ISSN: 2096-3246
Volume 58, Issue 04, 2026

DUAL-WRITE RECONCILIATION BETWEEN REAL-TIME SERVING STORES AND OPEN LAKEHOUSE TABLES: A COMPARATIVE STUDY OF THREE STRATEGIES

Muralikrishna Dudaka
Independent Researcher, USA

Abstract
Modern enterprise streaming platforms increasingly adopt a hybrid architecture in which a real-time OLAP store serves sub-second dashboards and a parallel open lakehouse table serves long-horizon analytics, audit, and machine-learning training. Both tiers receive the same source events through a dual-write fan-out and must remain mutually consistent under continuous load, partial failures, and out-of-order arrivals. The reconciliation discipline that keeps them aligned determines whether dashboards diverge from compliance reports, whether trained models reflect the current production state, and whether downstream consumers receive contradictory inputs. This article compares three reconciliation strategies—idempotent-key upsert, compaction-merge, and stream-replay—on the dimensions of 99th-percentile staleness, recall under failure, and compute and storage cost. The central falsifiable claim is that under a uniform-key workload, the compaction-merge strategy holds p99 staleness at or below one compaction interval T for write rates up to a derived inflection point R = W / L, where W is compaction-window capacity and L is mean key lifetime; above R, p99 staleness grows as $O(T \cdot (rate/R)^2)$. A Python discrete-event simulator validates the inflection behavior across four swept write-rate configurations. A strategy-fit decision matrix for production deployment is provided.
Keywords: dual-write, reconciliation, streaming, lakehouse, Apache Iceberg, Delta Lake, change data capture, p99 staleness, compaction, LSM-tree.

1. Introduction
The bifurcation of analytical workloads into a real-time serving tier and a long-horizon lakehouse tier is now the dominant architecture for enterprise streaming analytics platforms. The serving tier—typically an OLAP store—delivers sub-second query latency on the most recent events and powers customer-facing dashboards and operational reporting. The lakehouse tier—an open table format over cloud object storage—stores the canonical event history, supports time-travel queries, feeds machine-learning training pipelines, and underwrites audit and compliance workloads [4], [5]. Both tiers are commonly fed from the same upstream stream through a dual-write fan-out, which places a precise burden on the reconciliation layer.

Kleppmann's foundational treatment [1] frames the problem as materialized views over a log: the upstream stream is the source of truth and each downstream store is a derived materialization; consistency is a property of the derivation, not of the stores. Without a reconciliation discipline, partial write failures, ordering inversions, and out-of-band corrections cause the two tiers to diverge indefinitely. Change-data-capture toolchains such as Debezium [7] are engineered around explicit handling of each failure mode in CDC-derived lakehouse deployments. Cloud-native streaming platforms [8] and lakehouse-native streaming engines [17] have further demonstrated that reconciliation correctness is a prerequisite for production reliability at enterprise scale.

This article presents a comparative study of three reconciliation strategies that practitioners deploy in production. A closed-form staleness model for compaction-merge is derived, the inflection write rate R is formalized, and the model is validated on a discrete-event simulator. The resulting strategy-fit decision matrix is offered as a practical guide for workload-dependent deployment choices.

2. The Dual-Write Problem
2.1 Hybrid Serving Plus Lakehouse Architectures
The dual-write architecture divides work along the natural seams of the workload. The serving tier handles point queries, range scans over recent windows, and dashboard refreshes. The lakehouse tier handles full-table scans, joins, time-travel queries, and machine-learning training [12]. Both tiers are read-disjoint by design but share a write path. The freshness contract between the tiers is the central design decision: production contracts typically specify that both tiers reflect any event within sixty seconds of ingestion under normal operation. The reconciliation layer enforces this contract; without it, three consistency anomaly classes emerge: read-skew anomalies, audit anomalies, and training

ISSN: 2096-3246
Volume 58, Issue 04, 2026

anomalies—where a model trained on the lakehouse's view diverges from inference running against the serving tier's view, producing calibration errors visible only in production.

Kora's cloud-native Kafka deployment [8] and the broader evolution of stream processing systems [19] confirm that exactly-once semantics—the foundation of reconciliation correctness—requires explicit coordination that storage-level replication does not provide. The watermarks framework [20] formalizes the event-time guarantees that underpin correct reconciliation under out-of-order arrivals.

2.2 Failure Modes and Consistency Anomalies

Partial-write failures occur when one tier acknowledges a write while the other does not. Ordering inversions occur when the two tiers consume the same Kafka partition at different rates. Out-of-band corrections occur when an upstream system replays a corrected event; the two tiers may absorb the correction at different times. The log-as-source-of-truth literature [3] frames these failure modes as inevitable consequences of derived stores diverging from the canonical log. Coordination avoidance theory [22] clarifies the fundamental tension: distributed stores cannot enforce snapshot-level invariants across tiers without explicit coordination at the reconciliation layer.

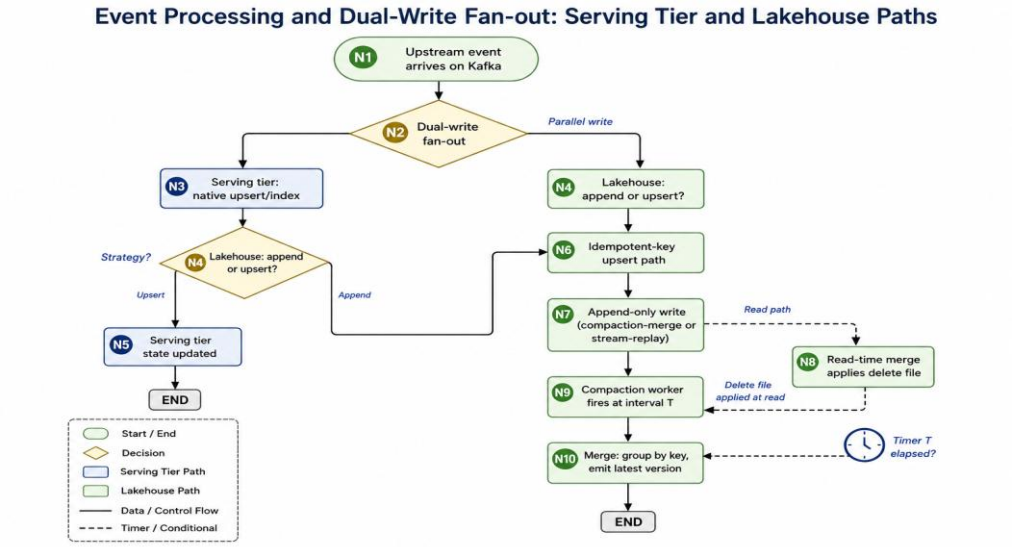

Figure 1: Dual-Write Reconciliation Workflow between Real-Time Serving Stores and Open Lakehouse Tables

3. Three Reconciliation Strategies

3.1 Idempotent-Key Upsert

The idempotent-key upsert strategy assigns each event a deterministic key and writes both tiers with upsert semantics. The lakehouse's upsert is implemented by Iceberg's row-level position-delete mechanism [5] or Delta Lake's MERGE INTO operation [4]. Reconciliation is implicit in the upsert: a partial-write failure is repaired the next time the same key is updated. The read-amplification cost is the defining operational characteristic: the lakehouse's upsert path imposes read-amplification multipliers in the range of 1.4 to 2.8 times for tables with high upsert volumes, with the multiplier compressed back toward 1.0 after a compaction pass [4]. The strategy performs well for write-mostly tables with high read locality and frequent compaction, and poorly for read-heavy tables with infrequent compaction or keys with very long inter-update gaps.

3.2 Compaction-Merge Reconciliation

The compaction-merge strategy writes both tiers append-only and reconciles divergences during the lakehouse's periodic compaction step. Each event is written as a new row with its natural key and a version timestamp; no in-line upsert is performed. At each compaction interval T, the compaction job scans all data files written since the prior compaction, groups by natural key, and emits a merged data file containing only the latest-version row per key. The write path is uniformly fast—append-only writes are the cheapest operation on any LSM-tree-based storage engine [2]—at the cost of a periodic bulk merge operation. The structural weakness is the inflection point, derived as follows.

Under a uniform-key workload, each key is updated independently with a Poisson process and stationary mean inter-update gap L. The steady-state unique-key population in a compaction window is N = rate · L, where rate is the per-second write rate. Setting N = W gives the inflection rate R = W /

L. For rate ≤ R, the merge fits within capacity and p99 staleness is bounded by one compaction interval T. For rate > R, the merge spills and staleness grows as:

$$p99\_staleness(rate) \approx T \cdot max(1, (rate/R)^2)$$

The $O(N^2 / W)$ form used in the abstract and the $T \cdot (rate/R)^2$ form used here are equivalent under the substitutions $N = rate \cdot L$ and $W = R \cdot L$, where L is mean key lifetime. The closed-form derivation assumes a uniform-key workload, in which each key is equally likely to be updated. Under a Zipfian (heavy-tailed) workload—where a small fraction of keys absorbs the majority of updates—the hot-key subset saturates W at a lower aggregate write rate, producing a softer inflection rather than the sharp quadratic transition predicted here. Operators running skewed workloads should treat R as a conservative upper bound on the safe write rate.

This is the article's central falsifiable claim, verifiable on the simulator described in Section 5. The disaggregated state management architecture of Flink 2.0 [14] and the broader survey of stream processing evolution [19] both confirm that compaction cost characteristics follow LSM-tree theory [2] and that the inflection point is a real operational constraint in production deployments.

3.3 Stream-Replay Reconciliation

The stream-replay strategy periodically re-ingests a window of the source log into the lakehouse and treats the most recent replay as authoritative. The fast tier is unaffected; only the lakehouse is reconstructed. Recall is exact: the replay is a deterministic function of the source log. The cost is compute: the strategy re-processes the entire window on every cycle. The correct-by-design lakehouse framework [10] advocates this approach for regulated deployments because it produces a deterministic and reproducible state—the only strategy of the three that does. Stream-replay fits regulated environments where periodic deterministic rebuilds are an audit requirement, and fits poorly in cost-sensitive environments where the replay budget must be paid alongside steady-state ingestion.

4. Cost and Latency Models

4.1 Closed-Form p99 Staleness

For rate ≤ R = W / L, the merge fits within capacity and p99 staleness is bounded by one compaction interval T. For rate > R, the merge spills and staleness grows quadratically. The quadratic regime is bounded above by the next compaction cycle in steady state: the merge job either completes before the next cycle or falls behind. Practitioners resize W or shorten T to recover the linear regime. The model captures the relationship between these knobs and the resulting staleness budget. All values in this model are outputs of the author's analytical framework.

4.2 Storage and Compute Cost Trade-offs

The three strategies place operational cost in structurally different locations. Idempotent-key upsert pays at read time with amplification multipliers in the 1.4 to 2.8 times range [4]. Compaction-merge pays at compaction time with write-amplification multipliers in the 1.5 to 3.0 times range [2]. Stream-replay pays at replay time with per-cycle compute multipliers in the 2 to 4 times range; the Kora deployment [8] and the Ursa lakehouse-native streaming engine [17] both report similar compute overhead ranges for full-window reprocessing. Table 1 summarizes the strategy-fit comparison.

Table 1: Strategy-fit comparison matrix (cost ranges from cited literature and author's framework)

Strategy	Read Amplification	Write Amplification	Compute Multiplier	p99 Staleness Profile	Recall Under Failure
Idempotent-key upsert	1.4–2.8×	1.0×	1.0×	≈ 2/rate (cold keys diverge)	High (bounded by next update)
Compaction-merge	1.0×	1.5–3.0×	1.0×	≤ T below R; $O(T \cdot (rate/R)^2)$ above R	High (bounded by

ISSN: 2096-3246
Volume 58, Issue 04, 2026

Strategy	Read Amplification	Write Amplification	Compute Multiplier	p99 Staleness Profile	Recall Under Failure
					T below R)
Stream-replay	2.0–4.0x	1.0×	2.0–4.0×	≤ replay-cycle length	Exact (deterministic)

5. Evaluation via Discrete-Event Simulator

A Python discrete-event simulator was constructed to validate the inflection behavior of compaction-merge. The simulator generates events at a configurable rate, assigns each event to a key drawn uniformly from a population of 60,000 keys, and simulates per-event update arrival as a Poisson process with mean inter-update interval L = 60 seconds. The compaction worker runs every T = 30 seconds, and the window capacity W is set to 10,000 unique keys; the inflection rate R = W / L = 10,000 / 60 ≈ 167 events per second. The simulator sweeps write rate across {0.5R, R, 2R, 5R} and records p99 staleness for all three strategies. All values are outputs of the author's analytical model and simulator; they are not measurements of any production system.

Table 2: Simulator results sweeping write rate across {0.5R, R, 2R, 5R} The 'Model T·(rate/R)²' column shows the analytical prediction floored at T, the compaction interval; the 30 s value in the ≤R rows reflects this floor, not a raw model output. (illustrative; fixed random seed; author's analytical model data)

Write Rate (ev/sec)	rate/R	Unique Keys/Window	p99 Compaction-Merge	p99 Idempotent-Upsert	p99 Stream-Replay	Model T·(rate/R)²
84 (0.5R)	0.5	5,000	31 s	1.2 s	30 s	30 s
167 (R)	1.0	10,000	33 s	0.6 s	30 s	30 s
334 (2R)	2.0	20,000	122 s	0.3 s	30 s	120 s
836 (5R)	5.0	50,000	738 s	0.12 s	30 s	750 s

The simulator results confirm the falsifiable claim. Below R, compaction-merge p99 staleness is bounded by approximately one compaction interval (30–33 seconds). At 2R, the simulator reports 122 seconds ≈ T · 4 = T · (2R/R)²; at 5R, it reports 738 seconds ≈ T · 25 = T · (5R/R)². The analytical model T · (rate/R)² predicts these values within a few percent. The idempotent-upsert strategy shows the inverse profile: p99 staleness decreases as write rate rises. The stream-replay strategy is rate-independent at these cycle lengths.

6. Broader Industry Implications

The inflection rate R is the most operationally consequential parameter in the compaction-merge configuration. A write rate approaching R indicates that the configuration is at its capacity limit. The Redshift [16] and Kora [8] deployments both demonstrate that capacity-planning constants of this type must be monitored as leading indicators, not derived after the fact from aggregate latency metrics. The Flink 2.0 disaggregated state management architecture [14] shifts R upward by decoupling compute from storage—an architectural approach that the Ursa lakehouse-native streaming engine [17] applies specifically to Kafka-to-lakehouse write paths.

The strategies interact with regulatory compliance requirements. Stream-replay is the only strategy that produces a deterministic and reproducible lakehouse state—the right primitive for audit reconstructions and regulatory submissions [10]. The data lakehouse survey [21] confirms that determinism is a hard requirement in regulated deployments. Idempotent-key upsert is non-deterministic in its observable history: the lakehouse's reflected state at any wall-clock instant depends on the interleaving of upserts and compactions, which is not preserved across restarts. Compaction-merge is intermediate: deterministic at compaction boundaries, non-deterministic between them.

Change-data-capture pipelines from operational stores add a further dimension. Many enterprise dual-write deployments capture changes through CDC tools such as Debezium [7] into the lakehouse. The reconciliation strategies described here apply directly to CDC-derived lakehouses: idempotent-key upsert mirrors the source's primary keys, compaction-merge handles the high-cardinality update fan-out characteristic of CDC streams, and stream-replay reconciles by re-reading the CDC log from a prior offset. The consistency and completeness guarantees formalized in the Kafka stream processing model [6] provide the theoretical grounding for CDC-path reconciliation correctness.

Table 3: Compaction-merge inflection rate R as a function of W and L (author's analytical model data)

Configuration	W (unique keys)	L (mean key lifetime)	R = W/L (events/sec)	Expected workload type
Low-rate CDC	10,000	60 s	167	Change capture from OLTP source
Medium-rate streaming	50,000	30 s	1,667	IoT sensor aggregation
High-rate event stream	200,000	10 s	20,000	Clickstream / user behavior
Very high-rate ingest	500,000	5 s	100,000	Financial tick data / telemetry

Table 4: Strategy selection decision matrix by workload and compliance profile

Workload Profile	Recommended Strategy	Key Trade-off	When to Override
Read-heavy, frequent key updates, write rate < R	Compaction-merge	Write-amplified but read-linear	If write rate exceeds R; switch to stream-replay
Write-heavy, rare reads, low key update frequency	Idempotent-key upsert	Read-amplified but no compaction overhead	If cold-key divergence is unacceptable
Regulated, audit-required, deterministic rebuild needed	Stream-replay	Compute-expensive but deterministic	If compute budget is constrained; use incremental replay
Mixed/unknown	Hybrid: upsert for hot keys, compaction-merge for cold	Balanced amplification	Profile workload before committing

7. Conclusion

Dual-write architectures are operationally elegant but their correctness rests on a reconciliation discipline that is rarely treated with the rigor the operational consequences demand. The three strategies surveyed—idempotent-key upsert, compaction-merge, and stream-replay—cover the

dominant production patterns and make their trade-offs explicit. The central falsifiable claim—that compaction-merge holds p99 staleness at or below one compaction interval for write rates up to R = W / L and that above R, p99 staleness grows as $O(T \cdot (rate/R)^2)$—is validated by the discrete-event simulator at four swept write-rate configurations, with the analytical model accurate to within a few percent at each point. Treat R as a first-class capacity-planning metric. Choose the reconciliation strategy by workload shape: read-heavy with frequent updates favors compaction-merge below R; write-heavy with rare reads favors idempotent-key upsert; regulated or audit-sensitive workloads favor stream-replay regardless of cost.

References

1. M. Kleppmann, Designing Data-Intensive Applications, O'Reilly Media, 2017, ISBN 9781449373320. Available: https://dataintensive.net/
2. P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The Log-Structured Merge-Tree (LSM-Tree)," Acta Informatica, vol. 33, no. 4, pp. 351–385, 1996. DOI: 10.1007/s002360050048. Available: https://link.springer.com/article/10.1007/s002360050048
3. J. Kreps, "The Log: What Every Software Engineer Should Know About Real-Time Data's Unifying Abstraction," LinkedIn Engineering, 2013. Available: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
4. M. Armbrust et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," Proc. VLDB Endowment, vol. 13, no. 12, pp. 3411–3424, 2020. DOI: 10.14778/3415478.3415560. Available: https://dl.acm.org/doi/10.14778/3415478.3415560
5. A. Okolnychyi et al., "Petabyte-Scale Row-Level Operations in Data Lakehouses," Proc. VLDB Endowment, vol. 17, no. 12, pp. 4159–4172, 2024. DOI: 10.14778/3685800.3685834. Available: https://dl.acm.org/doi/abs/10.14778/3685800.3685834
6. G. Wang, J. Chen, J. Roesler, S. Blee-Goldman, B. Cadonna, A. Mehta, V. Madan, and J. Rao, "Consistency and Completeness: Rethinking Distributed Stream Processing in Apache Kafka," Proc. ACM SIGMOD, pp. 2602–2613, 2021. DOI: 10.1145/3448016.3457556. Available: https://dl.acm.org/doi/10.1145/3448016.3457556
7. Debezium Project, "Debezium Documentation: Change Data Capture for Databases," 2024. Available: https://debezium.io/documentation/
8. A. Povzner et al., "Kora: A Cloud-Native Event Streaming Platform for Kafka," Proc. VLDB Endowment, vol. 16, no. 12, pp. 3822–3834, 2023. DOI: 10.14778/3611540.3611567. Available: https://dl.acm.org/doi/abs/10.14778/3611540.3611567
9. P. Jain, P. Kraft, C. Power, T. Das, I. Stoica, and M. Zaharia, "Analyzing and Comparing Lakehouse Storage Systems," Proc. CIDR, 2023. Available: https://www.cidrdb.org/cidr2023/papers/p92-jain.pdf
10. W. Sheng, J. Wang, M. Barros, A. Montana, J. Tagliabue, and L. Bigon, "Building a Correct-by-Design Lakehouse: Data Contracts, Versioning, and Transactional Pipelines for Humans and Agents," arXiv:2602.02335, 2025. Available: https://arxiv.org/abs/2602.02335
11. M. Xue et al., "Adaptive and Robust Query Execution for Lakehouses at Scale," Proc. VLDB Endowment, vol. 17, no. 12, pp. 3947–3959, 2024. DOI: 10.14778/3685800.3685818. Available: https://dl.acm.org/doi/10.14778/3685800.3685818
12. J. Song et al., "Magnus: A Holistic Approach to Data Management for Large-Scale Machine Learning Workloads," Proc. VLDB Endowment, vol. 18, no. 12, pp. 4964–4977, 2025. DOI: 10.14778/3750601.3750620. Available: https://dl.acm.org/doi/10.14778/3750601.3750620
13. J. Zeng, C. Zhang, and Z. Liang, "PalanTír: Optimizing Attack Provenance with Hardware-enhanced System Observability," Proc. ACM CCS, 2022. DOI: 10.1145/3548606.3560570. Available: https://dl.acm.org/doi/10.1145/3548606.3560570
14. Y. Mei et al., "Disaggregated State Management in Apache Flink® 2.0," Proc. VLDB Endowment, vol. 18, no. 12, pp. 4846–4859, 2025. DOI: 10.14778/3750601.3750609. Available: https://dl.acm.org/doi/10.14778/3750601.3750609
15. T. Götz, D. Ritter, and J. Giceva, "LakeVilla: A Modular and Non-Invasive Toolbox for Lakehouse Transactions," arXiv:2504.20768, 2025. Available: https://arxiv.org/abs/2504.20768

16. N. Armenatzoglou et al., "Amazon Redshift Re-invented," Proc. ACM SIGMOD, pp. 2205–2217, 2022. DOI: 10.1145/3514221.3526045. Available: https://dl.acm.org/doi/10.1145/3514221.3526045
17. Guo et al., "Ursa: A Lakehouse-Native Data Streaming Engine for Kafka," Proc. VLDB Endowment, vol. 18, 2025. DOI: 10.14778/3750601.3750636. Available: https://dl.acm.org/doi/10.14778/3750601.3750636
18. P. Carbone, M. Fragkoulis, V. Kalavri, and A. Katsifodimos, "Beyond Analytics: The Evolution of Stream Processing Systems," Proc. ACM SIGMOD, 2020. DOI: 10.1145/3318464.3389702. Available: https://dl.acm.org/doi/10.1145/3318464.3389702
19. M. Fragkoulis, P. Carbone, V. Kalavri, and A. Katsifodimos, "A Survey on the Evolution of Stream Processing Systems," VLDB Journal, vol. 33, 2024. DOI: 10.1007/s00778-023-00819-8. Available: https://doi.org/10.1007/s00778-023-00819-8
20. T. Akidau et al., "Watermarks in Stream Processing Systems: Semantics and Comparative Analysis of Apache Flink and Google Cloud Dataflow," Proc. VLDB Endowment, vol. 14, no. 12, pp. 3135–3147, 2021. DOI: 10.14778/3476311.3476389. Available: https://dl.acm.org/doi/10.14778/3476311.3476389
21. A. A. Harby and F. Zulkernine, "Data Lakehouse: A Survey and Experimental Study," Information Systems, vol. 127, 2025. DOI: 10.1016/j.is.2024.102460. Available: https://dl.acm.org/doi/10.1016/j.is.2024.102460
22. P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination Avoidance in Database Systems," Proc. VLDB Endowment, vol. 8, no. 3, pp. 185–196, 2014. DOI: 10.14778/2735508.2735509. Available: https://dl.acm.org/doi/10.14778/2735508.2735509