

An Introduction to WAL-on-S3 Architectures

Akihiro Okuno

Scalar Inc, Tech Talk / 2026-09-25

Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 **Background** The log as the source of truth: commits, recovery, disaggregation
- 2 **History and value** Logs to remote storage, data to object storage; what a database gains
- 3 **Architecture** Commit on S3: a composite view drawn from existing systems
- 4 **Design space** Where to acknowledge a commit, and how to batch the log
- 5 **Research and products** Recent papers and systems, mapped onto the design space
- 6 **Takeaways** Answers to the question above, in four points

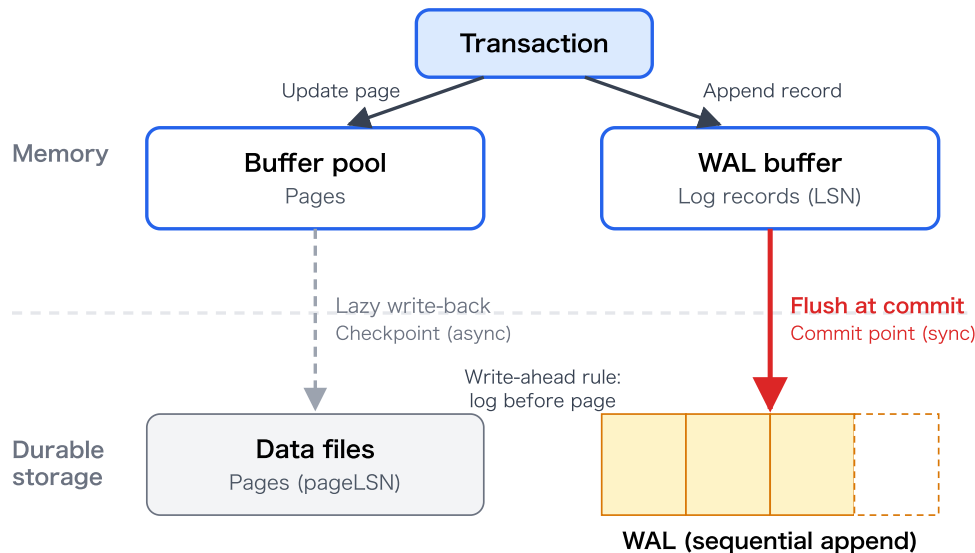
Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background** The log as the source of truth: commits, recovery, disaggregation
- 2 History and value Logs to remote storage, data to object storage; what a database gains
- 3 Architecture Commit on S3: a composite view drawn from existing systems
- 4 Design space Where to acknowledge a commit, and how to batch the log
- 5 Research and products Recent papers and systems, mapped onto the design space
- 6 Takeaways Answers to the question above, in four points

ARIES: WAL and Pages

ARIES: WAL and buffer management rules, plus the recovery protocol (next slide)



Buffer policy

	Force	No-force
No-steal	No undo, no redo	Redo only
Steal	Undo only	Undo + redo ARIES

Steal

Uncommitted pages on disk

→ **undo required**

No-force

Committed pages not written at commit

→ **redo required**

Steal / no-force: commit = WAL flush; pages written back lazily

ARIES: Recovery

Guarantee after a crash: effects of every committed transaction, none of any uncommitted one



1. Analysis



2. Redo



3. Undo



LSN

On every log record

pageLSN

On every page: idempotent redo

CLR

Compensation record: restartable undo

Fuzzy checkpoint

No quiescing

Pages are derived state: a base copy plus the log rebuilds them (redo, then undo)

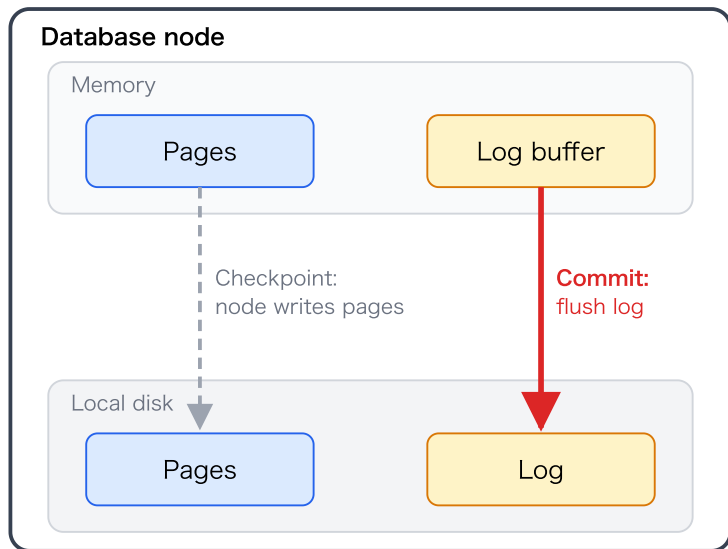
From Single-Node to Log-Centric Disaggregation

Commit writes only the log. Only the log leaves the compute node; storage builds pages

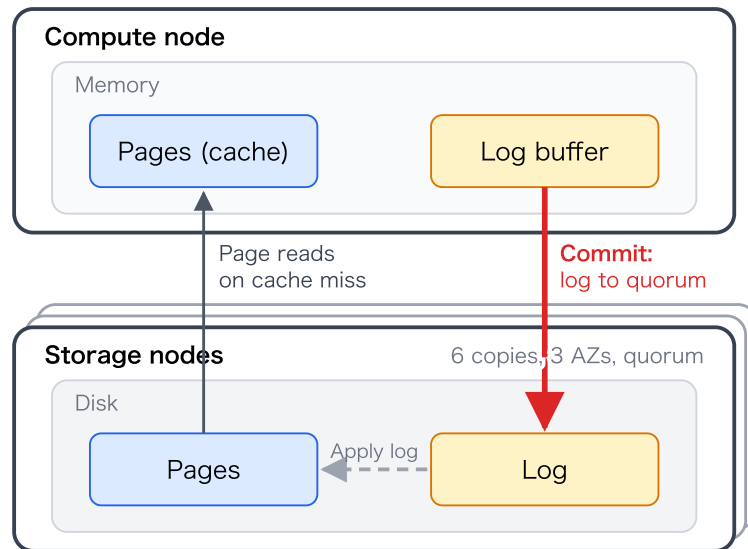
"The truth is the log. The database is a cache of a subset of the log."

Pat Helland, Immutability Changes Everything (CIDR 2015)

Single-node database



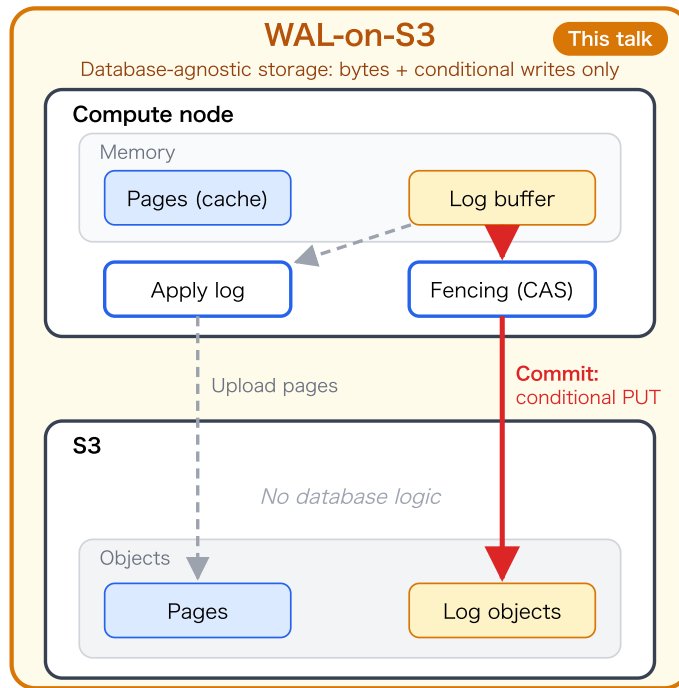
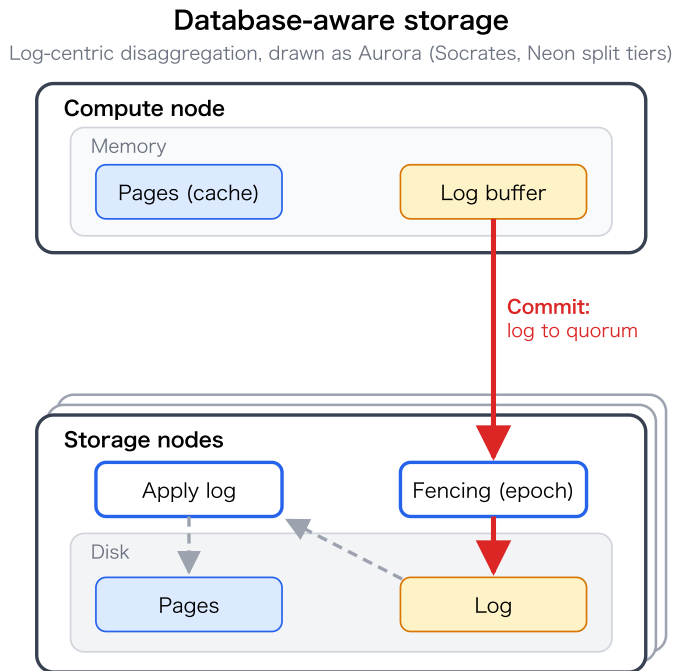
Log-centric disaggregation (Aurora)



"Log-centric disaggregation": term used in this talk for disaggregated databases where compute ships only the log and storage materializes pages (Aurora, Socrates, Neon). Other disaggregation styles, e.g. a transaction/data component split (Deuteronomy), are out of scope

Database-Aware vs. Database-Agnostic Storage

Same log and pages on both sides. Difference: where the log becomes pages, and where fencing runs



Blue box: database logic. Red: commit path (sync). Gray dashed: background (async). Fencing: rejecting writes from a replaced writer

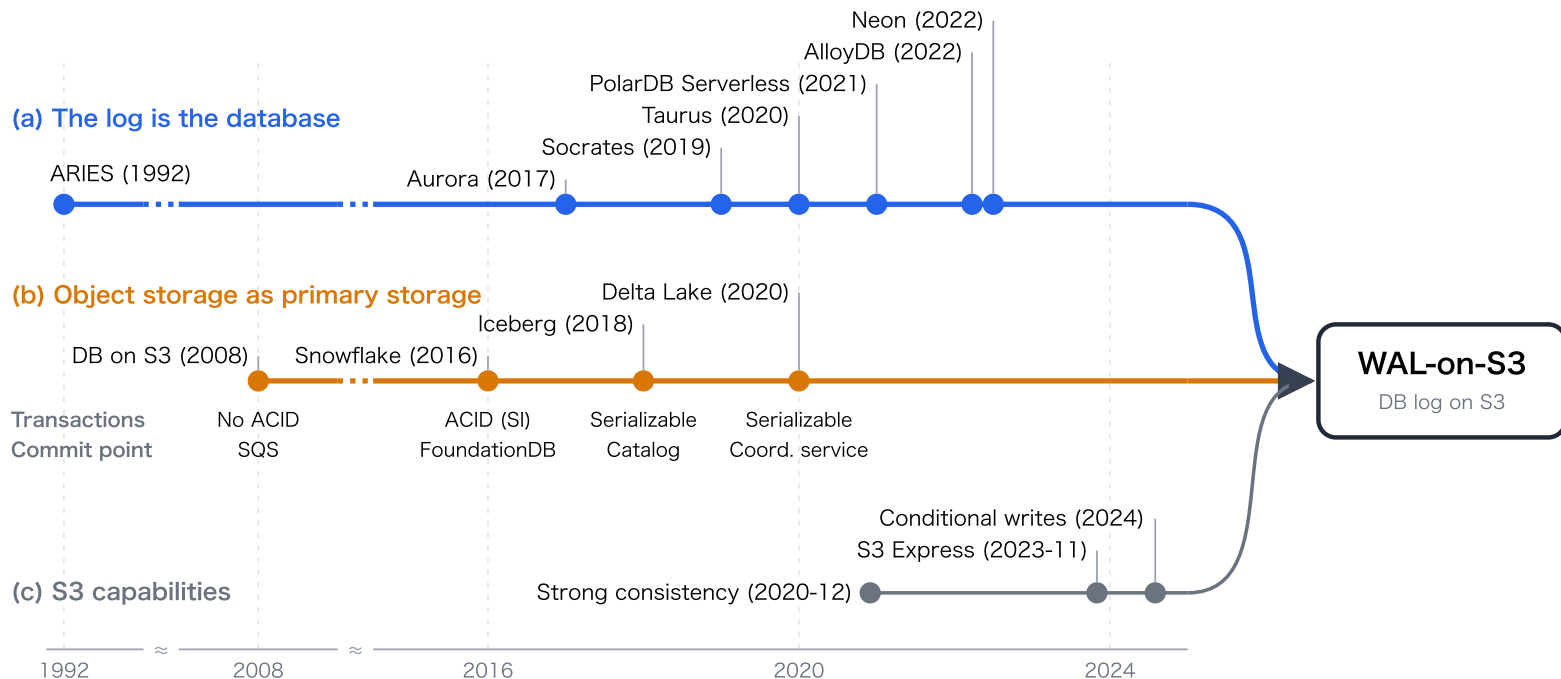
Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background
The log as the source of truth: commits, recovery, disaggregation
- 2 History and value**
Logs to remote storage, data to object storage; what a database gains
- 3 Architecture
Commit on S3: a composite view drawn from existing systems
- 4 Design space
Where to acknowledge a commit, and how to batch the log
- 5 Research and products
Recent papers and systems, mapped onto the design space
- 6 Takeaways
Answers to the question above, in four points

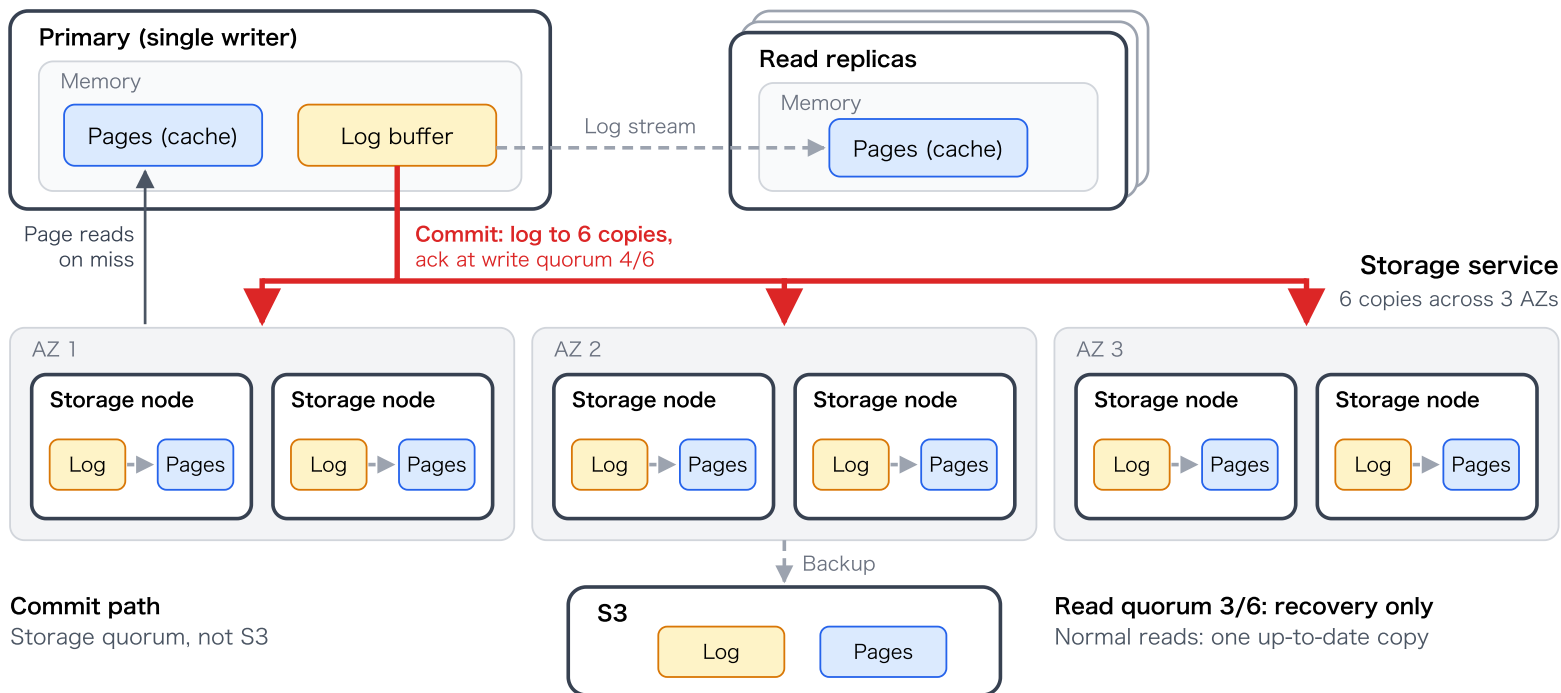
Two Lineages Leading to WAL-on-S3

(a) Remote DB logs, (b) object storage as primary storage, linked by S3 capabilities



Aurora: The Log Is the Database

Network as the bottleneck. Only log records cross the network; storage nodes build pages



Red: commit path (sync). Gray dashed: background (async), including log apply inside each storage node. Dark gray: read path

Verbitski et al., Amazon Aurora, SIGMOD 2017. §3 heading: "The Log Is The Database"; "the only writes that cross the network are redo log records"

Remote Logs After Aurora

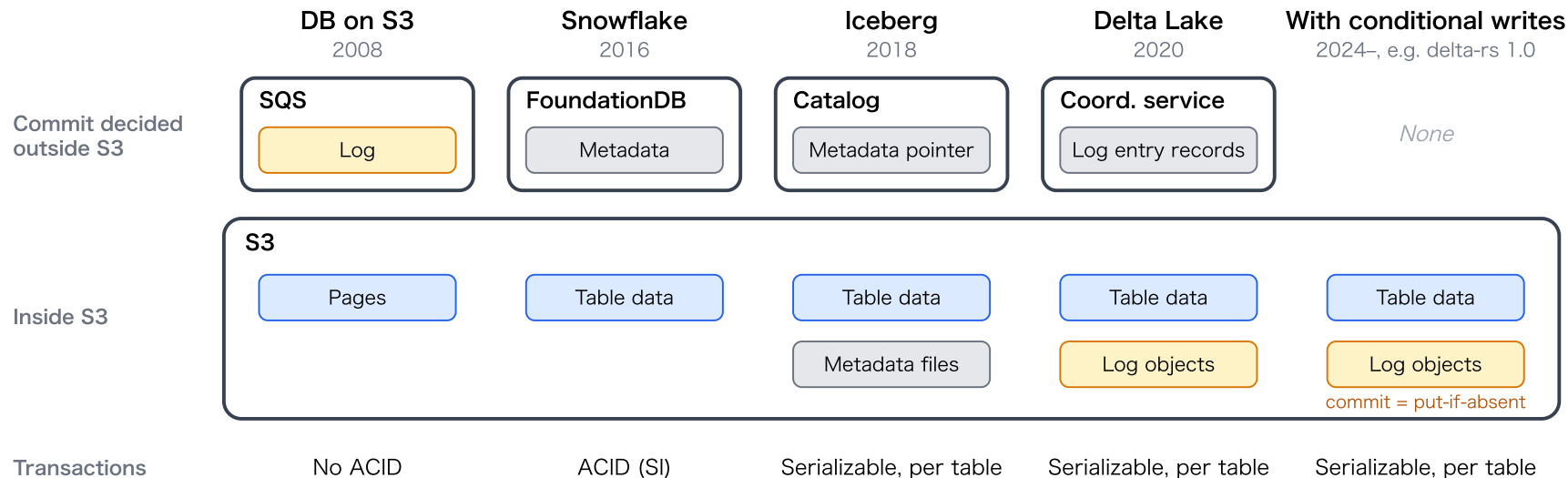
Common pattern: low-latency quorum for commit; object storage below the hot path

System	Commit point	Page materialization	Object storage role
Aurora (SIGMOD 2017)	4/6 quorum across 3 AZs	Storage nodes	Backup
Socrates / Azure SQL Hyperscale (SIGMOD 2019)	Landing zone (premium storage)	Page servers	Long-term log and pages (XStore)
PolarDB Serverless (SIGMOD 2021)	Log chunks on PolarFS (3 replicas, ParallelRaft)	Page chunks	—
Neon	Safekeepers (Paxos quorum)	Pageservers	Long-term history; "never on the hot query path"

Remote log: yes. Object storage: below the commit path, never on it

Transactions on Object Storage

Can S3 alone decide a commit? Data on S3 since 2008; commit point outside S3 until 2024



Blue: data. Orange: log. Gray: metadata. Delta Lake 2020: a Databricks coordination service (open source later: DynamoDB). Put-if-absent: create only if absent

Transactions on S3: yes, but always with an external commit point. Conditional writes move it into S3

WAL-on-S3: Where the Lineages Meet

The log of (a), stored as in (b), committed with the capabilities of (c)

(a) The log is the database

2017–2022

Log is the truth; pages are derived from it
Commit on a low-latency quorum, not on S3

(b) Object storage as primary storage

2008–2020

Data durable on S3
Transactions, but the commit decided outside S3

(c) S3 capabilities

2020–2024

Strong consistency, S3 Express, conditional writes
A commit can be decided inside S3

WAL-on-S3

This talk

A database's log with S3 as its durable tier

S3 is database-agnostic: bytes and conditional writes only

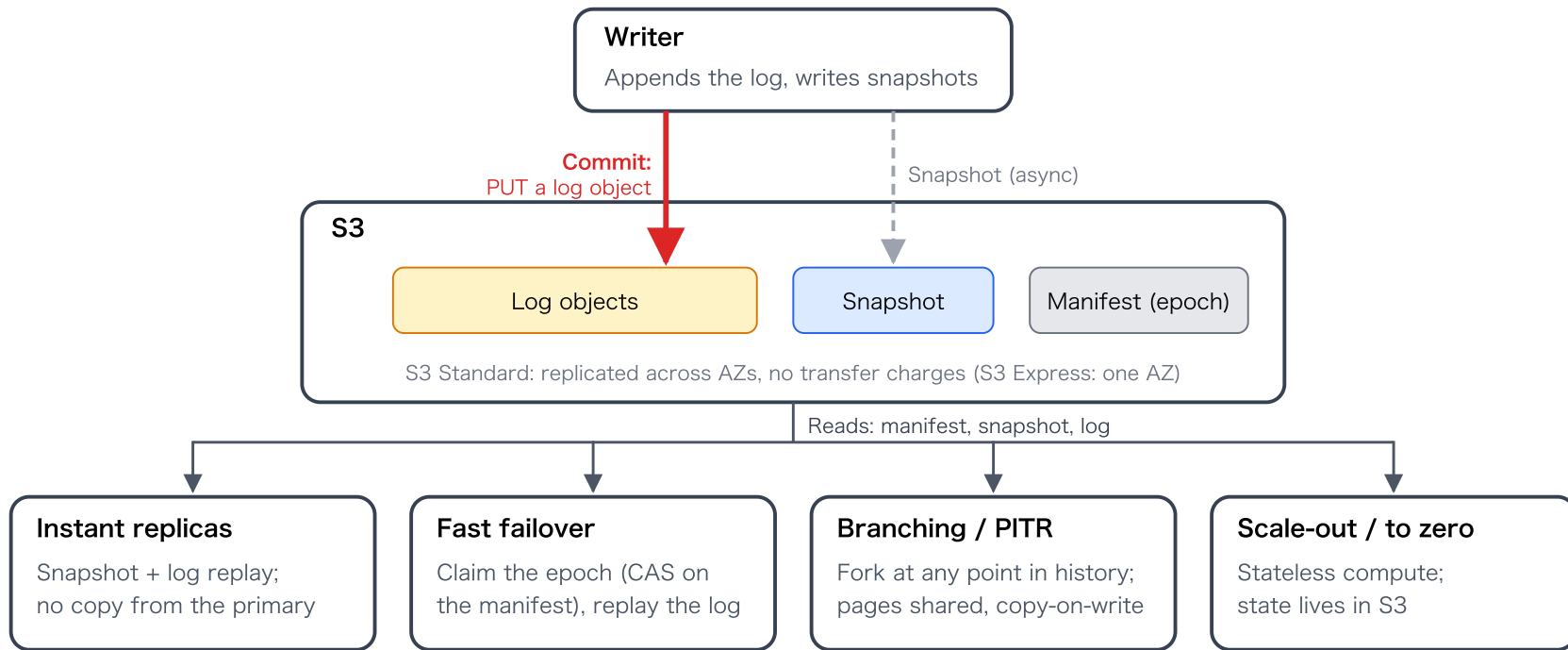
Commit Conditional PUT, or offset-checked append

Fencing CAS on S3 (e.g. a writer epoch)

Pages Derived by compute or an extra tier, not by S3

What WAL-on-S3 Offers a Database

Compute nodes share S3, with no storage service to run. Cost: commit latency



Orange: log. Blue: pages. Gray: metadata. Red: commit path (sync). Gray dashed: background (async). Dark gray: reads

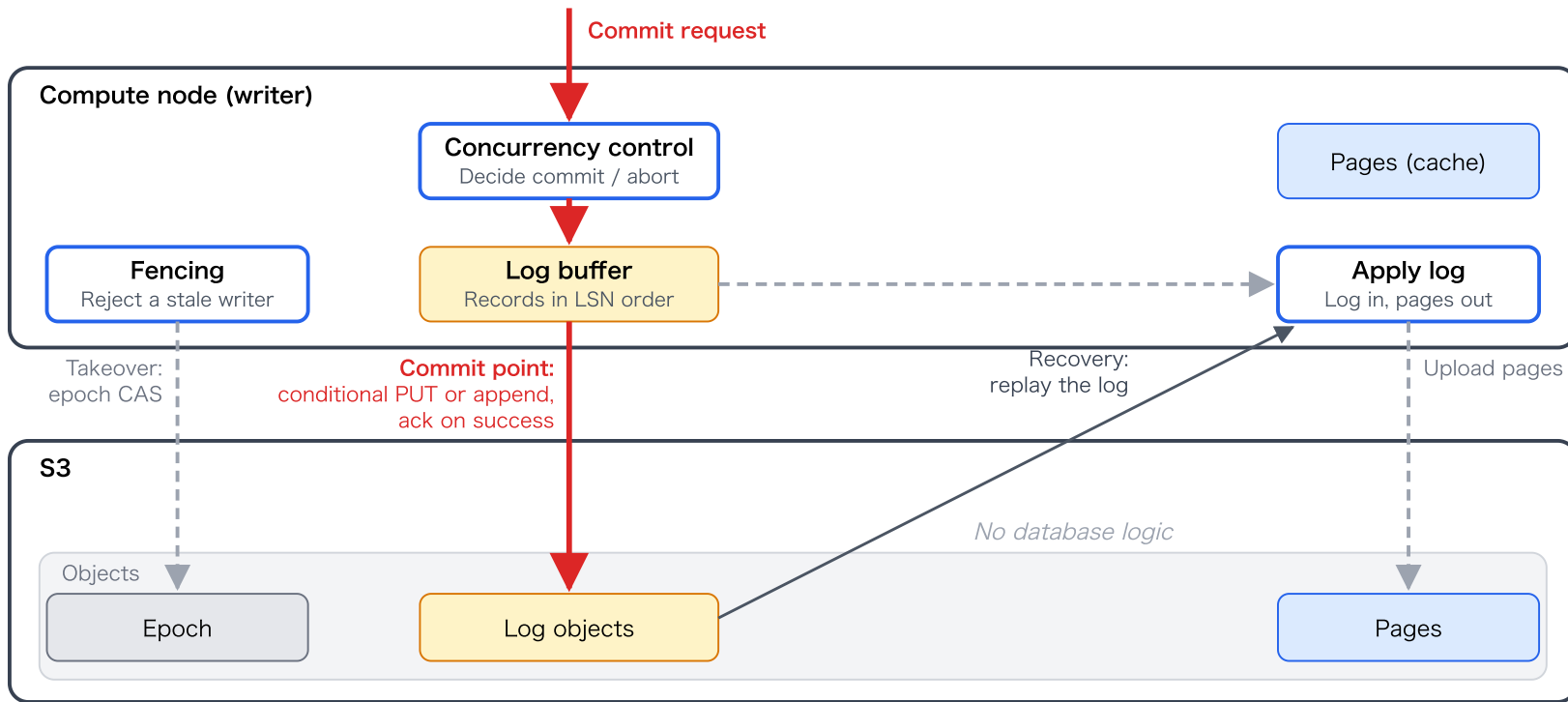
Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background The log as the source of truth: commits, recovery, disaggregation
- 2 History and value Logs to remote storage, data to object storage; what a database gains
- 3 Architecture** Commit on S3: a composite view drawn from existing systems
- 4 Design space Where to acknowledge a commit, and how to batch the log
- 5 Research and products Recent papers and systems, mapped onto the design space
- 6 Takeaways Answers to the question above, in four points

WAL-on-S3: Conceptual Architecture

Single writer on S3: S3 gives durability and conditional writes; the rest runs on compute or an extra tier



Red: commit path. Gray dashed: off the commit path. Dark gray: reads. Blue border: database logic

Not an established architecture: this talk's composite of parts from existing systems. SlateDB: commit by PUT to S3, epoch CAS fencing. Milliscale: commit on S3 Express. Neon: log apply on a separate tier (pageserver), pages on S3. No single system implements the whole picture

Roles of Logs on S3

Four of five roles replay the log in order to rebuild state (state machine replication, SMR), the database WAL included

	Role of the log	Replay rebuilds	Examples (research and products)
SMR Ordered log, replayed in order	Database WAL This talk	Database state	Commit on S3: SlateDB, Milliscale S3 after ack: Neon, BtrLog, Litestream
	Table commit log	Table snapshot	Delta Lake (Iceberg: pointer swap, no replay)
	Non-transactional WAL	State of that system Search index, Git repository	turbopuffer, Chroma wal3, Cursor Continuity
	Shared log service	Arbitrary Defined by each client	LogDrive (research)
	Stream (Kafka API)	None: consumers read the log	WarpStream, Bufstream, StreamNative Ursa, Confluent K2, Redpanda Cloud Topics

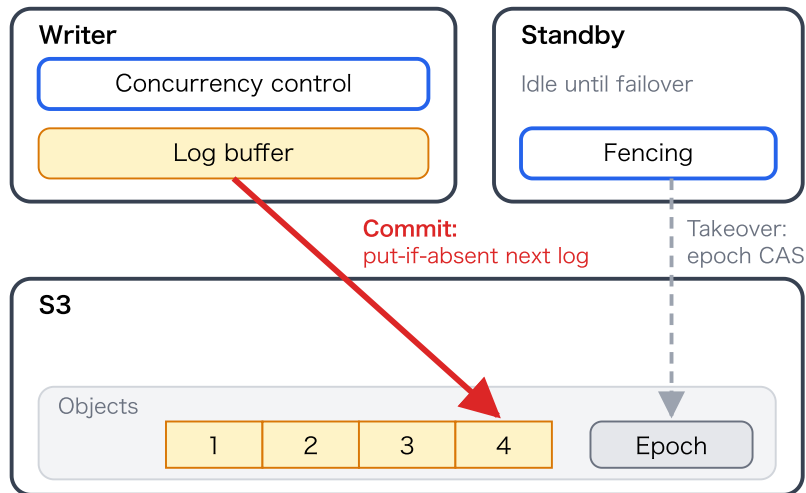
Scope: systems that persist a log on S3. Streams are not SMR but share ordered, durable append. Shared-log lineage (CORFU, Delos): same abstraction, not on S3. Gray: not a database transaction log

Classification by this talk, based on the surveyed papers and systems. Not an established taxonomy. LogDrive: S3 and S3 Express backends evaluated; production uses DynamoDB

Single-Writer vs. Multi-Writer

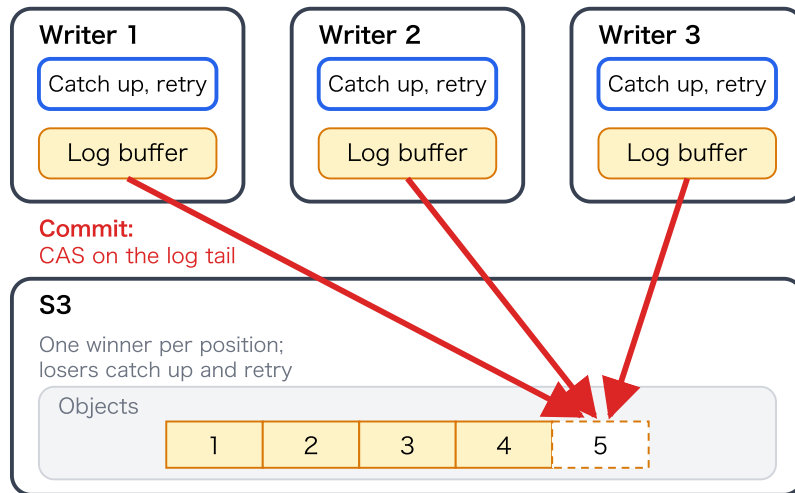
Where the log order is decided: inside one writer, or by a CAS on the log tail in S3

Single-writer (database mainstream)



Consensus	Writer takeover only
Concurrency	Inside the writer
On conflict	None on S3; an old writer is fenced
Limit	Batching, PUT latency
Examples	SlateDB, Milliscale

Multi-writer (non-database precedents)



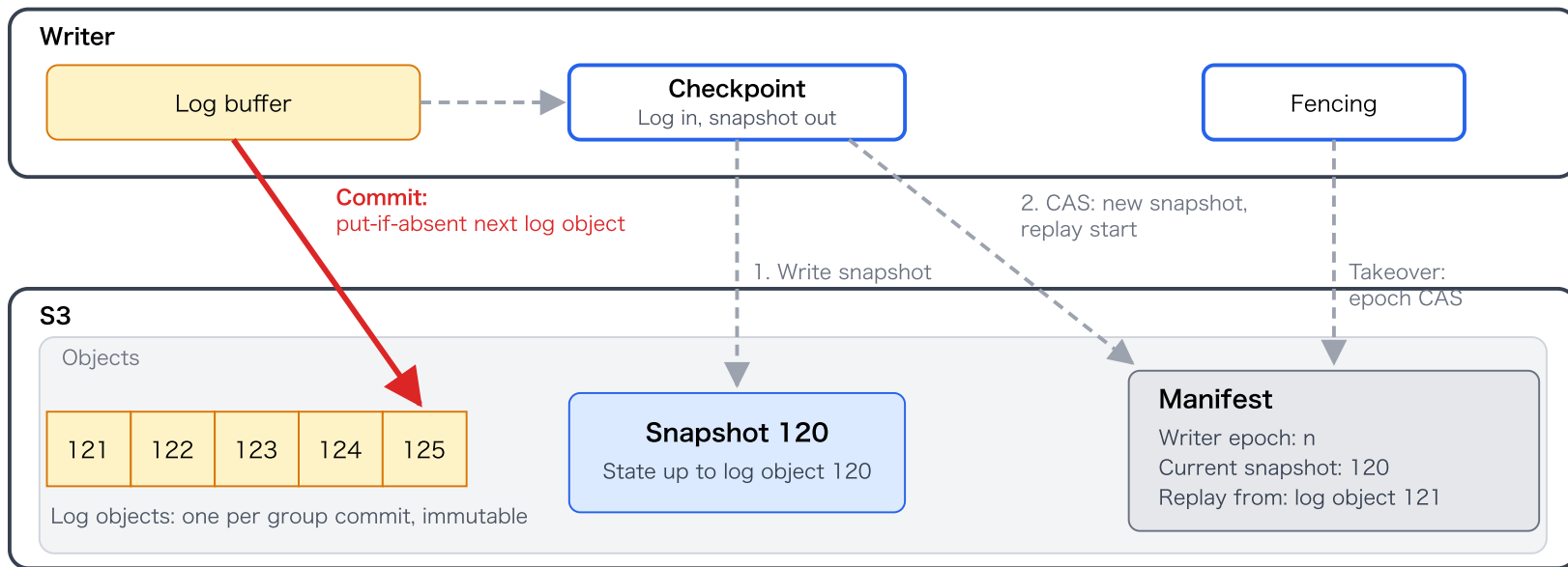
Consensus	Every commit
Concurrency	Order only; no conflict check on data
On conflict	Lost the tail: catch up, retry
Limit	Round trips to the consensus point
Examples	OSWALD, Continuity, wal3, turbopuffer

Red: commit path. Gray dashed: takeover, off the commit path. Blue border: database logic

This talk's framing. Aurora, Neon: also a single writer, but every commit is a quorum write (Neon: Paxos), not a write to S3. Multi-writer side from non-database precedents

Building Blocks on S3

Single writer: log objects, a snapshot, and a manifest that ties them together

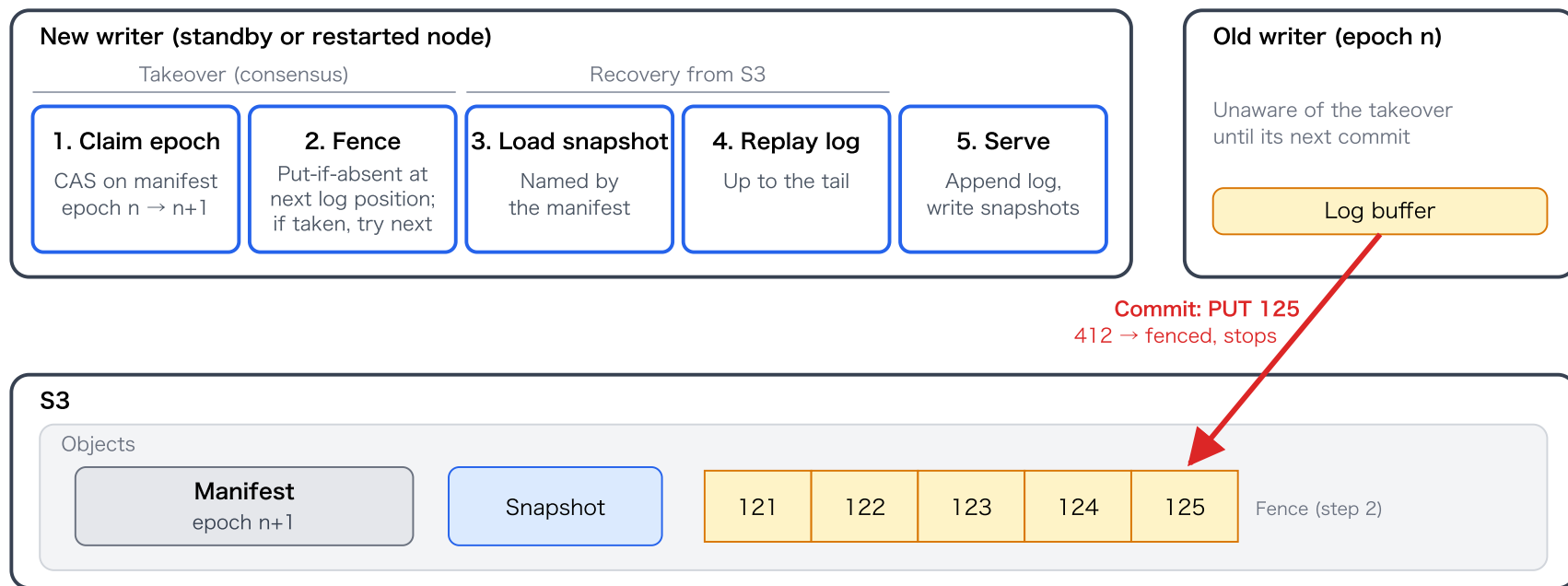


Red: commit path. Gray dashed: off the commit path. Blue border: database logic. Log objects before the replay start can be deleted

This talk's abstraction, following SlateDB's layout (manifest with writer epoch, WAL SSTs, compacted state)

Single-Writer: Writer Lifecycle on S3

Consensus only at writer takeover. The old writer stops at its next commit



Same steps, different uses. Failover: steps 1–5 on a standby, no data copy. Read replica: steps 3–4, then tail the log

Examples: SlateDB (epoch CAS + fence object), BtrLog (write token on S3 via `If-Match`)

Steps 1–5 follow SlateDB (writer epoch CAS on the manifest, empty WAL SST as the fence object). BtrLog fences with a different mechanism (write token)

Why Now: Required S3 Capabilities

S3 features used by the protocol steps, added in 2020–2024

Capability	Used for	Available since
Strong read-after-write consistency, including LIST	Finding the log tail during replay	2020-12-01
Low-latency tier (S3 Express One Zone)	Commit latency	2023-11-28
Put-if-absent (If-None-Match: *)	Commit of the next log object, fencing	2024-08-20
Append (S3 Express)	Appending log records to one object	2024-11-21
Compare-and-swap (If-Match)	Epoch claim on one mutable object (e.g. BtrLog write token)	2024-11-25

Strong consistency alone: concurrent PUTs to one key are last-writer-wins; no atomic multi-key update. Hence conditional writes. Later additions: conditional deletes (2024–25), atomic rename in S3 Express (2025), conditional CopyObject (2025). SlateDB: manifest versions are write-once objects, so put-if-absent suffices

Our reading: when Aurora was announced (2014), S3 had none of these: low latency, strong consistency, conditional writes

Outline

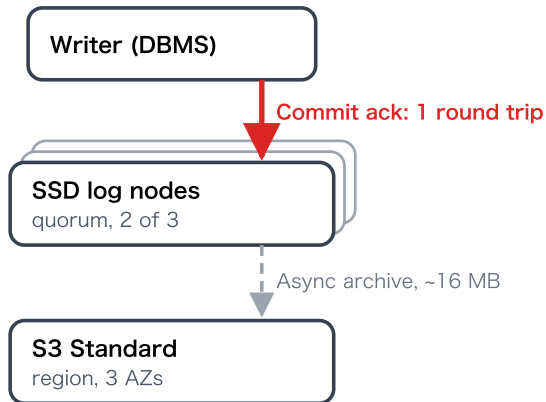
Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background The log as the source of truth: commits, recovery, disaggregation
- 2 History and value Logs to remote storage, data to object storage; what a database gains
- 3 Architecture Commit on S3: a composite view drawn from existing systems
- 4 Design space** Where to acknowledge a commit, and how to batch the log
- 5 Research and products Recent papers and systems, mapped onto the design space
- 6 Takeaways Answers to the question above, in four points

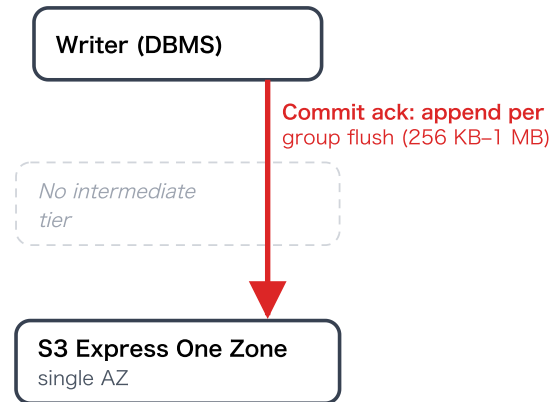
Axis: Commit Ack Placement

Which medium a commit waits on. It fixes commit latency, cost per commit, and durability scope

BtrLog (PVLDB 2026): ack before S3



Milliscale (arXiv 2026): ack on S3



Latency: $\geq$ one round trip to the ack medium, plus batching. Cost: per-request price $\times$ flushes. Durability: the medium's replicas

Ack medium	Used by	Commit latency	Cost per 1M appends	Durability scope
S3 Standard	—	tens of ms	~ \$5	Region (3 AZs)
S3 Express One Zone	Milliscale	~ 6 ms	~ \$1.1	Single AZ
EBS io2	—	0.3–0.5 ms	\$0.0036	Single AZ
SSD quorum	BtrLog	70 μ s (best median, 1 AZ)	\$0.00125 (1 AZ)	Single AZ (multi-AZ: 6 nodes)

Not in the table: a multi-AZ quorum also pays cross-AZ transfer (~\$0.02/GB); S3 includes cross-AZ durability

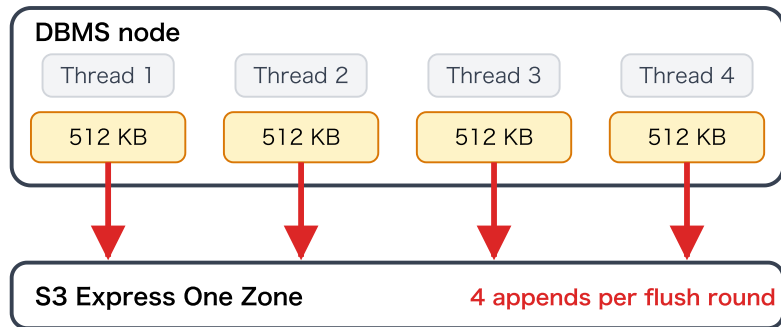
Numbers from the BtrLog paper; S3 and S3 Express values read from a log-scale figure (approximate). Group flush sizes from the Milliscale paper

Axis: Logging Scheme

With the ack on S3 Express, can per-thread logging meet it? Too many appends; share buffers instead

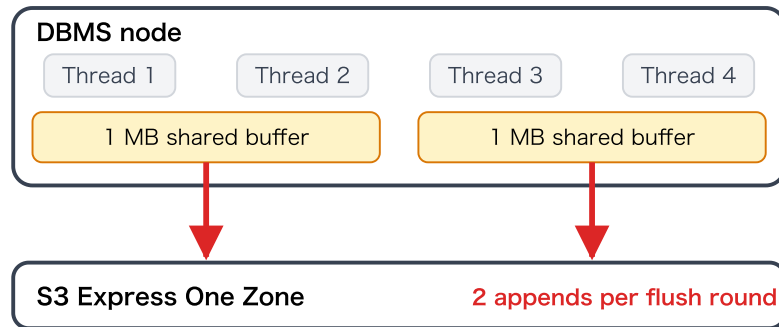
Per-thread logging (usual in OLTP engines)

Each append: one round trip and one billed request



Milliscale: thread-group logging

2 threads share a buffer twice as large



Why sharing works: S3 Express append latency is nearly flat up to ~1 MB (~8 ms at 128–512 KB), so one 1 MB append takes about as long as one 512 KB append. Result: ~50% fewer appends (TPC-C) at similar latency

Result (Milliscale: group logging + dependency tracking)	Value
YCSB-A tail latency (p99.9 / p99.99) vs. naive S3 Express	–52% to –62%
TPC-C mean commit latency vs. gp3	+23%

Database-specific: logging must be designed around request count, not only bytes

Not evaluated in the paper: failover, fencing, multi-AZ replication. Costs inferred, not measured

Design Space Summary

What a database on S3 has to decide, axis by axis

Axis	What the database decides	Conclusion
Commit ack placement	Which medium a commit waits on	Give up one of: an extra durable tier (quorum), durability scope (S3 Express), or latency (S3 Standard)
Logging scheme	How log records are batched into appends	Design around request count: fewer, larger appends
Durability scope	Which failure an acked commit survives	With S3 Express: replicate across AZs, at extra latency
Small objects	Batch size, caching, garbage collection	Batch size bounds commit latency; GC must be safe against stale writers
Ordering	Who assigns the log order	Settled: the single writer orders its own appends

For a database, the open choices are where to ack and how to batch; the single writer settles ordering and limits consensus to takeover

Durability scope, small objects, ordering: options and examples in the appendix (Other Design Axes)

Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background The log as the source of truth: commits, recovery, disaggregation
- 2 History and value Logs to remote storage, data to object storage; what a database gains
- 3 Architecture Commit on S3: a composite view drawn from existing systems
- 4 Design space Where to acknowledge a commit, and how to batch the log
- 5 Research and products** Recent papers and systems, mapped onto the design space
- 6 Takeaways Answers to the question above, in four points

Research: Databases

After Aurora and Socrates: what database research around WAL-on-S3 is asking now

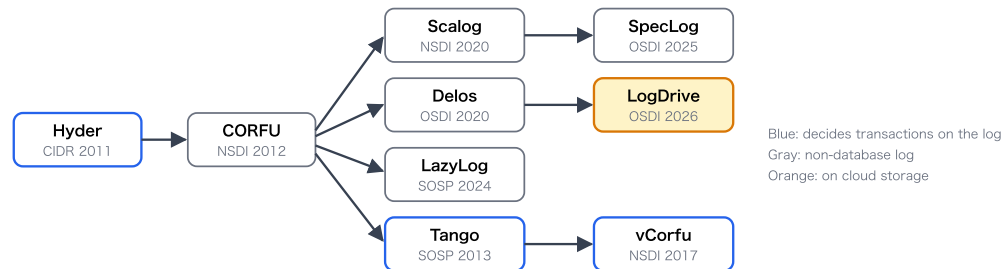
Research interest	Motivation	Work	Design-space axis explored
Choosing a cloud OLTP architecture	Many architectures and instance types; cost and durability tradeoffs unclear	OLTP in the Cloud (VLDBJ 2025)	Ack placement, durability scope: cost of each option
A reusable low-latency log service	Aurora-style log services are proprietary and engine-specific	BtrLog (PVLDB 2026)	Ack placement: a quorum tier, S3 below
Commit directly on object storage	Remove the separate log service altogether	Milliscale (arXiv 2026)	Ack placement: S3 Express; logging scheme
Object storage as the data tier	Cheap capacity for analytics without ETL	Colibri (PVLDB 2024)	Outside the commit path: data tier
Transactions on table formats	Multi-table ACID without a vendor-specific coordinator	LakeVilla (PVLDB 2026)	Outside OLTP: table-level commits by put-if-absent on S3

Frontier: where to ack (a generic quorum tier vs. S3 Express) and how to batch for it. Unevaluated: durability scope of an S3 Express ack

Survey scope: research on OLTP engines placing the WAL or commit on object storage, 2018–2026

Research: Shared Logs

Mostly non-database logs; Hyder, Tango, vCorfu decide transactions on the log



Research direction	Works	What it pursues	Storage it requires
Shared log	Hyder, CORFU	One ordered log on shared flash; CORFU adds a sequencer	Custom flash
Write first, order later	Scallog, SpecLog	Faster ordering: persist first, order later, speculate	Active custom servers
Pluggable durability	Delos, LogDrive	Consensus only at reconfiguration; swap storage	Seal, checkTail → weak writes on S3, DynamoDB
Deferred ordering	LazyLog	Lower append latency by ordering at read time	Custom sequencing layer
Transactions	Tango, vCorfu	Decide transactions from log order	CORFU-style log

Same idea in database logs (our reading): consensus at reconfiguration (Delos) ↔ at takeover (BtrLog, SlateDB)

Products: Databases

Which database products put the commit ack on S3?

Product	Where the commit is acked	Role of S3	Writers
Aurora	4/6 storage quorum across 3 AZs	Backup	Single
Neon	Safekeeper Paxos quorum	Long-term history, off the query path	Single; Paxos also fences the primary
Aurora DSQL	Journal	—	Multiple; adjudicators (OCC)
SlateDB	S3 (<code>await_durable</code>)	The WAL itself	Single; epoch CAS fencing
Litestream	Local disk; S3 asynchronously	WAL replica	Single; conditional-write lease (2025)

Among the products surveyed, only SlateDB, a KV engine, acks on S3

Products: Non-Database Precedents

Not database transaction logs: streaming (Kafka API), search and vector databases, code sync

Kind	Product	Where the append is acked	Who orders	Batching / latency
Streaming (Kafka API)	WarpStream	S3 write + metadata	Metadata service	≤ 250 ms / 8 MiB
	Confluent Freight / K2	S3 batch + metadata	Conflux	—
	Bufstream	S3 write + metadata	Postgres, Spanner, etcd	~100 ms
	StreamNative Ursa	WAL object + metadata	Oxia	≤ 200 ms / ~4 MB (S3 WAL)
	Kafka KIP-1150	Object storage write	Diskless coordinator	—
	Redpanda Cloud Topics	S3 upload	Raft	1–2 s
	AutoMQ	WAL (S3 / EBS / NFS)	—	—
Search / vector DB	turbopuffer	S3 WAL + CAS commit	CAS on S3	1 commit / s per namespace
	Chroma (wal3)	S3 WAL + manifest CAS	Manifest CAS	—
Code sync	Cursor Continuity (Git)	S3 WAL index CAS	Index CAS	—

For a database: acking on S3 with batching and CAS on S3 carry over; conflict-free appends do not

Outline

Question: how far a database's write-ahead log (WAL) can move to object storage

- 1 Background The log as the source of truth: commits, recovery, disaggregation
- 2 History and value Logs to remote storage, data to object storage; what a database gains
- 3 Architecture Commit on S3: a composite view drawn from existing systems
- 4 Design space Where to acknowledge a commit, and how to batch the log
- 5 Research and products Recent papers and systems, mapped onto the design space
- 6 Takeaways** Answers to the question above, in four points

Takeaways

How far a database's WAL can move to object storage

1 **WAL-on-S3: a database's log with database-agnostic S3 as its durable tier**

Same design as Aurora (the log is the truth, pages are derived); log apply moves to compute or an extra tier, fencing to a client protocol on S3

2 **Possible now: conditional writes (2024) let S3 decide a commit and stop a replaced writer**

With a single writer, consensus is needed only at writer takeover

3 **Durability moves to S3; the commit ack is the tradeoff**

Ack on S3 costs latency (Express: ms; Standard: tens of ms or more) or durability scope (Express: one AZ); many designs ack on a quorum above S3

4 **Today: among surveyed products, only SlateDB (a KV engine) acks on S3**

Research explores where to ack (a generic quorum tier vs. S3 Express); multi-writer at OLTP rates is open

Appendix

Appendix: References (1/2)

- Mohan et al., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, TODS 1992
- Helland, Immutability Changes Everything, CIDR 2015
- Kreps, The Log: What every software engineer should know about real-time data's unifying abstraction, 2013
- Verbitski et al., Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases, SIGMOD 2017
- Antonopoulos et al., Socrates: The New SQL Server in the Cloud, SIGMOD 2019
- Depoutovitch et al., Taurus Database: How to be Fast, Available, and Frugal in the Cloud, SIGMOD 2020
- Cao et al., PolarFS: An Ultra-low Latency and Failure Resilient Distributed File System for Shared Storage Cloud Database, PVLDB 2018
- Cao et al., PolarDB Serverless: A Cloud Native Database for Disaggregated Data Centers, SIGMOD 2021
- Google Cloud Blog, "AlloyDB for PostgreSQL under the hood", 2022
- Neon, Architecture overview (neon.com docs)
- Linnakangas, "Architecture decisions in Neon", 2022
- Brantner et al., Building a Database on S3, SIGMOD 2008
- Dageville et al., The Snowflake Elastic Data Warehouse, SIGMOD 2016
- Armbrust et al., Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores, PVLDB 2020
- Pavlo & Aslett, What's Really New with NewSQL?, SIGMOD Record 2016
- Corbett et al., Spanner: Google's Globally-Distributed Database, OSDI 2012
- Brooker, DSQL Vignette: Reads and Compute; Transactions and Durability (blog), 2024
- Brooker et al., Aurora DSQL: Scalable, Multi-Region OLTP, arXiv 2607.13276, 2026

Appendix: References (2/2)

- Bernstein et al., Hyder – A Transactional Record Manager for Shared Flash, CIDR 2011
- Balakrishnan et al., CORFU: A Shared Log Design for Flash Clusters, NSDI 2012
- Balakrishnan et al., Tango: Distributed Data Structures over a Shared Log, SOSP 2013
- Wei et al., vCorfu: A Cloud-Scale Object Store on a Shared Log, NSDI 2017
- Ding et al., Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log, NSDI 2020
- Balakrishnan et al., Virtual Consensus in Delos, OSDI 2020
- Luo et al., LazyLog: A New Shared Log Abstraction for Low-Latency Applications, SOSP 2024
- Bhat et al., Low End-to-End Latency atop a Speculative Shared Log with Fix-Ante Ordering (SpecLog), OSDI 2025
- Vickers et al., The LogDrive: Composable Durability for Cloud-Based Shared Logs, OSDI 2026
- Schmidt et al., Two Birds With One Stone: Designing a Hybrid Cloud Storage Engine for HTAP (Colibri), PVLDB 2024
- Haubenschild & Leis, OLTP in the Cloud: Architectures, Tradeoffs, and Cost, VLDBJ 2025
- Kuschewski et al., BtrLog: Low-Latency Logging for Cloud Database Systems, PVLDB 2026
- Zhou et al., Milliscale: Fast Commit on Low-Latency Object Storage, arXiv 2603.02108, 2026
- Götz, Ritter, El-Hindi, Giceva, Interoperable ACID Transactions for Open Table Formats (LakeVilla), PVLDB 2026
- Merli et al., Ursa: A Lakehouse-Native Data Streaming Engine for Kafka, PVLDB 2025
- Vanlightly, s3-wal-collection (GitHub)

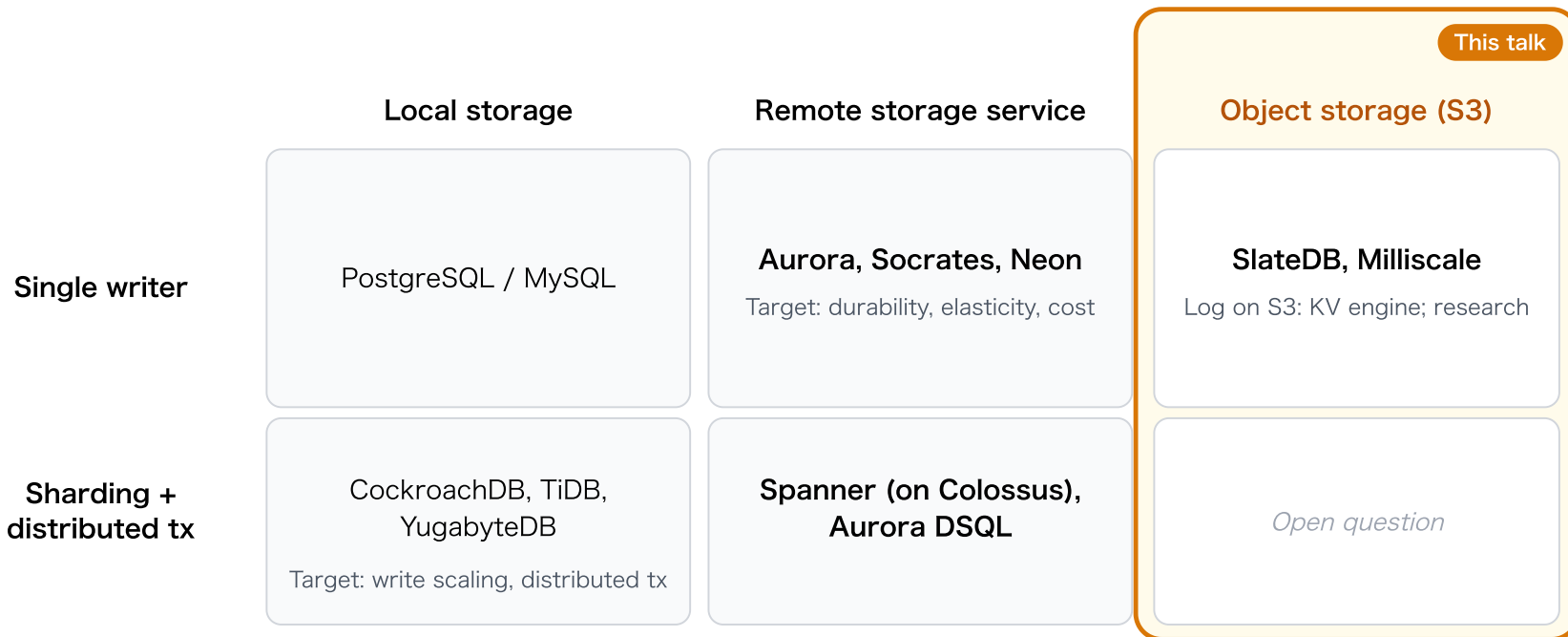
Appendix: Log-Is-the-Database Lineage

Same core in all six: compute ships only the log; storage builds pages

System	Commit durable at	Pages built by	Change from predecessors	Purpose
Aurora (2017)	4/6 quorum, 3 AZs	Storage nodes	Only the log crosses the network; storage applies it and serves pages	Cut network traffic, the bottleneck in the cloud
Socrates (2019)	Landing zone	Page servers	Log service split from page service. Object storage for long-term data	Provide durability (log) and availability (pages) separately
Taurus (2020)	All 3 Log Store replicas	Page Stores	No quorum: a failed write moves the log to 3 other nodes. Two tiers instead of Socrates' four	Higher write availability than Aurora's quorum, with 3 copies instead of 6; fewer network hops
PolarDB Serverless (2021)	3 log-chunk replicas (ParallelRaft)	Page-chunk leader	Log and page chunks added to the PolarFS shared file system, where compute had written pages	Reuse PolarFS with minimal changes; evict dirty pages without flushing; parallel recovery
AlloyDB (2022)	Regional log storage	Log processing service	Log applied in each zone; compute never checkpoints or writes full pages	Survive zone failures; read replicas share storage without a private copy
Neon (2022)	Safekeeper Paxos quorum	Pageservers	Open-source PostgreSQL. Consensus also fences the primary. Object storage as the bottom tier	Aurora-like design native to PostgreSQL; no split brain; immutable long-term history

Appendix: Two Axes Within NewSQL

Storage disaggregation vs. write scaling: independent axes



Pavlo & Aslett (2016): Aurora as DBaaS, Spanner as New Architectures

Appendix: Terms Used in This Talk

Architecture terms, and the two roles in a writer handoff: consensus and fencing

Term	Meaning in this talk
Log (WAL)	The database's write-ahead log. Called "log" unless the undo/redo distinction matters
Commit point	Where a commit becomes durable and is acknowledged, and where concurrent commits are decided
Hot path	The synchronous path of a commit acknowledgment (or of a query)
Log-centric disaggregation	This talk's term: compute ships only the log; storage builds pages (Aurora, Socrates, Neon)
Database-aware storage	Storage that applies the log, serves pages, and fences writers inside its own protocol
Database-agnostic storage	Storage that keeps bytes and offers conditional writes only (S3)
WAL-on-S3	A database's log whose durable tier is database-agnostic object storage
Single writer / multi-writer	One node appends to the log at a time / several nodes append concurrently
Consensus (in a handoff)	Deciding which node is the next writer and its epoch, exactly once

Appendix: Before Conditional Writes

External coordinators, now replaceable by S3 itself. Research primitives now offered by S3

External coordinators in practice

System	Before	After
DB on S3 (2008)	SQS	—
Delta Lake (multi-cluster)	DynamoDB	delta-rs 1.0: S3 conditional writes
Litestream	Consul lease	2025: conditional-write lease

Research primitives vs. S3

Required by research	S3 equivalent
Write-once address (CORFU, Tango)	If-None-Match: *
Seal / fence (CORFU, Delos)	Fence object, epoch CAS
Versioned register (Delos MetaStore)	If-Match

Delta Lake on Spark: still DynamoDB-based. LogDrive (OSDI 2026): the opposite choice, with no conditional writes on the log and a small conditional register on DynamoDB

Appendix: Other Design Axes

Three further choices, each with a question and its implication for a database

Axis	Question	Options seen	Implication for a database
Ordering vs. durability	Who assigns the log order?	The put-if-absent address is the order; sequence first (CORFU); write first, sequence later (Scalog, K2, WarpStream), where the metadata layer can dominate cost (LogDrive estimate for K2 on DynamoDB: ~75%)	Minor: a single writer assigns the order itself
Durability scope	Which failure does an acked commit survive?	S3 Standard: region. S3 Express: one AZ. Majority append across AZ buckets (Milliscale, proposed, not evaluated); quorum across stores (LogDrive)	An ack on S3 Express needs cross-AZ replication for region durability, at extra latency
Small objects	What does per-request pricing force?	Batching (adds latency); read caching; compaction and garbage collection	Batch size bounds commit latency; garbage collection must stay safe against stale writers

For a single-writer database, durability scope and small objects matter; ordering is settled by the writer

Appendix: Beyond a Single Writer

Multi-writer on S3 works where writes never conflict or commits are rare

Setting	How writers are coordinated	Conflict checks	Commit rate	Examples
Streaming, search (non-database)	CAS on the log tail; external metadata service; or a sequencer	None	Per batch (~100 ms to seconds)	wal3, turbopuffer; WarpStream, Ursa; CORFU, LogDrive
Open table formats	Put-if-absent log object (Delta Lake) or catalog pointer swap (Iceberg)	OCC per table	Seconds to minutes	Delta Lake, Iceberg
Database research	Deterministic replay, or a single CAS point	Every commit	Not on S3	Hyder, Tango; vCorfu
Database at OLTP rate on S3	—	Every commit	Milliseconds	No established design

Open problem: per-commit conflict decisions at millisecond commit rates on S3 (tail contention, small objects)