

WHITE PAPER · DATABASE CACHING

Database Caching Fundamentals

Strategies, architectures, eviction and pitfalls

White Paper 1 · Database Caching series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Database Caching Fundamentals: Strategies, Architectures, Eviction and Pitfalls

White Paper 1 · Database Caching series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Database Caching, paper 1 of 6
Author	Bassirah Technology — Innovation & R&D
Version	1.0 · 26 October 2026
Scope	A technology survey. It describes published research and publicly documented systems; it does not report Bassirah benchmarks or customer results.

Contents

1. Abstract	4
2. Introduction	4
2.1 Why databases are cached	4
2.2 When not to cache	4
2.3 Scope	4
2.4 Terms used in this paper	5
3. Core caching strategies	5
3.1 Read-based strategies	5
3.2 Write-based strategies	6
3.3 Combining strategies	6
4. Cache architecture types	7
4.1 In-process caches	7
4.2 Distributed caches	7
4.3 Near caches and client-side caching	8
4.4 Caches inside the database	8
4.5 HTTP and CDN caching for data APIs	8
4.6 Multi-tier compositions	8
5. Eviction and admission policies	9
5.1 The optimal policy and miss-ratio curves	9
5.2 Recency and frequency: LRU, LFU and CLOCK	9
5.3 Adaptive and scan-resistant policies	9
5.4 Admission: TinyLFU and W-TinyLFU	9
5.5 FIFO-based designs	10
5.6 Approximations in production caches	10
6. Invalidation and consistency	10

6.1 Expiration	10
6.2 Explicit invalidation on write and its races	11
6.3 Leases	11
6.4 Change data capture	11
6.5 Versioned keys	11
6.6 Measuring consistency	11
7. Common pitfalls and mitigations	12
7.1 Stampede	12
7.2 Penetration.....	12
7.3 Avalanche.....	13
7.4 Hot keys	13
7.5 Stale reads and races.....	13
7.6 Memory pressure and eviction storms.....	13
7.7 Cold start and cache dependence	13
7.8 Observability gaps	14
8. Comparative analysis.....	14
9. Open challenges and future opportunities.....	15
9.1 Policies that adapt to the workload	15
9.2 Caching by meaning	15
9.3 Memory and storage tiers	15
9.4 Precise invalidation	15
9.5 Hierarchies at the edge	15
9.6 Correctness and resilience under failure	16
10. What this means for platform decisions	16
References	16

1. Abstract

A cache keeps a copy of data closer to the code that reads it, so that repeated reads avoid the cost of going back to the database. That copy is also a second version of the truth, and most caching problems come from the gap between the two. This paper is the foundation of the Database Caching series. It describes the read-based strategies (cache-aside, read-through and refresh-ahead) and the write-based strategies (write-through, write-behind and write-around), and the consistency and failure behavior each implies. It then compares the main cache architecture types: in-process caches, distributed caches, near caches with server-assisted invalidation, the caches inside the database itself, and HTTP and CDN caches in front of data APIs. It surveys eviction and admission policies, from the optimal offline policy and classic recency and frequency policies to adaptive designs and recent FIFO-based work, and the invalidation mechanisms that keep cached data fresh: expiration, explicit invalidation, leases, change data capture and versioned keys. It closes with the common pitfalls and their mitigations, a qualitative comparison, open challenges that introduce papers 2 to 6, and notes for platform decisions.

2. Introduction

2.1 Why databases are cached

Databases are built to store data durably and answer arbitrary queries correctly. Many applications, however, read the same small set of records, pages or query results far more often than they change. Serving those repeated reads from memory placed near the application reduces response time, absorbs load that would otherwise reach the database, and can reduce the cost of database capacity provisioned only for peak reads. Large production deployments rely on this pattern. Facebook described a demand-filled, look-aside memcached tier in front of MySQL [1], and later a graph-aware caching layer, TAO, for its social graph [2]. Meta's CacheLib consolidated many specialized caches into one general-purpose caching engine [3], and Google described CliqueMap, a distributed caching system built on remote memory access [4]. A study of hundreds of production in-memory cache clusters at Twitter found many of them more write-heavy than earlier studies had suggested, and found time to live to be an important parameter of their working sets [5].

2.2 When not to cache

A cache is a trade: lower read cost in exchange for a second copy that can be stale, lost or inconsistent. It earns its place when reads repeat, when the data tolerates some staleness or can be invalidated reliably, and when the cost of a miss is high relative to a hit. It is a poor fit for data that changes on almost every read, for workloads with little reuse, for data whose correctness rules forbid any staleness unless invalidation is precise and tested, and for systems where the cache would mask a database that is undersized or missing indexes. A cache also adds a component that can fail, a dependency the system may come to rely on (Section 7.7), and a source of bugs that appear only under concurrency. Fixing the query or the schema is often the better first step.

2.3 Scope

This paper covers caching of database data at the application, service and database layers: strategies, architecture types, eviction and admission, invalidation and the operational pitfalls.

It concentrates on peer-reviewed work and official documentation, with recent work from 2021 to 2026 where it exists, and cites earlier work where it introduced an idea still in use. Statements about products reflect documentation accessed on 26 October 2026. It does not report benchmark figures, hit ratios, latency figures or cost figures. Where a paper reports results, we describe them qualitatively and attribute them to the authors. Papers 2 to 6 of the series examine learned replacement policies, semantic caching for LLM applications, tiered caching across memory and flash, automated invalidation, and multi-level caching at the edge.

2.4 Terms used in this paper

A **hit** is a read served from the cache and a **miss** is one that is not. The **hit ratio** is the share of reads that hit; the **miss ratio** is its complement, and it matters more to the database behind the cache, because every miss becomes database load. **Eviction** removes an entry to make room; **expiration** removes it because its time to live (TTL) has passed; **invalidation** removes or replaces it because the source data changed. **Admission** decides whether a new entry is stored at all. An entry is **stale** when it no longer matches the source of truth. The **backing store** is the database or service the cache fronts.

3. Core caching strategies

Caching strategies answer two questions: who loads data into the cache on a read, and what happens to the cache on a write. The answers are largely independent, so systems combine one read strategy with one write strategy (Section 3.3).

3.1 Read-based strategies

Cache-aside (lazy loading). The application owns the logic. On a read, it looks in the cache; on a miss, it reads the database, returns the result and stores a copy in the cache, usually with a TTL. The cache knows nothing about the database. Microsoft's Azure Architecture Center [6] and AWS's ElastiCache guidance [7] describe the pattern and its trade-offs, and Facebook's memcached deployment is a large-scale example of it [1]. Its strengths are simplicity, resilience (if the cache is unavailable, reads fall back to the database) and a cache that holds only data that has been requested. Its weaknesses are that the first read of each item always misses, that every application path must implement the logic consistently, and that a race between a read that repopulates the cache and a concurrent write can leave a stale value in place (Section 6.2).

Read-through. The cache itself loads missing data, through a loader the application registers. The application asks only the cache. In-process libraries such as Caffeine register the loader with the cache [8], and data grids such as Oracle Coherence load missing entries through a cache store [9]. Read-through centralizes loading logic and makes it easier to coalesce concurrent misses for the same key (Section 7.1), but it couples the cache to the data model and makes the cache a required component on the read path.

Refresh-ahead. The cache reloads an entry before it expires, so that frequently read entries are replaced without a reader waiting for the load. Caffeine's refresh-after-write policy reloads an entry asynchronously when it is read after a configured interval, while continuing to serve the old value until the new one is ready [8]; Coherence describes a refresh-ahead factor relative to the expiration time [9]. HTTP caching offers a related mechanism, stale-while-

revalidate, which lets a cache serve a stale response while it revalidates in the background [10], and reverse proxies such as NGINX can update expired entries in the background while serving the stale copy [11]. Refresh-ahead trades extra loads, some of which are never read, for fewer readers that wait on misses. It works best for a stable set of popular keys.

3.2 Write-based strategies

Write-through. Each write goes to the cache and to the backing store as part of one operation, and the write completes only when the store has accepted it. Data grids implement this through a store interface the cache calls synchronously, so that a write returns only after the data source has accepted it [9], and AWS describes an application-level variant in which the application updates the cache whenever it writes the database [7]. The cache stays populated with recently written data, which helps read-after-write workloads, but each write pays for both systems, the cache fills with data that may never be read [7], and the two updates are not atomic unless the platform makes them so: if one succeeds and the other fails, they diverge.

Write-behind (write-back). The cache acknowledges a write as soon as it holds it and writes to the backing store later, often in batches and with repeated updates to the same key coalesced. Coherence documents a write-behind queue that writes after a configured delay, coalesces repeated updates and writes in batches [9]. Write-behind absorbs write bursts and reduces load on the store, but it moves the point of durability into the cache, which Coherence describes as the cache temporarily becoming the system of record [9]: writes that have been acknowledged but not yet persisted can be lost if cache nodes fail, the store may see writes in a different order or combined, and other readers of the store see changes late. It suits data where a bounded loss window is acceptable or where the cache itself is replicated and persisted, and it requires the store to accept idempotent, retried writes.

Write-around. Writes go directly to the backing store and bypass the cache; the cache is populated only by reads. Combined with cache-aside and invalidation of the cached copy on write, this is a widely used pattern for database caching. It avoids filling the cache with data that is written but rarely read, at the cost of a miss on the first read after each write.

A related decision is whether a write **updates** the cached value or **deletes** it. Facebook chose deletion because deletes are idempotent [1], and the Azure guidance notes that the database should be updated before the cached item is removed [6]. Deletion makes the next read pay for a miss; updating keeps the cache warm but needs ordering, for example version numbers (Section 6.5).

3.3 Combining strategies

Table 1 summarizes the common combinations. The descriptions are qualitative; actual behavior depends on the platform, its configuration and the workload.

Combination	Who loads on a miss	What a write does	Consistency behavior	Failure behavior	Typical fit
Cache-aside with write-around and invalidate-on-write	Application	Writes the database, then deletes the cached entry	Stale until invalidation or TTL; races possible between read-fill and write	Reads fall back to the database if the cache is unavailable	General-purpose read-heavy services

Combination	Who loads on a miss	What a write does	Consistency behavior	Failure behavior	Typical fit
Read-through with write-through	Cache, through a loader	Writes cache and store synchronously	Cache reflects writes made through it; writes that bypass it are not seen	Writes fail or diverge if either system fails	Read-after-write workloads on one data platform
Read-through with write-behind	Cache, through a loader	Writes the cache; the store is updated asynchronously	Cache is the most current copy; the store lags	Acknowledged writes can be lost before they persist	Write-burst absorption where a loss window is accepted
Refresh-ahead added to any of the above	Cache, before expiry	Unchanged	Popular entries are reloaded on a schedule, bounding staleness by the refresh interval	Old values continue to be served if reloads fail	Stable sets of popular, expensive-to-compute entries

4. Cache architecture types

Where the cache runs determines its latency, how much memory it can hold, who shares it, and how invalidation reaches it.

4.1 In-process caches

An in-process cache lives in the application's own memory, so a hit costs a memory lookup with no network round trip or serialization. Libraries such as Caffeine for the JVM provide bounded caches with eviction, expiration and refresh [12], [8]. The limits are size (bounded by the process heap), duplication (each instance holds its own copy) and coherence: when data changes, every instance must be told, or each instance's copy is stale until it expires. In-process caches suit small, hot, rarely changing data, such as configuration and reference tables, and serve as the first tier in front of a shared cache.

4.2 Distributed caches

A distributed cache is a separate service that many application instances share over the network. Memcached is a key-value cache, deployed at large scale in front of MySQL at Facebook [1] and, with its derivatives, at Twitter [5]. Redis is an in-memory data store with key expiration [13], a choice of eviction policies [14] and server-assisted client-side caching [15]; it is available under several licenses, including AGPLv3 from Redis 8 [16]. Valkey is a Linux Foundation project, launched in 2024, that continues development from Redis 7.2.4 under the BSD license [17]. Clients usually spread keys across nodes by hashing, and each key lives on one primary node with optional replicas. A shared cache serves every instance from one copy, so invalidation needs to reach only one place, and it can be sized independently of the application. The costs are a network round trip per access, serialization, and a separate system to operate, scale and secure.

4.3 Near caches and client-side caching

A near cache combines the two: each application instance keeps a small local copy of entries it reads from the shared cache. Its difficulty is coherence, which some platforms address with server-assisted invalidation. In Redis client-side caching, the server tracks which keys a client has read and sends an invalidation message when one of them changes; a broadcasting mode instead notifies clients about every change to keys that match prefixes they subscribe to, which costs the server less memory but sends more messages [15].

Invalidation messages are asynchronous, so a near cache can serve a stale value for a short window after a write. The Redis documentation describes the race between a read reply and an invalidation, advises flushing the local cache when the invalidation connection is lost, and recommends a maximum TTL on every locally cached key [15].

4.4 Caches inside the database

Every database already caches. PostgreSQL keeps recently used pages in shared buffers and relies on the operating system's page cache as a second layer [18]; MySQL's InnoDB buffer pool uses a variant of LRU that inserts new pages at a midpoint, so that a large scan does not flush the pages that frequent queries need [19]. These caches are consistent by construction, because the database manages them under its own concurrency control. They cache pages, not query results, so a repeated query still pays for parsing, planning and execution. Caching query results is harder, because a write to one table can invalidate many cached results. Materialized views store a query's result as a table; in PostgreSQL they cannot be updated directly and are brought up to date with an explicit refresh [20], which makes them a cache with a refresh schedule. Paper 5 discusses incremental view maintenance as a more precise alternative.

4.5 HTTP and CDN caching for data APIs

When data is served through HTTP APIs, standard HTTP caching applies. RFC 9111 defines how responses declare their freshness lifetime and how caches validate stored responses with conditional requests [21]; RFC 5861 adds stale-while-revalidate and stale-if-error [10]. Content delivery networks apply these rules at many locations close to users. HTTP caching works well for public, cacheable responses, and poorly for responses that vary per user or must be invalidated precisely, because purging content from many edge locations is itself an asynchronous, distributed operation. Paper 6 examines multi-level caching at the edge.

4.6 Multi-tier compositions

Production systems often stack these types: a small in-process tier for the hottest keys, a shared distributed tier, the database's own buffer cache, and an HTTP tier in front. Each tier reduces load on the one behind it, and each adds an invalidation path. Facebook's deployment, for example, combined regional memcached pools, a separate pool to absorb load when servers failed, and invalidation driven by the database's commit log [1]. Tiering also changes what each tier sees: a lower tier receives only the misses of the tier above it, and that filtered stream can favor different eviction policies than the original traffic. Paper 4 examines tiers that span DRAM and flash.

5. Eviction and admission policies

A cache has finite capacity, so a full cache must decide which entry to remove and whether a new entry is worth storing. The right policy depends on the access pattern, and recent research has revisited several long-held assumptions.

5.1 The optimal policy and miss-ratio curves

Belady showed that the optimal offline policy evicts the entry whose next use is furthest in the future [22]. It needs knowledge of future requests, so no online cache can implement it, but it remains the reference against which policies are measured and the target that learned policies approximate (paper 2). A miss-ratio curve describes how a workload's miss ratio changes with cache size. Mattson et al. showed that, for a class of policies including LRU, the curve for every size can be computed in one pass over a trace [23], and SHARDS showed that spatial sampling of keys can approximate such curves at low cost [24]. Stack-distance methods apply only to policies with this inclusion property; Kosmo generates curves online for several policies without it, such as LFU, FIFO and 2Q [25]. Miss-ratio curves turn sizing into an evidence-based decision: they show where adding memory stops paying back.

5.2 Recency and frequency: LRU, LFU and CLOCK

Least-recently-used (LRU) eviction assumes that recently read entries will be read again soon. It is simple and effective for many workloads, but a single scan of many entries that will not be reused can push out the entries that matter. Least-frequently-used (LFU) eviction keeps entries that are read often, which resists scans but adapts slowly when popularity shifts, unless counts decay over time. CLOCK approximates LRU with a reference bit per entry and a circular scan, avoiding the cost of reordering a list on every hit. LRU-K evicts based on the time of the K-th most recent reference, separating occasional from frequent use [26].

5.3 Adaptive and scan-resistant policies

A family of policies combines recency and frequency and adapts to the workload. 2Q admits new entries to a probationary queue and promotes only those read again [27]. LIRS ranks entries by inter-reference recency, the number of distinct entries accessed between two uses of the same entry [28]. ARC keeps two LRU lists, for entries seen once and entries seen more than once, together with "ghost" lists of recently evicted keys, and continuously shifts capacity between the lists based on which ghost list receives hits [29]. CLOCK-Pro applies the LIRS idea within a CLOCK structure [30]. These policies resist scans and adapt without manual tuning, at the cost of more metadata per entry.

5.4 Admission: TinyLFU and W-TinyLFU

Eviction decides what leaves; admission decides what enters. TinyLFU estimates the access frequency of every key, including keys not in the cache, with a compact approximate counting sketch whose counts are periodically halved so that old popularity fades. A new entry is admitted only if it is estimated to be more frequent than the entry it would replace [31]. W-TinyLFU places a small LRU "window" in front of this filter so that new entries with a burst of recent use get a chance to prove themselves [31]. Caffeine implements W-TinyLFU and adapts the size of the window to the workload [12].

5.5 FIFO-based designs

Recent work has argued that simpler policies can match or improve on LRU. LRU must update shared metadata on every hit, which limits scalability under concurrency [32], whereas FIFO queues update metadata only on insertion and eviction. Yang et al. proposed two principles, lazy promotion (promote an entry only when it is about to be evicted, not on every hit) and quick demotion (remove new entries quickly unless they are reused), and argued that FIFO-based policies built on them can outperform LRU [33]. S3-FIFO applies these principles with three FIFO queues: a small queue for new entries, a main queue for entries that proved useful, and a ghost queue of recently evicted keys [34]. Its authors report, across a large collection of production traces, that many objects are accessed only once in a short window ("one-hit wonders") and that removing them quickly matters more than fine-grained recency ordering [34]. SIEVE goes further, with a single FIFO queue, one visited bit per entry and a pointer, the hand, that moves from the oldest toward the newest entries: a hit only sets the visited bit, and on eviction the hand clears the bit of visited entries, leaving them in place, until it finds one that has not been visited [35]. Its authors describe it as simpler than LRU, report that it compares favorably with established algorithms on web cache traces, and integrated it into several cache libraries [35]. A study of lazy promotion strategies on production traces compares how efficiently each turns promotions into hits [36]. These results come from simulation over traces and from integration in specific libraries; the best policy for a given workload remains an empirical question.

5.6 Approximations in production caches

Production caches rarely implement textbook policies exactly. Redis does not keep a full LRU list; it samples a small number of keys and evicts the best candidate among them, its LFU mode keeps a small probabilistic counter per key that decays over time, and recent versions add policies that evict the least recently modified keys [14]. Redis also lets operators choose whether eviction considers all keys or only keys with an expiration set, and its default policy is not to evict at all but to reject writes when memory is full [14]. That default suits Redis used as a data store; a cache usually needs an evicting policy. The study of Twitter's cache clusters found that TTLs shape what a cache holds as much as eviction does, and that FIFO works as well as or better than LRU for many of those workloads [5]; Segcache, which followed it, groups objects with similar creation and expiration times so that expired objects can be removed promptly [37].

6. Invalidation and consistency

A cache entry is correct only as long as the data it copies has not changed. Invalidation mechanisms trade freshness, cost and complexity.

6.1 Expiration

A time to live bounds how long an entry can be stale. It is simple and self-healing: an entry that was never invalidated still disappears. A TTL bounds staleness but does not provide consistency, and tight freshness needs are hard to meet with TTLs alone [38]. Its cost is a trade-off that has no right answer for every key: a short TTL means more misses and more database load, while a long one means longer staleness. TTLs also synchronize expirations when many entries are written together, which can cause a burst of misses (Section 7.3). Redis expires keys

both passively, when a client touches an expired key, and actively, by sampling keys with an expiration in the background [13].

6.2 Explicit invalidation on write and its races

When the application writes the database, it can delete or update the cached entry. The order matters. If a reader misses, reads the old value from the database, and is delayed, while a writer updates the database and deletes the cache entry, the reader can then write the old value back into the cache, where it stays until it expires. Meta describes this race in its own caches: a delayed fill carrying an old value lands after the invalidation and leaves the cache stale indefinitely [39]. Deleting the entry after the database commit, and again after a short delay ("double delete"), reduces but does not remove this window. Facebook's memcached leases address it: on a miss, the cache gives the reader a token, a delete invalidates outstanding tokens, and the cache refuses a set whose token is no longer valid [1]. The IQ framework uses two kinds of lease to give look-aside caches in front of a relational database strong consistency without modifying the database [40].

6.3 Leases

Gray and Cheriton introduced leases as time-limited grants that let a server promise a client that its cached copy will remain valid, or that the client will be told, for the lease period, and expire on their own so that a failed holder cannot block others indefinitely [41]. In caching practice, leases serve two purposes: they prevent the stale-set race above, and they limit how many clients reload the same missing key at once (Section 7.1) [1].

6.4 Change data capture

Invalidation can be driven by the database's own log of changes rather than by application code. Facebook's mcsqueal daemons read the MySQL commit log and broadcast deletes to the caches, so that invalidation follows every committed change regardless of which code path made it [1]. Change data capture platforms such as Debezium read the write-ahead log or binary log of PostgreSQL, MySQL and other databases and publish row-level change events [42], which a consumer can turn into cache invalidations; the Debezium project has shown this for an application's second-level cache, filtering out the application's own transactions [43]. The approach is complete for direct mappings from rows to cache keys, but derived entries, such as an aggregate or a rendered page that depends on many rows, still need a mapping from changes to keys. Paper 5 examines dependency tracking for that problem.

6.5 Versioned keys

Including a version or timestamp in the cache key, or storing a version with the value, turns invalidation into a lookup: readers ask for the current version, and old entries simply stop being read and age out. Versions also order concurrent updates, so a cache can refuse to overwrite a newer value with an older one. The cost is an extra read of the current version, which must itself be fresh.

6.6 Measuring consistency

Consistency between a cache and its source is rarely measured directly. Meta described Polaris, an external checker that receives invalidation events, queries replicas and retries over several timescales to separate transient lag from lasting inconsistency, treating cache consistency as a measurable reliability property rather than an assumption [39]. The guarantee a cache provides is a design choice to state explicitly: TAO first provided read-your-writes

consistency through write-through caching, and Facebook's FlightTracker later provided it across caches while keeping asynchronous replication [44].

7. Common pitfalls and mitigations

Most cache incidents follow a small number of patterns. Table 2 summarizes them; the subsections give detail.

Pitfall	What happens	Mitigations
Stampede (thundering herd)	Many readers miss the same popular key at once and all reload it	Request coalescing, leases, probabilistic early refresh, refresh-ahead
Penetration	Reads for keys that do not exist always miss and reach the database	Negative caching with short TTLs, Bloom filters, input validation
Avalanche	Many keys expire together, or a cache node fails, and the database receives a burst of misses	TTL jitter, staggered warm-up, replicas, load shedding
Hot keys	A few keys receive a large share of traffic and overload one node	Local tiers, replication of hot keys, key splitting
Stale reads and races	Concurrent read-fill and write leave an old value cached	Delete after commit, leases, versioned writes, CDC-driven invalidation
Memory pressure	The cache fills, evicts useful data or rejects writes	Bounded sizes, the right eviction policy, expiration, sizing from miss-ratio curves
Cold start and cache dependence	An empty or failed cache sends full load to a database sized for the hit path	Warm-up, gradual traffic ramp, admission control, capacity for cache loss
Observability gaps	Hit ratio looks healthy while latency, staleness or one tenant suffers	Per-key-class metrics, miss cost, staleness probes, tail latency

7.1 Stampede

When a popular entry expires or is invalidated, every concurrent reader misses at the same moment and sends the same query to the database. Request coalescing lets only one caller load the value while the others wait for its result; Go's singleflight package [45] and NGINX's proxy cache lock [11] implement this pattern in-process and at a proxy. Leases achieve the same across processes by giving only one client the right to reload [1]. Probabilistic early expiration recomputes an entry before it expires, with a probability that rises as expiration approaches and with how long the recomputation takes; Vattani et al. showed that this approach is optimal under their model [46]. Refresh-ahead (Section 3.1) avoids the expiration event altogether for keys that are kept warm.

7.2 Penetration

A read for a key that does not exist in the database misses the cache every time, because there is nothing to cache. Accidental or malicious traffic for absent keys therefore bypasses the cache entirely. Caching the absence itself (negative caching) with a short TTL stops repeated lookups. A Bloom filter, a compact probabilistic set that can report false positives but

not false negatives [47], can reject keys that certainly do not exist before they reach the database. Validating input at the edge removes obviously invalid keys.

7.3 Avalanche

If many entries are written at the same time with the same TTL, they also expire together, and the database receives their misses in one burst. The same happens when a cache node fails and its keys move elsewhere. Adding random jitter to TTLs spreads expirations; replicas and consistent hashing limit how many keys move when a node fails; and load shedding or circuit breakers protect the database while the cache refills.

7.4 Hot keys

Popularity in real workloads is skewed, so a few keys can receive a large share of requests. In a distributed cache, each key lives on one node, and that node can saturate while others are idle. Fan et al. showed that a small cache of the most popular items in front of a partitioned cluster can provably balance load across its back-end nodes [48]. In practice, teams place a local tier in front of the shared cache for hot keys, replicate hot keys across nodes and read from any copy, or split one logical key into several physical keys. Detecting hot keys requires sampling of access patterns, because averages hide them.

7.5 Stale reads and races

Beyond the read-fill race (Section 6.2), staleness arises when writes bypass the code that invalidates, when invalidation messages are lost or delayed, when replicas lag, and when an update and a delete arrive out of order. Mitigations stack: invalidation driven by the database log rather than by application code [1], [42], leases or versions that refuse stale writes, TTLs as a backstop for anything missed, and explicit statements of which reads may be stale and for how long.

7.6 Memory pressure and eviction storms

A cache without bounds grows until it fails, and a cache with the wrong policy evicts the data that matters. Redis rejects writes when it reaches its memory limit under its default policy [14], which is correct for a data store and surprising for a cache. Large values, many small keys with high per-key overhead, and fragmentation all change effective capacity. Sizing from miss-ratio curves (Section 5.1), setting expiration on cache entries, and choosing an evicting policy deliberately avoid most of these problems.

7.7 Cold start and cache dependence

A database sized for the miss stream of a warm cache cannot serve the full read load. When the cache is empty after a deployment, a failover or a flush, or when a cache tier fails, the database can be overloaded, slow down, and cause retries that keep it overloaded even after the cache has recovered. Bronson et al. describe such metastable failures and name the loss of a cache's hit ratio as a typical trigger [49]; Huang et al. found the pattern in public incident reports from many organizations, including look-aside caching cases [50], and recent work defines such failures as self-sustaining collapses that are hard to diagnose [51]. Mitigations include warming caches before they take traffic (Facebook let a cold cluster read from a warm one [1], and InnoDB can save and restore its buffer pool contents across restarts [52]), ramping traffic gradually, admission control and retry budgets at the database, a fallback tier for failed cache nodes (Facebook used a separate pool for this purpose [1]), and capacity planning that states whether the database must survive the loss of the cache.

7.8 Observability gaps

An aggregate hit ratio can look healthy while a particular key class misses constantly, while misses are concentrated on the most expensive queries, or while entries are stale. Useful signals include hit ratio by key class, the database cost of misses, eviction and expiration rates, memory use and fragmentation, hot-key detection, tail latency of both hits and misses, and direct probes that compare cached values with the source [39]. Traces of real access patterns allow policies and sizes to be evaluated offline before they are changed in production; public traces [53] and open-source simulators such as libCacheSim [54] make this practical.

8. Comparative analysis

Table 3 compares the architecture types in Section 4. The ratings are qualitative and relative to each other; actual behavior depends on the product, its configuration, the topology and the workload.

Architecture type	Hit cost	Capacity	Sharing across instances	Invalidation path	Consistency risk	Best-fit data
In-process cache	Very low (memory lookup)	Low (process heap)	None; each instance has its own copy	Must reach every instance	High unless data rarely changes	Configuration, reference data, the hottest keys
Distributed cache	Low (network round trip)	High; scales with nodes	Full	One place per key	Medium; races and lag between store and cache	Shared session, object and query-result caching
Near cache with server-assisted invalidation	Very low for local hits	Low locally, high behind it	Partial	Server pushes invalidations to clients	Medium; short windows after writes	Read-mostly hot data over a distributed cache
Database buffer cache	Low inside the database	Bounded by database memory	Full, inside the database	Managed by the database	Very low; consistent by construction	All data, at page level
Materialized view	Query-level reuse inside the database	Table storage	Full	Explicit refresh (or incremental maintenance where supported)	Stale between refreshes	Expensive aggregates read often
HTTP or CDN cache	Very low for users near an edge location	High; many locations	Across users for public content	Expiration, validation and purge	High for per-user or rapidly changing data	Public, cacheable API responses

9. Open challenges and future opportunities

The foundations in this paper are mature, but each area still has open problems. Each subsection names a challenge, a research direction and an opportunity, and points to the paper in the series that examines it.

9.1 Policies that adapt to the workload

Challenge. No single eviction or admission policy suits every workload, and workloads change over time. Recent work shows that simple FIFO-based policies can match more complex ones on many traces [34], [35], which raises the question of when complexity pays. **Research direction.** Machine-learned policies try to approximate the optimal offline policy from observed access patterns, and adaptive policies such as W-TinyLFU already tune themselves [12]. **Opportunity.** Policies that adapt safely, with low overhead and a predictable fallback, would reduce manual tuning. Paper 2 examines learned cache replacement.

9.2 Caching by meaning

Challenge. Exact-match keys cannot reuse the answer to a question asked in different words, which matters for LLM applications in which each miss is an expensive model call. **Research direction.** Semantic caches match requests by embedding similarity, and inference servers reuse intermediate computation across requests. **Opportunity.** Semantic caching can reduce cost and response time for AI applications, provided that false hits, access control and evaluation are handled. Paper 3 examines semantic caching for LLM applications and vector databases.

9.3 Memory and storage tiers

Challenge. DRAM accounts for much of the cost of a large cache, while flash is cheaper but slower, wears with writes and behaves differently under small random writes. **Research direction.** Hybrid caches place hot data in DRAM and warm data on flash [3], and memory expansion over CXL adds a tier between them. **Opportunity.** Tiered designs can hold larger working sets at lower cost, if placement policies and flash endurance are managed. Paper 4 examines non-volatile memory, NVMe SSDs and hybrid tiered caching.

9.4 Precise invalidation

Challenge. TTLs are imprecise, and explicit invalidation is hard to get right for derived data that depends on many rows. **Research direction.** Dependency tracking, incremental view maintenance and invalidation driven by change data capture aim to invalidate exactly what a change affects, production systems have begun to measure cache consistency directly [39], and MuCache comes with a proof that its invalidation protocol for caches between microservices preserves correctness [55]. **Opportunity.** Precise, automated invalidation would let teams cache derived data with stated freshness guarantees. Paper 5 examines dependency-graph and automated approaches.

9.5 Hierarchies at the edge

Challenge. Edge locations, regions and devices each add a cache tier with its own connectivity, capacity and consistency limits. **Research direction.** Hierarchical and cooperative caching, edge data stores and conflict-free replicated data types address placement and consistency across tiers. **Opportunity.** Multi-level designs can serve users and

devices locally while keeping a clear model of freshness. Paper 6 examines multi-level caching for geo-distributed, edge and IoT data.

9.6 Correctness and resilience under failure

Challenge. Caches fail partially, invalidations are lost, and recovery can trigger metastable overload [49], [50]. **Research direction.** Formal and empirical study of cache failure modes, consistency measurement [39] and load-aware recovery. **Opportunity.** Treating the cache as a component with stated guarantees and tested failure behavior, rather than as an optimization that is assumed to work, makes the systems around it more predictable.

10. What this means for platform decisions

Start without a cache and add one when measurements show repeated reads that the database cannot serve within its targets after query and schema tuning. Choose the strategy from the freshness the data requires: cache-aside with invalidation on write and a TTL backstop is a sound default for read-heavy data; write-through suits read-after-write workloads; write-behind needs an explicit decision about acceptable loss. Choose the architecture from who shares the data and how invalidation reaches it, and treat eviction policy and sizing as empirical questions to answer from access traces and miss-ratio curves. Plan for stampedes, hot keys and the loss of the cache before they happen, and measure staleness as well as hit ratio.

For distributed caching, Bassirah works with Redis as a Strategic Technology Partner, and Redis's eviction policies, expiration, client-side caching and replication cover the patterns in this paper. Other platforms named in this paper are ecosystem references. In each case, validate the design on representative workloads, failure scenarios and versions before relying on it.

References

1. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P. et al. (2013). Scaling Memcache at Facebook. Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13), 385–398. (<https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>)
2. Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H. et al. (2013). TAO: Facebook's Distributed Data Store for the Social Graph. Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13), 49–60. (<https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson>)
3. Berg, B., Berger, D. S., McAllister, S., Grosof, I., Gunasekar, S., Lu, J., Uhlar, M., Carrig, J., Beckmann, N., Harchol-Balter, M. et al. (2020). The CacheLib Caching Engine: Design and Experiences at Scale. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 753–768. (<https://www.usenix.org/conference/osdi20/presentation/berg>)
4. Singhvi, A., Akella, A., Anderson, M., Cauble, R., Deshmukh, H., Gibson, D., Martin, M. M. K., Strominger, A., Wenisch, T. F. and Vahdat, A. (2021). CliqueMap: productionizing an RMA-based distributed caching system. Proceedings of the 2021 ACM SIGCOMM Conference, 93–105. (<https://doi.org/10.1145/3452296.3472934>)

5. Yang, J., Yue, Y. and Rashmi, K. V. (2020). A large scale analysis of hundreds of in-memory cache clusters at Twitter. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 191–208.
(<https://www.usenix.org/conference/osdi20/presentation/yang>)
6. Microsoft (2025). Cache-Aside pattern. Azure Architecture Center, Microsoft Learn (ms.date 11 September 2025). (<https://learn.microsoft.com/en-us/azure/architecture/patterns/cache-aside>), accessed 26 October 2026.
7. Amazon Web Services (n.d.). Caching strategies for Memcached. Amazon ElastiCache User Guide.
(<https://docs.aws.amazon.com/AmazonElastiCache/latest/dg/Strategies.html>), accessed 26 October 2026.
8. Manes, B. (n.d.). Refresh. Caffeine wiki, GitHub. (<https://github.com/ben-manes/caffeine/wiki/Refresh>), accessed 26 October 2026.
9. Oracle (n.d.). Caching Data Sources. Developing Applications with Oracle Coherence, 14.1.2. (<https://docs.oracle.com/en/middleware/fusion-middleware/coherence/14.1.2/develop-applications/caching-data-sources.html>), accessed 26 October 2026.
10. Nottingham, M. (2010). HTTP Cache-Control Extensions for Stale Content. RFC 5861, Internet Engineering Task Force, May 2010. (<https://doi.org/10.17487/RFC5861>)
11. F5, Inc. / NGINX (n.d.). Module ngx_http_proxy_module. nginx documentation. (https://nginx.org/en/docs/http/ngx_http_proxy_module.html), accessed 26 October 2026.
12. Manes, B. (n.d.). Efficiency. Caffeine wiki, GitHub. (<https://github.com/ben-manes/caffeine/wiki/Efficiency>), accessed 26 October 2026.
13. Redis Ltd. (n.d.). EXPIRE. Redis command reference.
(<https://redis.io/docs/latest/commands/expire/>), accessed 26 October 2026.
14. Redis Ltd. (n.d.). Key eviction. Redis documentation.
(<https://redis.io/docs/latest/develop/reference/eviction/>), accessed 26 October 2026.
15. Redis Ltd. (n.d.). Client-side caching reference. Redis documentation.
(<https://redis.io/docs/latest/develop/reference/client-side-caching/>), accessed 26 October 2026.
16. Trollope, R. (2025). Redis is now available under the AGPLv3 open source license. Redis blog, 1 May 2025 (updated 23 May 2025). (<https://redis.io/blog/agplv3/>), accessed 26 October 2026.
17. The Linux Foundation (2024). Linux Foundation Launches Open Source Valkey Community. Press release, 28 March 2024. (<https://www.linuxfoundation.org/press/linux-foundation-launches-open-source-valkey-community>), accessed 26 October 2026.
18. The PostgreSQL Global Development Group (n.d.). 19.4. Resource Consumption. PostgreSQL 18 Documentation. (<https://www.postgresql.org/docs/current/runtime-config-resource.html>), accessed 26 October 2026.
19. Oracle (n.d.). 17.5.1 Buffer Pool. MySQL 8.4 Reference Manual.
(<https://dev.mysql.com/doc/refman/8.4/en/innodb-buffer-pool.html>), accessed 26 October 2026.
20. The PostgreSQL Global Development Group (n.d.). 39.3. Materialized Views. PostgreSQL 18 Documentation. (<https://www.postgresql.org/docs/current/rules-materializedviews.html>), accessed 26 October 2026.

21. Fielding, R., Nottingham, M. and Reschke, J. (2022). HTTP Caching. RFC 9111 (STD 98), Internet Engineering Task Force, June 2022. (<https://doi.org/10.17487/RFC9111>)
22. Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. IBM Systems Journal, 5(2), 78–101. (<https://doi.org/10.1147/sj.52.0078>)
23. Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L. (1970). Evaluation techniques for storage hierarchies. IBM Systems Journal, 9(2), 78–117. (<https://doi.org/10.1147/sj.92.0078>)
24. Waldspurger, C. A., Park, N., Garthwaite, A. and Ahmad, I. (2015). Efficient MRC Construction with SHARDS. Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15), 95–110. (<https://www.usenix.org/conference/fast15/technical-sessions/presentation/waldspurger>)
25. Shakiba, K., Sultan, S. and Stumm, M. (2024). Kosmo: Efficient Online Miss Ratio Curve Generation for Eviction Policy Evaluation. Proceedings of the 22nd USENIX Conference on File and Storage Technologies (FAST '24), 89–105. (<https://www.usenix.org/conference/fast24/presentation/shakiba>)
26. O'Neil, E. J., O'Neil, P. E. and Weikum, G. (1993). The LRU-K page replacement algorithm for database disk buffering. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, 297–306. (<https://doi.org/10.1145/170035.170081>)
27. Johnson, T. and Shasha, D. (1994). 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. Proceedings of the 20th International Conference on Very Large Data Bases (VLDB '94), Santiago, Chile, 439–450. (<https://www.vldb.org/conf/1994/P439.PDF>)
28. Jiang, S. and Zhang, X. (2002). LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 31–42. (<https://doi.org/10.1145/511334.511340>)
29. Megiddo, N. and Modha, D. S. (2003). ARC: A Self-Tuning, Low Overhead Replacement Cache. Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03), 115–130. (<https://www.usenix.org/conference/fast-03/arc-self-tuning-low-overhead-replacement-cache>)
30. Jiang, S., Chen, F. and Zhang, X. (2005). CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC '05), 323–336. (<https://www.usenix.org/conference/2005-usenix-annual-technical-conference/clock-pro-effective-improvement-clock-replacement>)
31. Einziger, G., Friedman, R. and Manes, B. (2017). TinyLFU: A Highly Efficient Cache Admission Policy. ACM Transactions on Storage, 13(4), 1–31. (<https://doi.org/10.1145/3149371>)
32. Qiu, Z., Yang, J., Zhang, J., Li, C., Ma, X., Chen, Q., Yang, M. and Xu, Y. (2023). FrozenHot Cache: Rethinking Cache Management for Modern Hardware. Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23), 557–573. (<https://doi.org/10.1145/3552326.3587446>)
33. Yang, J., Qiu, Z., Zhang, Y., Yue, Y. and Rashmi, K. V. (2023). FIFO can be Better than LRU: the Power of Lazy Promotion and Quick Demotion. Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS '23), 70–79. (<https://doi.org/10.1145/3593856.3595887>)
34. Yang, J., Zhang, Y., Qiu, Z., Yue, Y. and Vinayak, R. (2023). FIFO queues are all you need for cache eviction. Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23), 130–149. (<https://doi.org/10.1145/3600006.3613147>)

35. Zhang, Y., Yang, J., Yue, Y., Vigfusson, Y. and Rashmi, K. V. (2024). SIEVE is Simpler than LRU: an Efficient Turn-Key Eviction Algorithm for Web Caches. Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24), 1229–1246. (<https://www.usenix.org/conference/nsdi24/presentation/zhang-yazhuo>)
36. Chen, Q., Al-Araby, M. H. M., Qiu, Z., Chen, Z., Vinayak, R. and Yang, J. (2025). Demystifying and Improving Lazy Promotion in Cache Eviction. Proceedings of the VLDB Endowment, 19(4), 549–562. (<https://doi.org/10.14778/3785297.3785299>)
37. Yang, J., Yue, Y. and Vinayak, R. (2021). Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. Proceedings of the 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI '21), 503–518. (<https://www.usenix.org/conference/nsdi21/presentation/yang-juncheng>)
38. Mao, Z., Iyer, R., Shenker, S. and Stoica, I. (2024). Revisiting Cache Freshness for Emerging Real-Time Applications. Proceedings of the 23rd ACM Workshop on Hot Topics in Networks (HotNets '24), 335–342. (<https://doi.org/10.1145/3696348.3696858>)
39. Pan, L. (2022). Cache made consistent. Engineering at Meta, 8 June 2022. (<https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/>), accessed 26 October 2026.
40. Ghandeharizadeh, S., Yap, J. and Nguyen, H. (2014). Strong consistency in cache augmented SQL systems. Proceedings of the 15th International Middleware Conference (Middleware '14), 181–192. (<https://doi.org/10.1145/2663165.2663318>)
41. Gray, C. and Cheriton, D. (1989). Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. Proceedings of the twelfth ACM Symposium on Operating Systems Principles (SOSP '89), 202–210. (<https://doi.org/10.1145/74850.74870>)
42. Debezium Authors (n.d.). Debezium Architecture. Debezium Documentation (version 3.x). (<https://debezium.io/documentation/reference/stable/architecture.html>), accessed 26 October 2026.
43. Morling, G. (2018). Automating Cache Invalidation With Change Data Capture. Debezium blog, 5 December 2018. (<https://debezium.io/blog/2018/12/05/automating-cache-invalidation-with-change-data-capture/>), accessed 26 October 2026.
44. Shi, X., Pruett, S., Doherty, K., Han, J., Petrov, D., Carrig, J., Hugg, J. and Bronson, N. (2020). FlightTracker: Consistency across Read-Optimized Online Stores at Facebook. Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20), 407–423. (<https://www.usenix.org/conference/osdi20/presentation/shi>)
45. The Go Authors (n.d.). Package singleflight (golang.org/x/sync/singleflight). Go package documentation. (<https://pkg.go.dev/golang.org/x/sync/singleflight>), accessed 26 October 2026.
46. Vattani, A., Chierichetti, F. and Lowenstein, K. (2015). Optimal probabilistic cache stampede prevention. Proceedings of the VLDB Endowment, 8(8), 886–897. (<https://doi.org/10.14778/2757807.2757813>)
47. Bloom, B. H. (1970). Space/time trade-offs in hash coding with allowable errors. Communications of the ACM, 13(7), 422–426. (<https://doi.org/10.1145/362686.362692>)
48. Fan, B., Lim, H., Andersen, D. G. and Kaminsky, M. (2011). Small cache, big effect: provable load balancing for randomly partitioned cluster services. Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC '11), 1–12. (<https://doi.org/10.1145/2038916.2038939>)

49. Bronson, N., Aghayev, A., Charapko, A. and Zhu, T. (2021). Metastable failures in distributed systems. Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '21), 221–227. (<https://doi.org/10.1145/3458336.3465286>)
50. Huang, L., Magnusson, M., Muralikrishna, A. B., Estyak, S., Isaacs, R., Aghayev, A., Zhu, T. and Charapko, A. (2022). Metastable Failures in the Wild. Proceedings of the 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI '22), 73–90. (<https://www.usenix.org/conference/osdi22/presentation/huang-lexiang>)
51. Isaacs, R., Alvaro, P., Majumdar, R., Muniswamy-Reddy, K.-K., Salamati, M. and Soudjani, S. (2025). Analyzing Metastable Failures. Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS '25), 172–178. (<https://doi.org/10.1145/3713082.3730380>)
52. Oracle (n.d.). 17.8.3.6 Saving and Restoring the Buffer Pool State. MySQL 8.4 Reference Manual. (<https://dev.mysql.com/doc/refman/8.4/en/innodb-preload-buffer-pool.html>), accessed 26 October 2026.
53. Twitter, Inc. (2020). cache-trace: A collection of Twitter's anonymized production cache traces. GitHub repository (CC-BY-4.0). (<https://github.com/twitter/cache-trace>), accessed 26 October 2026.
54. Yang, J. et al. (n.d.). libCacheSim: a high-performance library for building cache simulators. GitHub repository. (<https://github.com/1a1a11a/libCacheSim>), accessed 26 October 2026.
55. Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S. and Liu, V. (2024). MuCache: A General Framework for Caching in Microservice Graphs. Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI '24), 221–238. (<https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran>)

Product and project names are trademarks of their respective owners. Their mention describes publicly documented technology and does not imply a business relationship, except where Bassirah names a Strategic Technology Partner.