such as a flexible design for handling diverse workloads, pre-trained data compression techniques, elastic threading mechanisms, and utilization of persistent memory.

Cost Optimization Framework: We introduce a framework for evaluating and optimizing costs for key-value stores. This framework includes strategies for adapting to diverse workloads and provides guidelines for making cost-effective decisions in system configuration and resource allocation.

To sum up, the main contributions are as follows:

- We propose a comprehensive Space-Performance Cost Model for Key-Value store that aligns with and extends the classic Five-Minute Rule, guiding decisions on optimal storage configuration selection.
- We present the architecture of TierBase, a distributed key-value store that leverages a tiered storage design to effectively balance performance and cost-efficiency across diverse workloads.
- We introduce and evaluate several cost-optimizing techniques implemented in TierBase, including pre-trained data compression, elastic threading mechanisms, and the utilization of persistent memory.
- We demonstrate the effectiveness of our cost optimization framework through extensive evaluations using synthetic benchmarks, real-world workloads, and case studies from Ant Group's production environments. These evaluations showcase TierBase's superior cost-effectiveness compared to existing solutions and its ability to achieve significant cost reductions in large-scale, data-intensive applications.



(a) Single-Tier Storage (b) Tiered Storage

Figure 2: Space-Performance Cost Model

2 SPACE-PERFORMANCE COST MODELING

In this section, we introduce a novel cost model for key-value storage systems, based on real-world implementations at Ant Group. Our model unifies performance and storage costs, providing a comprehensive framework for system optimization in online data serving with real-time latency requirements.

2.1 Cost Analysis Framework

Our framework is built upon two primary components: **Performance Cost (*PC*)** in key-value storage systems reflects the expenses associated with data transfer from storage media to end-users, encompassing resource utilization for both read and write operations. This includes CPU overhead, network I/O, disk IOPS consumption, and memory bandwidth usage.

Space Cost (*SC*) depends on the resources and expenses for data storage, varying by storage medium and space used. In caching and in-memory storage, it's linked to the data volume in RAM, which is fast but expensive. Disk-based storage costs are typically cheaper for large data volumes. These costs also account for storage overhead related to data structures and replicas necessary to ensure reliability and availability.

Space-Performance Cost Model. Our Space-Performance Cost Model is based on the observation that in enterprise data centers and cloud environments, resource instances (virtual machines or containers with compute and storage resources) are typically provided with pre-defined allocations. These allocations are usually evenly divided to maximize utilization, precluding arbitrary resource allocation.

Given this constraint, for a given workload w, which is a stream of read and write operations on a given resource instance i and storage system with configuration s, both the maximum performance ($MaxPerf(w, i, s)$) and the maximum storable data amount ($MaxSpace(w, i, s)$) are deterministic and quantifiable. These metrics are measured in queries per second (QPS) and gigabytes (GB), respectively.

In a distributed, shared-nothing architecture, we define the monetary cost C as the maximum of the performance cost (PC) and the space cost (SC) for a given workload on a set of resource instances i with the same configuration:

Definition 1 (Cost of workload w)**.** The cost of workload w is defined as the maximum of the performance cost and the space cost for a given workload on a set of resource instances i with a specific storage configuration s:

$$C(w,i,s) = \max(PC(w,i,s), SC(w,i,s)) \quad (1)$$

Where:

$$PC(w,i,s) = Cost(i) \times \left\lceil \frac{QPS(w)}{MaxPerf(w,i,s)} \right\rceil$$

$$SC(w,i,s) = Cost(i) \times \left\lceil \frac{DataSize(w)}{MaxSpace(w,i,s)} \right\rceil$$

Here, $Cost(i)$ is the monetary cost of a single resource instance i, $QPS(w)$ is the total queries per second for the workload w, $DataSize(w)$ is the total amount of data to be stored for workload w, and $MaxPerf(w,i,s)$ and $MaxSpace(w,i,s)$ are the maximum performance and space capacity for the given resource instance i and storage configuration s, respectively.

The maximum of performance cost and space cost is used because in a distributed, shared-nothing architecture, we must provision enough resources to meet both query processing and data storage demands simultaneously. This approach ensures that the system can handle whichever requirement is greater, whether it's query load or data volume.

For real-world deployments, we incorporate tolerance ratios for both $MaxPerf$ and $MaxSpace$, ensuring system redundancy and reliability. These ratios accommodate variations in workload distribution and access patterns, enabling adaptation to scenarios that deviate from the cost model's assumption of even data sharding.

In Figure 2(a), we illustrate both the performance and space costs. For workloads where the performance cost exceeds the space cost, we categorize them as *performance-critical workloads*. Conversely, when the space cost is dominant, we refer to these as *space-critical workloads*.

2.2 Cost Efficiency Metrics

Our cost model provides a framework for optimizing resource allocation and system configuration in distributed key-value storage systems. By leveraging the insights from this model, we can make informed decisions to minimize overall costs while meeting both performance and space requirements.

Consider a scenario where a workload typically requires multiple resource instances. We can simplify our model by removing the ceiling function and define the cost metrics as follows:

Definition 2 (Cost Metrics)**.** We define two key cost metrics:

$$CPQPS = Cost(i)/MaxPerf(w,i,s)$$

$$CPGB = Cost(i)/MaxSpace(w,i,s)$$

where CPQPS is the Cost per Query per Second, representing the performance cost incurred for processing each query per second, and CPGB is the Cost per GB, representing the space cost for storing each gigabyte of data.

For ease of presentation in subsequent discussions, we will consider the same i and w, allowing us to represent different configurations as subscripts in later formulae. Using these cost metrics, we can express the total cost of the storage system as:

$$C = \max(CPQPS \times QPS, CPGB \times DataSize) \quad (2)$$

This formulation allows us to evaluate and optimize the overall system cost by considering both performance and space requirements.

2.3 Space-Performance Trade-off and Optimal Cost Theorem

Our cost model reveals an inherent trade-off between performance and space costs in key-value storage systems. This trade-off forms the basis for our Optimal Cost Theorem.

Definition 3 (Space-Performance Trade-off of Storage Configurations)**.** Given a set of storage configurations S, the space-performance trade-off describes the relationship between the Cost per Query per Second ($CPQPS$) and the Cost per Gigabyte ($CPGB$), expressed as: $CPQPS_s = f(CPGB_s), \quad s \in S$ where f is a non-increasing function. As $CPGB_s$ decreases for a given configuration s, $CPQPS_s$ tends to increase, and vice versa.

A typical example of this trade-off is data compression: for a fixed compression algorithm, setting higher compression levels reduces space cost ($CPGB$) but increases performance cost ($CPQPS$) due to added computational overhead, allowing a trade-off between storage efficiency and query performance.

Based on the trade-off, we establish the Optimal Cost Theorem:

Theorem 2.1 (Optimal Cost C^*). *For a given workload w with requirements QPS and $DataSize$, and a set of storage configurations S, the optimal cost C^* is achieved by selecting the configuration $s^* \in S$ that minimizes the overall cost while balancing performance and space costs: $C^* = \min_{s \in S} \max(PC_s, SC_s)$*

The optimal configuration s^* is one that minimizes the absolute difference between performance and space costs: $s^* =_{s \in S} |PC_s - SC_s|$

Proof. Let s^* be the optimal configuration that minimizes the overall cost:

$C^* = \min_{s \in S} \max(PC_s, SC_s)$

Assume, for contradiction, that $PC_{s^*} \neq SC_{s^*}$. Without loss of generality, let $PC_{s^*} > SC_{s^*}$. Then:

$C^* = PC_{s^*}$

Now, consider a configuration s' that slightly reduces PC_{s^*} at the expense of increasing SC_{s^*}, such that:

$PC_{s'} = PC_{s^*} - \epsilon \; SC_{s'} = SC_{s^*} + \delta$

Where $\epsilon > 0$ and $\delta > 0$ are small positive values.

If we choose ϵ and δ such that $SC_{s^*} + \delta < PC_{s^*} - \epsilon$, then:

$\max(PC_{s'}, SC_{s'}) = PC_{s'} = PC_{s^*} - \epsilon < PC_{s^*} = C^*$

This contradicts the assumption that s^* is the optimal configuration. Therefore, our initial assumption must be false, and we must have $PC_{s^*} = SC_{s^*}$ for the optimal configuration.

Hence, the optimal configuration s^* is one that minimizes the absolute difference between performance and space costs:

$s^* =_{s \in S} |PC_s - SC_s|$

This proves both parts of the theorem. □

The proof demonstrates that any imbalance between performance and space costs can be optimized to yield a lower total cost, thus establishing the optimal point at their equality.

This theorem provides a guiding principle for optimizing key-value storage systems. It suggests that the most cost-effective configuration is one where the system's resources are balanced such that

neither space nor performance costs dominate. This balance point represents the optimal trade-off between space and performance for the given workload.

2.4 Cost Model with Tiered Storage

Tiered storage systems typically comprise two layers: a cache tier and a storage tier. The cache tier, often utilizing memory-based technologies, prioritizes performance, while the storage tier focuses on capacity and cost-effectiveness. This structure allows systems to balance performance and capacity requirements more efficiently than single-tier alternatives.

In a tiered storage system, the total cost is a function of both the cache tier and the storage tier. We propose a comprehensive cost model for tiered storage systems that accounts for both performance and capacity costs across tiers:

$$\begin{aligned} C_{tiered} = \max(&PC_{cache} + PC_{miss} \times MR, \\ &SC_{cache} \times CR) \\ &+ \max(PC_{storage} \times MR, SC_{storage}) \end{aligned} \tag{3}$$

where:

- CR is the cache ratio (cache capacity / total capacity)
- MR is the cache miss ratio (proportion of requests served by the storage tier)
- PC_{miss} is the additional performance cost incurred on a cache miss
- $PC_{cache/storage}$ and $SC_{cache/storage}$ represent performance and space costs for each tier

This model enables a comparison between tiered storage and single-tier alternatives. Tiered storage becomes cost-effective when its total cost is lower than both a pure cache solution and a pure storage solution, expressed as: $C_{tiered} < \min(C_{cache}, C_{storage})$. This approach can be particularly effective for workloads with skewed data access patterns.

The model provides insights into optimizing cache ratio (CR) and managing miss ratio (MR) to minimize overall system cost. Further detailed cost analysis on tiered storage is presented in Section 5.2.

2.5 Cost Optimization Strategies

Our cost model serves as a valuable guide for optimization efforts, assisting system designers and administrators in their decision-making process. We discuss optimization strategies for both single-tier and tiered storage systems, using the space-performance cost model as our framework.

2.5.1 Single-Tier Storage Optimization. In single-tier storage systems, the primary goal is to balance performance cost ($CPQPS$) and space cost ($CPGB$) to minimize overall cost. The optimization strategies depend on the workload characteristics:

Space-Critical Workloads: When the workload is space-critical. Here, reducing $CPGB$ becomes the primary goal. One approach is to enable data compression, which reduces $CPGB$ but may increase $CPQPS$ due to the additional computational overhead. The overall cost can still be optimized through such trade-offs according to the Optimal Cost Theorem. Other potential approaches include using instance with larger storage and implementing tiered storage solutions.

Performance-Critical Workloads: In this case, optimization efforts should focus on reducing $CPQPS$. Strategies may include: Optimizing query execution, tiered caching mechanisms, and utilizing faster storage media for frequently accessed data.

Resource Instance Selection: Choosing the right type of resource instance can significantly impact both $CPQPS$ and $CPGB$. This involves analyzing different instance types to find the optimal balance between performance capabilities and storage space for the specific workload.

2.5.2 Tiered Storage Optimization. Tiered storage systems offer additional opportunities for cost optimization by leveraging the strengths of different storage tiers. The effectiveness of tiered storage depends on three key factors:

Skewed Data Access Pattern: Optimal tiered storage performance occurs when both Cache Ratio (CR) and Miss Ratio (MR) are low. This scenario is typical in workloads with high temporal locality, where a small subset of "hot" data is frequently accessed. The cache tier can store this data, resulting in a low CR while serving most requests (low MR).

Cost Disparity between Tiers: The significant cost difference between cache and storage tiers is crucial for effective tiered storage. The cache tier offers high performance at a higher cost per unit capacity, while the storage tier provides lower performance at a much lower cost. This disparity allows the system to balance high performance for frequently accessed data with cost-effective storage for less accessed information.

Low Miss Penalty: The miss penalty (PC_{miss}) represents the additional performance cost when a request misses the cache. A low miss penalty is vital for effective tiered storage as it reduces the impact of cache misses, allowing for a smaller cache (lower CR) without significantly degrading overall performance. We will present our techniques for minimizing PC_{miss} in Subsection 4.1.

2.5.3 Workload-Driven Optimization Approaches. Instead of immediately diving into TierBase-specific techniques, we can introduce a more general framework for mapping workload characteristics to optimization strategies. This sets the stage for the detailed techniques that will be discussed in the following section.

Table 1 presents a mapping of various workload features to optimization strategies implemented in TierBase. This mapping illustrates how specific workload characteristics inform the choice of optimization techniques.

These optimization techniques are designed to address specific workload characteristics and leverage the strengths of both single-tier and tiered storage architectures. By applying these strategies guided by our cost model, we can iteratively refine system configurations, resource allocations, and feature enablements.

In the following sections, we will detail each of these optimization techniques, explaining how they work and how they contribute to overall system cost-effectiveness. Later, in Section 6, we will demonstrate how these strategies are applied in practice. Our evaluation methodology involves replaying real-world workloads and assessing costs across various configurations, providing empirical validation of our cost-optimization approach.

Table 1: Workload Features and Optimization Options

Workload Features	Optimization Options
Skewed access patterns (a small subset of data accessed frequently)	Tiered Storage Elastic Threading
Low latency requirements	In Memory Mode PMem Usage
Space-critical (Large volume, low throughput)	Larger Storage Instance Tiered Storage Pre-trained Compression
Performance-critical (High throughput, small volume)	In Memory Mode PMem for Persistence
Read-heavy, Write-less	Elastic Threading Pre-trained Compression
Write-heavy	Write-back Caching PMem for WAL

3 TIERBASE SYSTEM DESIGN

Building upon our unified space-performance cost model, we present TierBase, a high-performance, distributed key-value storage system designed to optimize cost for large-volume online storage. TierBase leverages a tiered storage architecture to provide low-latency, cost-effective data access while addressing the challenges highlighted in our cost model.

TierBase extends Redis's capabilities by supporting not only basic key-value operations like GET and SET, but also advanced data structures such as lists, sets, and sorted sets. Additionally, it provides CAS (Compare-And-Set) operations, wide-column data handling and vector search.

TierBase supports vector search by integrating the VSAG library[9], a vector indexing library developed by Ant Group for similarity search, which enables efficient ANN queries over high-dimensional vectors within our key-value infrastructure. The integration supports dynamic vector operations, including real-time insertion and deletion in memory, demonstrating performance improvements of 3-4x compared to conventional algorithms such as HNSW.

TierBase integrates a persistence mechanism to ensure data reliability, safeguarding against data loss in the event of system failures. Its geo-replication component facilitates inter-cluster replication and backups, enabling data to be securely replicated across geographical regions for enhanced disaster recovery capabilities.

Furthermore, TierBase incorporates a tiered storage architecture, which separates the caching layer from the storage layer, allowing each to scale independently based on workload requirements. The system also features memory compression capabilities, reducing memory footprint without significant performance overhead, and elastic threading support, dynamically adjusting the number of threads based on workload demands for optimal resource utilization.

The design target of TierBase is to achieve a space–performance balance with flexible configuration of several system knobs, such as pre-trained compression, elastic threading, and different persistence mechanisms. In the following sections, we will introduce the overall architecture of TierBase, exploring its primary components.

3.1 Architecture Overview

As shown in Figure 3, the architecture of TierBase is structured into three primary tiers: the client tier, cache tier, and storage tier. The cache tier employs in-memory hash tables stored in DRAM or persistent memory (PMem) for efficient random access performance, while the storage tier typically utilizes a LSM-tree structure stored on SSD or HDD to optimize write performance and storage capacity. This architecture allows TierBase to effectively balance high-speed data access with efficient data storage across different storage media.

Client Tier. The client tier consists of the TierBase clients and TierBase proxy services. The TierBase client has compatible interfaces with native Redis clients (e.g. Jedis [2], Lettuce [4]), and retrieves cluster routing information from the coordinator cluster to access data directly on specific instances. Additionally, the client automatically handles failover, cluster scaling, Quality of Service (QoS). The TierBase client is highly suitable for scenarios involving large-scale applications. Applications using the client connect directly to data instances, achieving optimal performance, latency and minimizing costs.

For relatively small-scale scenarios requiring native Redis clients, TierBase also provides a proxy service that includes the client capabilities. This facilitates rapid integration for public cloud users.

Cache Tier. The cache tier in our architecture is composed of a collection of TierBase instances and coordinators. Each data instance within TierBase functions as a data node, featuring data sharding.The cache instances implements hash tables for efficient key-value storage. These instances are capable of maintaining data in a single-replica mode, primarily serving as a caching system. TierBase also offers a multi-replica mode to further enhance data reliability within memory. It implements asynchronous and synchronous master-replica replication, along with the RAFT [47] synchronization protocol for different levels of data reliability.

It is important to note that the cache tier can function as an independent shared-nothing caching system, for in-memory storage use cases, providing high-speed data access similar to systems like Redis and Memcached. When equipped with local persistent storage, such as SSDs and persistent memory, it can also serve as an in-memory database.

Coordinators play a pivotal role in overseeing the entire cluster, managing data node failover, and administering tenant resource allocation. The coordinator group includes an elected leader, who manages the cluster, while followers distribute client layout retrieval workload. The coordinator uses Zookeeper [33] for metadata management, covering tenant details, cluster layout, configurations, and leader election among coordinators.

Storage Tier. The storage tier provides data persistence through a disaggregated storage system. In the disaggregated cache-storage architecture, the cache tier directly accesses the storage tier for data fetching in cases of cache misses. Data writing employs either a write-through or a write-back policy, ensuring data persistence across both cache and storage tiers.

TierBase offers an array of disaggregated storage options. The experiments focus on evaluating the performance and cost-effectiveness of using the Universal Configurable Storage (UCS)[1], a sophisticated real-time serving and analytical storage engine. UCS implements an LSM-Tree with a shared disk architecture and remote compaction.

[1]Note that UCS is an internal system and does not have a publicly available reference at the time of writing this paper.

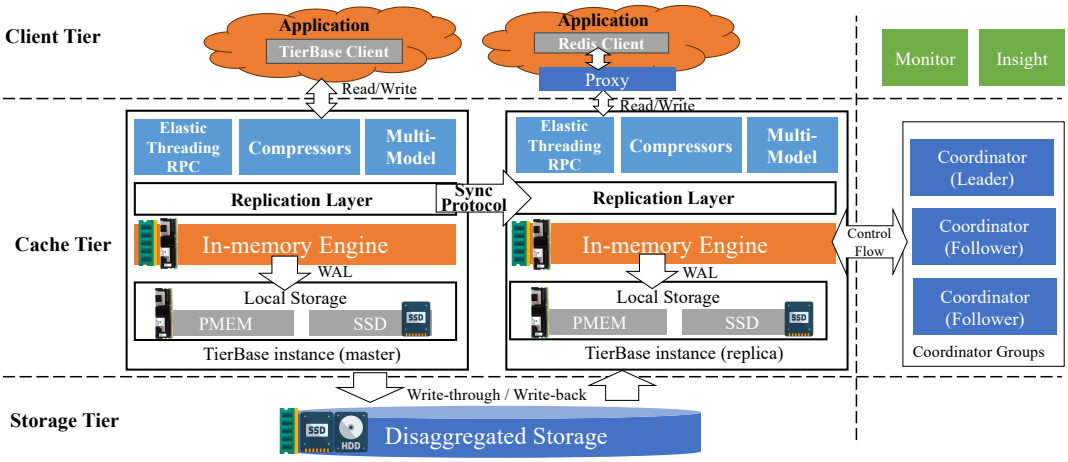


Figure 3: TierBase architecture overview

This design ensures optimal online performance while supporting both row and columnar storage formats.

Although our experiments focus on an LSM-tree storage engine, the pluggable storage adapter in TierBase allows integration with various disaggregated storage systems based on different data structures. Consequently, the cost optimization techniques in the cache tier and cost evaluation with the cost model can be applied to a wide range of key-value stores.

TierBase also includes a monitoring service, as well as a related storage analysis tool named Insight. Monitor service is responsible for the collection and display of TierBase's real-time metrics data and log data. Insight utilizes this data for problem diagnosis and to provide user suggestions according to the workload. Furthermore, TierBase is integrated with Cougar [55] for automatic scaling, commonly referred to as the serverless mode in the cloud service.

4 COST OPTIMIZATION STRATEGIES

In this section, we introduce the well-designed features which aim to optimize the cost of TierBase.

4.1 Tiered Storage

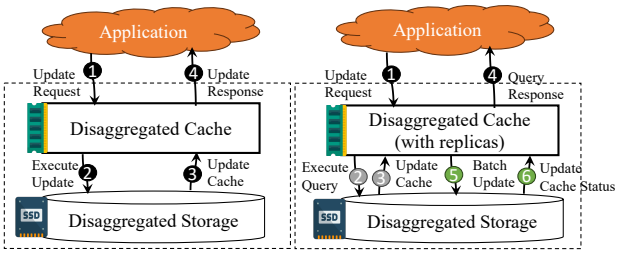
TierBase introduces a tiered storage architecture that disaggregates cache and storage components, allowing for independent optimization. This approach directly addresses the space-performance trade-off highlighted in our cost model. The cache tier is optimized for speed (minimizing PC_{cache}), while the storage tier is designed for capacity and durability (optimizing $SC_{storage}$). Both tiers can scale independently, accommodating diverse workloads and data access patterns.

To ensure data consistency and reliability in this disaggregated architecture, we adapt the well-known caching techniques "write-through" and "write-back". These strategies are commonly used in conventional hardware scenarios to synchronize data between cache and storage. However, applying these techniques to a disaggregated architecture presents unique challenges. In traditional contexts, write-through and write-back policies are implemented within a single tier, where the cache and storage are tightly coupled. This tight coupling allows for simpler coordination and synchronization mechanisms between the cache and storage, as they reside on the same physical node or have low-latency communication channels. In contrast, the separation of cache and storage tiers in TierBase introduces new complexities in maintaining data consistency and synchronization, requiring careful design and implementation of coordination protocols and synchronization strategies.

4.1.1 Write-through Caching. In the write-through caching policy (Figure 4(a)), TierBase prioritizes data consistency between the cache and storage tiers. When a write request is received, it is first executed on the cache tier and then synchronously passed to the disaggregated storage tier before acknowledging completion to the application. If the storage update succeeds, the cache tier maintains the updated data; otherwise, the corresponding cache entry is invalidated, and an error is returned to the application.

To ensure data consistency in the presence of failures while reducing PC_{miss}, TierBase employs several key techniques:

- **Temporary Update Buffer.** Each connection maintains a temporary update buffer. Incoming update requests are initially performed on this buffer, and the results are used to update the main cache. If the storage write succeeds, the data is seamlessly transferred from the temporary buffer to the main cache. In case of a storage write failure, the corresponding entry in the main cache is removed, ensuring subsequent reads fetch the data from the storage, maintaining consistency.
- **Sequential Write Ordering.** As storage updates are performed asynchronously, maintaining the sequential order of writes to the same key is crucial. TierBase achieves this by using a per-key write queue. When a key already has pending writes, subsequent write operations are appended to the queue, ensuring sequential execution order in the cache tier.
- **Write Coalescing.** To optimize write-heavy workloads and effectively handle write hotspots, TierBase incorporates write coalescing within an event loop. The event loop is the main thread's processing loop that handles incoming client connections, processes commands, and manages the overall operation from the implementation of Redis server. Instead of executing each update command separately on the store, multiple write commands targeting the same key



(a) Write-through caching policy (b) Write-back caching policy

Figure 4: Caching policy for write operation

are coalesced into a single operation similar to the concept of group commit in database systems. This coalesced operation is then executed, efficiently updating the store with the final result while reducing the number of write operations lowering PC_{miss}.

This write-through strategy is particularly effective in environments where read operations significantly outnumber write operations, and high data reliability are critical.

4.1.2 Write-back Caching. The write-back caching policy in TierBase prioritizes performance by optimizing PC_{cache} and reducing PC_{miss}. Updates are first written to the cache tier with immediate response to the application, while data synchronization to the storage tier is deferred and performed asynchronously in batches. This approach minimizes $PC_{storage}$ by reducing the frequency of writes to the storage tier.

If the requested data is not present in the cache during an update operation, TierBase queries the storage tier to fetch the data, as illustrated in steps 2 and 3 of Figure 4(b). The fetched data is then updated in the cache tier. Data that has been updated in the cache but not yet synchronized to the storage tier is marked as "dirty". These dirty data updates are periodically propagated to the storage tier in batches to minimize the frequency of remote calls to the storage tier. Furthermore, when multiple updates are pending for the same key, these updates can be merged in the batch to reduce write operations.

Implementing write-back caching in a disaggregated architecture introduces unique challenges in ensuring data reliability and optimizing synchronization efficiency between cache and storage:

- **Replication of Cache.** In write-back caching, there is a risk of data loss if the cache tier fails before the dirty data is synchronized to the storage tier. TierBase addresses this issue by enabling data replication in the cache tier. By maintaining multiple replicas of the dirty data and the cache, TierBase ensures that even if one cache node fails, the data can still be recovered from the replicas, preventing data loss.
- **Managing Dirty Data.** A crucial aspect of the write-back approach involves maintaining dirty data in the cache tier at a reasonable scale. Overly large dirty data may result in significant overhead for a single remote call, whereas an insufficient amount may yield only marginal improvements in storage performance. In TierBase, this equilibrium is achieved by restricting the size of dirty data and establishing a maximum interval time for the batch updates. Additionally, as dirty data approaches a predefined safe threshold, especially under heavy write workloads, TierBase activates a backpressure mechanism directed at clients. This prevents situations where synchronization of dirty data to the storage tier cannot be achieved within an acceptable timeframe.
- **Optimizing Update.** To optimize write performance in write-back caching, TierBase employs several strategies. Firstly, it minimizes the frequency of remote calls to the storage tier by batching the updates for dirty data. Secondly, for updates pertaining to the same key, TierBase merges these updates in the batch to further reduce write operations. The coalescing is even more effective as it can coalesce more writes in the batch instead the writes in the event loop introduced in write-through caching.
- **Deferred Cache-fetching.** TierBase introduces a mechanism to handle cases where an update operation is performed on a key missing in the cache tier. When such an update operation occurs, TierBase accumulates a certain number of these operations and submits a batch read task to fetch the corresponding data from the storage tier into the cache tier. The fetched data is then updated correspondingly in the cache tier. This approach reduces the number of read requests to the storage tier, minimizing costs in both tiers. If a user's read request accesses data pending in the batch read, it is considered a cache miss, triggering immediate retrieval and fetch from the storage tier. This optimization is suitable for scenarios with frequent write operations on keys not present in the cache tier.

We argue that the consistency level of data in write-back caching is determined by the cache tier's configurable coherent protocol, including master-replica or RAFT. This approach ensures that updates generated within the cache tier are eventually propagated to the underlying storage, which supports strong consistency.

The write-back caching strategy is particularly well-suited for scenarios with frequent writes and reads, especially when reads occur shortly after writing, and where the requirements for data reliability are moderately lower.

4.1.3 Write-through versus write-back. TierBase supports both write-through and write-back caching policies, each offering different trade-offs:

Write-through caching provides lower SC_{cache} as it doesn't store dirty data, but potentially higher PC_{cache} for write-heavy workloads due to synchronous storage updates. Write-back caching offers lower PC_{cache} and PC_{miss} for write-heavy workloads due to deferred and batched storage updates, but incurs higher SC_{cache} due to storing dirty data and potential replication.

The choice between these policies depends on specific workload characteristics and relative costs of cache and storage resources. Write-back caching may provide better cost-performance for write-heavy workloads with good temporal locality, while write-through caching may be more cost-effective for read-heavy workloads or when storage writes are relatively inexpensive.

TierBase's flexible configuration allows users to select the appropriate caching policy based on their application's requirements

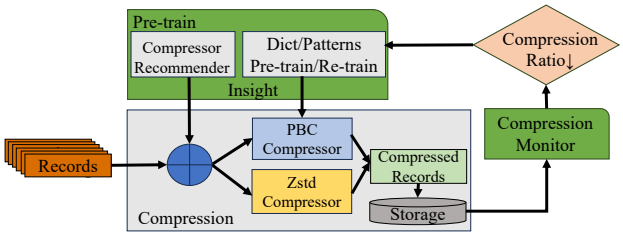


Figure 5: The framework of pre-trained based compression

and cost structure, optimizing the space-performance trade-off for various workloads and scenarios.

4.2 Pre-trained Compression Mechanism

In the context of our space-performance cost model, in-memory data compression plays a crucial role in optimizing the trade-off between storage costs (*SC*) and performance costs (*PC*) within the memory tier.

Ant Group processes a significant volume of machine-generated data, characterized by recurring phrases and structures, from various application serialization methods. To efficiently compress this in-memory data, we've developed a pre-trained compression mechanism, leveraging two high-efficiency algorithms: our newly developed Pattern-Based Compression (PBC) [63] and the widely adopted Zstandard (Zstd) [16] by Meta.

The framework of pre-trained compression is shown in Figure 5. During the pre-training phase, Zstd builds a dictionary by identifying frequent strings in the data, while PBC employs hierarchical clustering and a unique similarity metric to pinpoint and extract data patterns. In the compression phase, the resulting patterns, along with residual strings, are then compressed further using string compression techniques.

Initially, we construct the dictionary (patterns) in an offline manner using samples from data records. Then we apply these dictionary (patterns) to all instances of the workload. Ultimately, instances can employ this dictionary to perform data compression and decompression.

One of the key challenges in utilizing pre-trained compression in production is the necessity to restart sampling and re-training of the datasets if the value pattern changes; otherwise, the compression ratio will be affected. To address this, TierBase introduces a monitoring service that continuously tracks compression efficiency and initiates re-sampling and re-training cycles when necessary. Specifically, this service tracks the compression ratio and the number of data records that do not align with a pattern. A re-sampling and retraining process is triggered when the compression ratio falls below a baseline level or when the rate of unmatched records surpasses a predefined threshold.

Additionally, TierBase's Insight service features a compressor recommender that automatically proposes the optimal compressor based on specific data types and performance needs. This adaptive compression mechanism is capable of recommending an appropriate compressor for different application data types, dynamically rebuilding the dictionary or pattern base to accommodate changes in data patterns.

The pre-trained compression mechanism efficiently manages large volumes of machine-generated data by achieving high compression ratios and adaptively adjusting strategies based on changing patterns and requirements. Our experiments demonstrate the cost-effectiveness of our pre-trained compression mechanism, enabling TierBase to dynamically adjust its compression strategy and balance the space-performance trade-off based on evolving data patterns and workloads. Despite a moderate increase in performance cost (*PC*) for write operations due to compression overhead, the significant reduction in space cost (*SC*) and high decompression speed for read operations optimize overall cost-effectiveness. By adjusting compression levels, TierBase can fine-tune the balance between *SC* and *PC*, achieving an optimal point in the cost model to minimize total cost while maintaining high performance across diverse workloads.

4.3 Persistent Memory Utilization

A standout component in TierBase's storage strategy is the adoption of Intel's Optane Memory [34], which leverages 3D XPoint technology. Optane Memory distinguishes itself through its substantial capacity, affordability compared to DRAM, swift memory-like access speeds, and its non-volatile nature. The PMem enhances the overall performance while reduce the cost of TierBase in two folds:

- **DRAM Extension**: Serving as an economical DRAM supplement, PMem allows for efficient memory use by keeping frequently accessed (hot) data, such as key-value pairs and index structures, in DRAM, while less accessed (cold) data is stored in PMem. This strategy optimizes the balance between space cost (*SC*) and performance cost (*PC*).
- **WAL Persistence**: PMem greatly improves TierBase's Write-Ahead Log (WAL) persistence by overcoming the I/O operations per second (IOPS) bottleneck found in disk or cloud storage, ensuring faster and more consistent data synchronization. Crucially, WAL files are first written to a PMem-based persistent ring buffer, then batch-moved to cloud storage, achieving high throughput and real-time persistence, thus significantly boosting performance in high-demand scenarios.

In integrating PMem into TierBase, one of the key challenges faced was addressing the performance disparities between PMem and DRAM, particularly in terms of write latency. To mitigate this, TierBase optimized its memory allocator strategy to efficient data placement. Keys and indexes, which are relatively small and frequently accessed, are stored in DRAM, while larger value data is stored in PMem. Additionally, write operations to PMem are optimized by assembling data structures in DRAM and then copying them to PMem in batches, reducing the number of PMem write operations.

Our experience of using PMem in production environments has shown that even with straightforward data structures, PMem can deliver strong performance when incorporated into a tiered storage architecture. In this approach, PMem is used alongside DRAM and SSDs, with each tier being optimized to leverage its unique strengths.

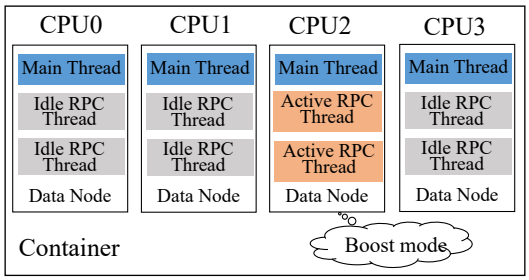


Figure 6: Elastic threading

4.4 Elastic Threading

TierBase implements an innovative elastic threading approach to optimize performance cost (*PC*) within allocated node resources. This dynamic method, shown in Figure 6, seamlessly switches between single-thread and multi-thread modes based on workload demands, enhancing system responsiveness and resource efficiency without external scaling.

Elastic threading in TierBase synergizes the benefits of both single-threaded and multi-threaded architectures. Single-threaded models are efficient in CPU resource usage, avoiding complexities like locking and context switching associated with multi-threading. However, they often struggle with high-concurrency demands, leading to bottlenecks during peak periods. Conversely, multi-threaded systems enhance performance and resource utilization of a single instance but introduce synchronization challenges and higher CPU overhead.

In normal conditions, TierBase operates in a default single-thread mode, utilizing an event-driven model with epoll. This approach offers high CPU efficiency and lower *PC* for typical workloads. We claim that the efficiency of a single-threaded process per data shard generally outperforms that of multi-threading due to reduced locking overhead, a principle supported by Amdahl's Law[7].

When the workload on a particular instance increases significantly, TierBase seamlessly transitions to multi-threaded mode by dynamically adding threads within the container's pre-allocated CPU resources. Containers are provisioned with CPU capacity based on anticipated peak workloads, but this capacity isn't fully utilized during normal operations. Elastic threading allows TierBase to leverage these underutilized CPU resources when needed, boosting the affected instance's performance without exceeding resource limits or incurring additional costs. When the workload subsides, TierBase switches back to single-thread mode, allowing CPU resources to be used by other processes within the container and maximizing resource efficiency.

The transition employs a lock-free implementation that incorporates lock-free queues and epoll eventfd, along with sub-second real-time monitoring of instance load. These techniques ensure minimal synchronization overhead and sustain performance during both the transition and operation in multi-thread mode.

Elastic threading is particularly effective for skewed workloads like dynamic hotspots.As shown in Figure 6, typically, one instance might switch to multi-threaded mode while others remain in single-threaded mode within the same container, optimizing resources across the system. If the container's overall CPU load remains consistently high, the system recognizes the need to scale out to further enhance tenant performance.

This approach improves responsiveness to immediate demands and optimizes resource use, preventing unnecessary allocation during low-activity periods. By dynamically balancing between single-threaded efficiency and multi-threaded performance, elastic threading significantly contributes to overall CPU efficiency. The elastic threading model aligns with modern computing trends that emphasize balancing performance with resource consumption.

5 FURTHER COST ANALYSIS AND FRAMEWORK

This section provides an in-depth analysis of our cost model by examining space-performance trade-offs in storage systems, relating these to our Optimal Cost Theorem and the classic Five-Minute Rule. We adapt the Five-Minute Rule for modern distributed systems and provide a framework for cost optimization for tiered storage.

5.1 Adapting the Five-Minute Rule for Modern Storage Systems

The Five-Minute Rule[10, 24, 25], introduced by Jim Gray and Gianfranco Putzolu in 1985, has been a cornerstone in database system design. Originally formulated for single-server environments, it provided a simple heuristic for deciding whether data should be kept in memory or on disk based on its access frequency:

$$BreakEvenInterval = \left(\frac{PagesPerMBofRAM}{AccessPerSecondPerDisk}\right) \times \left(\frac{PricePerDiskDrive}{PricePerMBofRAM}\right) \quad (4)$$

However, in today's distributed and cloud-based systems, we need to consider a broader range of factors and trade-offs. We propose an adapted version of the Five-Minute Rule that aligns with our cost model:

$$BreakEvenInterval = \frac{CPQPS_{slow}}{CPGB_{fast} \times AverageRecordSize} \quad (5)$$

Where $CPQPS_{slow}$ is the Cost Per Query Per Second for slower, space-optimized storage, $CPGB_{fast}$ is the Cost Per Gigabyte for faster, performance-optimized storage, and $AverageRecordSize$ is the average size of data records in the workload.

To illustrate how our adapted rule relates to the original, we provide the following mapping.

- The ratio $\left(\frac{PricePerDiskDrive}{AccessPerSecondPerDisk}\right)$ effectively represents the $CPQPS_{slow}$.
- $PricePerMBofRAM$ correlates to $CPGB_{fast}$.
- $PagesPerMBofRAM$ is conceptually similar to $\left(\frac{1GB}{AverageRecordSize}\right)$.

This formulation determines the optimal point on the space-performance trade-off spectrum for a given workload in a distributed environment. It can be illustrated by comparing fast, in-memory storage systems like Redis [54] with slower, more space-efficient systems like HBase[58], or by examining different configurations of the same database system.

The break-even interval analysis, derived from our adapted Five-Minute Rule, optimizes data placement by comparing data access

intervals to the break-even point. It guides the choice between fast, performance-oriented storage and slower, space-efficient options.

Our Cost Optimal Theorem extends this concept by determining the best overall storage settings for a given workload, aiming to minimize system cost by balancing performance cost (*PC*) and space cost (*SC*). It considers the entire system configuration, including multiple storage tiers and complex workload characteristics, to guide high-level design and resource allocation decisions.

Our comprehensive experiments assessed the performance and cost efficiency of our system against leading open-source key-value databases using this integrated approach. The evaluation results on real application workloads demonstrate the model's effectiveness in guiding cost-saving database system designs. The case study in Section 6.5 showcases how the break-even interval, derived from the Five-Minute Rule, helps choose the most cost-effective Tier-Base configuration within the framework established by the Cost Optimal Theorem.

5.2 Cost Analysis of Tiered Storage

Revisiting Equation 3, we can focus on optimizing the cost of the cache tier. In disaggregated storage systems with a sufficiently large storage pool, the storage tier cost is dominated by *SC* when $MR < SC_{storage}/PC_{storage}$ for skewed access patterns. As illustrated in Figure 2(b), this allows us to concentrate on the cache tier cost:

$$Cost_{cache} = \max(PC_{cache} + PC_{miss} \times MR, \\ SC_{cache} \times CR) \tag{6}$$

To find the optimal cost, we consider the relationship between the Miss Ratio (*MR*) and the Cache Ratio (*CR*), typically represented by the Miss Ratio Curve[31], where $MR = f(CR)$, and f is a non-increasing function.

Theorem 5.1 (Optimal Cache Tier Cost). *The optimal cost for the cache tier of a tiered storage is achieved when the performance cost equals the space cost:*

$$PC_{cache} + PC_{miss} \times f(CR^*) = SC_{cache} \times CR^* \tag{7}$$

where CR^* is the optimal cache ratio.

Let CR^* be the optimal cache ratio that minimizes the overall cost of the cache tier:

$Cost^*_{cache} = \min_{0 \le CR \le 1} \max(PC_{cache} + PC_{miss} \times f(CR), SC_{cache} \times CR)$

Define two functions:

$g(CR) = PC_{cache} + PC_{miss} \times f(CR)$ $h(CR) = SC_{cache} \times CR$

Note that $g(CR)$ is non-increasing (as $f(CR)$ is non-increasing) and $h(CR)$ is increasing linearly with *CR*.

The optimal cost occurs at the intersection of these two functions. To see why, consider:

1. If $g(CR) > h(CR)$, we can decrease *CR* to reduce cost. 2. If $g(CR) < h(CR)$, we can increase *CR* to reduce cost. 3. The minimum cost occurs when neither of these improvements is possible, i.e., at $g(CR) = h(CR)$.

Therefore, the optimal cache ratio CR^* satisfies:

$PC_{cache} + PC_{miss} \times f(CR^*) = SC_{cache} \times CR^*$

This equality represents the balance point where performance cost (including miss penalty) equals space cost, minimizing the overall cache tier cost. This theorem provides a principle for optimizing tiered storage systems: the most cost-effective configuration is one where the cache tier's performance cost (including the cost of cache misses) equals its space cost. This balance point represents the optimal trade-off between performance and space for the cache tier.

In practice, estimating the exact CR^* is challenging, as $f(CR)$ can be complex and highly dependent on specific workload characteristics. Nevertheless, this theorem serves as a valuable target for optimization efforts, guiding cache size tuning. To address this challenge, we propose an evaluation-based approach in Section 5.3 to find the optimal *CR*.

Furthermore, this analysis provides a way for determining when to use tiered storage over single-tier solutions and how to optimally configure the cache tier within a tiered storage system, enhancing our understanding of cost-effective storage design in modern disaggregated environments.

5.3 Cost Optimization Framework

In order to speedup the cost optimization procedure, we develop a sample-based method to calculate the cost for various configurations with regard to real-world workloads. The method involves the following steps:

(1) **Sample**: Sample data snapshots and record a representative period of workload from production instances.
(2) **Load**: Load the sampled data snapshot into a testing instance with a specific configuration.
(3) **Replay**: Replay the recorded real-world key-value operation traces on the testing instance, measuring and collecting the maximum performance and maximum space utilization for the workload.
(4) **Calculation**: Calculate the workload cost based on measurements.
(5) **Iteration**: Repeatedly perform steps 2-4 with different configurations to approach cost-optimal configuration.

This method simulates key-value store behavior under realistic conditions, providing accurate performance and cost assessments. By using real workload traces and access patterns, we obtain a precise representation of system performance. This method enables comprehensive exploration of the configuration space, ensuring identification of the most cost-effective configuration for each individual workload.

While the configuration space for cost optimization can be large, in practice, we focus on the most impactful parameters specific to the workload, guided by user input and prior experience. This approach narrows the candidate configurations substantially. Optimization computations are performed offline and parallelized to accelerate the process, ensuring that the time invested is minor compared to the long-term cost savings achieved. To address the cold start problem, we initialize the system with configurations based on user inputs and best practices.The detailed optimization results and cost savings are analyzed with case studies in Section 6.

6 EXPERIMENTS

6.1 Settings

The experimental evaluation is conducted on the following servers. For the cache tier, we use three servers with dual Intel Xeon Platinum 8263C CPUs at 2.50 GHz, 192GB DRAM, and eight 128GB Intel® Optane™ DCPMM 100 series(App Direct Mode), while for storage server, we use three servers with Intel Xeon Platinum 8163 CPU at 2.50 GHz, 64GB DRAM, and 8TB SSDs. For all servers, NUMA is enabled and Hyper-threading is disabled for linear performance scaling and to avoid contention between logical cores. The system and software environments are as follows: Linux kernel version 4.19.91, OpenJDK version 1.8.0, GCC version 10.2.1, Dragonfly 1.23.0, Redis 6.0.17, Cassandra 4.0.11, HBase 2.4.1, and Memcached 1.6.20.

YCSB (Yahoo! Cloud Serving Benchmark) [17] is utilized which encompasses the load and run phase. Our experiments utilize two distinct default workloads from YCSB: Workload A, characterized by a predominance of write operations, and Workload B, distinguished by a higher proportion of read operations. We have adapted YCSB to accept user-specified datasets for data insertion, as opposed to the default use of random strings as values. In particular, the cities dataset [1] is designated as the default for our tests. We deploy 16 YCSB threads for single-thread cases and 48 for multi-thread cases.

We focus our comparisons on widely-used production systems to evaluate TierBase's performance and cost-effectiveness in real deployment scenarios, which have essential features implemented, such as full failure recovery. These systems can reflect the performance and cost accurately in real world applications. We selected Redis [54], Memcached [35] and Dragonfly [19] for caching system comparison. Redis and Memcached are established caching systems, extensively utilized across diverse applications. Dragonfly is a newly introduced, high-performance caching system. For databases with persistence, we select Redis with AOF, Cassandra [38], and HBase [58] as competitors. Specifically, Redis-AOF ensures data durability by logging writes to disk, which can impact performance due to the additional disk I/O overhead.

Instances represent the fundamental units of resource allocation. In systems operating in single-thread mode, each instance is allocated 1 CPU core and 4GB of memory. In multi-thread mode and databases with persistence, the allocation for each instance increases to 4 CPU cores and 16GB of memory. These specifications are common instance configurations used by Ant Group.

6.2 Performance Evaluation

6.2.1 Caching Systems. Figure 7 illustrates the performance comparison of four caching systems: TierBase, Redis, Memcached, and Dragonfly. During this evaluation, TierBase was tested in its default mode without any cost optimization techniques enabled. We evaluate the performance for single-thread and multi-thread mode separately and report the throughput and 99th percentile tail latency respectively.

In single-thread mode (Figures 7(a) and 7(b)), TierBase and Redis exhibit similar performance, outperforming Memcached and Dragonfly across all workloads. This distinction arises because Memcached and Dragonfly are principally engineered for multi-thread environments, while in contrast, Redis is meticulously optimized for

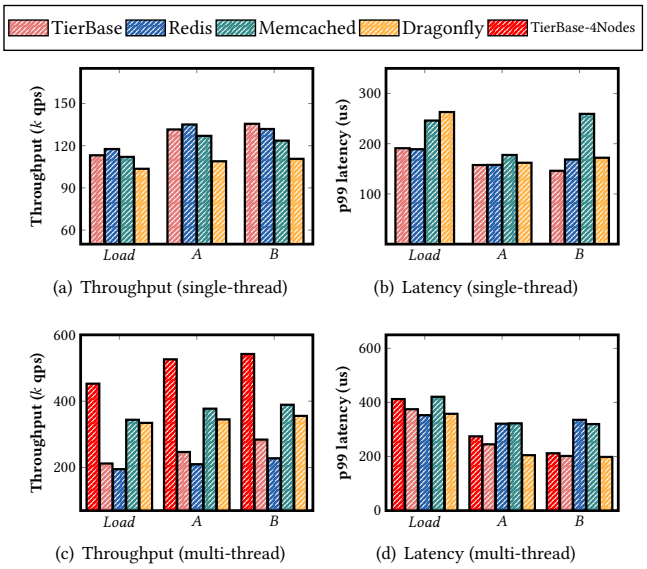


Figure 7: Performance of single-thread and multi-thread Mode

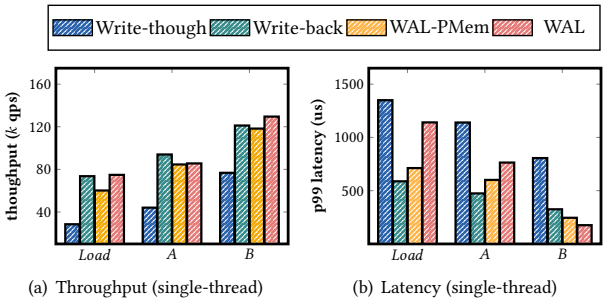


Figure 8: Performance of different data persistence mechanisms

single-thread mode. TierBase maintains the lowest latency across most workloads. During the load phase, the latency of TierBase and Redis is significantly lower than that of Memcached and Dragonfly.

In multi-thread mode (Figures 7(c) and 7(d)), Memcached and Dragonfly surpass TierBase and Redis. Memcached combines a streamlined caching approach with a lightweight threading model, minimizing inter-thread contention and enhancing speed. Dragonfly benefits from a shared-nothing architecture for threads, boosting its parallel processing. Although TierBase's per-instance throughput is slightly lower in multi-thread mode, it excels in real-world scenarios through efficient scaling. Figure 7(c) shows that 4 single-threaded TierBase instances outperform a single multi-threaded instance of Memcached or Dragonfly using equivalent resources, leading to a lower performance cost. In practice, scaling out across multiple instances meets performance requirements, and TierBase's cost-effectiveness makes this economically viable. Thus, TierBase effectively balances performance and cost, aligning with the demands of large-scale applications.

6.2.2 Persistence Mechanisms. Figure 8 outlines TierBase's performance with four persistence mechanisms in single-thread mode: WAL, WAL while using PMem as the persistent ring buffer(WAL-PMem), write-back and write-through introduced in Section 4.1.

For the throughput performance(Figure 8(a)), write-back significantly outperforms write-through in the load phase by 92.52%, due

Table 2: Evaluation of compression techniques

Datssets		cities	KV1	KV2
Comp. Ratio	PBC	**0.2003**	**0.2341**	**0.2297**
	Zstd	0.2920	0.4042	0.4096
Overall Comp. Ratio	PBC	**0.4919**	**0.6884**	**0.6219**
	Zstd	0.5508	0.7594	0.7117
Thoughput (SET)	PBC	54469	74878	65329
	Zstd	61667	75018	67558
	Raw	**122324**	**122414**	**120496**
Thoughput (GET)	PBC	109998	119688	115888
	Zstd	96861	112866	102616
	Raw	**138045**	**132397**	**132961**

to its deferred writing mechanism that reduces immediate write operation overhead. In various read-write workloads, write-back's throughput is about twice of write-through, demonstrating its efficiency in write-intensive tasks. The WAL-PMem mode, while not as effective as write-back, still surpasses write-through, indicating benefits by using persistent memory. However, WAL mode outperforms WAL-PMem due to its use of SSDs and asynchronous disk flushes every second, while WAL-PMem synchronizes to PMem per transaction, potentially incurring higher synchronization overhead.

In terms of latency (Figure 8(b)), write-through experiences the highest latency due to its immediate write to storage, being around 3 times higher than write-back in the load phase. Write-back, with its deferred write, significantly lowers latency, particularly in write-heavy scenarios. WAL-PMem offers a middle ground, with lower latency than write-through but higher than write-back.

6.3 Features Evaluation

6.3.1 Compression. As introduced in 4.2, TierBase has implemented pre-trained based compression strategies to mitigate memory utilization. We evaluate effectivness of the pre-trained based compression methods. Basic Zstd[16] (denoted as Zstd-b) which is without pre-trained dictionaries, Zstd with pre-trained dictionaries(denoted as Zstd-d) and Pattern-Based Compression[63] (PBC). We also include the raw data without compression(Raw) as the bar of throughput.

As shown in Table 2, the pre-trained based methods, PBC and Zstd-d, consistently outperform Zstd-b. It demonstrates that the pre-trained mechanism, by prior analysis and storage of common data patterns, is able to enhance the compression ratio. Notably, PBC consistently achieves higher compression ratios than Zstd. In KV datasets, the distinctive patterns within the values lead to a more significant improvement in PBC's compression performance. Specifically, PBC surpasses Zstd-d by 43% and Zstd-b by 74% in average compression ratios. These enhanced ratios contribute to PBC's sustained superiority in overall compression performance.

In the evaluation of throughput. all three compression methods perform worse compared to Raw, especially in SET operations. Among the three compression mechanisms tested, Zstd-d demonstrated the highest performance. In public datasets, the throughput of Zstd-d was approximately twice as high as PBC and about 3.5 times higher than Zstd-b. This notable difference is primarily due to the higher computational overhead in PBC's compression process which involves pattern matching and string encoding. Meanwhile, Zstd-b, without pre-trained dictionary, necessitates online data analysis during compression, which hampers throughput. In contrast, when considering average GET operation throughput, PBC not only surpasses Zstd-d but also nearly parallels the velocity of Raw. Zstd-b still demonstrates the least favorable performance.

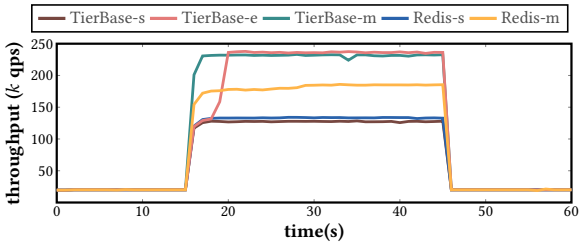


Figure 9: Performance of TierBase and Redis for workload boosting

The implementation of compression introduces a calculated trade-off, modestly impeding throughput performance in exchange for substantial memory conservation, thereby augmenting the judicious utilization of resources. Based on the pre-trained compression strategy, it is possible to preserve common patterns in the data in advance, avoiding real-time computation during compression and decompression, thereby enhancing the efficiency and effectiveness of the compression process. The Section 6.4 will provide an in-depth exploration of the multifaceted benefits engendered by compression.

6.3.2 Elastic threading. In order to show the effect of elastic threading, we simulate a scenario of workload burst to test the system's adaptability to sudden workload influxes. At the beginning, the workload maintenance a low QPS(20,000). Then, we increase in the number of client requests to simulate a surge in workload at the 15 second. This state lasts for 30 seconds and finally the workload returned to normal, the low QPS state. We represent single-thread, multi-thread and elastic threading modes as *s*, *m* and *e* (e.g. TierBase-s, TierBase-m, TierBase-e).

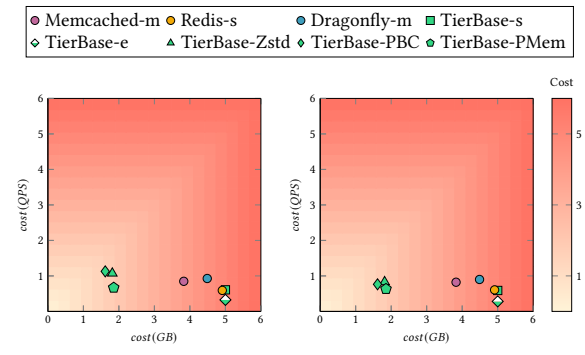
As shown in Figure 9, under normal conditions, all databases manage well, with Redis showing some jitter in multi-thread mode. Upon increased workload, systems' throughput reaches their limits, and latency rise. In single-thread mode, TierBase has the highest latency, followed by Redis. However, with elastic threading, TierBase initially faces higher latency but quickly adjusts to have the lowest latency, equal to its multi-thread mode performance. In terms of throughput, both TierBase and Redis hit 120,000 QPS in single-thread mode, with Redis peaking at 180,000 QPS and TierBase at over 240,000 QPS in multi-thread mode.

In summary, the experiment result indicates that elastic threading allows TierBase to operate in a cost-saving single-thread mode under normal scenario, while automatically switching to a multi-thread mode to achieve higher throughput performance during workload spikes, without the need for manual intervention.

6.4 Cost Evaluation

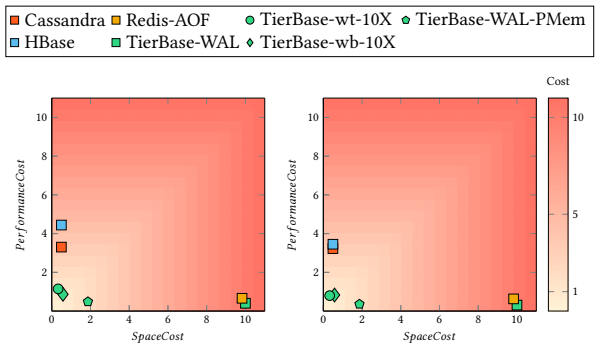
In the following two subsections, we evaluate the cost-effectiveness of TierBase and compare it with other representative systems under different configurations and synthetic workloads using the space-performance cost model.

6.4.1 Evaluation setup. We employ the framework introduced in Section 5.3 to evaluate synthetic workloads generated by YCSB



(a) 50%write 50%read (b) 95%read 5%write

Figure 10: Cost of caching system



(a) 50% read 50%write (b) 95% read 5%write

Figure 11: Cost of database with persistence

using public datasets for write operations. Our simulated workload comprises 10GB data with 80,000 QPS for caching systems, and 10GB data with 40,000 QPS for databases with persistence. While our cost model is applicable to various workloads, we selected these specific parameters as a representative miniature of typical workloads at Ant Group.

The cost unit presented is relative, based on a standard container with 1 CPU core and 4GB of memory. All systems are tested within this standard container on a single instance, with *CPQPS* and *CPGB* calculated accordingly. This evaluation methodology allows us to assess the cost-effectiveness of various systems under controlled conditions.

For systems employing replication (e.g., Redis with AOF, TierBase with WAL, and TierBase with write-back policy), we implement a master-replica setup in the cache tier to ensure data reliability. This configuration effectively doubles the cache tier cost.

We denote single-thread, multi-thread, and elastic threading as *s*, *m*, and *e* respectively. TierBase-PMem is for PMem activation in TierBase, and TierBase-Zstd and TierBase-PBC for compression. Redis-AOF and TierBase-WAL denote Redis with AOF and TierBase with WAL, respectively. Write-through and write-back policies in TierBase are abbreviated as *wt* and *wb*. Workloads with a cache ratio of 10 are labeled as 10*X*.

6.4.2 Cost analysis for caching systems. Figure 10 presents the cost results for caching systems. The primary cost driver for caching systems is memory storage expenses. Memcached has the lowest storage cost, followed by Dragonfly, while Redis and TierBase without additional features have relatively higher storage costs.

In terms of performance costs, TierBase, Redis, and Memcached exhibit similar low costs in single-thread mode, while Dragonfly shows a higher performance cost. When elastic threading is enabled, TierBase demonstrates improved throughput, leading to a significant reduction in performance costs, nearly half that of single-thread Redis, by efficiently utilizing surplus CPU resources within the containers.

Furthermore, when TierBase employs PMem to extend memory, it achieves a substantial 60% reduction in storage costs compared to the base configuration, with minimal performance impact. This cost is considerably lower than Memcached. Activating compression in TierBase leads to an additional decrease in storage costs.

The results show that TierBase's features can effectively reduce both performance and storage costs. Similar trends are observed across different workload settings, as illustrated in Figure 10(b).

6.4.3 Cost analysis for databases with persistence. Figure 11 presents the results for databases with persistence. The traditional key-value store like Cassandra and HBase are observed to have relatively high performance costs while the storage costs are notably low. For Redis with AOF and TierBase with WAL, both systems ensure data persistence and adopt a dual-replica strategy for data safety. These approaches result in lower performance cost but significantly higher storage costs.

TierBase demonstrates a good balance in terms of both performance and storage costs. On the one hand, its inherent characteristics as a caching system enable high throughput. On the other hand, its persistence mechanism does not require storing all data in memory, which contributes to its overall lower costs. It is important to note that under the write-back approach, where data is stored in duplicate copies, the storage cost is higher compared to write-through. However, due to different data update characteristics, the write-back approach exhibits higher throughput in write-intensive scenarios, translating to lower performance costs. This advantage diminishes or even disappears in read-heavy scenarios.

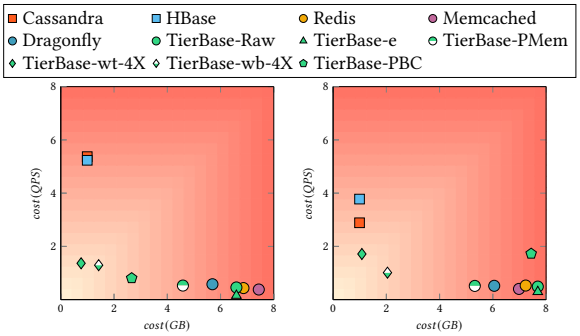
Using PMem for data persistence is a cost-effective choice. Although its space cost is relatively higher compared to write-through and write-back, its performance cost is sufficiently low due to PMem's near-memory speed.

6.5 Case Study

TierBase is extensively utilized across a wide range of scenarios at Ant Group, with over 3,000 applications leveraging its capabilities. These applications span various use cases, employing hundreds of thousands of CPU cores and several petabytes of memory. Due to space constraints, we will focus on two representative case studies in this paper.

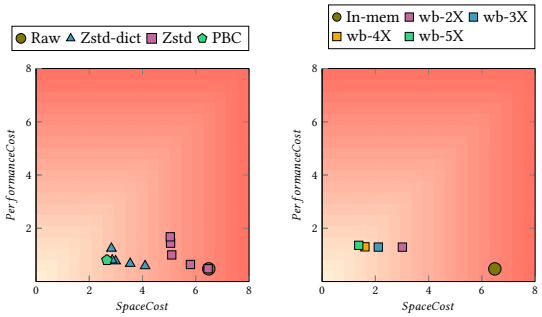
Case 1: User Info Service.

The User Info Service at Ant Group manages basic user profile data, serving numerous applications through a second-party library with TierBase client. During peak hours on a typical day, this service handles approximately 500,000 updates and 16,000,000 reads per second, indicating a significantly read-heavy workload. Given the service's primary focus on online users, high availability and reliability are of paramount importance.



(a) User Info Service (b) Capital Reconciliation

Figure 12: Cost of case study



(a) Compression level trade-offs (b) Cache ratio trade-off

Figure 13: Space-Performance Cost Trade-offs

6.5.1 Systems comparison. To assess the cost-effectiveness of various systems, we replayed a real business trace with all databases configured for dual-replica reliability. Figure 12(a) shows that in-memory stores like Redis, Memcached, and Dragonfly have low performance costs but higher storage expenses. TierBase, using compression, halves its original volume, significantly reducing costs compared to Redis. This is particularly advantageous in this read-heavy scenario where performance cost is not primary. The trade-off between performance and storage efficiency demonstrates TierBase's adaptability to specific workload characteristics, optimizing overall cost while maintaining performance efficiency. Activating compression in TierBase yields a 62% cost reduction compared to TierBase-Raw, showcasing its effectiveness in balancing performance and storage requirements in read-heavy, availability-critical scenarios.

6.5.2 Space-performance cost trade-offs. We demonstrate the performance space trade-off using our proposed cost model under use case 1. Figure 13 shows the workload's space cost significantly exceeds its performance cost. We employ compression techniques, sacrificing some performance to save substantial space. We tested Zstd compression levels -50, -10, 1, 15, and 22, both with and without a dictionary. As shown in Figure 13(a), higher compression levels increase space savings but have an upper bound, beyond which compression ratio gains become marginal while performance costs grow considerably. Practically, we may select compression level 1 for better performance tolerance. Additionally, pre-trained compression yields more substantial cost savings compared to compression without pre-training.

We also evaluate the cost-effectiveness of TierBase with write-back policy using four cache ratios ranging from 2X to 5X. Higher cache ratios result in lower space costs but higher performance costs.

The result reveals that a cache ratio of 5X approximately achieves the optimal balance between performance and storage costs as predicted by our model. The results validate our cost model for accurately guiding cost optimization, confirming that the achieved effects align with expectations.

6.5.3 Break-even interval. Furthermore, we calculate several sets of break-even intervals between the fast and slow storage configurations on TierBase based on the analyses in Section 5.1. As shown in Table 3, if the average access interval for a key in the workload is less than 98 seconds, the default TierBase is the most cost-effective choice. For access intervals between 98 and 264 seconds, TierBase with PMem mode is recommended. When the average access interval exceeds 264 seconds, employing compression becomes the optimal solution. By collecting the average access interval for a key in the real workload, we observe that it exceeds 1018 seconds. Consequently, TierBase is employed as a single-layer caching system, leveraging pre-trained compression (PBC) to optimize memory usage. This approach achieves a 25% compression rate for values and realizes cost savings of 50%, which is significant considering the hundreds of thousands of CPU cores utilized in this case.

Table 3: Break-even interval between different configurations.

Fast Storage	Slow Storage	Time Interval(s)
Raw	PMem	98
Raw	Compression(PBC)	184
PMem	Compression(PBC)	264

Although write-through caching could potentially be applicable in this scenario and achieve a 60% cost reduction, the decision to prioritize compression over write-through caching is driven by the client's stringent requirements for low latency and high stability when serving online requests. TierBase's adaptability allows it to configure for aligning with the specific needs and priorities of each client, ensuring an optimal balance between cost efficiency and performance in real-world scenarios.

Case 2: Capital Reconciliation. TierBase is also deployed in the capital reconciliation business at Ant Group. As a risk control scenario focused on financial auditing and verification, the capital reconciliation business is particularly sensitive to costs. During peak shopping seasons, the overall QPS for capital reconciliation can reach tens of millions. TierBase's write-through and write-back caching strategies are selectively employed based on different scenarios. For this case study, we choose one of the main scenarios where the read and write operations are close to a 1:1 ratio. In this scenario, data from different channels is written into TierBase and then read out by the reconciliation system for verification.

Figure 12(b) illustrates the cost breakdown for this scenario. Disk-based key-value stores like HBase and Cassandra exhibit low space and performance costs. When TierBase is configured with write-through, performance costs are lowered by 35% compared to Cassandra. In high-throughput scenarios, enabling write-back mode further enhances performance. With the same configuration,

TierBase can achieve 2.6x the performance of Cassandra. Overall, TierBase reduces costs by at least 37% compared to both Cassandra and HBase. Additionally, it cuts costs by 70% compared to TierBase default configuration.

The observations in the capital reconciliation case study reveal that recent data is frequently accessed in the cache, while long-term data is occasionally retrieved. Online statistics shows that TierBase with write-through mode achieves a cache hit rate of approximately 80%, with only 1% of the hottest data stored in the cache tier. This demonstrates the effectiveness of TierBase's cache-storage disaggregation in significantly reducing costs for workloads with temporal access skewness.

7 RELATED WORK

7.1 Key-Value Stores

Key-value stores, a type of NoSQL databases, play a crucial role in Internet applications. Diverging from traditional relational databases, they provide fast, scalable, and efficient data access essential for a wide range of online applications.

Memcached [35] is a distributed caching system enhancing web application performance by minimizing database load and optimizing memory across servers. Redis [54], unlike Memcached's focus on caching, is a multifunctional in-memory database with a wide variety of data types, extended functionality, and persistence, suitable for complex, high-performance applications. KeyDB [3] enhances Redis with multi-threading. Dragonfly [19] offers compatibility with Redis and Memcached, using multi-threaded, shared-nothing architecture. Etcd [51] is crucial for Kubernetes, providing consistent configuration across clusters. EVCached [46], ElastiCache [56], and Azure Cache for Redis [44] are managed caching solutions, improving cloud applications by reducing database loads and enabling quick data access.

7.2 Cost Optimization

7.2.1 Cost model. Constructing a reasonable cost model is crucial for the optimization of database costs. Total Cost of Ownership (TCO)[45] refers to the sum of all related costs throughout the entire lifecycle of an item or service, including purchasing, operating, maintaining, and disposing of it. This concept is widely applied across various fields. Some cost models [29, 41, 43] primarily focus on estimating or optimizing query execution time or overall system performance. They consider factors like memory access patterns, cache behavior, and data structure efficiency to predict or improve query processing speed. Other cost models [39, 50] explicitly consider financial costs, such as the cost of operating cloud resources or the cost differences between different types of storage medium. However, these methods often struggle to adequately represent the complex relationships between configurations and costs, especially when considering diverse workload characteristics. As a result, they are not directly applicable to our study's objective.

7.2.2 Optimization strategy. Contrasting with TierBase, which concentrates on a comprehensive cost model that balances performance and storage expenses, some studies [18, 27, 36, 61] achieve more effective resource allocation or configuration parameter adjustments by monitoring workload variations within databases, thereby enhancing resource utilization and reducing costs. Cosine [13] actively tailors its storage engine architecture to optimize costs, adapting to workload demands, cloud budgets, and specific performance objectives.

7.2.3 Compression. In database technology, Lempel-Ziv(LZ) algorithms are widely used for data compression to improve storage and transfer efficiency. TiDB [32] and Hadoop [58] use LZ4, while LevelDB [22] uses Snappy [23]. Zstandard (Zstd) [16] is chosen by Facebook's RocksDB [20] and Redshift [6] for its efficiency. Apache Cassandra [38] also uses compression to reduce disk space use. For memory efficiency, SlimCache [37] and zExpander [62] compress data in caches, and COLLATE [30] applies lightweight compression in in-memory databases.

7.2.4 Thread management. Thread management techniques play a pivotal role in DBMSs by optimizing the handling of concurrent client requests and maximizing resource utilization. In systems like [20, 48, 49, 57], dynamic adjustment of thread pools is utilized to efficiently handle fluctuating workloads. On the other hand, databases such as [35, 54] maintain a fixed thread count, ensuring consistent resource allocation. [15, 26, 28, 52], use coroutines in certain scenarios for managing concurrent requests, balancing thread overhead with concurrency. [53] discusses thread management in the NUMA-aware task scheduling. [5] merges these approaches, applying dynamic thread pools for general requests and fixed threads for specific tasks.

7.2.5 Persistent memory. Non-volatile memory, also known as persistent memory is a type of memory that keeps its data even when the power is turned off. Many current efforts are integrating PMem into storage systems [8, 11, 14, 21, 42, 59, 64] to enhance performance. However, PMem has its own set of challenges and issues. Some studies [12, 40, 60] are exploring how systems should navigate between PMem and DRAM, addressing concerns about optimal choice and management.

8 CONCLUSION

This paper introduces TierBase, a distributed key-value store developed by Ant Group to address the challenges of managing large-scale online data serving systems. TierBase employs a tiered storage architecture and incorporates cost-effective optimizations. We propose a space-performance cost model to guide the selection of optimal storage configurations for diverse workloads. Extensive evaluations using both synthetic benchmarks and real-world workloads demonstrate TierBase's superior cost-effectiveness compared to existing solutions.

REFERENCES

[1] Cities. https://www.geonames.org/search.html?q=&country=.
[2] Jedis. https://github.com/redis/jedis.
[3] Keydb. https://docs.keydb.dev/docs/.
[4] Lettuce. https://github.com/lettuce-io/lettuce-core.
[5] Inc. Aerospike. Aerospike: The flash-optimized, in-memory nosql database. 2010.
[6] Amazon. Redshift. https://aws.amazon.com/cn/redshift/.
[7] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA*, volume 30 of *AFIPS Conference Proceedings*, pages 483–485. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967.

[8] Mijin An, Jonghyeok Park, Tianzheng Wang, Beomseok Nam, and Sang-Won Lee. NV-SQL: boosting OLTP performance with non-volatile dimms. *Proc. VLDB Endow.*, 16(6):1453–1465, 2023.

[9] AntGroup. Vsag. https://github.com/alipay/vsag.

[10] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. The five-minute rule thirty years later and its impact on the storage hierarchy. In Rajesh Bordawekar and Tirthankar Lahiri, editors, *International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures, ADMS@VLDB 2017, Munich, Germany, September 1, 2017*, pages 1–8, 2017.

[11] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. Viper: An efficient hybrid pmem-dram key-value store. *Proc. VLDB Endow.*, 14(9):1544–1556, 2021.

[12] Lawrence Benson, Leon Papke, and Tilmann Rabl. Perma-bench: Benchmarking persistent memory access. *Proc. VLDB Endow.*, 15(11):2463–2476, 2022.

[13] Subarna Chatterjee, Meena Jagadeesan, Wilson Qin, and Stratos Idreos. Cosine: A cloud-cost optimized self-designing key-value storage engine. *Proc. VLDB Endow.*, 15(1):112–126, 2021.

[14] Lixiang Chen, Ruihao Chen, Chengcheng Yang, Yuxing Han, Rong Zhang, Xuan Zhou, Peiquan Jin, and Weining Qian. Workload-aware log-structured merge key-value store for NVM-SSD hybrid storage. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 2207–2219. IEEE, 2023.

[15] Kristina Chodorow. Mongodb: A scalable, high-performance, open source NoSQL database. 2010.

[16] Yann Collet and Murray S. Kucherawy. Zstandard compression and the application/zstd media type. *RFC*, 8878:1–45, 2018.

[17] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In Joseph M. Hellerstein, Surajit Chaudhuri, and Mendel Rosenblum, editors, *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010*, pages 143–154. ACM, 2010.

[18] Carlo Curino, Evan P.C. Jones, Samuel Madden, and Hari Balakrishnan. Workload-aware database monitoring and consolidation. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, page 313–324, New York, NY, USA, 2011. Association for Computing Machinery.

[19] DragonflyDB. Dragonflydb. https://www.dragonflydb.io/.

[20] Facebook. Rocksdb. https://github.com/facebook/rocksdb.

[21] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Qian Long, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, and Feifei Li. Tair-pmem: a fully durable non-volatile memory database. *Proc. VLDB Endow.*, 15(12):3346–3358, 2022.

[22] Google. Leveldb. https://github.com/google/leveldb.

[23] Google. Snappy compression library. https://github.com/google/snappy.

[24] Goetz Graefe. The five-minute rule 20 years later (and how flash memory changes the rules). *Commun. ACM*, 52(7):48–59, 2009.

[25] Jim Gray and Franco Putzolu. The 5 minute rule for trading memory for disc accesses and the 10 byte rule for trading memory for cpu time. *ACM SIGMOD Record*, 16(3):395–398, 1987.

[26] Mail.Ru Group. Tarantool: An in-memory database and application server. 2012.

[27] Rui Han, Chi Harold Liu, Zan Zong, Lydia Y. Chen, Wending Liu, Siyi Wang, and Jianfeng Zhan. Workload-adaptive configuration tuning for hierarchical cloud schedulers. *IEEE Transactions on Parallel and Distributed Systems*, 30(12):2879–2895, 2019.

[28] Yongjun He, Jiacheng Lu, and Tianzheng Wang. Corobase: Coroutine-oriented main-memory database engine. *Proc. VLDB Endow.*, 14(3):431–444, 2020.

[29] Moditha Hewasinghage, Alberto Abelló, Jovan Varga, and Esteban Zimányi. A cost model for random access queries in document stores. *The VLDB Journal*, 30(4):559–578, mar 2021.

[30] Juliana Hildebrandt, Dirk Habich, Patrick Damme, and Wolfgang Lehner. Compression-aware in-memory query processing: Vision, system design and beyond. In Spyros Blanas, Rajesh Bordawekar, Tirthankar Lahiri, Justin Levandoski, and Andrew Pavlo, editors, *Data Management on New Hardware*, pages 40–56, Cham, 2017. Springer International Publishing.

[31] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Trans. Storage*, 14(2), apr 2018.

[32] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. Tidb: A raft-based htap database. *Proc. VLDB Endow.*, 13(12):3072–3084, aug 2020.

[33] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin C. Reed. Zookeeper: Wait-free coordination for internet-scale systems. In Paul Barham and Timothy Roscoe, editors, *2010 USENIX Annual Technical Conference, Boston, MA, USA, June 23-25, 2010*. USENIX Association, 2010.

[34] Intel. Intel® optane™ memory. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/overview.html.

[35] Danga Interactive. Memcached. http://memcached.org/.

[36] Zhaoxuan Ji, Zhongle Xie, Yuncheng Wu, and Meihui Zhang. Lbsc: A cost-aware caching framework for cloud databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*, pages 4911–4924, 2024.

[37] Yichen Jia, Zili Shao, and Feng Chen. Slimcache: An efficient data compression scheme for flash-based key-value caching. *ACM Trans. Storage*, 16(2), jun 2020.

[38] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.*, 44(2):35–40, 2010.

[39] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proc. VLDB Endow.*, 15(11):2867–2880, jul 2022.

[40] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. Cost modelling for optimal data placement in heterogeneous main memory. *Proc. VLDB Endow.*, 15(11):2867–2880, 2022.

[41] Yan Li, Liwei Wang, Sheng Wang, Yuan Sun, and Zhiyong Peng. A resource-aware deep cost model for big data query processing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, pages 885–897, 2022.

[42] Gang Liu, Leying Chen, and Shimin Chen. Zen+: a robust NUMA-aware OLTP engine optimized for non-volatile main memory. *VLDB J.*, 32(1):123–148, 2023.

[43] Stefan Manegold, Peter Boncz, and Martin L. Kersten. Generic database cost models for hierarchical memory systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, page 191–202. VLDB Endowment, 2002.

[44] Microsoft. Azure cache for Redis. https://azure.microsoft.com/en-us/services/cache/.

[45] Lars Mieritz and Bill Kirwin. Defining gartner total cost of ownership. *L. Mieritz, B. Kirwin*, 2005.

[46] Netflix. Evcache. https://github.com/Netflix/EVCache.

[47] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX annual technical conference (USENIX ATC 14)*, pages 305–319, 2014.

[48] Oracle. Oracle database concepts. https://www.oracle.com/, 2020.

[49] Oracle. Mysql. https://www.mysql.com, 2021.

[50] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel oltp systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 61–72, New York, NY, USA, 2012. Association for Computing Machinery.

[51] Brandon Philips. etcd: A reliable distributed key value store. In *CoreOS Fest*, 2015.

[52] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: A practical approach for robust index joins. *Proc. VLDB Endow.*, 11(2):230–242, 2017.

[53] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *Proc. VLDB Endow.*, 8(12):1442–1453, 2015.

[54] Redis. Redis. https://redis.io/, 2009.

[55] Bo Sang, Shuwei Gu, Xiaojun Zhan, Mingjie Tang, Jian Liu, Xuan Chen, Jie Tan, Haoyuan Ge, Ke Zhang, Ruoyi Ruan, and Wei Yan. Cougar: A general framework for jobs optimization in cloud. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*, pages 3417–3429, 2023.

[56] Amazon Web Services. Amazon elasticache. https://aws.amazon.com/elasticache/.

[57] Justin Sheehy and David Smith. Bitcask: A log-structured hash table for fast key/value data. *Basho White Paper*, 2010.

[58] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[59] Jason Sun, Haoxiang Ma, Li Zhang, Huicong Liu, Haiyang Shi, Shangyu Luo, Kai Wu, Kevin Bruhwiler, Cheng Zhu, Yuanyuan Nie, Jianjun Chen, Lei Zhang, and Yuming Liang. Accelerating cloud-native databases with distributed pmem stores. In *39th IEEE International Conference on Data Engineering, ICDE 2023, Anaheim, CA, USA, April 3-7, 2023*, pages 3043–3057. IEEE, 2023.

[60] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. Managing non-volatile memory in database systems. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1541–1555. ACM, 2018.

[61] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, 2019.

[62] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. Zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[63] Jiujing Zhang, Zhitao Shen, Shiyu Yang, Lingkai Meng, Chuan Xiao, Wei Jia, Yue Li, Qinhui Sun, Wenjie Zhang, and Xuemin Lin. High-ratio compression for machine-generated data. *Proc. ACM Manag. Data*, 1(4):245:1–245:27, 2023.

[64] Wenhui Zhang, Xingsheng Zhao, Song Jiang, and Hong Jiang. Chameleondb: a key-value store for optane persistent memory. In Antonio Barbalace, Pramod Bhatotia, Lorenzo Alvisi, and Cristian Cadar, editors, *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021*, pages 194–209. ACM, 2021.