Rethinking the Cost of Distributed Caches for Datacenter Services

Ziming Mao[†] Jonathan Ellithorpe[‡] Atul Adya[‡]
Rishabh Iyer[†] Matei Zaharia[†] Scott Shenker[†*] Ion Stoica[†]

[†]UC Berkeley [‡]Databricks [*]ICSI

Abstract

This paper systematically studies the cost impact of distributed in-memory caches on datacenter services. While memory used for these caches is often perceived to be expensive, we find that the resulting CPU savings from these in-memory caches far outweigh the cost of added memory. In fact, across a variety of both synthetic and production workloads, we find that adding distributed in-memory caches can lower total operating costs by $3-4\times$, even without considering their latency benefits. These cost savings can vary significantly across various architectures, such as storage layer caches, remote lookaside caches, and in-memory linked caches. We additionally evaluate cost for two emerging scenarios: caching rich application objects and strongly consistent cache. For the former, we find that caching application objects provides outsized benefits compared to their denormalized, key-value-style variants, up to 8× compared to reading from storage. For the latter, we observe that even a minimal version check for consistency can eliminate most of the cost benefits, calling for new designs for cost-effective consistent cache.

CCS Concepts

• **Information systems → Information storage systems**; • **Computer systems organization**;

Keywords

Caching, Data Freshness, Consistent Cache

HotNets '25, November 17–18, 2025, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/25/11
https://doi.org/10.1145/3772356.3772388

ACM Reference Format:
Ziming Mao, Jonathan Ellithorpe, Atul Adya, Rishabh Iyer, Matei Zaharia, Scott Shenker, Ion Stoica. 2025. Rethinking the Cost of Distributed Caches for Datacenter Services. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3772356.3772388

1 Introduction

In-memory caches are widely used to achieve low latency in datacenter services that tolerate eventual consistency [5, 14, 39, 48, 50, 51]. Popular implementations include remote caches like Memcached and Redis [7, 19, 41, 49] which often serve as lookaside caches. Additionally, many storage systems natively support automatic caching of popular tables and rows for fast lookups [17, 37, 40, 42, 47]. More recently, linked in-memory key-value caches have been proposed to further reduce latency by avoiding costly network hops, data (de)serialization, and over-reads [2].

While in-memory caches are valuable for bringing down service latency and are widely deployed, we find a lack of research on their monetary cost. While it is well-known that caches improve performance, the cost impact of these in-memory caches (beyond performance benefits) for datacenter services is not well understood. From discussing with practitioners in both the database and systems community, many believe that the cost of memory in the cloud makes achieving low latency appear to be expensive [26, 30, 31]. DRAM is approximately 32× more expensive than storage [35], and cloud operators reported that memory can constitute 50% of server cost [29] and 37% of total cost of ownership [33].

This paper presents a systematic cost study of cache deployments for datacenter services with three new observations. First, adding these low-latency in-memory caches can reduce demand for CPU by so much that it not only makes up for the added cost of the memory but significantly reduces total operating costs of the application. Across both open-sourced production traces from Meta [7] and traces of a large scale data platform at Databricks, operating costs can be largely reduced by $3-4\times$. Additionally, we compare the cost savings across various caching architectures and show that cost can be further reduced by another 2× using linked in-memory caches which saves CPU in the application itself for data (de)serialization. Compared to provisioning the backend data store with more memory, deploying external in-memory caches brings surprisingly large amount of cost saving by saving CPU cores.

Second, we observe that caching benefits rich object workloads disproportionately. While the caching research community has focused on traditional key-value scenarios, rich object workloads are also prevalent in practice, and often emerge organically as application and storage schemas evolve over time. For example, in our experience at Databricks and prior large-scale deployments elsewhere, simple read paths gradually accumulate logic to join, filter, or verify various metadata. As a result, even ostensibly simple operations (e.g., retrieving metadata for a user-defined table) translate to multiple expensive storage queries. In a representative case—our internal data governance platform, Unity Catalog [13], which offers a unified namespace for data assets and employs an entity–relationship data model—serving a single rich object can require as many as eight SQL queries. Our key insight is that linked in-memory caches– embedded directly in the application–offer outsized benefits for these workloads, 2× greater cost savings than a denormalized, key-value-style variant of the same application. Rich object workloads exhibit a fundamentally different caching profile and requirements than traditional key-value scenarios. While the typical strategy today is for these applications to read directly from storage (expressing application logic in a sequence of SQL queries), given the massive cost benefits, these applications deserve dedicated architectural attention with better caches.

Third, we observe that emerging requirements for consistent caches can nullify most of the cost benefits of these distributed caches. In Databricks, driven by customer requirements, there is a need to scale services with low latency while preserving consistency (e.g. linearizability). Continuing with the cost study, we observe that even a minimal version check in storage for reads can significantly increase cost. Matching the cost advantages of an eventually consistent cache while maintaining strongly consistent read is challenging. We argue that this need not be the case, and cost-efficient strongly consistent distributed caches should be considered an important area of future work.

2 Background and motivation

Distributed caches are deployed to reduce latency in datacenter services, yet their contribution to monetary cost has been unclear. We observe that practitioners often associate adding caches with additional cost due to the added DRAM. This work begins with the simple question: *do distributed caches add or save cost, if so, how much?* (§2.1). We additionally take the cost study to two emerging scenarios: caching rich application objects (§2.2) and strongly consistent caches (§2.3), across several cache architectures (§2.4).

2.1 The cost of distributed caches

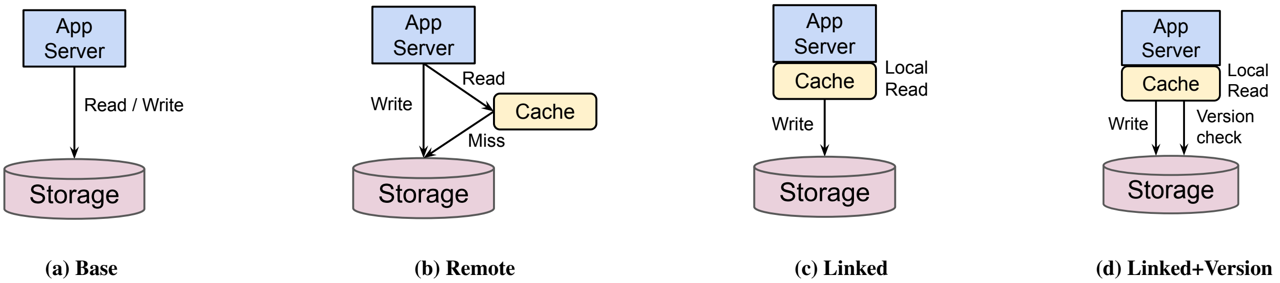
Caches are widely deployed to improve performance for datacenter services [2], as storage systems are typically slower in comparison. While it is known that caches reduce backend load, memory is expensive [34, 55] and constitutes a significant portion of datacenter cost [29, 33]. It is unclear whether in-memory caches might cost more than the load saved at the backend. Practitioners often assume that distributed caches improve latency at an additional cost [26, 30, 31], or they are unaware of the full impact distributed caches have on cost. Moreover, since caches can be deployed in various ways (§2.4), such as storage layer cache, remote cache, or application-linked cache, it remains unclear which deployment strategy has a greater cost impact and how this influences resource usage (e.g., compute and memory). Consequently, determining the optimal caching architecture for cost savings is a challenge.

While cost trade-offs have been studied in HPC *jobs*, where caching working set reduces job makespan, the datacenter services differ in two respects: multi-tenant services run continuously, and resource billing is decoupled across CPU, memory, and storage (compared to node-based allocation in HPC). Despite their prevalence, distributed caches and their cost impact on datacenter services have not been systematically studied and reported.

2.2 The need for caching rich objects

Caching literature has been largely focused on workloads for simple key and value pairs [41, 49, 50]. For instance, the Meta traces [1, 7] have a median value size of approximately 10 bytes with approximately 70% reads; the Twitter traces [49] have a median value size of 230 bytes and mixed read-write patterns. These traces reflect classic key–value access patterns—individual put, get, and delete operations on small items. While Redis supports caching data structures such as lists, it does not support graphs, nested structures, or any object with internal semantics. Although research prototypes have added richer semantics such as transactional consistency [38], production caches are still deliberately lightweight and single-operation–oriented.

However, rich application object workloads are prevalent in practice. These objects have internal application-specific semantics. A single request to retrieve an object (e.g. table metadata) gets translated to multiple operations, such as retrieving permission, dependencies, constraints, and data lineage. Each piece of this information is stored in separate structured tables in the backend, accessible via SQL. After retrieval, the application logic decides the return result to client



(a) Base **(b) Remote** **(c) Linked** **(d) Linked+Version**

Figure 1: Architecture comparisons. Different components (application server, cache, storage) are *services* that span multiple nodes.

based on the values retrieved. Often, these application logics cannot be easily expressed as simple computations and often involve semantics that do not fit neatly into basic data structures. These application objects can also be "parametized" based on each request. For example, a dashboard call such as "top-N user-relevant logs in the past T minutes" is parameterized and assembled on the fly by joining log rows, severity tags, and ownership metadata— which are difficult to express in fixed key-value blobs. These complex application logic can also subject to updates by developers. Since traditional caches cannot express application-level logic, the most common approach to supporting applications with rich objects is to express application requirements in a sequence of storage layer queries (e.g. SQL) that executes on every request and rely on applications to process and compose records. Unlike traditional key-value scenarios where values are small, the combined retrieved records tend to be large, on the order of hundreds of KBs or even MBs.

2.3 The need for consistent cache

While caches are traditionally used for services that tolerate eventual consistency (e.g. social media and web applications), many emerging distributed applications require *strong consistency*—particularly *linearizable* reads that always reflect the latest write. For instance, Unity Catalog, which centralizes access control and auditing user data, requires linearizable reads for correctness yet is also expected to offer low latencies for a real-time user experience. Likewise, a system in Databricks that lets customers schedule and execute SQL queries on elastic compute clusters, is tuned for fast responses but also requires strongly consistent session state as any inconsistency can yield incorrect query behavior. These requirements are not unique to Databricks: other large-scale companies such as Dropbox [44], Google [10], and Meta [18] have also reported similar requirements, and recent work has highlighted the need for strong consistency in latency-critical domains such as financial trading [9, 15] and online ad bidding [53]. However, for these scenarios, the common approach is to read directly from storage, bypassing the cache, as caches typically cannot provide data consistency.

2.4 Caching architectures

Storage layer caches *(Base) Figure 1a* We define storage layer caches as those which are natively managed by the storage system, such as row or block-level caches that are colocated with the storage nodes. They serve primarily to reduce latency and improve throughput, since serving records from memory is orders of magnitude faster than disk. While storage layer caches save applications from needing to manage copies of data, they can be more expensive to access since they incur up-front overheads for query processing in storage. Typically, only popular rows or records are cached in storage, since the storage is unaware of application semantics.

Remote caches *(Remote) Figure 1b* Remote distributed caches are used by applications to cache frequently accessed data such as popular query results, or database records. Examples include Memcached and Redis [7, 19, 41, 49], and are typically designed to offer relatively low latency read and write operations. Remote caches have the advantage of being shareable by multiple applications interested in the same data, and so can make more economical use of expensive memory resources. However, being remote means that applications add a network hop on the serving path and incur CPU cost for remote access and data (de)serialization.

Linked caches *(Linked) Figure 1c* Linked caches are similar to remote caches, but instead of remote to the application server, they are linked directly into the application as a library [2]. To avoid replicating the cache in every application server, linked caches are typically sharded [3] so that each server handles only a partition of the cache. Compared to remote caches, linked caches improve latency by eliminating remote data fetches, and also save CPU resources by avoiding data (un)marshalling overheads and the overreads problem [2]. However, since these caches are not shared, each application incurs additional memory costs for its own linked cache if the same data is requested across multiple applications.

Linked cache with version check *(Linked+Version) Figure 1d* For emerging applications that require consistent reads (§2.3), the typical approach is for users to read directly from storage. We describe and compare a simple baseline for maintaining data consistency: checking the data version upon each read. When a read request is issued, the application server concurrently issues a version check to the storage system (or a separate service [44]) and waits for its reply before returning the requested data. The version check contains only the key, therefore more lightweight than retrieving the full value. After the read verifies the matching data version, the cache read can return the cached data to the clinet.

3 Hypotheses

This section outlines four hypotheses on how caches affect the overall cost of datacenter services.

Caches save compute, offsetting DRAM costs. Since workloads are often skewed [49], adding extra memory can reduce CPU usage by minimizing communication between application servers, storage, and remote cache. CPU savings can be impactful because proto (de)serialization, which are computationally expensive [2], consume a substantial portion of CPU cycles in production clusters [27]. We hypothesize that reducing communication overhead can lower compute costs to potentially offset the expense of additional DRAM. To contextualize these costs, on GCP, 1 vCPU core costs approximately \$17 per month, while 1 GB of memory costs about \$2 per month [16]. Storage, required for persistence regardless, costs \$2 per 100 GB per month. While we initially expect that compute savings would merely offset the cost of DRAM, we were surprised to find that the compute savings were substantial enough to drive significant overall cost reductions (§5.3) for datacenter services.

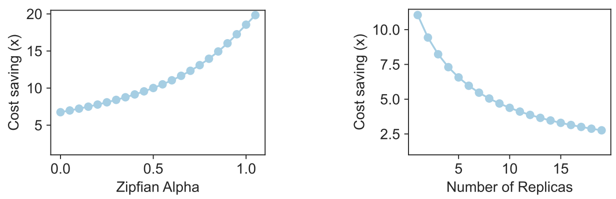
More distributed in-memory caches, less storage layer caches We observe that there has not yet been a cost study comparing different caching architectures (§2.4). From discussions with industry practitioners, a common practice is to provision more memory (storage layer cache) in the backend to improve performance. In Databricks, by default, backend storage is provisioned with tens of GBs of memory per instance for storage layer cache. We hypothesize that by provisioning more distributed caches external to storage, we can reduce the memory allocation for storage layer caches. This approach can allow for a more efficient memory allocation while potentially maintaining the same or even smaller total memory footprint to achieve a target performance.

Caching rich objects reduce query processing at storage For rich object workloads, applications often have to execute *multiple* read queries (e.g., SQL statements that retrieve rows from multiple tables) to storage, even if the queried data (e.g. rows) is already cached in storage. The overhead of such "query amplification" might lead to significant compute cost in query processing in the backend. We hypothesize that caching external to storage can alleviate this query processing, and the caches are best colocated with the application so that applications can directly operate the cached data in-memory.

Version check increases storage overhead To enable consistent cache, we originally expect version checks to be simple and incur minor cost in backend, which contains only the key queried. However, we find that might not hold true for modern distributed storage systems (§5.5), where even a version check upon read can incur query processing and RPC communication overhead, largely removing the cost benefit of caching.

4 Theoretical Model

We develop a simple theoretical model. We focus on linked caches but the approach can extend to remote caches.



(a) Zipfian alpha α

(b) Number of replicas N_r

Figure 2: The effect of varying the number of replicas N_r and Zipfian alpha α on cost saving. Comparing Linked (s_A = 8GB. s_D = 1GB) with Base (1GB of in-storage cache).

Let the cache size in the app server be s_A and the cache size in the storage system be s_D. A miss in the linked cache or the storage layer cache incurs a cost of c_A and c_D *per request*. To estimate c_A and c_D, we empirically measure the CPU cores used and divide that by the request rate. Let N_r be the number of replicas, and $MR(x)$ be the miss ratio with a cache size x, The total cost is: $T = QPS \cdot (MR(s_A) \cdot c_A + MR(s_A + s_D) \cdot c_D) + c_M \cdot (s_A \cdot N_r + s_D)$.

Takeaways We omit derivations for space constraints. We find that adding a unit of cache at the application leads to *larger* cost reduction compared to adding a unit of cache at the storage system, or $\left|\frac{\partial T}{\partial s_A}\right| > \left|\frac{\partial T}{\partial s_D}\right|$. This effect is more pronounced with higher workload skew (Figure 2a). This means we should provision more linked cache and less storage layer cache. The optimal cache allocation is to use as much linked cache as possible, up to where the marginal benefit of adding cache equals the marginal cost of a unit of memory, or when $\left|\frac{\partial T}{\partial s_A}\right| = 0$. Surprisingly, even when memory becomes expensive (up to 40× than the current cost) or caches have more replication (larger N_r, Figure 2b), adding caches still saves cost.

5 Evaluation

We validate our hypotheses (§3) and theoretical model (§4) by comparing designs on synthetic and production workloads. We first show that the CPU cores saved by adding distributed caches far outweigh the provisioned memory (§5.3). Next, we highlight that caching rich application objects brings outsized cost benefits (§5.4), up to 8× compared to directly reading from storage. Lastly, we find that to support consistent cache, even a minimal version check can remove most of the benefits (§5.5) due to the added overhead in backend storage. These observations point to future caching research directions in supporting these emerging workloads.

5.1 Experiment

Cache The remote cache and the application are implemented as gRPC services [46]. Each application server is provisioned with 6GB of memory for cache.

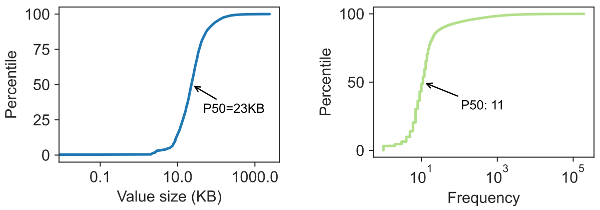
Database We used TiDB [24] as a representative distributed database, with 3 TiKV pods and 3 TiDB pods. Each TiKV pod is provisioned with 15GB of memory. These TiKV pods rely on *block caches* for caching popular rows [45].

Cost model Total cost is based on compute (vCPU cores) and memory (GB) used, multiplied by their unit costs on GCP [16]. Although cloud providers charge per-VM-minute, most production platforms today are provisioned according to peak CPU utilization. Lower steady-state CPU demand maps directly to smaller VM shapes or fewer replicas when auto-scalers trigger on compute utilization. While practical constraints like VM sizes and the need for overprovisioning can slightly skew results, we believe custom VM shapes, such as those supported by GCP [22], and advancements in resource disaggregation [4, 23] can help mitigate these issues.

5.2 Workloads

Synthetic workloads: We use 100K keys with Zipfian access pattern ($\alpha = 1.2$ [49]). We vary the read ratio r from 50% to 99%, and the value size from 1KB to 1MB.

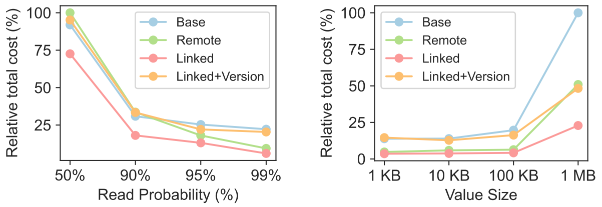
Open-sourced workloads: We use traces from Meta [1, 7], which have 30% writes with the median value ≈ 10 bytes. The trace contains reads and writes over key-value pairs.



(a) Value size **(b) Access Frequency**

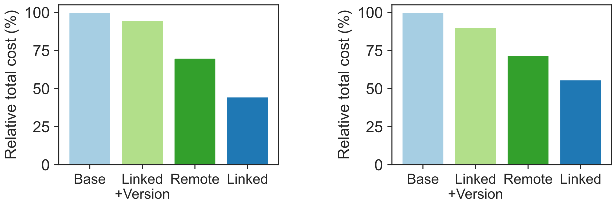
Figure 3: Analysis of Unity Catalog Trace distribution

Production workloads: We use production traces from an internal data governance service Unity Catalog [13] in Databricks (Figure 3). Unity Catalog models data using a hierarchical namespace. At the top is the metastore, which defines a three-level identifier— `catalog.schema.table`—and tracks metadata, ownership, and privileges. Within a metastore, catalogs group related domains (for example by business unit) and contain schemas; schemas then hold securable objects such as tables, views, and functions. Privileges are granted to principals (users, groups, service principals) on any level, and inherit downward. The workload is read-heavy (≈93%), handling approximately 40K complex queries per second (QPS). The median value size is ≈ 23KB with large values at the tail. `getTable` is the most common operation and incurs most cost, which retrieves data governance and access control information about a particular table. `getTable` translates to up to 8 SQL queries directed at multiple tables in the database—such as permissions, constraints, and privileges.



(a) Read Ratio **(b) Value Size**

Figure 4: Comparison of total cost for different architectures based on varying value size and read ratio.

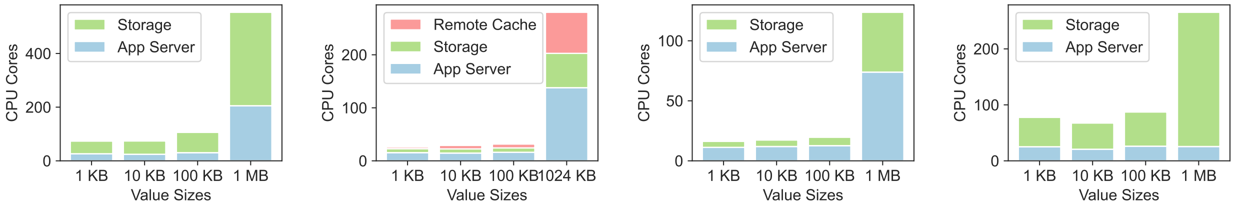


(a) Unity Catalog-KV **(b) Meta**

Figure 5: Cost comparison of Unity Catalog-KV and Meta.

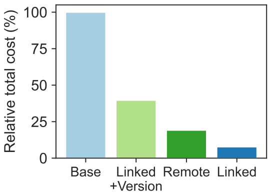
5.3 Compute saved by caches largely exceed the cost of added DRAM

We compare cost for synthetic workload in Figure 4. Compared to Base, Linked demonstrates significant cost savings, 7.3× for large values (1MB) and a 3.9× for small value sizes (1KB), since larger value incurs more (de)serialization. We observe significant cost savings of Remote and Linked for Unity Catalog-KV and Meta (Figure 5) as well. Remote has less cost saving than Linked, due to CPUs needed for gRPC communication between application and remote cache.



(a) Base **(b) Remote** **(c) Linked** **(d) Linked+Version**

Figure 6: Comparison of CPU usage at app server, remote cache and storage based on varying value size.



(a) Unity Catalog-Object - Cost

Figure 7: Running Unity Catalog where reading rich objects requires issuing multiple SQL statements, mimicking production.

In Figure 6, we present a breakdown of relative compute cost at the cache and the storage layer. As the value size increases, writes become more expensive and the cost *at the storage layer* required to serve these requests constitutes a larger portion of the total cost. Among the CPU cycles spent in the database, $40-65\%$ of compute is spent on managing connection, query processing, and execution planning. The rest is spent executing key-value lookups and communication (e.g. replication and executing queries), which accounts for $10-40\%$ of cycles. For Linked, preparing and issuing read or write requests to storage accounts for most CPU cycles at the application ($\approx 60\%$), followed by communication between client and the application server ($\approx 31\%$), and the remaining cycles are spent on application server servicing requests, including reading from the local cache. Across the runs, memory contributes $6-22\%$ of total cost for Linked, and $1-5\%$ for Base (due to higher compute cost). Therefore, compute saved by these caches largely exceed the cost of added DRAM.

5.4 Caching rich objects brings outsized cost savings.

We present two versions for Unity Catalog: (1) **Unity Catalog-Object** (Figure 7): each read request translates to multiple SQL statements to compose a rich application object (how Unity Catalog works in production). (2) **Unity Catalog-KV** (Figure 5): each read request is executed as a single key-value lookup, assuming a heavily denormalized schema which has all needed information (e.g. permission, constraints, privileges) in a single row. We note that Unity Catalog-KV is significantly simplified and only reflects the data accessing logic: actual applications often need to operate on intermediate results, such as conditionally accessing different sets of tables based on user permissions.

Comparing (2) to (1) shows that real-world applications benefit significantly more from caching due to handling large, complex application objects. Specifically, we observe larger savings in Unity Catalog-Object compared to Unity Catalog-KV as caching application objects directly saves the application server multiple SQL queries from querying the object, widening the cost benefits (by up to 2×). We expect the cost difference to be larger in practice when factored into more application logic in data retrieval decisions. By storing the fully materialized object in a linked cache, we eliminate query amplification entirely, thereby reducing CPU usage in storage.

5.5 Version checks removes most cost benefits

We observe that maintaining data consistency nullifies most cost benefits. Comparing Linked+Version (Figure 6d) with Linked (Figure 6c), we observe that version checks upon read significantly increases the storage load. Even a seemingly trivial version check—returning the row's 8-byte version column—still traverses the main read path: the SQL front-end parses and plans the query, TiDB's transaction layer validates Raft leases, and the request is sent over gRPC to TiKV, which fetches the full row and ships it back. The extra computation and communication erase most of the cache's cost savings. A key-value store could sidestep some of this overhead—version metadata can be read with a GET—but doing so forfeits SQL features such as joins, and transactional semantics, and still incurs a network round-trip and consensus validation (e.g. a read quorum) in replicated deployments. A RPC-bypass design (e.g., leveraging on-NIC atomics) could eliminate this overhead; investigating that is future work.

6 Open challenges and future directions

Extending cost benefits to consistent cache Many applications with strong consistency requirements rely on reading directly from storage, forfeiting the latency and cost benefits of distributed caches. Even minimal version checks significantly erode these benefits. We believe that future caches can do better — for example, by leveraging techniques built on top of auto-sharders that efficiently provide strong ownership over key ranges [3] to optimize away per-read version checks. Exploring this approach could unlock significant potential for consistent and cost-effective distributed caching.

In addition to addressing the inefficiencies of per-read version checks, another critical challenge arises: *delayed writes*. Even with strong ownership guarantees that help optimize consistency checks, delayed writes can lead to correctness issues that compromise consistency between the cache and storage. For instance, consider the following scenario: (1) an application sends a write to storage, but the write is delayed; (2) a different instance of the cache, perhaps due to resharding or node failure, reads the current value from storage; and (3) the delayed write eventually succeeds, leaving the cache and storage out of sync. Developing cost-effective mechanisms to ensure both consistency and correctness in such cases represents an exciting avenue for future exploration.

A dedicated architecture for caching application objects We demonstrated that applications with rich objects that are computationally expensive to reconstruct gain significantly more from caching than applications with smaller, simpler key-values. While most existing research has focused on the latter [34, 49, 50], we believe caching for rich objects represents a promising direction for future exploration, especially given its significant cost benefits. Large objects introduce unique challenges, such as partial eviction of objects, handling fine-grained updates while maintaining snapshot consistency, and efficiently managing memory and state during resharding or replication. Handling the internal semantics of application objects is challenging, and we believe that caching these objects is best done with linked caches due to cost benefits and that applications can directly operate on them. Addressing these challenges to realize these cost benefits requires caches dedicated for rich application objects.

7 Related work

Distributed caches and cost Kangaroo [34] considers caching tiny objects on flash. This paper targets DRAM caches and shows that they, too, can offer significant cost savings; SSD caches may further improve cost. [2] argues that linked caches can avoid (de)serialization. However, none of the prior work investigates whether in-memory caches reduce the *total* operating cost across various architectures and workloads.

Maintaining data consistency TTL has been widely used [6–8, 11, 12, 20, 21, 25, 43, 49, 50] and studied for in-memory caches. Cache invalidation has been explored [18, 28, 32, 36, 52, 54]. However, to the best of our knowledge, none of the prior work explores consistent caches. We believe that extending *cost benefits* to caches with stronger consistency guarantees presents a new class of research challenges.

8 Conclusions

This work represents the first step in quantifying the impact of adding distributed in-memory caches on cost across a variety of caching architectures for datacenter services. We discuss several emerging scenarios—caching rich application objects and consistent caches—and believe these cost observations point to significant opportunities for future research.

References

[1] 2024. Evaluating SSD hardware for Facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/cachebench-fb-hw-eval. Accessed: 2024-06-22.

[2] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. 2019. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 113–119.

[3] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, et al. 2016. Slicer:{Auto-Sharding} for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 739–753.

[4] Marcos K Aguilera, Emmanuel Amaro, Nadav Amit, Erika Hunhoff, Anil Yelam, and Gerd Zellweger. 2023. Memory disaggregation: Why now and what are the challenges. *ACM SIGOPS Operating Systems Review* 57, 1 (2023), 38–46.

[5] Microsoft Azure. 2025. Azure Cache Pricing - Microsoft Azure. https://azure.microsoft.com/en-us/pricing/details/cache/ Accessed: 2025-01-12.

[6] Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, and Ramesh Sitaraman. 2018. Adaptive TTL-based caching for content delivery. *IEEE/ACM transactions on networking* 26, 3 (2018), 1063–1077.

[7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The {CacheLib} caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 753–768.

[8] Daniel S Berger, Philipp Gland, Sahil Singla, and Florin Ciucu. 2014. Exact analysis of TTL cache networks. *Performance Evaluation* 79 (2014), 2–23.

[9] Laura Bright and Louiqa Raschid. 2002. Using latency-recency profiles for data delivery on the web. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier, 550–561.

[10] Mike Burrows. 2006. The Chubby lock service for loosely-coupled distributed systems. In *Proceedings of the 7th symposium on Operating systems design and implementation*. 335–350.

[11] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. 2019. TTL-based cloud caches. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 685–693.

[12] Damiano Carra, Giovanni Neglia, and Pietro Michiardi. 2020. Elastic provisioning of cloud caches: A cost-aware TTL approach. *IEEE/ACM Transactions on Networking* 28, 3 (2020), 1283–1296.

[13] Ramesh Chandra, Haogang Chen, Ray Matharu, Sarah Cai, Jeff Chen, Priyam Dutta, Bogdan Ghita, Todd Greenstein, Gopal Holla, Peng Huang, et al. 2025. Unity Catalog: Open and Universal Governance for the Lakehouse and Beyond. In *Companion of the 2025 International Conference on Management of Data*. 310–322.

[14] Yue Cheng, Aayush Gupta, and Ali R Butt. 2015. An in-memory object caching framework with adaptive load balancing. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–16.

[15] James Cipar. 2014. *Trading freshness for performance in distributed systems*. Technical Report. Citeseer.

[16] Google Cloud. 2025. Google Cloud Compute Engine Pricing. https://cloud.google.com/compute/vm-instance-pricing Accessed: 2025-01-06.

[17] Oracle Corporation. 2023. Introducing Oracle True Cache. https://blogs.oracle.com/database/post/introducing-oracle-true-cache

[18] Facebook Engineering. 2022. Cache Made Consistent: Improving Cache Efficiency and Data Freshness. *Facebook Engineering Blog* (June 2022). https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/

[19] Brad Fitzpatrick. 2004. Distributed caching with memcached. *Linux journal* 2004, 124 (2004), 5.

[20] Nicaise Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. 2014. Performance evaluation of hierarchical TTL-based cache networks. *Computer Networks* 65 (2014), 212–231.

[21] Alexander Fuerst and Prateek Sharma. 2021. FaasCache: keeping serverless computing alive with greedy-dual caching. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 386–400.

[22] Google Cloud. n.d.. Create a VM with a Custom Machine Type. https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type. Accessed: 2025-01-14.

[23] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G Shin. 2017. Efficient memory disaggregation with infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. 649–667.

[24] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, et al. 2020. TiDB: a Raft-based HTAP database. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3072–3084.

[25] Jaeyeon Jung, Arthur W Berger, and Hari Balakrishnan. 2003. Modeling TTL-based Internet caches. In *IEEE INFOCOM 2003. Twenty-second Annual Joint Conference of the IEEE Computer and Communications Societies (IEEE Cat. No. 03CH37428)*, Vol. 1. IEEE, 417–426.

[26] Farhana Kabir and David Chiu. 2012. Reconciling cost and performance objectives for elastic web caches. In *2012 International Conference on Cloud and Service Computing*. IEEE, 88–95.

[27] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *Proceedings of the 42nd annual international symposium on computer architecture*. 158–169.

[28] Alexandros Labrinidis and Nick Roussopoulos. 2003. Balancing performance and data freshness in web database servers. In *Proceedings 2003 VLDB Conference*. Elsevier, 393–404.

[29] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[30] David Lomet. 2018. Cost/performance in modern data stores: How data caching systems succeed. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*. 1–10.

[31] Vigneshwaran Manivelmurugan. 2025. Designing Resilient Systems: A Guide to Distributed Caching for Modern Applications. *International Journal of Engineering Research & Technology (IJERT)* 14, 1 (January 2025).

[32] Ziming Mao, Rishabh Iyer, Scott Shenker, and Ion Stoica. 2024. Revisiting Cache Freshness for Emerging Real-Time Applications. In *Proceedings of the 23rd ACM Workshop on Hot Topics in Networks* (Irvine, CA, USA) *(HotNets '24)*. Association for Computing Machinery, New York, NY, USA, 335–342. https://doi.org/10.1145/3696348.3696858

[33] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. Tpp: Transparent page placement for cxl-enabled tiered-memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 742–755.

[34] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S Berger, Nathan Beckmann, and Gregory R Ganger. 2021. Kangaroo: Caching billions of tiny objects on flash. In *Proceedings of the ACM SIGOPS 28th symposium on operating systems principles*. 243–262.

[35] M. Mead. n.d.. RAM, Hard Drive, and SSD Prices Over Time. https://azrael.digipen.edu/~mmead/www/Courses/CS180/ram-hd-ssd-prices.html. Accessed: 2025-01-04.

[36] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. 385–398.

[37] Percona. 2023. MySQL Caching Methods and Tips. https://www.percona.com/blog/mysql-caching-methods-and-tips/

[38] Dan RK Ports, Irene Zhang, Samuel Madden, and Barbara Liskov. 2010. Transactional consistency and automatic management in an application data cache. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*.

[39] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*. 557–573.

[40] RavenDB. 2023. Caching Data: Automatic Database Caching. https://ravendb.net/articles/caching-data-automatic-database-caching

[41] Salvatore Sanfilippo and Redis Core Team. 2024. Redis. https://redis.io. Accessed: 2024-09-20.

[42] Amazon Web Services. 2023. Database Caching Strategies Using Redis. https://aws.amazon.com/caching/database-caching/

[43] Sari Sultan, Kia Shakiba, Albert Lee, Paul Chen, and Michael Stumm. 2024. TTLs matter: Efficient cache sizing with TTL-aware miss ratio curves and working set sizes. In *Proceedings of the Nineteenth European Conference on Computer Systems*. 387–404.

[44] Dropbox Tech Team. 2023. Meet Chrono: Our Scalable, Consistent Metadata Caching Solution. Dropbox Tech Blog. https://dropbox.tech/infrastructure/meet-chrono-our-scalable-consistent-metadata-caching-solution Accessed: 2025-03-19.

[45] PingCAP Documentation Team. n.d.. Tune TiKV Memory Performance. https://docs.pingcap.com/tidb/stable/tune-tikv-memory-performance. Accessed: 2025-01-04.

[46] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. 1993. GRPC: A communication cooperation mechanism in distributed systems. *ACM SIGOPS Operating Systems Review* 27, 3 (1993), 75–86.

[47] Wikipedia. 2023. List of In-Memory Databases. https://en.wikipedia.org/wiki/List_of_in-memory_databases

[48] Juncheng Yang, Ziming Mao, Yao Yue, and KV Rashmi. 2023. {GL-Cache}: Group-level learning for efficient and high-performance caching. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 115–134.

[49] Juncheng Yang, Yao Yue, and KV Rashmi. 2021. A large-scale analysis of hundreds of in-memory key-value cache clusters at twitter. *ACM Transactions on Storage (TOS)* 17, 3 (2021), 1–35.

[50] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design

and Implementation (NSDI 21). 503–518.

[51] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 130–149.

[52] Haobo Yu, Lee Breslau, and Scott Shenker. 1999. A scalable web cache consistency architecture. *ACM SIGCOMM Computer Communication Review* 29, 4 (1999), 163–174.

[53] Shuai Yuan, Jun Wang, and Xiaoxue Zhao. 2013. Real-time bidding for online advertising: measurement and analysis. In *Proceedings of the seventh international workshop on data mining for online advertising*. 1–8.

[54] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. 2024. {MuCache}: A General Framework for Caching in Microservice Graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. 221–238.

[55] Qizhen Zhang, Philip A Bernstein, Daniel S Berger, Badrish Chandramouli, Vincent Liu, and Boon Thau Loo. 2022. Compucache: Remote computable caching using spot vms. In *Annual Conference on Innovative Data Systems Research (CIDR'22)*.