

WHITE PAPER · DATABASE CACHING

Consistency and Invalidation: Dependency-Graph and Automated Approaches

Knowing exactly what to invalidate when data changes

White Paper 5 · Database Caching series · Version 1.0 · 26 October 2026

Bassirah Technology — Innovation & R&D

Consistency and Invalidation: Dependency-Graph and Automated Approaches

White Paper 5 · Database Caching series · Bassirah Technology — Innovation & R&D · 26 October 2026

Document control

Field	Value
Series	Database Caching, paper 5 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 The problem of derived data	4
2.2 How this paper builds on the series.....	4
2.3 Scope.....	4
2.4 Terms used in this paper	5
3. Why invalidation goes wrong.....	5
3.1 Races between reads and writes	5
3.2 Lost, duplicated and reordered messages	5
3.3 Results that depend on many inputs.....	5
3.4 A taxonomy of choices	5
4. Sources of change information	6
4.1 The application's write path.....	6
4.2 The commit log and change data capture	6
4.3 Native database notifications.....	6
4.4 Validation instead of notification.....	7
4.5 Leases as a contract.....	7
5. Ordering, idempotence and backstops	7
6. Dependency tracking.....	8
6.1 Object dependency graphs	8
6.2 Query-result caches with precise invalidation	8
6.3 Automatic tagging of cached results	8
6.4 Dependencies between services	8
7. Incremental view maintenance as precise invalidation	9
7.1 From invalidation to maintenance	9

7.2 Dataflow caches: Noria and partial state	9
7.3 Incremental maintenance for rich queries: DBSP	9
7.4 Differential dataflow and streaming databases.....	9
7.5 A wire-compatible cache built on dataflow: ReadySet	9
7.6 Materialized views in databases and platforms	10
8. Lineage for derived data across systems	10
9. Consistency models for caches	10
10. Measuring consistency	10
11. Derived data in AI systems.....	11
12. Comparative analysis.....	11
13. Open challenges and future opportunities.....	12
13.1 Precision without write-path cost	12
13.2 Coverage of query classes	12
13.3 Dependencies across services and systems.....	12
13.4 Reliable change streams	12
13.5 Consistency guarantees that users can rely on	12
13.6 Continuous measurement	12
13.7 Freshness for real-time and AI workloads	12
13.8 Verification of invalidation logic	13
14. What this means for platform decisions	13
References	13

1. Abstract

Every cache holds copies of data that can change, and invalidation, knowing exactly which copies to remove or update when data changes, is where most caching errors begin. Expiration bounds how long an error lasts but does not prevent it, and explicit invalidation written by hand misses the cached results that depend on data in ways the code does not see. This paper examines how production systems and research make invalidation correct and automatic. It explains why invalidation fails: races between reads and writes, lost and reordered messages, and derived results that depend on many rows. It describes the sources of change information, from the application's write path to database commit logs, change data capture and native notifications, and the ordering and idempotence that consumers of those signals need. It then covers dependency tracking, from object dependency graphs to automatic tagging of cached results, and incremental view maintenance, in which a cache is updated from the change itself. It discusses lineage standards for derived data that crosses systems, consistency models for caches, and how to measure staleness. It closes with a qualitative comparison, open challenges and notes for platform decisions.

2. Introduction

2.1 The problem of derived data

A cache entry for a single row is easy to invalidate: when the row changes, remove the entry. Most valuable cache entries are not single rows. They are query results, aggregates, rendered pages and API responses that depend on many rows, sometimes in several tables or services. When one of those rows changes, something must determine which entries depend on it. Applications usually answer that question by hand, in code that runs on writes, and the answer is easy to get wrong: a new write path forgets to invalidate, a change made outside the application is never seen, or a cached result depends on data the developer did not consider. Engineers who build application caches report that mapping database updates to cache entries is the hardest part of the work [1].

2.2 How this paper builds on the series

Paper 1 introduced the invalidation mechanisms in their basic form: expiration, invalidation on write and its races, leases, change data capture and versioned keys. This paper examines each in depth and adds the approaches paper 1 left for later: dependency tracking, incremental view maintenance and lineage. Paper 3 described the freshness problem for semantic caches and retrieval results; Section 11 returns to it briefly.

2.3 Scope

This paper covers invalidation and consistency for caches of database data: look-aside and integrated caches, query-result caches, materialized views used as caches, and caches between services. It concentrates on peer-reviewed work and official documentation, and draws on engineering reports from large operators where they describe mechanisms in detail. Engineering blogs and preprints are identified as such in the references. Statements about products reflect documentation accessed on 26 October 2026. It does not report hit ratios, consistency percentages, latency or throughput figures; where a source reports results, we describe them qualitatively and attribute them to the authors.

2.4 Terms used in this paper

Base data is the authoritative data in the database. A **derived entry** is a cached value computed from base data. A **dependency** links a derived entry to the base data it was computed from. **Invalidation** removes a derived entry when its base data changes; **propagation** updates it instead. A **delta** is the set of changes to base data. **Staleness** is the gap, in versions or in time, between a cached value and the current base data. A **push** mechanism notifies the cache of changes; a **pull** mechanism has the cache ask whether its copy is still valid.

3. Why invalidation goes wrong

3.1 Races between reads and writes

The classic failure involves a reader and a writer. A reader misses the cache, reads the old value from the database and is delayed; meanwhile a writer updates the database and deletes the cache entry; the reader then writes the old value into the cache, where it stays until it expires. Facebook described this "stale set" problem in its memcached deployment and addressed it with leases: a miss returns a token, a delete voids outstanding tokens, and the cache refuses a set whose token is no longer valid [2]. Meta later described the same race in its caches in detail, with a delayed fill landing after an invalidation, and showed how a transient error can turn it into a lasting inconsistency [3]. The IQ framework uses two kinds of lease to give look-aside caches in front of a relational database strong consistency without changing the database [4].

3.2 Lost, duplicated and reordered messages

Invalidations travel as messages, and messages fail. Change data capture platforms typically deliver events exactly once under normal operation but at least once after crashes or network faults, so consumers see duplicates [5]. PostgreSQL's logical decoding may resend recent changes after a crash [6]. Redis keyspace notifications are delivered without acknowledgment, so events sent while a subscriber is disconnected are lost [7]. Updates and deletes can also arrive in a different order from the one in which they were committed. Section 5 describes how production systems make consumers robust to all three.

3.3 Results that depend on many inputs

A cached aggregate, join or page may depend on many rows. Invalidation by key cannot see those dependencies, so systems either over-invalidate, removing more than necessary and losing hits, or under-invalidate, leaving stale results. Section 6 describes how dependencies can be tracked explicitly, and Section 7 how they can be derived from the query itself.

3.4 A taxonomy of choices

Franklin, Carey and Livny classified transactional cache-consistency algorithms by whether they avoid or detect stale access, and by whether a change invalidates copies or propagates new values to them [8]. The same choices recur throughout this paper. Gupta and Mumick's survey of materialized view maintenance gives a second framing: a derived entry is a materialized view, and invalidation is the simplest form of view maintenance [9]. Table 1 summarizes the main failure modes and the mechanisms that address them.

Failure mode	What goes wrong	Mechanisms that address it
Stale fill after invalidation	A delayed read writes an old value back	Leases, versions compared on write, invalidation markers
Missed write path	A change made by other code or tools never invalidates	Invalidation driven by the commit log or change data capture
Lost invalidation	A message is dropped or a subscriber is offline	Durable change streams, resumable consumers, expiration as a backstop
Duplicate or reordered events	A consumer applies the same change twice or out of order	Idempotent consumers, ordering by commit position or timestamp
Hidden dependencies	A derived entry depends on rows its key does not show	Dependency tracking, incremental view maintenance, lineage
Unmeasured staleness	Inconsistency exists but is not detected	Consistency checkers, shadow comparison, staleness distributions

4. Sources of change information

4.1 The application's write path

The simplest signal is the write itself: the code that updates the database also deletes or updates the cached entry. Facebook's writers updated the database and then deleted the key, because deletes are idempotent, and deleted it in their own cluster immediately so that users saw their own writes [2]. Write-path invalidation is fast but incomplete, because any write that bypasses the code, such as another service, a migration or a manual fix, is never seen. Uber added write-path invalidation to its integrated cache only once its storage engine could return the keys each transaction modified together with its commit timestamp, which made the signal precise [10].

4.2 The commit log and change data capture

Driving invalidation from the database's own log of committed changes sees every change regardless of which code made it, and never sees uncommitted data. Facebook annotated SQL statements with the cache keys they affect and ran daemons that tailed the MySQL commit log and broadcast deletes, so an invalidation lost by a crashed web server was still replayed from the log [2]. Uber drives invalidation of its integrated cache from a service that tails the MySQL binary log [11], and LinkedIn keeps a cache tier in front of its Espresso store up to date with stream jobs that consume committed changes [12]. General-purpose platforms make the pattern available to any team. LinkedIn's Databus captured committed changes in commit order and fanned them out to caches and indexes, bootstrapping consumers from snapshots when they fell behind [13]. Netflix's DBLog interleaves the change log with chunked table reads, so a consumer can obtain both a snapshot and the ongoing stream [14]. Debezium reads the logs of PostgreSQL, MySQL and other databases and runs inside Kafka Connect, as a standalone server or embedded in an application [15], [16]; its project has shown the embedded form evicting entries from an ORM's second-level cache when rows change outside the application, while noting that the approach is asynchronous and therefore eventually consistent [17].

4.3 Native database notifications

Databases also offer notification features, each with documented limits. PostgreSQL's NOTIFY delivers messages only when the sending transaction commits and only to sessions listening at that moment; its payload is small, and its queue is bounded [18]. PostgreSQL's logical

decoding is the durable alternative, with replication slots that persist the change stream, at the cost that an abandoned slot retains log data and can exhaust disk [6]. MySQL's binary log records changes in commit order but is a log to tail, not a notification API, and its retention settings determine how far a consumer can fall behind [19]. MongoDB's change streams deliver majority-committed changes and can resume only while the operation log still holds their position [20]. Redis offers two features: client-side caching, in which the server tracks which keys each client has read and pushes invalidations when they change [21], and keyspace notifications, which are disabled by default, are not acknowledged and, in a cluster, are local to each node, making them a hint rather than a reliable log [7].

4.4 Validation instead of notification

A cache can also ask. HTTP separates freshness, during which a cache serves a stored response without asking, from validation, in which it sends a conditional request with a validator such as an entity tag and reuses its copy if the origin confirms it is unchanged [22], [23]. Validation trades a round trip for certainty and needs no notification channel; push invalidation avoids the round trip but needs a reliable channel. Many systems combine the two.

4.5 Leases as a contract

Gray and Cheriton introduced leases as time-limited grants during which a server promises to notify a cache holder before a conflicting write, so that staleness is bounded and an unreachable holder simply lets its lease expire [24]. Memcache leases and IQ leases adapt the idea to look-aside caches [2], [4]. Table 2 compares the signals in this section.

Signal	Completeness	Durability	Typical delay	Main caveat
Application write path	Only writes made by instrumented code	None beyond the request	Immediate	Misses changes made elsewhere
Commit log or change data capture	Every committed change	Durable and resumable	Short, asynchronous	At-least-once delivery; consumers must be idempotent
PostgreSQL NOTIFY	Transactions that call it	Not replayed for disconnected listeners	Immediate after commit	Small payload; bounded queue
Redis client-side tracking	Keys the client has read	Connection-bound	Immediate	Flush the local cache if the connection drops
Redis keyspace notifications	Key events on each node	Not acknowledged	Immediate	Lost while disconnected; node-local in clusters
HTTP validation	Per request, on demand	Not applicable	One round trip	Cost of validation requests
Expiration	All entries, eventually	Not applicable	Up to the time to live	Bounds staleness; does not prevent it

5. Ordering, idempotence and backstops

Production systems that drive invalidation from change streams converge on the same rules. Make consumers idempotent, because events can repeat [5], [6]. Order changes by a

version, commit timestamp or sequence number rather than by arrival: Uber compares row timestamps inside an atomic script so that a racing fill and invalidation resolve to the newer value [11], and LinkedIn orders updates by a per-partition commit sequence number with last-writer-wins [12]. Prefer markers to deletes, so that a late fill cannot bring back a value that was invalidated: Uber writes invalidation markers [10], and LinkedIn writes tombstones for deletes [12]. Keep a time to live as a backstop for anything the other mechanisms miss; Uber notes explicitly that expiration bounds worst-case staleness while change data capture only shortens it [10], and LinkedIn uses a finite time to live to purge entries whose deletes were lost [12]. Finally, layer mechanisms: Uber combines write-path invalidation, a change-log tailer and expiration, ordered by commit timestamps [10].

6. Dependency tracking

6.1 Object dependency graphs

Challenger, Iyengar and Dantzig introduced data update propagation for dynamic web content: a graph records which cached objects depend on which underlying data, and when data changes, traversing the graph finds exactly the objects to invalidate or update [25]. They deployed it for the official web site of the 1998 Olympic Winter Games and credited it with much better cache performance than their earlier approach [25]. The idea is general: if dependencies are recorded when entries are computed, invalidation becomes a graph lookup rather than a guess.

6.2 Query-result caches with precise invalidation

Query-result caches apply dependency tracking to SQL. DBProxy cached the results of overlapping, changing queries at the edge of the network as materialized views over the origin database [26]. Ferdinand, a cooperative query-result cache, distributed its consistency management over a publish-and-subscribe system, so that updates reached exactly the proxies caching affected results [27].

6.3 Automatic tagging of cached results

Several systems remove invalidation code from applications by deriving dependencies automatically. TxCache caches the results of application functions, tags each result with the database data its queries touched, receives invalidations for those tags from the database, and gives read-only transactions a consistent snapshot across cache and database [28]. CacheGenie offers caching abstractions for common ORM query patterns and uses database triggers to invalidate or update cached data, without changes to the database or cache server [29]. Cachematic lets developers mark functions as cacheable and manages the dependency tree automatically; its industrial case study found faster reads but markedly slower writes, showing that automated invalidation moves cost onto the write path [30]. A recent preprint proposes making invalidation transparent by having the database supply the information the cache needs and keeping an invalidation index in the cache [1].

6.4 Dependencies between services

In microservice applications, a cached response may depend on state held by several services. MuCache adds caches between services and keeps them consistent with a non-blocking invalidation protocol designed for graph-shaped call topologies; its authors prove that any execution observed with the caches could also have occurred in the original

application without them [31]. Pegasus takes a related approach inside storage: a coherence directory tracks which servers hold copies of hot objects, so writes can be kept consistent while reads are spread across copies [32].

7. Incremental view maintenance as precise invalidation

7.1 From invalidation to maintenance

Invalidation discards a derived entry and recomputes it on the next miss. View maintenance updates the entry from the change itself, so the cache stays warm and correct. Gupta and Mumick's survey classifies the maintenance problem by the class of views, the information available and the kinds of change [9]. For a cache, the attraction is that the dependency set is derived from the query, so no developer has to state it.

7.2 Dataflow caches: Noria and partial state

Noria compiles an application's parameterized queries into a dataflow graph; writes enter at base tables and flow through filters, joins and aggregates to reader nodes that hold cached results, so only results that depend on the changed rows are touched, and they are updated in place [33]. Its partially stateful design lets parts of the state be evicted and rebuilt on demand, and lets the system adapt to query changes online [33]. Gjengset's thesis frames incrementally maintained views explicitly as a query-result cache, in which missing entries are filled on demand through the dataflow and races between updates and these fills are handled to keep the cache eventually consistent [34].

7.3 Incremental maintenance for rich queries: DBSP

DBSP defines incremental view maintenance mathematically, as computation over streams of changes, and gives an algorithm that incrementalizes any program in its language; relational algebra, aggregation, nested relations and recursion can all be expressed, so SQL and Datalog views can be maintained automatically [35]. An extended version appeared in a journal [36], and the Feldera engine builds on it, updating SQL views by processing only changes and documenting view state that matches what a batch computation over the same input would produce [37].

7.4 Differential dataflow and streaming databases

Differential dataflow extended incremental computation to nested iteration in a data-parallel setting, so that even iterative results can be updated as inputs change [38]. Materialize keeps SQL results current by updating indexed and materialized views incrementally as data arrives from databases and streams; its default isolation is strict serializability, and it offers a bounded-staleness option that serves reads within a stated bound [39].

7.5 A wire-compatible cache built on dataflow: ReadySet

ReadySet applies the Noria approach as a cache in front of PostgreSQL or MySQL: the application changes its connection string, selected queries are cached and others pass through, ReadySet consumes the database's replication stream, and changes flow through a dataflow graph so that cached results are maintained incrementally [40]. It stores results only for requested parameters and evicts under a memory limit, and its documentation describes it as eventually consistent, with a short delay before a write appears in the cache [40]. It is source-available under the Business Source License, with terms set by its license file [41].

7.6 Materialized views in databases and platforms

Database-native materialized views vary in how precisely they are maintained. PostgreSQL's built-in materialized views are refreshed by re-running their query and replacing the contents, with an option to keep the view readable during refresh [42]. The `pg_ivm` extension adds incrementally maintained views that apply only changes, within the same transaction as the base-table change, for a defined subset of SQL [43]. Snowflake's dynamic tables are refreshed in dependency order toward a declared target lag, incrementally when the query allows it [44], and Databricks materialized views choose between incremental and full refresh with a cost model that depends on change tracking in the source tables [45]. Precise maintenance is always bounded by the class of queries a system can maintain incrementally, and it shifts work onto the write path [30], [43].

8. Lineage for derived data across systems

Many derived results cross system boundaries: a warehouse table feeds a dashboard, a feature store or a retrieval index, and no single engine sees all the queries. Lineage metadata supplies the dependency graph there. OpenLineage defines an open event model in which jobs consume and produce datasets during runs, with a facet that maps each output column to the input fields it depends on [46]. DataHub records table-level and column-level lineage and provides impact analysis that lists everything downstream of a table or column [47]. Provenance semirings give the theory: a query result can carry an annotation recording exactly which input tuples it was derived from [48]. Lineage is metadata rather than a trigger: a team must still connect lineage events to refresh or invalidation actions, and lineage is only as complete as what systems emit [47].

9. Consistency models for caches

The guarantee a cache provides is a design choice, and stating it explicitly helps more than any single mechanism. Caches kept current by asynchronous invalidation are eventually consistent: given no new writes, cached values converge on the database, as the Debezium project notes for invalidation driven by change data capture [17]. Read-your-writes, a guarantee users notice directly, can be added per session: Facebook's writers invalidated their own cluster immediately [2]; TAO provided read-after-write consistency within a cache tier [49]; and FlightTracker provides read-your-writes across many caches and replicas by tracking each user's recent writes rather than imposing one global protocol [50]. Stronger guarantees are possible at a cost: TxCache gives transactions a consistent snapshot across cache and database [28], and IQ leases provide strong consistency for a look-aside cache [4]. Some systems make staleness a declared contract instead, such as a target lag for derived tables [44] or a bounded-staleness isolation level [39]. For real-time applications, Mao et al. argue that expiration cannot cheaply provide the tight freshness required and propose choosing adaptively between invalidating and updating cached entries [51].

10. Measuring consistency

Consistency is rarely measured, and systems that measure it find problems that tests did not. Lu et al. measured how often Facebook's eventually consistent, cache-fronted store violated

stronger consistency models, using sampled request logs, and found anomalies rare but real [52]. Meta's Polaris checker consumes the same invalidation events as the cache, queries cache replicas for versions older than each event, and rechecks over several timescales so that replication lag is not counted as an error; Meta reports the share of cache writes that become consistent within a time window [3]. Uber compares cache contents with the database by replaying its change stream on a delay and exporting staleness histograms and mismatch rates per table; it used the results to justify longer expiration times once write-path invalidation was in place [10]. Probabilistically Bounded Staleness provides the conceptual tool: staleness is measured in versions and in time and reported as a probability that a read reflects the latest write after a given delay, not as a yes-or-no property [53]. For caches, the lesson is to report a staleness distribution against the source of truth, including its tail.

11. Derived data in AI systems

Retrieval indexes, embeddings and agent context are derived data, and they inherit the invalidation problem. Paper 3 described freshness for semantic caches. Recent preprints apply the ideas of this paper to AI data: one monitors source pages and re-indexes only content whose meaning changed, rather than rebuilding a retrieval index periodically [54], and another adapts cache-coherence protocols to shared state in multi-agent systems [55]. Both are early and unreviewed, but they suggest that dependency tracking and incremental maintenance will matter as much for AI pipelines as for database caches.

12. Comparative analysis

Table 3 compares the approaches in this paper. The ratings are qualitative and relative to each other; actual behavior depends on the implementation, workload and configuration.

Approach	Precision	Freshness	Write-path cost	Coverage of changes	Operational effort
Expiration only	Low	Bounded by the time to live	None	All entries, eventually	Low
Write-path invalidation by key	Medium	Immediate for instrumented writes	Low	Instrumented code only	Low to Medium
Commit-log or change-data-capture invalidation	Medium (by key)	Short delay	None on the application	All committed changes	Medium (pipeline to operate)
Leases with invalidation	Medium	Immediate	Low	Instrumented code or log	Medium
Dependency tracking and automatic tagging	High	Immediate to short delay	Medium to High	Queries the system observes	Medium
Incremental view maintenance (dataflow, DBSP, pg_ivm)	Very high (per result)	Short delay or same transaction	Medium to High	Supported query classes	Medium to High
Lineage-driven refresh across systems	Medium (per dataset)	Depends on refresh schedule	None on sources	Whatever systems emit	Medium

13. Open challenges and future opportunities

13.1 Precision without write-path cost

Challenge. Precise invalidation and incremental maintenance move work onto writes [30], [43]. **Research direction.** Asynchronous maintenance from change streams [40], [37], partial state that maintains only results in demand [33], [34]. **Opportunity.** Systems that keep derived results exact while leaving the write path fast would make precise caching the default.

13.2 Coverage of query classes

Challenge. Incremental maintenance supports only some queries, and the supported set differs by system [42], [43], [44]. **Research direction.** General incrementalization of rich query languages [35] and differential computation for iterative queries [38]. **Opportunity.** Maintenance that covers the queries applications actually run would remove the need for hand-written invalidation.

13.3 Dependencies across services and systems

Challenge. Derived data crosses services, warehouses and AI pipelines that no single engine sees [31], [46]. **Research direction.** Invalidation protocols for service graphs [31], lineage standards with column-level dependencies [46], [47], and provenance as a dependency record [48]. **Opportunity.** Connecting lineage events to invalidation would let organizations keep derived data fresh across system boundaries.

13.4 Reliable change streams

Challenge. Change signals can be lost, duplicated or delayed [5], [18], [7]. **Research direction.** Durable, resumable change capture with snapshot integration [14], [13] and storage engines that return modified keys and commit timestamps [10]. **Opportunity.** Change streams with clear delivery guarantees would simplify every consumer.

13.5 Consistency guarantees that users can rely on

Challenge. Many caches offer only eventual consistency, and stronger guarantees cost latency or complexity [50], [28], [4]. **Research direction.** Session guarantees layered over eventually consistent caches [50], transactional caches [28], and declared staleness contracts [39], [44]. **Opportunity.** Caches that state and meet a guarantee per use case would let teams choose the right trade-off explicitly.

13.6 Continuous measurement

Challenge. Inconsistency that is not measured is not fixed [52]. **Research direction.** Black-box consistency checkers [3], shadow comparison against change streams [10], and probabilistic staleness models [53]. **Opportunity.** Staleness reported as a service-level indicator, beside latency and errors, would make cache correctness observable.

13.7 Freshness for real-time and AI workloads

Challenge. Real-time applications need tighter freshness than expiration can provide cheaply [51], and AI pipelines derive indexes and context from changing sources [54]. **Research direction.** Adaptive choice between invalidation and update [51] and change-driven maintenance of AI artifacts [54], [55]. **Opportunity.** Applying the techniques of this paper to retrieval indexes and agent state would reduce wrong answers caused by stale context.

13.8 Verification of invalidation logic

Challenge. Invalidation code is concurrent and rarely tested under failure, and its bugs appear only as rare stale reads [3]. **Research direction.** Formal guarantees for invalidation protocols [31] and tracing of race windows [3]. **Opportunity.** Verified or systematically tested invalidation would turn one of caching's most common sources of incidents into an engineered component.

14. What this means for platform decisions

Decide first what consistency each cached dataset needs and write it down. For simple keyed entries, drive invalidation from committed changes rather than from scattered application code, make consumers idempotent and ordered by commit position or timestamp, use markers rather than deletes where late fills are possible, and keep a time to live as a backstop. For derived results, prefer approaches that derive dependencies from the query, such as automatic tagging, incremental view maintenance or a dataflow cache, over hand-written invalidation, and check which queries each tool can maintain. For data that crosses systems, use lineage to find what must be refreshed. Measure staleness against the source of truth continuously.

For in-memory caching, Bassirah works with Redis as a Strategic Technology Partner; Redis client-side caching provides server-assisted invalidation for local caches, and keyspace notifications can serve as hints alongside a durable change stream. Other products and projects named in this paper are ecosystem references. In each case, validate invalidation behavior under concurrency and failure, on representative workloads and versions, before relying on it.

References

1. Ji, Y., Zhou, X., Zhou, Y. and Wang, K. (2023). The Case of Transparent Cache Invalidation in Web Applications. arXiv preprint arXiv:2311.02384. (<https://arxiv.org/abs/2311.02384>)
2. 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. 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), pp. 385–398. (<https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala>)
3. Pan, L. (2022). Cache made consistent. Engineering at Meta (blog), 8 June 2022. (<https://engineering.fb.com/2022/06/08/core-infra/cache-made-consistent/>)
4. Ghandeharizadeh, S., Yap, J. and Nguyen, H. (2014). Strong consistency in cache augmented SQL systems. Proceedings of the 15th International Middleware Conference (Middleware '14), pp. 181–192. (<https://doi.org/10.1145/2663165.2663318>)
5. Debezium Project (2026). Frequently Asked Questions. Debezium Documentation. Official documentation, accessed 26 October 2026. (<https://debezium.io/documentation/faq/>)
6. PostgreSQL Global Development Group (2026). Logical Decoding Concepts. PostgreSQL 18 Documentation, Chapter 47 (Logical Decoding). Official documentation, accessed 26 October 2026. (<https://www.postgresql.org/docs/current/logicaldecoding-explanation.html>)

7. Redis Ltd. (2026). Redis keyspace notifications. Redis Documentation (Develop > Pub/Sub). Official documentation, accessed 26 October 2026. (<https://redis.io/docs/latest/develop/pubsub/keyspace-notifications/>)
8. Franklin, M. J., Carey, M. J. and Livny, M. (1997). Transactional client-server cache consistency: alternatives and performance. *ACM Transactions on Database Systems*, 22(3), pp. 315–363. (<https://doi.org/10.1145/261124.261125>)
9. Gupta, A. and Mumick, I. S. (1995). Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2), pp. 3–18. (<http://sites.computer.org/debull/95JUN-CD.pdf>)
10. Pozniansky, E. and Narayanareddy, P. (2025). How Uber Serves over 150 Million Reads per Second from Integrated Cache with Stronger Consistency Guarantees. *Uber Engineering Blog*, 26 August 2025. (<https://www.uber.com/us/en/blog/how-uber-serves-over-150-million-reads/>)
11. Narayanareddy, P., Pozniansky, E. and Kutsia, Z. (2024). How Uber Serves Over 40 Million Reads Per Second from Online Storage Using an Integrated Cache. *Uber Engineering Blog*, 15 February 2024. (<https://www.uber.com/us/en/blog/how-uber-serves-over-40-million-reads-per-second-using-an-integrated-cache/>)
12. Pham, E. and Lu, G. (2023). Upscaling LinkedIn's Profile Datastore While Reducing Costs. *LinkedIn Engineering Blog*, 9 May 2023. (<https://www.linkedin.com/blog/engineering/data-management/upscaling-profile-datastore-while-reducing-costs>)
13. Das, S., Botev, C., Surlaker, K., Ghosh, B., Varadarajan, B., Nagaraj, S., Zhang, D., Gao, L., Westerman, J., Ganti, P. et al. (2012). All aboard the Databus!: LinkedIn's scalable consistent change data capture platform. *Proceedings of the Third ACM Symposium on Cloud Computing (SoCC '12)*, pp. 1–14. (<https://doi.org/10.1145/2391229.2391247>)
14. Andreakis, A. and Papapanagiotou, I. (2020). DBLog: A Watermark Based Change-Data-Capture Framework. *arXiv preprint arXiv:2010.12597*. (<https://arxiv.org/abs/2010.12597>)
15. Debezium Project (2026). Debezium Architecture. *Debezium Documentation* (stable release line at time of access). Official documentation, accessed 26 October 2026. (<https://debezium.io/documentation/reference/stable/architecture.html>)
16. Apache Software Foundation (2026). Kafka Connect: Overview. *Apache Kafka Documentation* (4.4 docs tree). Official documentation, accessed 26 October 2026. (<https://kafka.apache.org/44/kafka-connect/overview/>)
17. 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/>)
18. PostgreSQL Global Development Group (2026). NOTIFY. *PostgreSQL 18 Documentation*, SQL Commands. Official documentation, accessed 26 October 2026. (<https://www.postgresql.org/docs/current/sql-notify.html>)
19. Oracle Corporation (2026). The Binary Log. *MySQL 8.4 Reference Manual*, Section 7.4.4. Official documentation, accessed 26 October 2026. (<https://dev.mysql.com/doc/refman/8.4/en/binary-log.html>)
20. MongoDB, Inc. (2026). Change Streams. *MongoDB Manual* (current). Official documentation, accessed 26 October 2026. (<https://www.mongodb.com/docs/manual/changeStreams/>)

21. Redis Ltd. (2026). Client-side caching reference. Redis Documentation (Develop > Reference). Official documentation, accessed 26 October 2026. (<https://redis.io/docs/latest/develop/reference/client-side-caching/>)
22. Fielding, R., Nottingham, M. and Reschke, J. (eds.) (2022). HTTP Semantics. RFC 9110 (STD 97), Internet Engineering Task Force, June 2022. (<https://doi.org/10.17487/RFC9110>)
23. Fielding, R., Nottingham, M. and Reschke, J. (eds.) (2022). HTTP Caching. RFC 9111 (STD 98), Internet Engineering Task Force, June 2022. (<https://doi.org/10.17487/RFC9111>)
24. 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), pp. 202–210. (<https://doi.org/10.1145/74850.74870>)
25. Challenger, J., Iyengar, A. and Dantzig, P. (1999). A scalable system for consistently caching dynamic Web data. IEEE INFOCOM '99, vol. 1, pp. 294–303. (<https://doi.org/10.1109/INFCOM.1999.749295>)
26. Amiri, K., Park, S., Tewari, R. and Padmanabhan, S. (2003). DBProxy: a dynamic data cache for web applications. Proceedings of the 19th International Conference on Data Engineering (ICDE 2003), pp. 821–831. (<https://doi.org/10.1109/ICDE.2003.1260881>)
27. Garrod, C., Manjhi, A., Ailamaki, A., Maggs, B., Mowry, T., Olston, C. and Tomasic, A. (2008). Scalable query result caching for web applications. Proceedings of the VLDB Endowment, 1(1), pp. 550–561. (<https://doi.org/10.14778/1453856.1453917>)
28. Ports, D. R. K., Clements, A. T., Zhang, L., Madden, S. and Liskov, B. (2010). Transactional consistency and automatic management in an application data cache. 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10), pp. 279–292. (<https://www.usenix.org/conference/osdi10/transactional-consistency-and-automatic-management-application-data-cache>)
29. Gupta, P., Zeldovich, N. and Madden, S. (2011). A Trigger-Based Middleware Cache for ORMs. In Middleware 2011, Lecture Notes in Computer Science, vol. 7049, Springer, pp. 329–349. (https://doi.org/10.1007/978-3-642-25821-3_17)
30. Holmqvist, V., Nilsfors, J. and Leitner, P. (2019). Cachematic - Automatic Invalidation in Application-Level Caching Systems. Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering (ICPE '19), pp. 167–178. (<https://doi.org/10.1145/3297663.3309666>)
31. Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S. and Liu, V. (2024). MuCache: A General Framework for Caching in Microservice Graphs. 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24), pp. 221–238. (<https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran>)
32. Li, J., Nelson, J., Michael, E., Jin, X. and Ports, D. R. K. (2020). Pegasus: Tolerating Skewed Workloads in Distributed Storage with In-Network Coherence Directories. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 387–406. (<https://www.usenix.org/conference/osdi20/presentation/li-jialin>)
33. Gjengset, J., Schwarzkopf, M., Behrens, J., Araújo, L. T., Ek, M., Kohler, E., Kaashoek, M. F. and Morris, R. (2018). Noria: dynamic, partially-stateful data-flow for high-performance web applications. 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18), pp. 213–231. (<https://www.usenix.org/conference/osdi18/presentation/gjengset>)
34. Gjengset, J. F. R. (2021). Partial state in dataflow-based materialized views. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science. (<https://hdl.handle.net/1721.1/130767>)

35. Budiu, M., Chajed, T., McSherry, F., Ryzhyk, L. and Tannen, V. (2023). DBSP: Automatic Incremental View Maintenance for Rich Query Languages. Proceedings of the VLDB Endowment, 16(7), pp. 1601–1614. (<https://doi.org/10.14778/3587136.3587137>)
36. Budiu, M., Ryzhyk, L., Zellweger, G., Pfaff, B., Suresh, L., Kassing, S., Gyawali, A., Budiu, M., Chajed, T., McSherry, F. et al. (2025). DBSP: automatic incremental view maintenance for rich query languages. The VLDB Journal, 34(4), article 39. (<https://doi.org/10.1007/s00778-025-00922-y>)
37. Feldera, Inc. (2026). Feldera documentation: introduction and Literature/Publications pages. Official documentation, accessed 26 October 2026. (<https://docs.feldera.com/> ; <https://docs.feldera.com/literature/papers/>)
38. McSherry, F., Murray, D. G., Isaacs, R. and Isard, M. (2013). Differential dataflow. 6th Biennial Conference on Innovative Data Systems Research (CIDR 2013). (https://www.cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf)
39. Materialize, Inc. (2026). Materialize documentation: Views concept page and Isolation levels page; MaterializeInc/materialize repository README. Official documentation, accessed 26 October 2026. (<https://materialize.com/docs/fundamentals/concepts/views/> ; <https://materialize.com/docs/serve-results/isolation-level/> ; <https://github.com/MaterializeInc/materialize>)
40. Redyset, Inc. (2026). Redyset documentation: Redyset Platform overview and "Deep caching" concepts page. Official documentation, accessed 26 October 2026. (<https://redyset.io/docs/> ; <https://redyset.io/docs/concepts/deep-caching>)
41. Redyset, Inc. (2026). redysettech/readysset GitHub repository: README and LICENSE. Official repository, accessed 26 October 2026. (<https://github.com/readyssettech/readysset>)
42. PostgreSQL Global Development Group (2026). REFRESH MATERIALIZED VIEW. PostgreSQL 18 Documentation. Official documentation, accessed 26 October 2026. (<https://www.postgresql.org/docs/current/sql-refreshmaterializedview.html>)
43. SRA OSS (2026). pg_ivm: Incremental View Maintenance for PostgreSQL. GitHub repository README. Official documentation, accessed 26 October 2026. (https://github.com/sraoss/pg_ivm)
44. Snowflake Inc. (2026). Dynamic tables (About dynamic tables). Snowflake Documentation. Official documentation, accessed 26 October 2026. (<https://docs.snowflake.com/en/user-guide/dynamic-tables-about>)
45. Databricks, Inc. (2026). Use standalone materialized views. Databricks on AWS documentation. Official documentation, accessed 26 October 2026. (<https://docs.databricks.com/aws/en/views/materialized>)
46. OpenLineage Project (2026). OpenLineage specification: Object Model, and Column Level Lineage Dataset Facet. Official documentation, accessed 26 October 2026. (<https://openlineage.io/docs/spec/object-model> ; https://openlineage.io/docs/spec/facets/dataset-facets/column_lineage_facet)
47. DataHub Project (2026). About DataHub Lineage. DataHub documentation. Official documentation, accessed 26 October 2026. (<https://docs.datahub.com/docs/features/feature-guides/lineage>)
48. Green, T. J., Karvounarakis, G. and Tannen, V. (2007). Provenance semirings. Proceedings of the 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '07), pp. 31–40. (<https://doi.org/10.1145/1265530.1265535>)

49. 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. 2013 USENIX Annual Technical Conference (USENIX ATC 13), pp. 49–60.
(<https://www.usenix.org/conference/atc13/technical-sessions/presentation/bronson>)
50. 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. 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), pp. 407–423.
(<https://www.usenix.org/conference/osdi20/presentation/shi>)
51. 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), pp. 335–342. (<https://doi.org/10.1145/3696348.3696858>)
52. Lu, H., Veeraraghavan, K., Ajoux, P., Hunt, J., Song, Y. J., Tobagus, W., Kumar, S. and Lloyd, W. (2015). Existential consistency: measuring and understanding consistency at Facebook. Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15), pp. 295–310. (<https://doi.org/10.1145/2815400.2815426>)
53. Bailis, P., Venkataraman, S., Franklin, M. J., Hellerstein, J. M. and Stoica, I. (2012). Probabilistically bounded staleness for practical partial quorums. Proceedings of the VLDB Endowment, 5(8), pp. 776–787. (<https://doi.org/10.14778/2212351.2212359>)
54. Zerhoudi, S., Dinzinger, M., Granitzer, M. and Mitrovic, J. (2026). OwlerLite: Scope- and Freshness-Aware Web Retrieval for LLM Assistants. arXiv preprint arXiv:2601.17824.
(<https://arxiv.org/abs/2601.17824>)
55. Parakhin, V. (2026). Token Coherence: Adapting MESI Cache Protocols to Minimize Synchronization Overhead in Multi-Agent LLM Systems. arXiv preprint arXiv:2603.15183.
(<https://arxiv.org/abs/2603.15183>)

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.