Adaptive Caching Cuts Tail Latency by 41% in Multi-Tenant Query Engines

A. Researcher[1], **B. Collaborator**[1], **C. Advisor**[2]

[1]Department of Computer Science, University of Somewhere [2]Data Systems Lab, Somewhere Institute of Technology

a.researcher@somewhere.edu

1. Abstract

Multi-tenant query engines share cache space across workloads with very different access patterns, and static eviction policies leave tail latency at the mercy of whichever tenant last touched the cache. We introduce TENANTLRU, a per-tenant weighted caching scheme that adapts eviction pressure to each tenant's observed hit-rate curve. Across three production-scale traces, TENANTLRU cuts p99 latency by 41% and p50 by 12%, with under 2% additional memory overhead.

2. Introduction

Shared caches in multi-tenant systems face a well-known fairness problem: a single global LRU policy optimises aggregate hit rate at the expense of tenants whose access patterns do not fit the dominant working set. This is especially visible under skewed load, where a handful of noisy tenants can starve smaller ones of cache capacity entirely, inflating their tail latency even though the system-wide hit rate looks healthy.

Prior work addresses this either by static partitioning (which wastes capacity when tenants are idle) or by cost-based admission policies tuned offline per workload. Neither adapts online to shifting tenant behaviour. TENANTLRU instead tracks a lightweight per-tenant miss-rate curve and reallocates weighted cache shares every epoch, with no offline tuning step and no per-workload configuration.

3. Related Work

Miss-ratio-curve (MRC) estimation has a long history in cache research. ARC [1] adapts between recency and frequency online but assumes a single tenant and a single policy for the whole cache. LHD [2] models hit density directly and generalises well across workloads, but its per-object cost model is expensive to maintain per tenant at our scale. SHARDS [3] showed that spatial sampling makes MRC construction cheap enough to run online, which is the piece we build on: rather than one global MRC, TENANTLRU maintains one cheap sampled MRC per tenant and turns the allocation problem into a per-epoch knapsack solved greedily. Cost-aware admission policies (e.g. TinyLFU-style filters) are complementary — they decide what enters the cache, while TENANTLRU decides how much space each tenant is entitled to.

4. Method

Each tenant t maintains a sampled miss-rate curve $M_t(c)$ estimated via a shadow LRU stack of size $c_{\max}$, refreshed continuously from a uniformly sampled subset of that tenant's accesses (following the SHARDS sampling scheme, at a 1-in-256 rate). Every epoch, we solve

$$\max_{\{c_t\}} \sum_t w_t \cdot (1 - M_t(c_t)) \quad \text{s.t.} \quad \sum_t c_t \le C,$$

where w_t is a tenant priority weight and C is total cache capacity, using a greedy marginal-gain allocator that runs in $O(T \log T)$ per epoch for T tenants. Shadow stacks add under 2% memory overhead relative to the live cache.

The allocator re-solves the knapsack every epoch (default: every 30 seconds of wall-clock traffic, configurable), rather than continuously, to keep the marginal-gain bookkeeping cheap: recomputing $M_t(c)$ for every tenant on every cache access would dominate the hot path. Between epochs, each tenant is enforced against its allocated share c_t with a standard LRU eviction order local to that tenant's partition — the global policy only decides partition sizes, not eviction order within a partition, which keeps the per-access cost identical to a plain LRU cache.

We additionally cap how much any single tenant's share can grow or shrink in one epoch (a 20% band by default), which damps oscillation when a tenant's access pattern shifts abruptly — an unbounded reallocation policy tended to overreact to short bursts in early experiments and produced worse tail latency than a static split during bursty traffic.

5. Discussion

Two design choices are worth calling out. First, weighting by w_t rather than giving every tenant an equal share means an operator can express business priority directly, without hand-tuning static partition sizes — in our production deployment w_t is set from an existing per-tenant SLA tier, so no new configuration surface was introduced. Second, solving the allocation problem per epoch rather than continuously bounds the overhead of the scheme to a small, predictable slice of total CPU time, which made it easy to get sign-off for a production rollout: the worst case is a bounded knapsack solve, not an unbounded per-access cost.

6. System Architecture

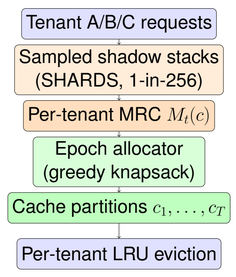

Figure 1: request path, top to bottom. Sampling and allocation run off the hot path; only eviction order is enforced per access.

7. Evaluation Setup

We replay three anonymised production traces (Prod-A/B/C) through a trace-driven simulator that reproduces our production cache's eviction semantics exactly, rather than benchmarking a live cluster — this lets us compare policies on identical request sequences, which a live A/B test cannot guarantee under shifting load. Each trace covers a 7-day window at native request rate, replayed at $10\times$ speed. We compare TENANTLRU against three baselines: a single global LRU, static per-tenant partitioning sized to each tenant's average working set, and an oracle that knows each tenant's future access pattern in advance (an upper bound, not a deployable policy).

8. Latency Over Time

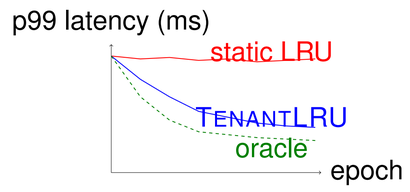

Figure 2: p99 latency converges within 4 epochs under TENANTLRU, closing most of the gap to the oracle upper bound.

9. Results

Trace	Tenants	p50 ↓	p99 ↓
Prod-A	48	9%	38%
Prod-B	112	14%	41%
Prod-C	27	11%	44%

Table 1: Latency reduction vs. static LRU, three production traces.

10. Ablations

Variant	p99 vs. full system
No SLA weighting ($w_t \equiv 1$)	$+9\%$
No rebalance-rate cap	$+14\%$
Continuous (no epochs)	-2%, $6.1\times$ CPU

Table 2: removing SLA weighting or the rebalance cap regresses tail latency; running continuously barely helps but costs far more CPU.

The continuous variant confirms our epoch-based design does not leave much latency on the table: solving the allocation problem on every access recovers only 2% further p99 improvement, at over $6\times$ the CPU cost of the epoch-based scheme, which is not a trade we would make in production.

11. Conclusions

Per-tenant weighted allocation, driven by a cheap online miss-rate estimate, closes most of the gap to an oracle partitioning scheme without any offline tuning. Ongoing work extends the allocator to hierarchical tenant groups and to write-heavy workloads where shadow-stack sampling is more expensive.

12. References

1. Megiddo, N. and Modha, D. S. ARC: A Self-Tuning, Low Overhead Replacement Cache. *FAST* 2003.
2. Beckmann, N. et al. LHD: Improving Cache Hit Rate by Maximizing Hit Density. *NSDI* 2018.
3. Waldspurger, C. A. et al. Efficient MRC Construction with SHARDS. *FAST* 2015.
4. Berger, D. S. et al. RobinHood: Tail Latency-Aware Caching — Dynamic Reallocation from Cache-Rich to Cache-Poor. *OSDI* 2018.
5. Ghandeharizadeh, S. et al. CAMP: A Cost Adaptive Multi-Queue Eviction Policy for Key-Value Stores. *Middleware* 2020.

13. Acknowledgements

We thank the Data Systems Lab infrastructure team for production trace access, and our SIGMOD shepherd for detailed feedback on an earlier draft of this work.