

Coordinated Design of Cache Consistency and Invalidation Recovery in High Concurrency Backends

Xiaoming Ma

Hangzhou Yiyi Zhixing Technology, Hangzhou, China

Corresponding author: 929367421@qq.com

Abstract

Databases and caches jointly support typical read and write paths in high concurrency backends. When they are operated independently, successful writes, cache invalidation, and stale query fills can interleave and create races. This study examines single deletion, delayed double deletion, epoch token validation, and a combined strategy that integrates validation with invalidation replay in the cache-aside pattern. A finite-state model with one key, one reader, and one writer enumerates ten event schedules under specified partial-order constraints. Three delivery conditions are considered: normal delivery, loss of the first invalidation, and loss of all invalidations. Model execution shows that, under normal delivery, single deletion and double deletion leave stale cached data in two schedules and one schedule, respectively. Token validation eliminates these residual states under the stated assumptions but still fails when the first invalidation is lost. With successful replay, the combined strategy restores convergence within the model. These results distinguish two responsibilities: preventing stale requests from filling the cache and ensuring that invalidation eventually executes. Both require coordinated treatment. The contribution is a set of reproducible concurrency checks and a design framework that separates fill control, event recovery, and application read semantics. The experiment does not measure production-cluster throughput or establish global strong consistency.

Keywords: backend architecture, cache consistency, concurrency control, transactional outbox, failure recovery

1 Introduction

In product detail, user profile, and content services, caches reduce repeated queries and preserve database resources for requests that are harder to reuse. However, storing a copy of application data also introduces the problem of invalidating that copy. A request may retrieve an old value before a database update but fill the cache only after the update completes and the cache has been deleted. All write operations may succeed, while stale data still reappears in the cache. Consequently, API success rates and cache hit rates do not adequately describe system correctness.

The Redis cache-aside documentation describes the basic paths for on-demand population and invalidation after writes, but individual systems must also handle request interleavings and failed operations that require retries (Redis, n.d.-f). Research on large-scale caching has likewise treated stale fills and concurrent database loads as problems requiring explicit coordination (Nishtala et al., 2013). This article compares several common engineering strategies within the same event model to clarify the problem each addresses, its assumptions, and the failures under which it breaks down.

This study asks two questions. First, can delayed double deletion eliminate every stale request fill? Second, can epoch validation independently resolve message loss? Its main contributions are executable, reviewable counterexamples and a design framework that separates fill control, event recovery, and application read semantics. The framework supports analysis of backend read paths such as order displays and inventory queries. It does not replace transaction design for inventory deductions or financial accounting.

2 Related Work and Consistency Boundaries

The meaning of the latest value in a distributed system requires an explicit ordering relation. Lamport (1978) explains the logical order of distributed execution through precedence relations between events, providing a basis for analyzing read and write interleavings. Arrows in this article denote required precedence constraints; they do not assume that physical clocks on different servers can be compared directly. Waiting a few milliseconds is a scheduling policy and does not automatically establish a causal guarantee across processes.

Dynamo balances availability and consistency through object versioning and conflict resolution, illustrating the need to make replica-management semantics explicit (DeCandia et al., 2007). COPS and Eiger investigate causal consistency and stronger semantics for operations across keys (Lloyd et al., 2011, 2013). These studies provide conceptual background, but their conclusions do not establish that combining Redis with a database inherently ensures consistency. Database replication protocols and application-level invalidation operate at different layers; component names alone do not establish end-to-end properties.

This article distinguishes three goals: cache convergence after recovery completes in the absence of concurrent writes; read-your-writes behavior within a session; and linearizability for arbitrary reads. Linearizability requires operations to appear in a single order that respects real-time precedence (Jepsen, n.d.). This requirement is substantially stronger than eventually deleting stale cached data. The experiment checks only one terminal-state property associated with the first goal and does not certify the other goals.

Database isolation also has boundaries. PostgreSQL transaction isolation governs concurrent visibility within the database; it does not automatically incorporate Redis commands into the same transaction (PostgreSQL Global Development Group, n.d.). Conversely, Redis transactions coordinate Redis command sequences and do not provide atomicity for database commits (Redis, n.d.-h). Applications therefore need a propagation and recovery protocol spanning both storage systems.

3 How Stale Cache Fills Arise

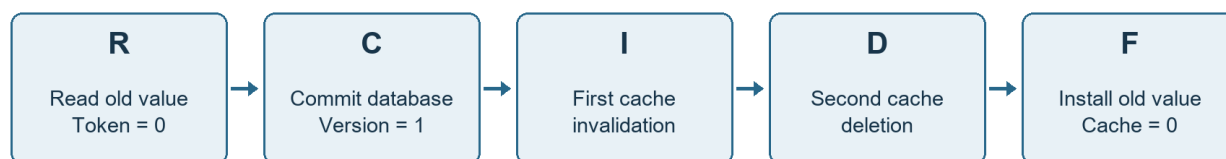
Suppose that an object initially has database version zero and the cache is empty. A reader obtains version zero from the database and then waits for network activity or application processing. A writer updates the database to version one and successfully deletes the cache entry. The earlier reader subsequently resumes and writes version zero into the cache. Later cache hits again return the old value. The failure occurs because the old query remains eligible to fill the cache, even though the deletion itself succeeded.

Figure 1 shows the critical path of this counterexample. R denotes snapshot capture, C the database commit, I the first cache invalidation, F the cache fill, and D a later opportunity for a second deletion or replay. With the order $R \rightarrow C \rightarrow I \rightarrow D \rightarrow F$, the second deletion cannot remove a stale fill that has not yet occurred. Setting a fixed, empirically chosen deletion delay therefore covers only part of the latency distribution. Without an upper bound on request duration, it does not constitute a complete proof.

Figure 1

A Stale Fill Can Occur After Both Cache Deletions

A Late Fill Can Outlive Both Cache Deletions



Logical event order only. No wall-clock delay or production failure rate is implied.

Note. Created by the author from the finite-state model in this study.

An expiration policy can limit the lifetime of a cache entry, but the starting point matters. If an old value is inserted much later and receives a new lifetime, the application may experience a stale interval that includes both the stalled query duration and the new entry lifetime. Redis EXPIRE provides key expiration but does not check whether a value is stale relative to the database (Redis, n.d.-d). A thirty-second expiration therefore does not imply recovery within thirty seconds of the database commit.

4 Coordinated Control Strategies

4.1 Revoking Fill Eligibility

The epoch token strategy maintains a monotonically advancing epoch marker for each cached object. A reader obtains a token before loading from the database. Invalidation deletes the entry and advances the epoch. The reader may fill the cache only if the current epoch still matches the original token. A request overtaken by an invalidation thus loses its fill eligibility. The lease mechanism described by Nishtala et al. (2013) addresses related concerns. This study uses a simplified epoch model and does not claim to reproduce their production implementation.

Comparison and installation must occur in the same atomic step. Otherwise, a concurrent write may intervene after the check succeeds. Redis scripts can organize server-side checks and writes (Redis, n.d.-c), but they provide atomicity only within the cache. Database snapshot capture, epoch reads, script execution, and failover remain separate events. Implementations should also bound script work rather than concentrate complex application logic on the cache server.

The epoch marker cannot be treated as ordinary disposable cache data. Suppose the epoch advances from zero to one after invalidation. If the marker is then evicted and recreated with a default value of zero, an old request may pass the check again, producing an ABA-like reuse problem. Redis eviction and replication behavior must therefore be considered in deployment assumptions (Redis, n.d.-e, n.d.-g). Possible safeguards include nonreused namespaces, a persistent source of versions, or explicit recovery barriers, whose additional costs require separate evaluation.

4.2 Binding Event Creation to the Database Commit

Epoch control depends on invalidation actually occurring. If a process exits immediately after committing the database change, neither deletion nor epoch advancement may execute. A transactional outbox records the application change and the pending event in the same database transaction, after which an independent publisher propagates the event. This reduces the gap in which data have changed but no event has been recorded (Richardson, n.d.). The Debezium outbox event router offers implementation support, but adopting a tool does not by itself make the application protocol correct (Debezium, n.d.).

Consumers should handle duplicate events for each object and define behavior for out-of-order delivery. If a message merely revokes fill eligibility, processing it again can advance the epoch once more, generally reducing the hit rate without reintroducing an old value. If a message carries data and directly overwrites the cache, application versions must be compared to prevent a late old message from replacing a newer value. Event versions, fill tokens, and consumer progress serve different purposes: data ordering, eligibility checks, and recovery. A single ambiguous version number cannot substitute for all three designs.

Durable notifications and incremental processing have an established research basis (Peng & Dabek, 2010), but an outbox does not make asynchronous invalidation instantaneous. Systems should monitor commit-to-consumption delay, record failure reasons, and support replay. Unparseable events need quarantine and manual intervention paths rather than unlimited retries. Timeouts, backoff, and jitter control retry load and improve recovery behavior; they do not prove data consistency (Amazon Web Services, n.d.).

4.3 Explicit Guarantees for Different Reads

Ordinary display reads may tolerate bounded staleness, whereas a confirmation page after a write may need to show the change immediately. Such reads can temporarily bypass the cache and query the primary database, or carry the minimum application version produced by the commit and load from the database when the cached version is insufficient. That version should come from the database commit result rather than the client clock. This concentrates consistency costs on reads that require stronger guarantees.

For critical decisions such as inventory deductions, payment-state transitions, and permission revocation, cached values should not be the sole authority. Constraints should be enforced through conditional updates, transactions, or appropriate concurrency control in the authoritative store. A cache may support displays and candidate information, but a high hit rate cannot replace checks of application invariants. Multiple client-cache layers add further invalidation stages. Redis client-side cache tracking provides invalidation notifications, but applications must still manage connection and local-copy lifecycles (Redis, n.d.-a).

5 Finite State Experiment Design

The study implements deterministic event enumeration in Python. Initially, the database version is zero, the cache is empty, and the epoch is zero. There is one reader and one writer. R captures the database value and epoch; F attempts a cache fill; C advances the database to version one; I performs the first invalidation; and D represents a later opportunity for a second deletion or successful replay. The constraints R before F and C before I before D yield ten valid total orders. For strategies that do not use D, it remains a no-op to preserve a common comparison grid.

Four strategies are compared. Single deletion executes only I. Double deletion executes I and D. Epoch validation advances the epoch at I and rejects F when its token does not match. The combined strategy advances the epoch and clears the cache at both I and D. Three fault conditions are considered: I is delivered normally; I is lost but D can execute; and both I and D are lost. Each condition covers four strategies and ten schedules, producing 120 traces. The outcome is whether a nonempty cache contains a version older than the database after all events finish.

The model abstracts the database read and token acquisition at R into one step, and validation and installation at F into another atomic step. This reduces the state space and limits the conclusions. The main experiment does not model time to live (TTL), database replica lag, cache eviction, or process restart. D represents one opportunity after I rather than a fixed wall-clock delay. The program records the database, cache, and epoch state at every step. An additional negative control resets an evicted epoch marker to zero to examine a risk outside the main model.

6 Experimental Results and Interpretation

Table 1 reports the number of schedules that end with stale cached data. Every denominator is ten, representing coverage in the common enumeration grid rather than a production error rate. In particular, no-op events cause some effective executions to appear more than once in the grid, so the counts cannot be used to compare real-world failure probabilities.

Table 1

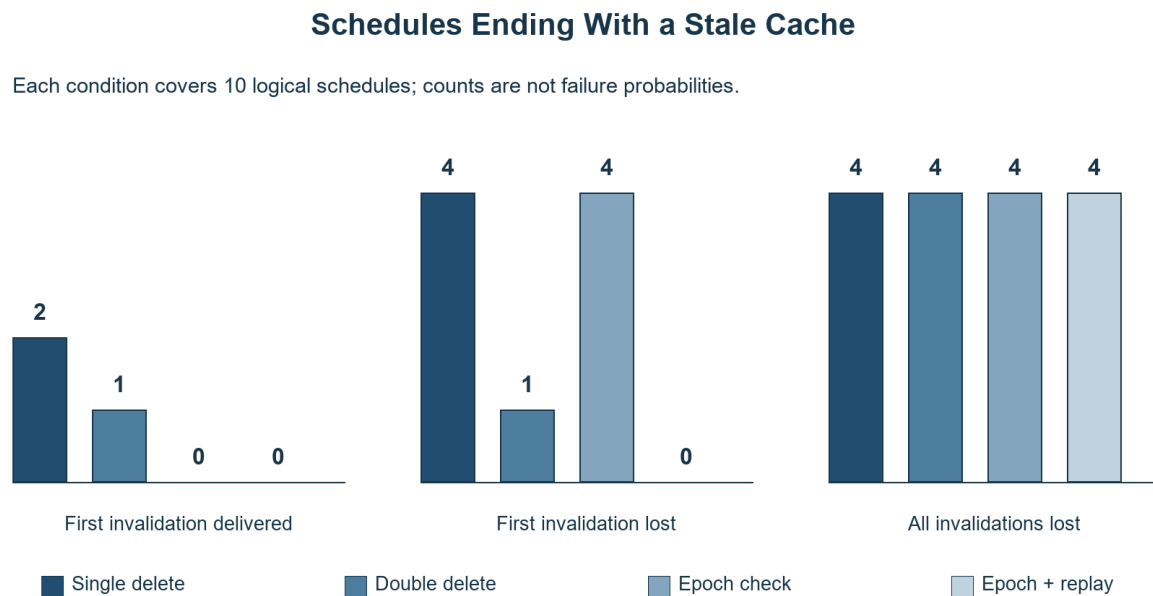
Schedules Ending With Stale Cached Data Under Different Fault Conditions

Fault condition	Single deletion	Delayed double deletion	Epoch validation	Epoch validation and replay
First invalidation delivered	2/10	1/10	0/10	0/10
First invalidation lost	4/10	1/10	4/10	0/10
All invalidations lost	4/10	4/10	4/10	4/10

Under normal delivery, single deletion fails in the schedules $R \rightarrow C \rightarrow I \rightarrow F \rightarrow D$ and $R \rightarrow C \rightarrow I \rightarrow D \rightarrow F$. Double deletion removes the stale fill in the first schedule but cannot remove the later F in the second. It therefore covers a time window rather than the entire lifetime of an old request. Figure 2 summarizes the model coverage results under the three conditions.

Figure 2

Number of Schedules Ending With Stale Cached Data



Note. Created by the author from the finite-state model in this study.

Epoch validation leaves no stale cached data under normal delivery because an old token cannot pass F after I. However, when the first invalidation is lost, the epoch does not change, and the strategy fails in four schedules, just as single deletion does. The combined strategy leaves no terminal stale entry when D executes successfully: D clears old values installed before it, and stale fills after D are rejected. These mechanisms complement each other by clearing an existing incorrect state and preventing its reconstruction.

When all invalidations are lost, all four strategies fail in four schedules. Successful replay is therefore a premise of the combined strategy's result. The negative control also confirms that a previous token can match again after an epoch marker is deleted and recreated with an old value. The experiment was executed programmatically, but it sent no Redis commands and recorded neither service throughput nor real-world failure probabilities. Zero counterexamples means only that the terminal-state check passed within the bounded state space; it does not establish that the system can never return stale data.

Terminal convergence and read correctness during execution are different metrics. Even if the combined strategy eventually clears the old value, another request may hit stale cached data after C has committed but before I executes. The main model does not include this observer and therefore cannot rule out such reads. If every new request after a database commit must avoid returning an old value, stronger read and write coordination or authoritative version checks are required, and their costs must be evaluated.

Similarly, an outbox records the event together with the application change but does not independently bound recovery time. A stopped consumer, a congested retry queue, or unavailable persistent storage can extend the stale interval. Successful replay in the combined strategy is a liveness condition that requires monitoring and operational support. A bounded-staleness commitment also requires bounds on propagation delay, database-load snapshots, and maximum request lifetime. Without them, the appropriate claim is conditional eventual recovery.

7 Joint Evaluation of Reliability and Performance

Backend evaluations should report cache hit rate, database-load requests, rejected fills, invalidation backlog, and stale-state duration together. A high rejection count may indicate effective control, but it may also indicate that frequent writes to a hot object repeatedly force readers back to the database. It must be interpreted alongside database load. Standard trace context can connect request paths and event propagation, although applications must define correlation across queues and retries (W3C, 2021).

Request coalescing suppresses simultaneous database loads for misses on the same key, but it addresses a different problem from consistency validation. Even a single loader can overlap a write. Conversely, correct epoch validation does not ensure that the volume of database loads is sustainable. Distributed locks used for coalescing or serialization require explicit holder identity, timeouts, and behavior after expiration. The presence of a lock key does not establish that an old holder can never continue executing (Redis, n.d.-b).

Total cost can be divided into cache access, database loading, message propagation, and consistency checking. Additional epoch queries, scripts, and durable events create overhead whose acceptability depends on write frequency, object size, and the application's staleness budget. This study does not measure those costs and therefore proposes evaluation dimensions only. Subsequent comparisons should use the same hardware and workloads and record tail latency during failure recovery rather than reporting only average response times without faults.

8 Limitations and Further Validation

The experiment covers one key, one read, and one write; it does not exhaust real-system behavior. Multiple writers introduce event reordering and version contention, while multiple keys raise snapshot and application-relationship issues. Reads that hit the cache are also absent from the main model. The abstraction at R hides additional interleavings between token acquisition and the database query. In particular, a stale snapshot returned by a database replica may invalidate the simplified reasoning. This experiment is suitable for finding explicit counterexamples and checking local mechanisms, but it cannot replace verification of the full protocol.

Further experiments should proceed in three stages. First, refine the model by separating token reads, primary or replica reads, and validation and installation, then add timeouts, restarts, and epoch reclamation. Second, run an actual database, Redis, and event consumer in containers with controlled process termination and network faults.

Third, conduct load tests based on application traffic distributions and record versions rather than API status codes alone. Each stage should retain random seeds, software versions, fault-injection times, and complete histories to distinguish model errors, implementation errors, and measurement noise.

Staleness should be defined using both version lag and elapsed time. Version lag detects regression but does not necessarily represent equal application loss. Time-based measures require trustworthy commit and observation records; subtracting uncalibrated timestamps from different machines is insufficient. Acceptance criteria for permission revocation and inventory should reflect the consequences of incorrect decisions, whereas content displays may tolerate short periods of version lag. Evaluation can determine whether an optimization succeeds only after the acceptable semantics are explicit.

9 Conclusion

Cache consistency involves both concurrent event ordering and invalidation recovery. The finite-state experiment shows that delayed double deletion covers some stale fills but cannot exclude late fills without a delay bound. Epoch tokens revoke stale requests' fill eligibility but do not independently compensate for lost invalidation events. A convergence argument under explicit assumptions requires atomic fill checks, durable event recording, replayable consumption, and defined application read guarantees.

In practice, authoritative databases should maintain critical invariants, while caches accelerate reads that tolerate staleness. Stronger semantics, such as confirmation after a write, should receive separate treatment. Evaluation should extend beyond hit rates to version regression, propagation backlog, and recovery duration. This study contributes executable counterexamples and a validation path with explicit boundaries. Further experiments on real clusters should examine whether the model assumptions hold and whether the additional coordination costs fit application budgets.

References

- Amazon Web Services. (n.d.). *Timeouts, retries, and backoff with jitter*.
<https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/>
- Debezium. (n.d.). *Outbox event router*. Retrieved October 18, 2026, from
<https://debezium.io/documentation/reference/stable/transformations/outbox-event-router.html>
- DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., & Vogels, W. (2007). Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles* (pp. 205–220). ACM. <https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf>
- Jepsen. (n.d.). *Linearizability*. <https://jepsen.io/consistency/models/linearizable>
- Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 558–565.
<https://lamport.azurewebsites.net/pubs/time-clocks.pdf>
- Lloyd, W., Freedman, M. J., Kaminsky, M., & Andersen, D. G. (2011). Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles* (pp. 401–416). ACM. <https://www.cs.cmu.edu/~dga/papers/cops-sosp2011.pdf>
- Lloyd, W., Freedman, M. J., Kaminsky, M., & Andersen, D. G. (2013). Stronger semantics for low-latency geo-replicated storage. In *10th USENIX Symposium on Networked Systems Design and Implementation* (pp. 313–328). USENIX Association.
<https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd>
- Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., & Venkataramani, V. (2013). Scaling Memcache at Facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation* (pp. 385–398). USENIX Association.
<https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>
- Peng, D., & Dabek, F. (2010). Large-scale incremental processing using distributed transactions and notifications. In *9th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association.
<https://www.usenix.org/conference/osdi10/large-scale-incremental-processing-using-distributed-transactions-and>

- PostgreSQL Global Development Group. (n.d.). *Transaction isolation*. Retrieved October 18, 2026, from <https://www.postgresql.org/docs/current/transaction-iso.html>
- Redis. (n.d.-a). *Client-side caching introduction*. <https://redis.io/docs/latest/develop/clients/client-side-caching/>
- Redis. (n.d.-b). *Distributed locks with Redis*. <https://redis.io/docs/latest/develop/clients/patterns/distributed-locks/>
- Redis. (n.d.-c). *Eval*. <https://redis.io/docs/latest/commands/eval/>
- Redis. (n.d.-d). *Expire*. <https://redis.io/docs/latest/commands/expire/>
- Redis. (n.d.-e). *Key eviction*. <https://redis.io/docs/latest/develop/reference/eviction/>
- Redis. (n.d.-f). *Redis cache-aside*. <https://redis.io/docs/latest/develop/use-cases/cache-aside/>
- Redis. (n.d.-g). *Redis replication*. https://redis.io/docs/latest/operate/oss_and_stack/management/replication/
- Redis. (n.d.-h). *Transactions*. <https://redis.io/docs/latest/develop/using-commands/transactions/>
- Richardson, C. (n.d.). *Pattern: Transactional outbox*. Microservices.io. <https://microservices.io/patterns/data/transactional-outbox.html>
- W3C. (2021). *Trace context*. <https://www.w3.org/TR/trace-context/>

Research Transparency

The empirical material reported in this article consists of executed finite-state enumerations and their traces, rather than production-cluster observations or Redis benchmarks. The counts do not represent failure probabilities, and no throughput improvement or causal effect is reported.

AI Use Declaration

The author declares that no generative AI tools were used in the preparation of this manuscript.

Copyright and License

Copyright © 2026 Xiaoming Ma. This article is licensed under the Creative Commons Attribution 4.0 International License (CC BY 4.0). <https://creativecommons.org/licenses/by/4.0/>