WHITEPAPER

APACHE FLUSS™:

THE FOUNDATION OF THE UNIFIED STREAMING LAKEHOUSE

A LAKEHOUSE-NATIVE STREAMING STORAGE FOR REAL-TIME ANALYTICS, CONTEXT ENGINEERING, AND AI.

AUTHORS

GIANNIS POLYZOS
Ververica

JARK WU
Alibaba

CONTENT

CONTENT

INTRODUCTION

Apache Fluss© is an open-source, lakehouse-native storage layer designed to unify the fragmented infrastructure that real-time analytics, Machine Learning (ML) pipelines, and AI systems require. This paper traces the architecture that makes that possible, from the storage engine internals to the stateless compute model to the unified feature and context store. We start with Ververica Streamhouse™ because Fluss is its foundational layer, and the clearest way to understand what Fluss is built to do is to see where it sits in a production intelligent system.

PROLOGUE

VERVERICA STREAMHOUSE™

& THE ROLE OF APACHE FLUSS

WHAT IS VERVERICA STREAMHOUSE?

Ververica Streamhouse is a unified streaming lakehouse platform built around a single guiding principle: one copy of data, serving every workload. These workloads include streaming ingestion, low-latency operational analytics, batch processing, machine learning features, and AI context; all without incurring a multiple systems tax that conventional architectures accumulate. Streamhouse combines an open-source streaming storage layer (Apache Fluss), an open lakehouse tier (like Apache Paimon™ or Apache Iceberg®), a unified batch-and-stream processing engine (VERA, 100% compatible with Apache Flink®), declarative data pipelines (Materialized Tables), a freshness-aware workflow scheduler, and an autoscaling service (Autopilot) into a single operational substrate.

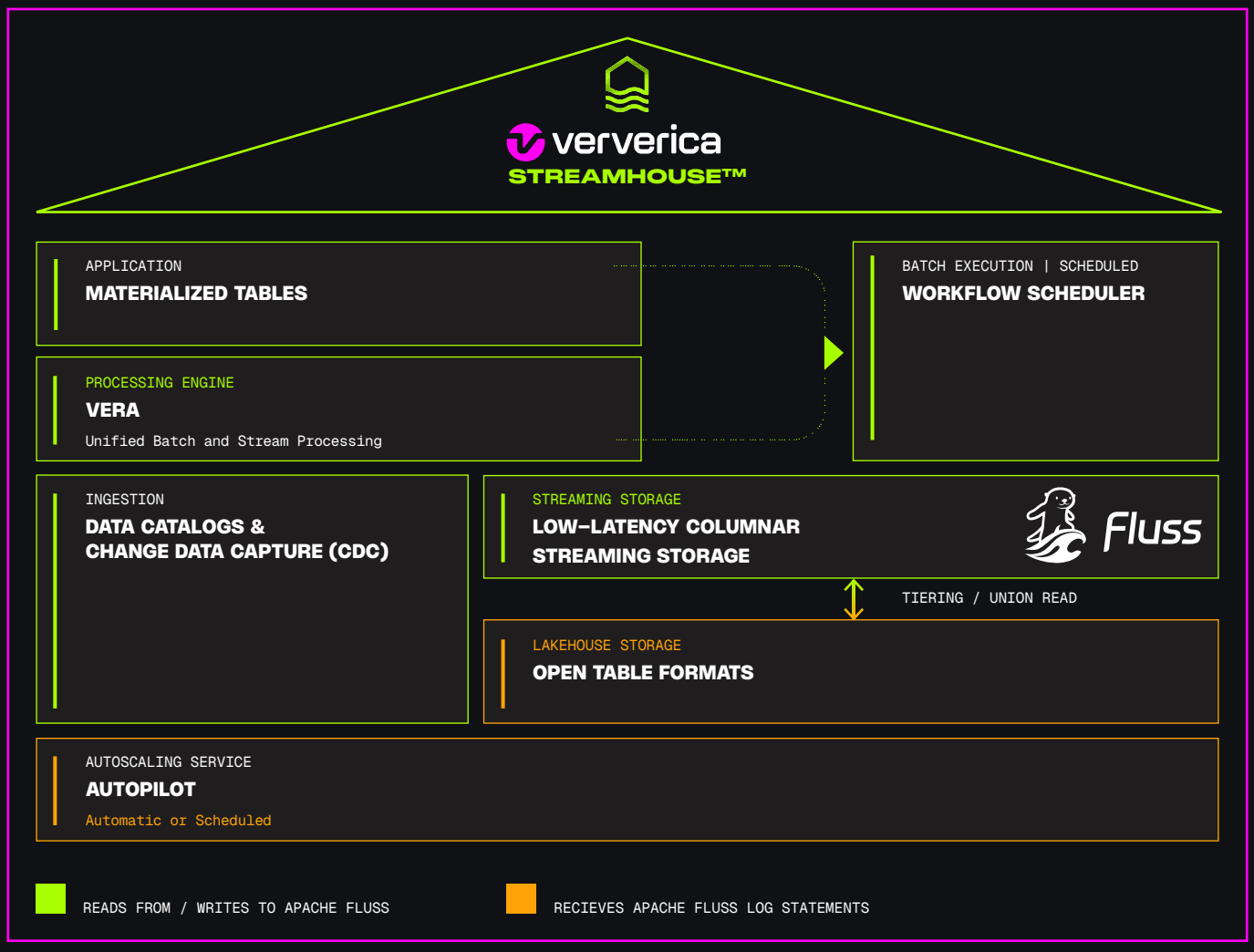


DIAGRAM 1: STREAMHOUSE REFERENCE ARCHITECTURE

Apache Fluss occupies the streaming storage layer, the low-latency columnar substrate on which the rest of Ververica's Platform stands.

THE ROLE OF APACHE FLUSS

Apache Fluss is the low-latency columnar streaming storage layer at the heart of Ververica's Streamhouse. Everything pictured in the diagram above, including Materialized Tables, VERA, and Workflow Scheduler, reads from and writes to Fluss. Everything pictured below Fluss, including the open lakehouse tier on object storage, receives Fluss's log segments through the Tiering Service, and is transparently readable from above via Union Read (covered below in Part IV).

Fluss collapses what would otherwise require several individual components, including:

» **A MESSAGE QUEUE** (Apache Kafka®)
» **A KEY-VALUE STORE** (Redis®)
» **OLAP HOT STORE** (StarRocks, Apache Doris, ClickHouse)

Fluss combines the functionality of these otherwise separate technologies into a single storage primitive with three access modes (KV lookup, streaming log read, and columnar batch scan) on the same underlying data.

This whitepaper is a deep technical investigation of how Fluss achieves this, based on the architecture that is captured in diagram one above.

THE SIX CAPABILITY PILLARS

There are six distinct capability pillars that Streamhouse delivers which support and benefit engineering teams. Each pillar is a direct consequence of one or more of the architectural mechanisms explored in this whitepaper (Parts II–VII) and each maps to a concrete operational outcome.

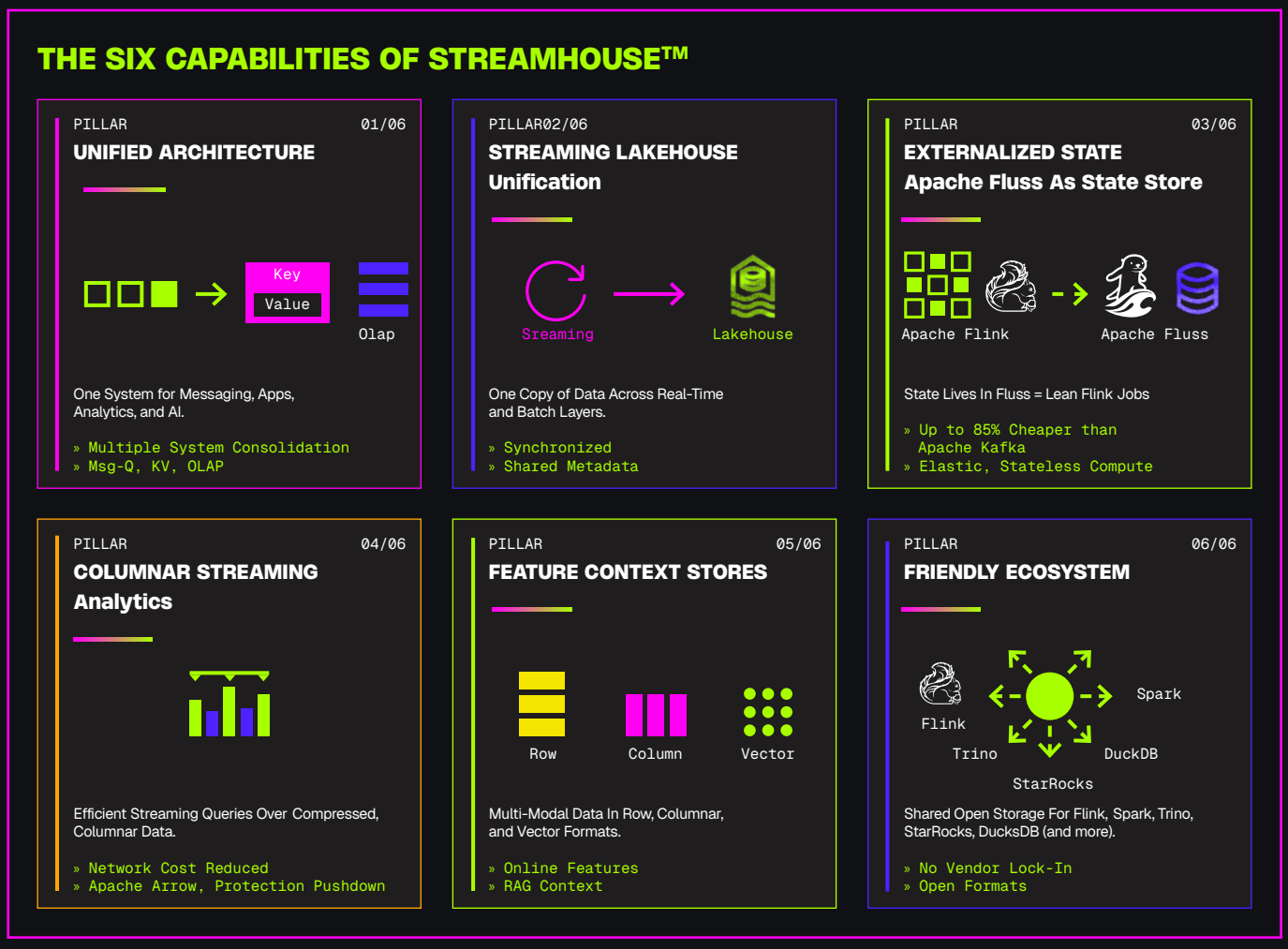


DIAGRAM 2: THE SIX CAPABILITY PILLARS OF STREAMHOUSE
The operational benefits of Streamhouse, each grounded in a specific architectural mechanism explored in depth later in this paper.

CAPABILITY 1
UNIFIED ARCHITECTURE

One system for messaging, applications, analytics, and AI; consolidating the roles previously played by a message queue, a key-value store, and an OLAP engine. The architectural basis is the dual representation of PK Tables (log + RocksDB KV on the leader) explored in Parts II–III.

CAPABILITY 2
STREAMING & LAKEHOUSE UNIFICATION

One copy of data across the real-time and batch layers, with synchronised metadata. The architectural basis is the Tiering Service and Union Read mechanism explored in Part IV. The hot Fluss tier and the cold open-format tier share the same logical schema and are queryable as a single substrate.

CAPABILITY 3
EXTERNALIZED STATE

State lives in Apache Fluss, not in Apache Flink, enabling lean, elastic, stateless compute with fast recovery, up to 85% cheaper than comparable Kafka-based topologies. The architectural basis is the stateless compute model explored in Part VI. Using Fluss as a state store significantly decreases recovery time and decouples Recovery Point Objective (RPO) from Recovery Time Objective (RTO).

CAPABILITY 4
COLUMNAR STREAMING ANALYTICS

Efficient streaming queries over compressed columnar data, with server-side projection pushdown in ARROW format, reducing network transfer costs materially. The architectural basis is the compound pruning stack explored in Part V. Column projection, predicate pushdown, and partition pruning compound into order-of-magnitude reductions.

CAPABILITY 5
FEATURE & CONTEXT STORES

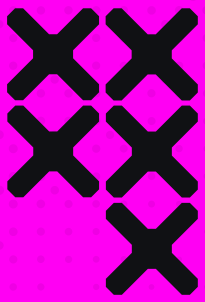
Multi-modal data in row, columnar, and vector formats, online feature serving and RAG-ready semantic context on the same substrate. The architectural basis is the unified substrate explored in Part VII. The feature store, context store, and real-time entity profile collapse into one PK Table accessed through different views.

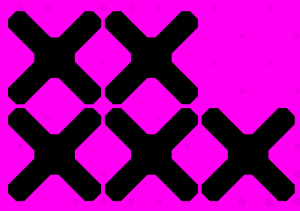
CAPABILITY 6
ECOSYSTEM OPENNESS

Shared open storage readable by Flink, Apache Spark™, Trino, StarRocks, DuckDB, open formats with no vendor lock-in. The architectural basis is Streamhouse's commitment to open lake formats (Paimon, Iceberg, Apache Lance™) in the cold tier, with native connectors for the hot Fluss tier.

HOW TO READ THE REST OF THIS PAPER

The remainder of the whitepaper is organized as a top-down drilldown into the Fluss storage layer that makes all six capability pillars possible. Part I sets up the problem space, including the conventional multiple systems tax Streamhouse is designed to eliminate. Parts II–VII trace, in order: the topology, the storage primitives, the lakehouse foundation, query execution, the stateless compute model, Flink integration, and the unified feature/context store. Finally, the Appendix is an operator's reference for configuration, split lifecycles, and KV snapshot mechanics.





PART I

THE PROBLEM SPACE

WHY EXISTING INFRASTRUCTURE FALLS SHORT

No organization builds a real-time AI data platform from scratch on day one. They arrive at it gradually, through a sequence of expanding ambitions, each stage solving a real problem, each stage adding infrastructure that makes perfect sense at the time of implementation.

HOW ORGANIZATIONS ARRIVE AT REAL–TIME AI INFRASTRUCTURE

STAGE 1

STREAMING INGESTION & CHANGE DATA CAPTURE (CDC)

The journey most enterprises encounter with building a real-time infrastructure almost always begins the same way: a growing operational database that batch ETL jobs can no longer drain fast enough, and a first encounter with a message broker.

Engineers commonly start by deploying Apache Kafka and begin streaming application events (including clickstreams, transactions, user actions) into a data lake. Alongside this, Change Data Capture (CDC) tools tap the transaction logs of operational databases and publish every insert, update, and delete as a stream of events. This is a foundational act that liberates data from the boundaries of a single operational system and makes it available to downstream consumers in near-real time.

At this stage, infrastructure is relatively simple. The challenges are primarily operational, and include connector reliability, schema change handling, partition management, and consumer lag monitoring. At this initial stage, “real-time” means seconds to minutes.

STAGE 2

STREAMING ANALYTICS & OPERATIONAL DASHBOARDS

Once data is flowing continuously, buwsiness teams quickly discover that waiting until tomorrow morning for yesterday‘s numbers is untenable. The demand for live operational dashboards emerges, often with requests for active sessions, revenue by the minute, error rates, and funnel conversion metrics provided in real time.

In response, engineers implement a stream processing framework like Apache Flink, which begins consuming event topics and materializing business-relevant metrics like windowed aggregations for counts in the last five minutes, rolling 24-hour totals, and session activity metrics. The patterns introduced here (like tumbling windows, sliding windows, watermarks, late-arriving data handling) solve a genuinely hard set of problems. But they also introduce the first taste of infrastructure complexity: Flink jobs carrying internal state must be checkpointed and recovered, and as a result, friction grows due to needing to query both a streaming store and a historical batch store in the same pipeline.

At this stage, everything still works, but tension in the infrastructure is already beginning to show, and will snowball with time.

STAGE 3

REAL-TIME DATA PROCESSING & COMPLEX EVENT PROCESSING

The third phase is the shift from observing data to acting on it. Use cases like fraud detection, dynamic pricing, and real-time personalization require events to be enriched against reference data, joined across multiple streams, and evaluated against rules, all within a latency budget measured in milliseconds.

This is where infrastructure begins to fracture. Enrichment joins against reference tables require either broadcasting the entire reference table into Flink operator state, which introduces bootstrap lag, memory pressure and staleness risk; or routing every event to an external lookup against Redis, which adds network round trips and another entirely separate system to operate. The stateful Flink job that seemed elegant at small scale becomes a fragile, slow-recovering system once at production scale.

STAGE 4

REAL-TIME MACHINE LEARNING & FEATURE ENGINEERING

The fourth forced change occurs when data science teams deploy ML models that need to score events in real time. These models are retrained nightly on historical data and consume feature vectors computed from the same streaming pipelines that serve operational analytics. This is where the gap between streaming infrastructure and ML infrastructure becomes painful.

Training pipelines read from an offline store (typically Iceberg or Apache Parquet on Amazon S3) while serving pipelines read from an online store (like Redis or Amazon DynamoDB). But because these two stores are populated by separate pipelines, the data begins to diverge silently over time. This phenomenon, known as training-serving skew, is the most common source of silent ML degradation in production. It is not a code bug; it is a structural consequence of a two-store architecture.

STAGE 5

REAL-TIME AI SYSTEMS & CONTEXT ENGINEERING

The current frontier is real-time AI systems. These are pipelines in which large language models, retrieval-augmented generation, and agentic workflows operate on data current to the second. One example of such a real-time AI system is a loan advisory system that explains a credit decision in natural language, grounded in the applicant's live feature vector, current policy rules, and semantically relevant documents. An agentic underwriting assistant then re-evaluates in-flight applications the moment a policy rule changes.

Use cases and complex systems like this require everything the previous stages required, plus three new categories of data infrastructure:

- **SESSION MEMORY** a durable, replayable record of the agent's interaction history
- **ENTITY MEMORY** a consistent, live key-value store of facts about each entity the AI reasons about
- **VECTOR KNOWLEDGE BASE** a continuously-updated semantic index over documents and policies

Each new piece of infrastructure adds another consistency boundary and an additional system to operate. The multiple systems tax of the traditional ML feature store compounds further for the AI-native platform, with more systems, more boundaries, and more divergence.

SUMMARY

A PATTERN EMERGES: THE INFRASTRUCTURE FRACTURES

The pattern across all five stages is the same: each new capability is layered onto the infrastructure that came before it, and each reasonable request adds a new specialized system to each layer boundary. The result is not a platform, but rather a stack of systems, each with its own operational model, and each introduces a new surface for data to silently diverge.

In addition, because data architectures use multiple specialized systems that each handle one specific job (event logs, key-value lookups, analytics, etc), teams end up exerting enormous effort and cost to keep the systems in sync. Eventually, as complexity grows, the architecture simply fails.

The question this whitepaper addresses below is whether a fundamentally different design that is built from the ground up utilizing a unified, lakehouse-native streaming storage primitive, can collapse the majority of that stack into a single real-time infrastructure foundation. With Fluss, the answer is yes.

THE INFRASTRUCTURE REAL–TIME INTELLIGENCE ACTUALLY REQUIRES

In order to successfully build a unified, single-source-of truth infrastructure, there are specific feature requirements. These include:

- **EVENT INGESTION** Low-latency, durable event ingestion with replay semantics, equivalent to what Kafka provides, but without requiring Kafka as a separate system.
- **FAST KV SERVING** Consistent key-value serving equivalent to what Redis provides, but without a separate store that diverges from the log. In addition, any entity's current state must be accessible via a sub-millisecond KV lookup from the same system that maintains the event history for that entity.
- **STREAMING FEATURES** Streaming feature computation including aggregated signals like rolling sums and velocity counts, without requiring long-running stateful Flink operators that are expensive to recover and impossible to scale elastically.
- **HISTORICAL STORE** A durable historical store in an open format that training pipelines, analytical queries, and regulatory audits can read with arbitrary time range and column filters, without duplicating data into a separate offline system.
- **SEMANTIC LAYER** A semantic knowledge layer for document retrieval and vector similarity search, schema-consistent with the structured feature store.
- **AUDIT TRAIL** A complete, deterministic audit trail provided as a first-class architectural primitive, not an afterthought.

THE LAKEHOUSE–NATIVE HYPOTHETICAL

What if a single storage layer could natively handle all six requirements at once: append-only event log, live KV store, streaming computation substrate, cold analytical archive, vector-compatible data layer, and audit trail, while tiering seamlessly to open formats on object storage?

You'd eliminate the synchronization problem entirely. No divergent systems. Nothing to sync.

No such system has existed. Until now.

Apache Fluss is that single layer. Because the data never diverges by design, the synchronization headache doesn't get managed. It disappears.

THE MULTIPLE SYSTEMS TAX

A conventional real-time feature store at enterprise scale requires several separate infrastructure components:

- A **MESSAGE BROKER** (Kafka) for streaming event transport
- A **STREAM PROCESSOR** (Flink or Spark) for computing derived features
- An **ONLINE STORE** (Redis or Amazon DynamoDB) for sub-millisecond feature lookup
- An **OFFLINE STORE** (Iceberg or Parquet on Amazon S3) for training and historical analysis
- And a **SYNCHRONIZATION LAYER** to keep the online and offline stores consistent, monitor freshness gaps, and manage schema changes across every system in the stack

Every boundary between these systems is a seam where data can silently diverge. The sum of managing this is the multiple systems tax, which requires continuous engineering investments not used for building better models, but with keeping the infrastructure consistent.

LLM-based and agentic systems multiply this tax further. Diagram Three demonstrates what a fragmented multi-system stack looks like in comparison to an infrastructure consolidated into a single Fluss-centric substrate. With Fluss, sources (APIs and clients) write directly to Fluss via either the Flink or Fluss clients, Flink runs as a stream processing layer on top of Fluss, with the lakehouse acting as a cold tier, all surfaced through a unified Union read path.

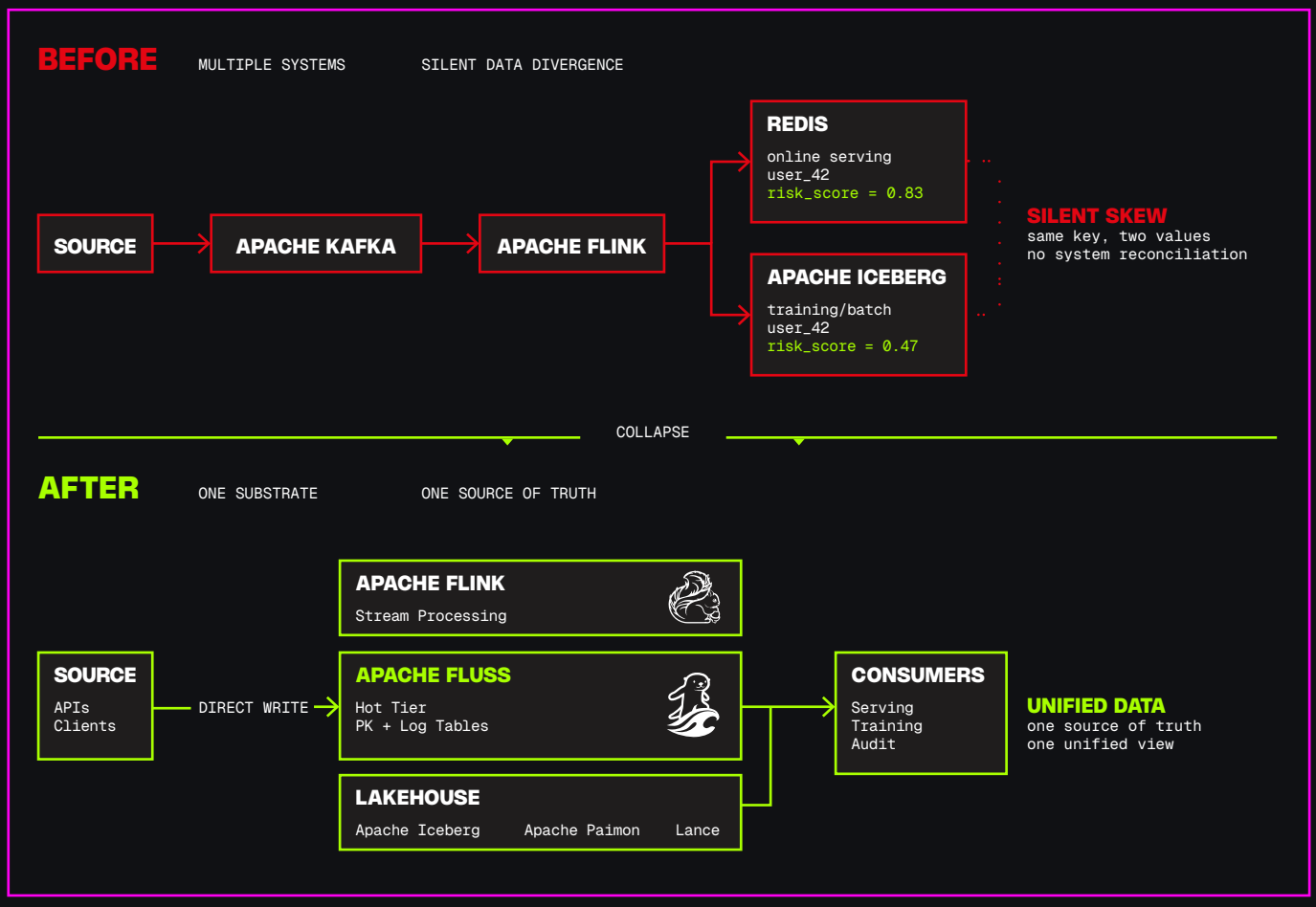


DIAGRAM 3: MULTIPLE SYSTEMS TAX VISUALIZATION: BEFORE AND AFTER
A fragmented multi-system stack consolidated into a single Fluss-centric substrate.

PART II

SYSTEM ARCHITECTURE

THE CLUSTER AS ONE LOGICAL SYSTEM

DISTRIBUTED SYSTEM TOPOLOGY

As shown below in Diagram Four, a Fluss cluster consists of three logical tiers: a CoordinatorServer (the cluster brain), a fleet of TabletServers (the data plane), and a metadata store. Remote object storage (Amazon S3 or compatible) is a first-class participant for the remote tier, written to by the TabletServers themselves through the internal log-tiering subsystem and the KV-snapshot process; the external Flink-based Tiering Service consumes from there and projects the data into open lake formats in the Lakehouse tier. Clients operate against the TabletServers directly after routing metadata is resolved from the CoordinatorServer.

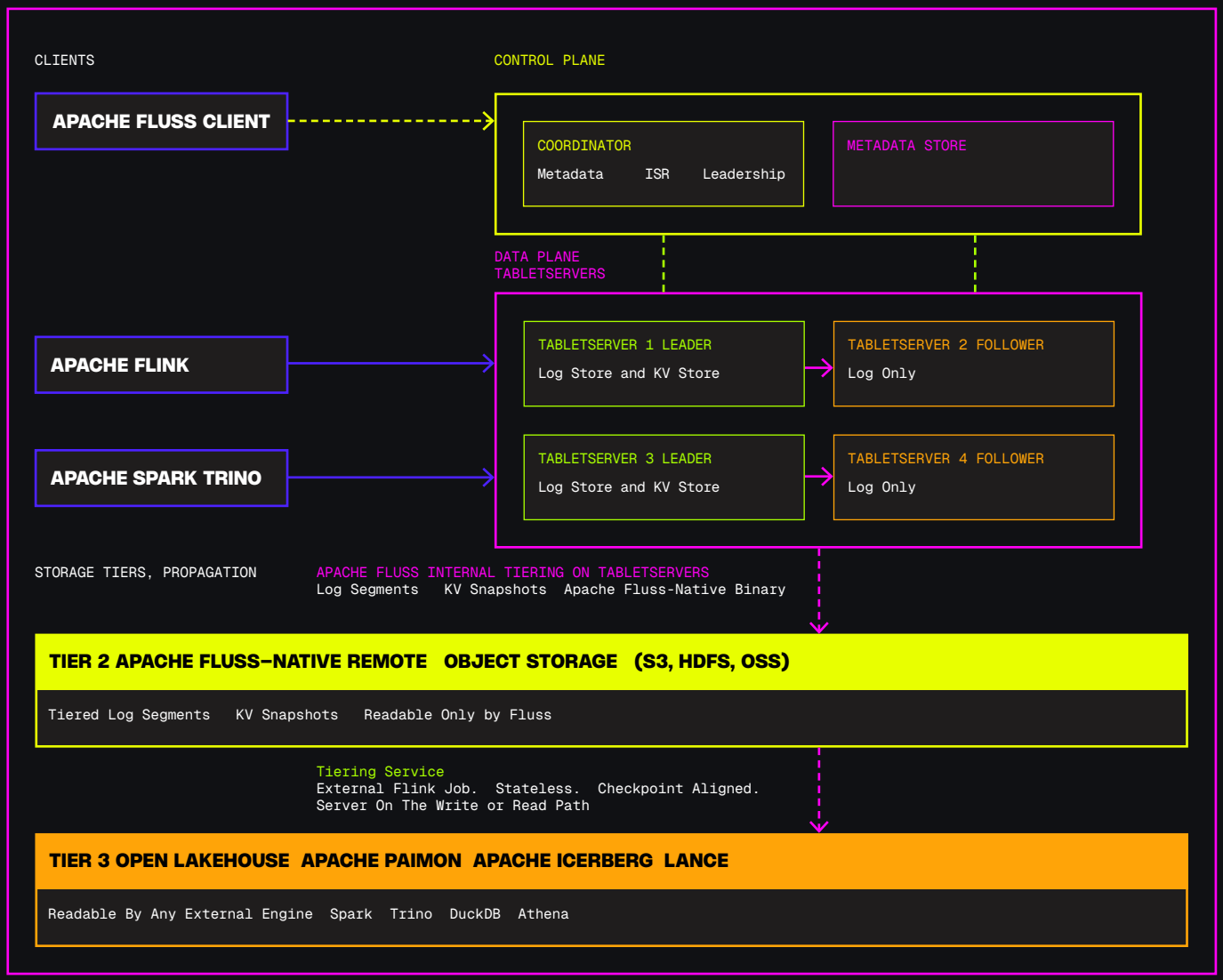


DIAGRAM 4: FLUSS CLUSTER TOPOLOGY

Clients contact the Coordinator for metadata and then write directly to TabletServer leaders, which replicate to followers over the ISR protocol. The TabletServers themselves tier sealed log segments and KV snapshots into the Remote tier (Fluss-native binary on object storage); the external Flink Tiering Service consumes from the Remote tier and projects the data into the Lakehouse tier in open table formats.

THE COORDINATORSERVER

The CoordinatorServer is the single logical brain of the cluster. It manages In-Sync Replica (ISR) sets, triggers leader re-election on TabletServer failure, coordinates schema evolution, and drives the Tiering Service. It does not participate in the data path, as all reads and writes bypass it once the client has resolved routing metadata.

TABLETSERVERS: LOG & KV STORAGE

Each TabletServer manages a set of buckets. Each bucket maintains two independent storage structures on the leader: a log storage subsystem (append-only segments) and, for PK Tables, a KV storage subsystem (RocksDB LSM-tree). The server exposes three access modes: streaming log reads (offset-based, for Flink CDC sources), KV lookups (primary key, for async lookup joins and feature serving), and batch scans with server-side column projection and predicate filtering (for analytics and training pipelines). All three are served from the same TabletServer process without protocol translation.

STORAGE ENGINE INTERNALS & LOG FORMATS

The log storage engine is an append-only segment store that backs every table's changelog, which is the durable, offset-ordered record of every write. Fluss defines two primary log formats in active use for that changelog: ARROW (columnar Apache Arrow IPC, the default format that enables server-side columnar pruning and predicate pushdown for analytics workloads) and **COMPACTED** (a compact row-oriented changelog format that can be selected for PK Tables to reduce storage overhead for row-at-a-time CDC consumption).

Both Log Tables and PK Tables use ARROW by default for their changelog. On a PK Table the changelog doubles as the write-ahead log (WAL) that drives the leader's KV materialization, so the same ARROW segments that ship to streaming and analytic consumers are also the durable record from which the RocksDB current-state view is rebuilt on recovery. COMPACTED is an opt-in alternative on either table type via `table.log.format`, typically chosen for PK Tables that are consumed primarily as row-at-a-time CDC. The KV storage engine is a RocksDB LSM-tree maintained exclusively on the bucket leader and followers carry no KV state.

The two engines on the leader share no data structures. Their consistency is maintained by the write sequence described later in this paper.

The encoding of rows inside the leader's RocksDB store is governed by a separate setting, `table.kv.format`, which defaults to `compacted`. It is independent of `table.log.format`, the log format controls how records are serialised on disk and over the wire for streaming/analytic consumption, while the KV format controls how the materialized current-state row is encoded for sub-millisecond KV lookup.

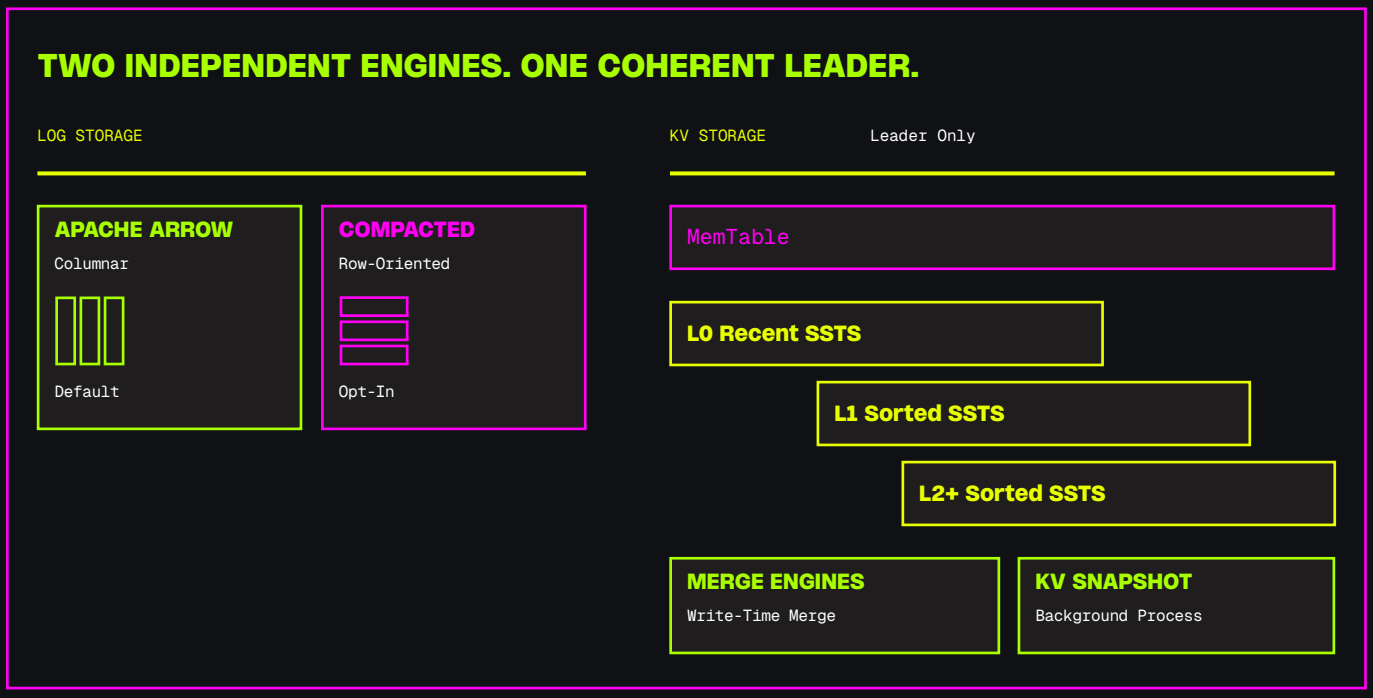


DIAGRAM 5: STORAGE ENGINE INTERNALS

The log storage layer supports two formats (ARROW columnar by default, COMPACTED row-oriented as opt-in). On the leader, a RocksDB LSM tree maintains current KV state alongside merge engines and periodic snapshots.

WRITE PATH: READ PATH, REPLICATION

All writes for a bucket arrive at its leader. The leader appends the record to its local log, stages the KV update in an in-memory pre-write buffer that holds pending writes awaiting ISR quorum acknowledgement, and in parallel replicates the log to the in-sync replicas (ISR). The write is acknowledged to the producer only after the configured ISR quorum has persisted it. Followers carry only the log, no KV state.

The pre-write buffer is consulted by subsequent writes on the same key, so the leader can compute correct UPDATE_BEFORE values for the changelog before its predecessor has been applied, but is not visible to read paths. Pending records may still be rolled back if the ISR quorum fails, so reads bypass the buffer entirely and query RocksDB directly.

Once a write has cleared ISR quorum and the producer ack has been issued, the leader applies the corresponding entry from the pre-write buffer into RocksDB. Only at that moment does the new value become visible to sub-millisecond KV lookups. The visibility window therefore tracks the local RocksDB apply, not the ack itself: an ack means the write is durable on the log; the post-apply read is what guarantees a KV lookup returns the new value. In normal operation this gap is sub-millisecond, but the distinction matters for any test that races a producer ack against a downstream lookup.

The write path differs in one important way between the two table types: a Log Table finishes at the log append with no KV side-effect, while a PK Table also materializes the row into RocksDB on the leader so KV lookups return current state. The figure below traces both paths against the same physical leader.

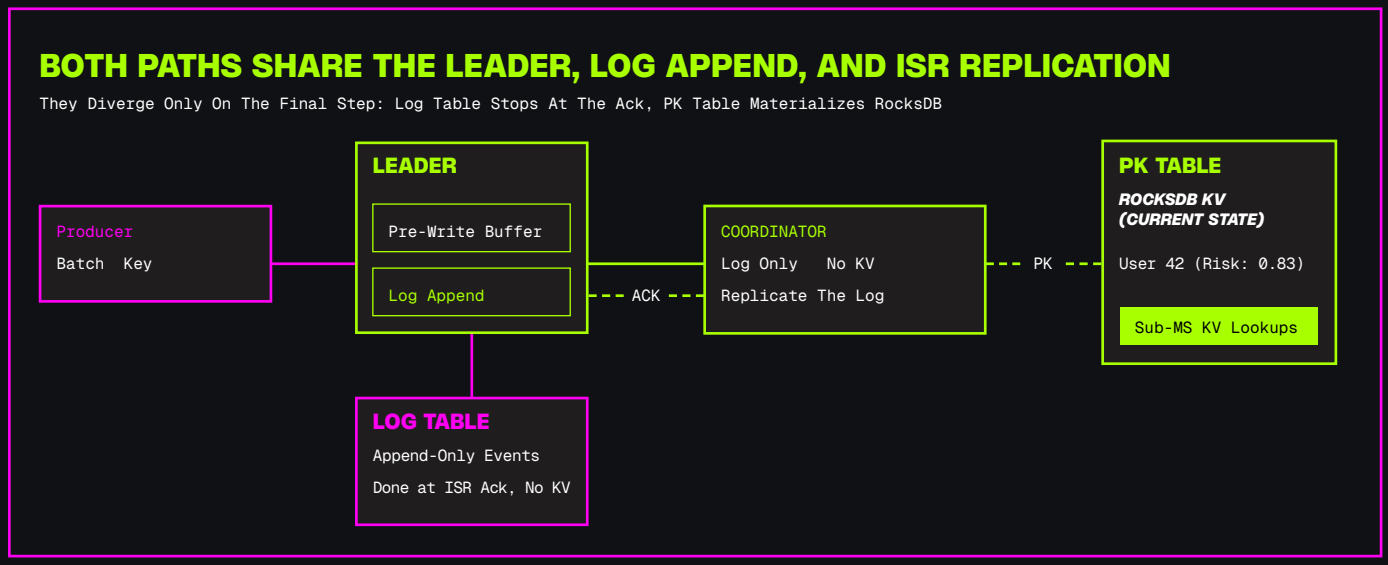


DIAGRAM 6: WRITE PATH: LOG TABLE VS PK TABLE

Both paths share the leader, the pre-write buffer, the log append, and ISR replication. A Log Table finishes at the ISR ack. A PK Table additionally applies the buffered write into the RocksDB KV store on the leader so the row is immediately visible to sub-millisecond KV lookups.

READS TAKE ONE OF THREE SHAPES

» **A KV LOOKUP** hits the leader's RocksDB in sub-milliseconds, as the feature-serving and async-lookup-join path.

» **STREAMING LOG READ** returns an offset-sequential, replayable record stream, used by CDC consumers and the Tiering Service.

» **BATCH SCAN WITH PRUNING** evaluates column projection and predicate filters server-side over ARROW columnar buffers before any bytes cross the wire.

Failover is bounded by the KV snapshot interval and log delta size, not by operator-state size. ISR followers replicate the log in lock-step with the leader, and KV standby replicas continuously pre-fetch the leader's snapshots from remote storage, so the snapshot is already local before failover. On leader loss the CoordinatorServer promotes a surviving follower, which replays the small log delta from the snapshot's offset and begins serving. The remote download is off the critical path.

Flink connects to the same data plane through the Fluss source and sink. For Log Tables and standard PK Tables, exactly-once semantics fall out of deterministic replay from committed log offsets at checkpoint boundaries; no 2PC required.

ARCHITECTURAL SYNTHESIS

The pattern that makes Fluss coherent

Fluss is built on a single principle: **the log is the source of truth.**

The KV store on each leader is not a separate system. It is the same data viewed as a table. This is stream/table duality by design, with no external synchronization layer sitting between the two views. Everything else follows from the log. The cold tier is a derived archive of it. KV snapshots are durability checkpoints created in the background by the leader. All three stay consistent because they share a single write path. One source, three views, and no sync layer.

PART III

STORAGE PRIMITIVES

HERE, WE'LL DISCUSS THREE SPECIFIC STORAGE PRIMITIVES:

LOG TABLES

Log Tables: Handle immutable event streams.

PK TABLES

PK Tables: Deliver full stream/table duality: the same primary-key data readable as either an offset-ordered changelog or a current-state table, in a single system.

CHANGELOG

The changelog: Supports five semantically precise change types: insert (+I), before-update(-U), after-update(+U), delete(-D), and accumulate(+A). This is enough to power Change Data Capture (CDC), point-in-time training, and regulatory audit without any secondary infrastructure.

LOG TABLES: THE APPEND-ONLY EVENT STORE

Log Tables store immutable, offset-ordered event sequences. Unlike a classic Kafka topic, a Fluss Log Table is directly queryable: the default ARROW (columnar) format lets the server evaluate column projection and predicate pushdown against the log itself, so a downstream consumer pays only for the columns and rows it actually needs.

The same table is also a live streaming source for Flink, a point-readable record store for an analytic engine, and a checkpointable archive on the cold tier, without any of the auxiliary connectors, schema-registry hops, or sink jobs a topic-based pipeline would require.

Log Tables are the natural home for transaction events, clickstreams, application lifecycle events, and agent session records. Records in Log Tables use the `+A` (APPEND_ONLY) change type, semantically distinct from the `+I` (INSERT) type used in PK Table changelogs.

(Note for engineers debugging downstream CDC: some Flink consumers surface Log Table appends as `+I` rather than `+A`, because Flink's `RowKind` enum has no APPEND_ONLY value and the connector maps it to INSERT on conversion. The on-disk record type is still `+A`; only the consumer-facing `RowKind` is reshaped.)

PRIMARY-KEY TABLES: STREAM AND TABLE IN ONE SUBSTRATE

A PK Table is the literal embodiment of stream/table duality: the same data is exposed as both an ordered, durable changelog (in ARROW format by default, with COMPACTED available as an opt-in via `table.log.format`) and a current-state KV view of the latest row per primary key. When a write arrives, both views advance together on the leader. They are maintained in close consistency with no external synchronization pipeline, the changelog is the source of truth, and the KV view is the same data materialized. As described in Part II ("Write path, read path, replication"), the log-visibility step and the KV flush proceed in sequence rather than as a single atomic commit; reads consult the pre-write buffer so that KV lookups reflect the latest committed writes.

A PK Table can optionally configure a merge engine via `table.merge-engine` to control how an incoming write is combined with the current row. Three values are supported: `FIRST_ROW`, `VERSIONED`, and `AGGREGATION`.

`FIRST_ROW` gives you exactly-once deduplication out of the box. The first write for a key wins; every subsequent write for that same key is silently dropped at the leader. There is no separate deduplication pipeline, no Bloom filter to maintain, no, "have I seen this key before" lookup table to keep in sync. Idempotent ingestion (replays after a connector restart, at-least-once upstream sources, retry storms during incident recovery) all collapse to a one-line table option. The same primary key landing twice produces the same table state as it landing once, by construction. `FIRST_ROW` is also the typical configuration for the append-only operand of a delta join (Part VI), where each key's first observed row is the one that participates in the join.

`VERSIONED` resolves out-of-order CDC ingestion: the row with the largest value in a designated version column wins, regardless of arrival order, so event-time semantics (not arrival-time) determine the winner. `AGGREGATION` wires per-column aggregator factories that combine incoming values into the existing row at write time and is explored in depth in Part VI ("The Aggregation Merge Engine"). When `table.merge-engine` is unset, the table uses last-write-wins replace semantics (handled internally by the default row merger). Partial-updates (Part VI, "Partial-updates") are a separate, writer-driven mechanism on top of PK Tables; they are not a value of `table.merge-engine`.

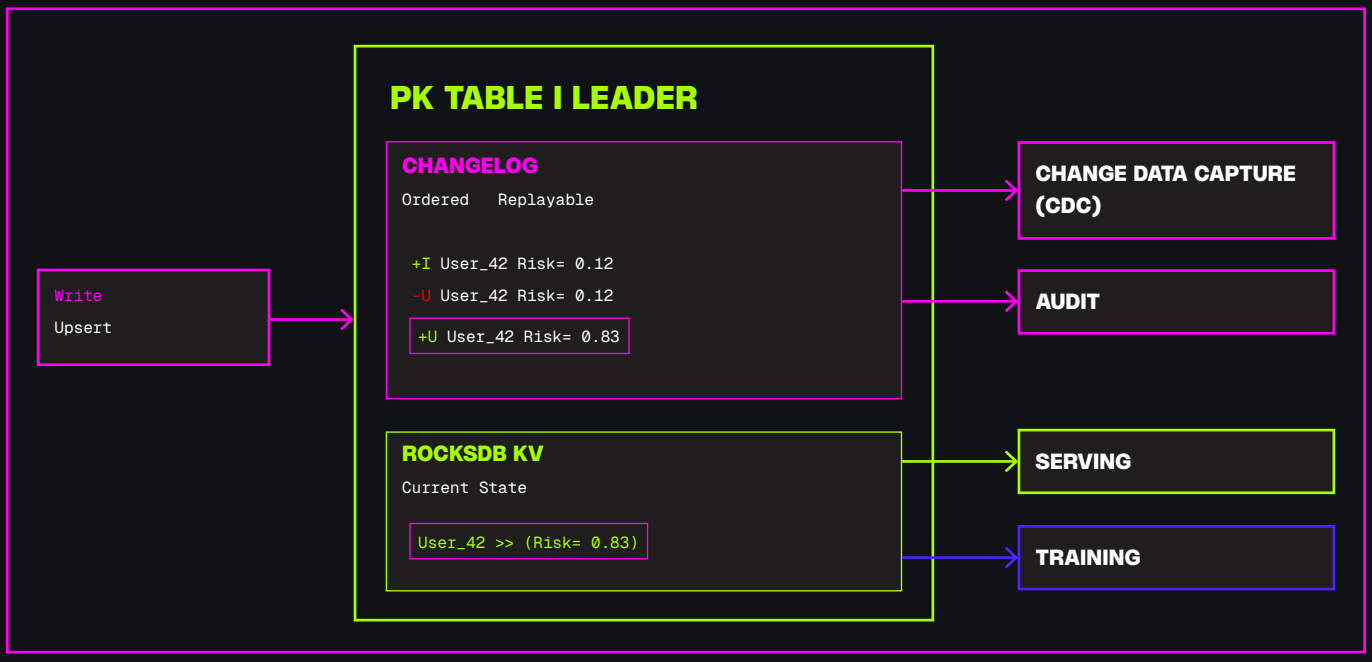


DIAGRAM 7: PK TABLE: DUAL REPRESENTATION

The PK Table on the leader. A single write creates entries in two representations: an offset-ordered changelog (source of truth) and a materialized current-state row in RocksDB. Different consumers read different representations without coordination.

THE CHANGELOG AS A FIRST-CLASS CITIZEN

Anyone who has run a relational database has stumbled on this issue. A database stores a primary-key table on disk, but every write goes through a write-ahead log (WAL) first. The WAL records what changed: insert this row, update that one to a new value, delete this one. The table is a materialized view of those changes, applied row-by-row. If the database crashes, recovery is just "replay the WAL from the last checkpoint and rebuild the table". The table is reconstructible; the WAL is the source of truth. Fluss generalizes that pattern, and exposes the WAL itself as a first-class read primitive: the changelog.

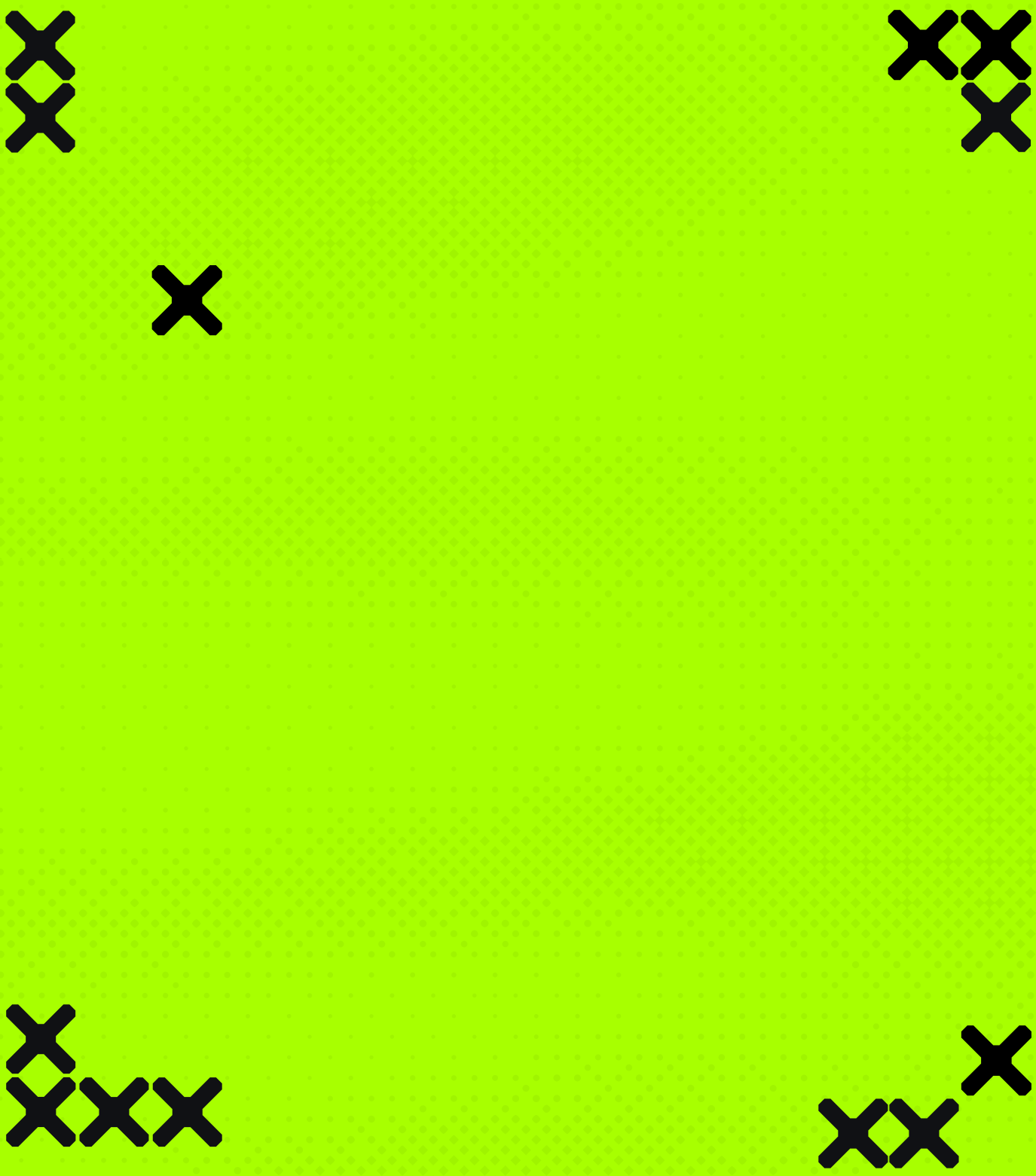
Fluss defines five change types: `+A` (APPEND_ONLY, immutable Log Table events), `+I` (INSERT, new PK Table entity), `-U` (UPDATE_BEFORE, prior state snapshot), `+U` (UPDATE_AFTER, new state after update), and `-D` (DELETE). Log Tables emit only `+A`; PK Tables emit `+I (-U) +U, -D`. Every PK Table update emits a `-U/+U` pair. Every delete emits a -D record carrying the last known row payload, downstream consumers reconstruct the deleted state from the `-D` itself. Together these change types enable point-in-time correct training, streaming CDC consumption, and deterministic regulatory audit without secondary log infrastructure.

DIAGRAM 8: CHANGELOG SEMANTICS

Every PK Table update produces a -U / +U pair. Every delete emits a -D record carrying the last known row payload, downstream consumers reconstruct the deleted state from the -D itself. This enables point-in-time correct reads across the entire history.

PART IV

THE LAKEHOUSE–NATIVE FOUNDATION



THREE STORAGE TIERS, ONE SUBSTRATE

Fluss organizes data into three storage tiers with a single logical substrate:

» HOT LOCAL TIER FOR LIVE WRITES
» FLUSS-NATIVE REMOTE TIER FOR LONG RETENTION AND FAILOVER
» OPEN LAKEHOUSE TIER FOR BATCH AND EXTERNAL ANALYTICS

Data flows through them in sequence: the TabletServers themselves tier sealed log segments and KV snapshots from local disk into the Remote tier, and an external Flink-based Tiering Service then projects from the Remote tier into the Lakehouse tier in open table formats. Each step is decoupled from the live write path.

ONE SUBSTRATE: THREE STORAGE TIERS

The hot tier (Tier 1) holds live writes on TabletServer local disk plus the leader's RocksDB KV plus an in-memory pre-write buffer. Data leaves it in two stages. First, Fluss internal tiering, running on the TabletServers themselves, ships sealed log segments and KV snapshots into Fluss-Native Remote storage (Tier 2) in Fluss's own binary format, readable only by Fluss itself, used for failover, point-in-time training reads, and long-tail streaming consumers. Second, the Tiering Service, an external Flink job, consumes from Tier 2 and projects the log into an open table format (Iceberg, Paimon, or Lance) on object storage (Tier 3) so any external engine can read it. Tier 2 and Tier 3 exist for different reasons, serve different consumers, and operate on independent cadences.

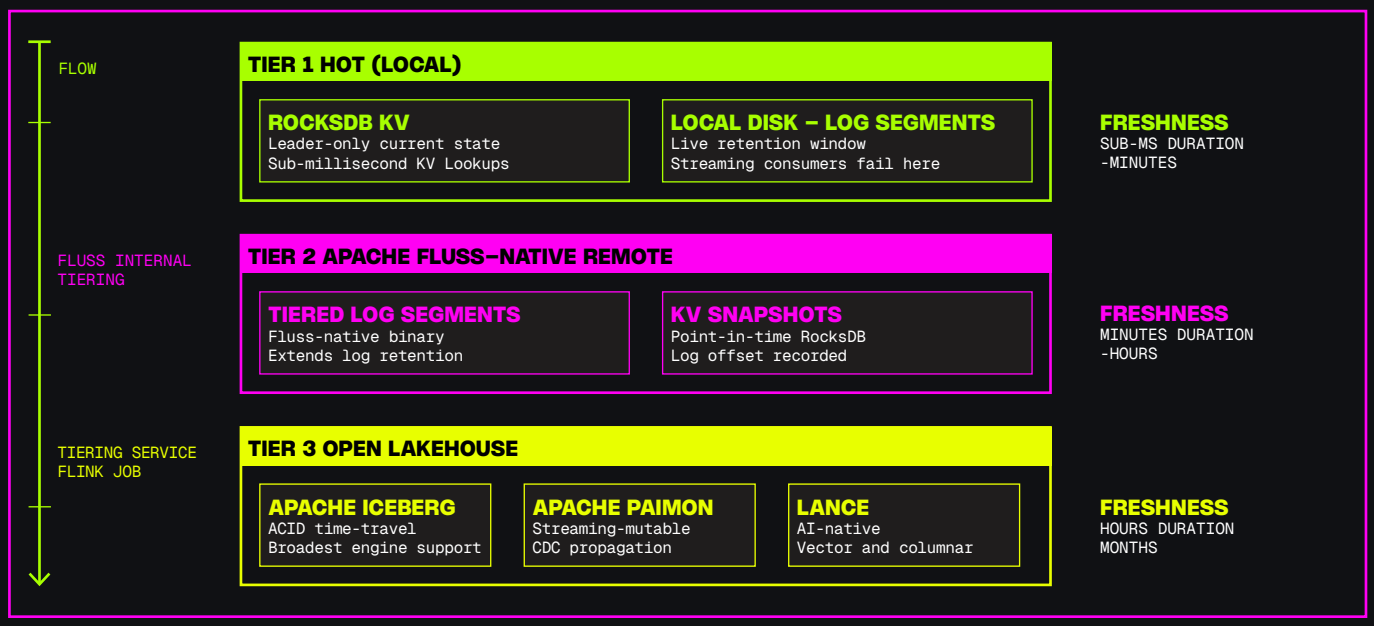


DIAGRAM 9: THREE STORAGE TIERS, ONE WRITE PATH

Three storage tiers, one substrate. Tier 1 holds the leader's RocksDB current state and live log segments on local disk. Fluss internal tiering moves both log segments and periodic KV snapshots down into Tier 2 (Fluss-Native Remote) in Fluss-native binary format, read only by Fluss itself for failover and point-in-time training reads. The Tiering Service Flink job, in parallel, projects the log into Tier 3 (Open Lakehouse) as Iceberg, Paimon, or Lance tables, the format any external engine can read.

PATHWAYS OUT OF THE HOT TIER

Data leaves Tier 1 in two stages, both decoupled from the live write path. The first stage runs inside Fluss on the TabletServers themselves and produces Tier 2; the second stage is an external Flink job that consumes Tier 2 and produces Tier 3. Neither stage blocks a write acknowledgement.

STAGE 1A, FLUSS-NATIVE LOG TIERING INTO TIER 2. A built-in tiering subsystem (`RemoteLogManager` driving per-bucket `LogTieringTasks`) running on the TabletServers themselves continuously copies sealed log segments from local disk to remote object storage in Fluss's own binary format. Once a segment has been tiered and the local retention window has elapsed, it can be reclaimed locally; consumers that need offsets past the local horizon read transparently from the remote archive. This pathway operates on the log only and is internal to Fluss.

STAGE 1B, KV SNAPSHOT CAPTURE INTO TIER 2. A periodic background process on each bucket leader takes a consistent point-in-time view of its RocksDB instance, writes the snapshot to the same Tier 2 remote object storage in Fluss-native binary format, and records the snapshot's log offset in cluster metadata. Snapshots have two jobs: they are the source for leader initialisation on failover (Part II), and they provide the consistent point-in-time materialization used for training dataset construction. Together with Stage 1a, this is the only set of writers into Tier 2, and they are both internal to Fluss.

STAGE 2, THE TIERING SERVICE FROM TIER 2 INTO TIER 3. The Tiering Service is an external Apache Flink job that consumes the Fluss-native log segments already sitting in Tier 2 and projects them into an open lake table format (Iceberg, Paimon, or Lance) on its checkpoint cadence. It operates on the log only, it does not interact with the RocksDB KV store, does not create KV snapshots, and does not write to Tier 2. Tier 3 is the open-format home of the log; any engine that speaks the chosen lake format can read it directly.

MENTAL MODEL

The flow is **sequential, not parallel**: Tier 1 → Tier 2 → Tier 3.

Tier 2 is Fluss-internal, written only by Fluss itself: tiered log segments (from the log-tiering subsystem on the TabletServers) and periodic KV snapshots (from the KV snapshot process on each leader). It exists so Fluss can bootstrap leaders, serve point-in-time reads, and consume offsets past local retention without anything outside Fluss being involved. Tier 3 is the open-format projection of the log, written by the external Tiering Service Flink job from Tier 2 onwards, and is the form any external engine reads. The Tiering Service is therefore not a writer into Tier 2 and not a participant in the Tier-1-to-Tier-2 step at all.

OPEN TABLE FORMATS: CHOOSING THE COLD-TIER TARGET

For an open lakehouse format, Apache Iceberg is an appropriate default with ACID transactions, time-travel, and broad engine compatibility (Spark, Trino, Flink batch, DuckDB, Athena). Apache Paimon is the right choice when the cold tier must participate actively in streaming change propagation, when entities need to be updatable after aging out of the hot retention window. Lance is the right choice when the cold tier serves AI / ML workloads directly, vector search, embedding storage, and random-access feature retrieval at training time, with native columnar storage and zero-copy random access designed for ML pipelines.

- **ICEBERG** Default choice (ACID) time-travel (read by Spark, Trino, Flink batch, DuckDB, Athena) broadest engine compatibility.
- **PAIMON** Choose when cold tier must remain mutable and participate in streaming change propagation after entities age out of the hot retention window.
- **LANCE** Choose when the cold tier feeds AI / ML workloads directly, native vector indexing, fast random row access, and columnar storage tuned for embedding retrieval, RAG, and training pipelines. Note: as of the time of this writing, Lance integration in fluss-lake-lance is the most recent of the three lake integrations and has the smallest production footprint to date; teams choosing it should validate against their workload before committing.

Apache Hudi™ support is also a work in progress in the Fluss community as of the time of this writing, expect it to land alongside Iceberg, Paimon, and Lance as a fourth cold-tier target once the integration stabilizes.

UNION READ: THE BOUNDARY DISAPPEARS

A Flink or Spark job querying data across the hot-cold boundary issues a single query. The system determines which data lives in the hot Fluss tier and which in the cold lakehouse tier, reads from both, and returns a unified result. The compound pruning stack is applied independently in each tier. Training pipelines and monitoring jobs spanning the retention boundary need no custom hot-cold bridging logic, that complexity is absorbed by the storage layer.

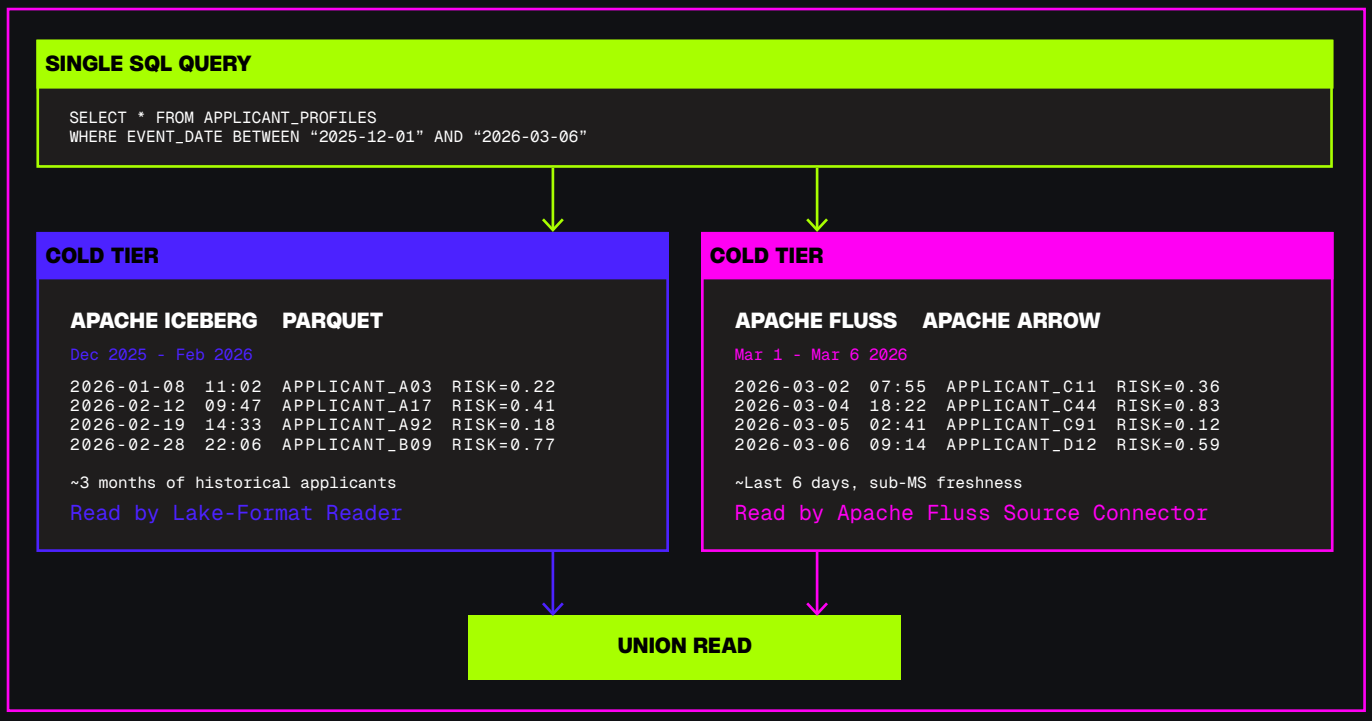


DIAGRAM 10: UNION READ, ON QUERY, TWO TIERS

Union Read. A single SQL query reads transparently from both the cold lakehouse tier and the hot Fluss tier. The engine handles the hot-cold boundary; application code has no bridging logic.

PART V

QUERY EXECUTION EFFICIENCY

COMPOUND PRUNING: THREE MECHANISMS, ONE STACK

Column pruning, predicate pushdown, and partition pruning each independently reduce the data that leaves the TabletServer. Applied together from a standard SQL expression, they compound into order-of-magnitude reductions in I/O, network bandwidth, dezerialisation cost, and task-slot memory pressure.

Fluss stores its log in Apache Arrow columnar format by default, the same format that anchors modern analytic engines. Because each column is a contiguous, self-describing byte array, the storage layer can answer a query the way a database would, not the way a topic would: it can project, filter, and partition-prune on the server, against the columnar buffers themselves, before any bytes leave the TabletServer. That is the foundation for the three compounding mechanisms below, each driven by a standard SQL clause and applied automatically through the Flink source connector.

This applies equally to PK Tables: their changelog (the WAL) is also ARROW by default, so the same column-projection, predicate-pushdown, and partition-pruning path runs against PK Table reads. The pruning path applies to any table whose `table.log.format` is ARROW; for PK Tables explicitly configured with `table.log.format = compacted`, the log is row-oriented and columnar pruning does not apply.

SERVER-SIDE COLUMN PRUNING

When a consumer specifies a column projection, the server reads only the requested column buffers from ARROW-format storage and transmits only those buffers. Unrequested columns never touch the network or client memory. For a 50-column feature table where a model requests 3 columns, this yields approximately a 94% reduction in disk I/O, network bandwidth, and deserialization cost.

```
SELECT user_id, risk_score, last_txn_ts
FROM applicant_profiles;
```

The `SELECT` list above tells Fluss to read three column buffers, `user_id`, `risk_score`, and `last_txn_ts`, and skip the remaining 47. The other columns are never deserialized, never sent across the network, and never allocated in the consumer's heap.

SERVER-SIDE PREDICATE PUSHDOWN

A predicate pushed to the server is evaluated before rows are transmitted. Filters are evaluated against the per-batch statistics carried in the Arrow IPC metadata (per-column min/max/null-count summaries), so an entire Arrow batch can be skipped or accepted without ever decoding the row buffers underneath. No rows are deserialized on the storage node, and only the surviving batches cross the wire, the evaluation cost collapses to a handful of statistical comparisons per batch rather than per row. The Flink source connector propagates predicates automatically from `SQL WHERE` clauses.

```
SELECT user_id, risk_score
FROM applicant_profiles
WHERE risk_score > 0.8
  AND country = 'DE';
```

The `WHERE` clause is evaluated server-side against the `risk_score` and `country` Arrow vectors; rows below the threshold or in other countries are dropped before they cross the wire.

PARTITION PRUNING AND THE COMPOUND EFFECT

Partition pruning eliminates entire data partitions before any storage I/O begins. For time-bounded queries (which describe almost every training and monitoring job in practice) it is the most impactful of the three mechanisms.

```
SELECT user_id, risk_score
FROM applicant_profiles
WHERE event_date BETWEEN '2026-03-01' AND ,'2026-03-06'
  AND risk_score > 0.8;
```

The `event_date BETWEEN...` predicate restricts the scan to six daily partitions out of potentially hundreds; predicate pushdown then filters by `risk_score` within those; and column pruning trims the remaining buffers down to the two columns actually requested. All three compound simultaneously, transparently, from a single standard SQL expression.

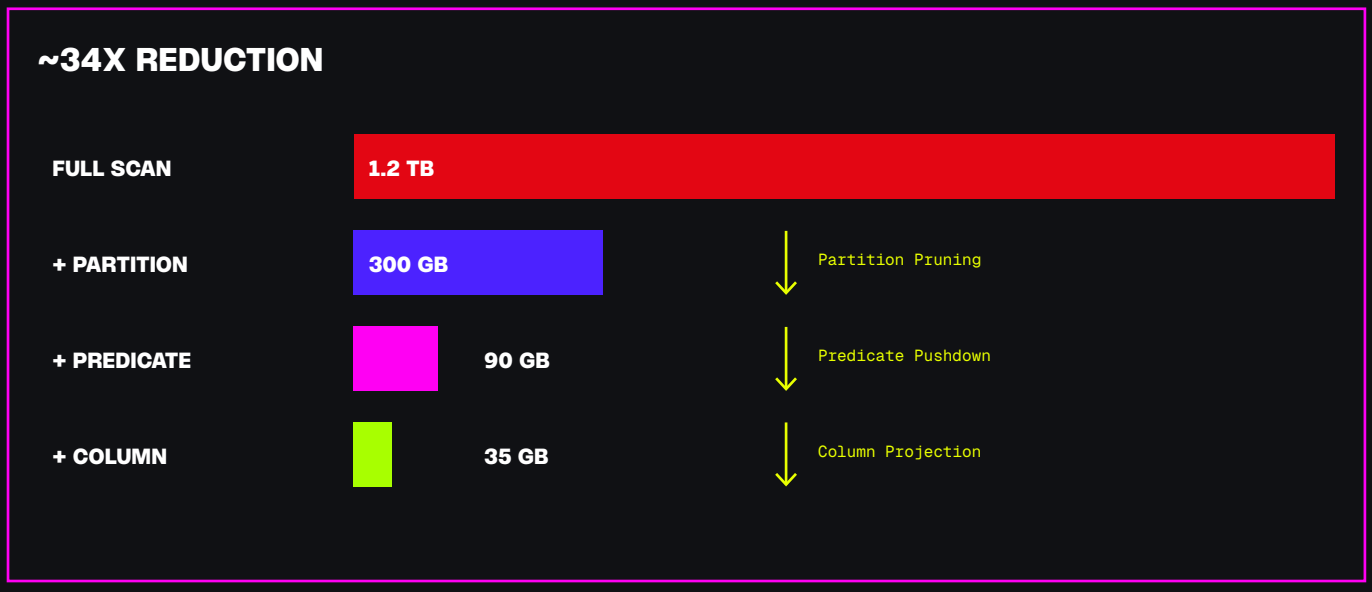


DIAGRAM 11: COMPOUND PRUNING, ILLUSTRATIVE REDUCTION

Compound pruning. Partition pruning, predicate pushdown, and column projection each reduce the data that leaves the TabletServer. Applied together, they compound into order-of-magnitude reductions for a typical analytical query.

SAVING STORAGE IS JUST THE START. WHEN DATA IS PRUNED BEFORE IT EVER LEAVES THE SERVER, IT NEVER TRAVELS THE NETWORK TO REACH FLINK AT ALL. THAT CUTS THE REAL COST OF LONG-RUNNING JOBS.

IMPACT ON NETWORK DATA TRANSFER COSTS

The efficiency story does not end at storage. The three-layer pruning stack has a second-order effect inside the stream-processing layer that often dominates the economics of long-running Flink jobs: bytes that are pruned server-side never cross the network to reach a Flink task manager.

In a conventional architecture, the full scan volume travels over the wire to Flink, which then discards the vast majority of it inside its own processes before any actual computation occurs. In Diagram 11, that proportion is ~97%, and while the precise figure is workload-specific, the structural pattern is universal across partition-selective, predicate-selective, and column-projecting queries. Task managers carry the cost of receiving, deserialising, and filtering those bytes, CPU cycles spent on data that contributes nothing to the final result.

Server-side pruning relocates that work into the TabletServer itself. Partition elimination, predicate filtering over ARROW columnar buffers, and column projection all execute on the storage node, so only the surviving bytes ever reach the Flink source operator. The reduction is not just I/O, it is also deserialization cycles, task-slot memory pressure, and checkpoint barrier alignment time, all of which scale with the volume of bytes that enter the job. For long-running analytical streams and continuous feature computation jobs, this translates to smaller task slots, fewer task managers for the same workload, and tighter back-pressure envelopes during load spikes.

The design consequence is that the filter predicate lives on the data‘s side of the wire, not the compute‘s side. The read path becomes: storage node → local pruning over ARROW columnar buffers → already-filtered bytes on the wire → Flink source receives only what the query needs. The traditional pattern of „read everything, filter in Flink“ produces the inverse, wire-saturating bulk transfers followed by in-memory discard in every task manager. For Flink jobs that re-scan the same fact table continuously under varying predicates (a common pattern for rolling-window feature stores and continuous monitoring pipelines) the difference compounds over every checkpoint interval, not just at job start.

DESIGN CONSEQUENCE

The filter predicate lives on the data side of the wire, not the compute side. Fluss discards irrelevant bytes before they ever reach Flink. The result is fewer task managers, smaller slots, and tighter backpressure, not because the job is better optimized, but because it never touches the data it doesn‘t need.

PART VI

STATELESS COMPUTE MODEL

STATE BELONGS IN FLUSS, NOT IN FLINK

ONE PRINCIPLE, SIX CAPABILITIES

Six capabilities: Stateless stream-stream joins, the Aggregation Merge Engine, async lookup joins, partial updates, Fluss as a state store, and redefined recovery semantics, all following the same principal design decision. Because the state lives in Fluss, via the RocksDB KV store that lives on the leader TabletServer and is accessible via sub-millisecond KV lookup, Flink jobs no longer need to carry state internally.

EXTERNALIZED STATE: APACHE FLUSS AS A STATE STORE

In the conventional Flink model, stateful streaming jobs maintain operator state internally, which makes the state private to a single job, slow to recover, and rigid to scale. Fluss resolves all three by externalizing state into PK Tables, where it becomes a shared substrate. Every Flink job (and every other consumer) reads and writes the same authoritative state via the same APIs, so a feature aggregated by one job is immediately visible to a serving job, a training job, and a downstream enrichment job, without any state-handoff or duplication.

The Flink job itself becomes a pure compute process, so recovery is fast (read current state from the leader's KV store rather than rehydrating operator state), scaling is elastic (no state redistribution between task managers), and state is always accessible via sub-millisecond KV lookup.

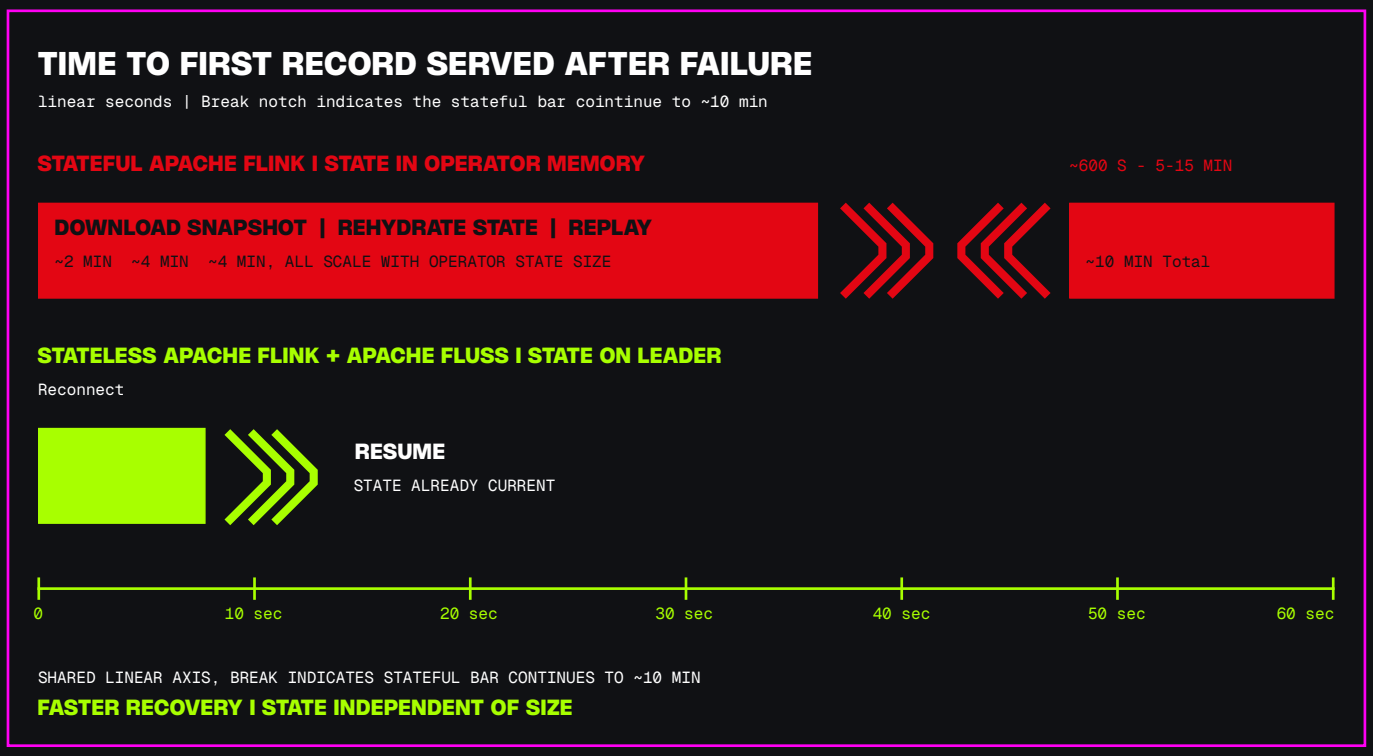


DIAGRAM 12: RECOVERY, STATEFUL VS. STATELESS

Recovery is no longer state-bounded. When state lives on the leader, Flink's recovery problem shrinks to reading an offset and reconnecting. The visual length ratio is the architectural claim.

DELTA JOINS: STATELESS STREAM–STREAM JOINS

The traditional stream-stream join is the heaviest operator in the streaming toolkit. Each side of the join maintains its own full state buffer in the processor memory, waiting for the matching key to arrive from the other side. For joins on high-cardinality keys or wide time windows, this state grows without bound, gigabytes per task slot, checkpoints that take minutes to snapshot, recovery that takes even longer.

The delta join inverts the pattern. Because Fluss PK Tables already maintain the live current state of every entity on the leader, the stream processor no longer has to buffer either side. The Flink planner detects the pattern and rewrites the join into a pair of index-key lookups against Fluss: when a record arrives on the left input, the planner-generated operator performs an index-key lookup against the right-side Fluss table; when a record arrives on the right, it performs the symmetric lookup on the left. Both sides are persisted, both sides are queryable, neither side lives in operator memory. The join becomes stateless on the processor, the state is already in Fluss.

Multi-stream delta joins extend the pattern to three or more inputs. The same record-arrives-triggers-lookup protocol generalizes that each incoming record triggers concurrent lookups against every other participating PK Table, and the enriched result emerges once all lookups return. The processor holds nothing across events, no watermark alignment, no state redistribution on scale-out, no recovery cost beyond what the Fluss leaders already provide.

Translating that architectural change into a measured workload makes the cost difference obvious. The same dual-stream join was run against a Kafka and stateful Flink baseline and against Fluss with a delta-join operator under two merge engine configurations (`versioned` and `first_row`).

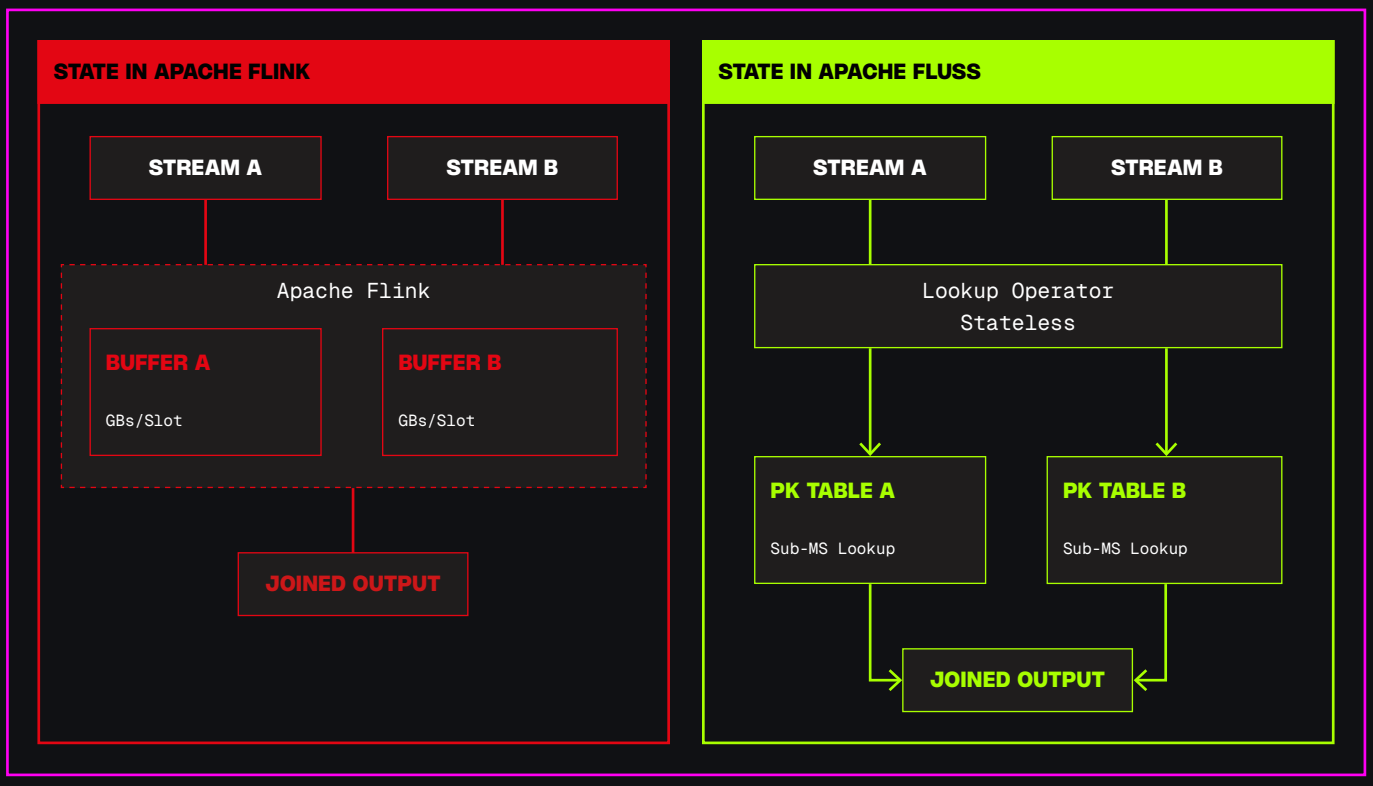


DIAGRAM 13: DELTA JOINS, STATE IN FLINK VS. FLUSS

In a traditional stream-stream join, each side buffers its full state in the Flink operator's memory. In a delta join, both sides live in Fluss PK Tables and the operator performs stateless lookups on each arriving record.

RESULTS: THE FLUSS PATH MAKES THE JOIN MORE COST EFFECTIVE ACROSS EVERY DIMENSION THAT MATTERS: OPERATIONALLY, CPU, MEMORY, ON-THE-WIRE STATE SIZE, NETWORK PAYOUT, AND CHECKPOINT LATENCY.

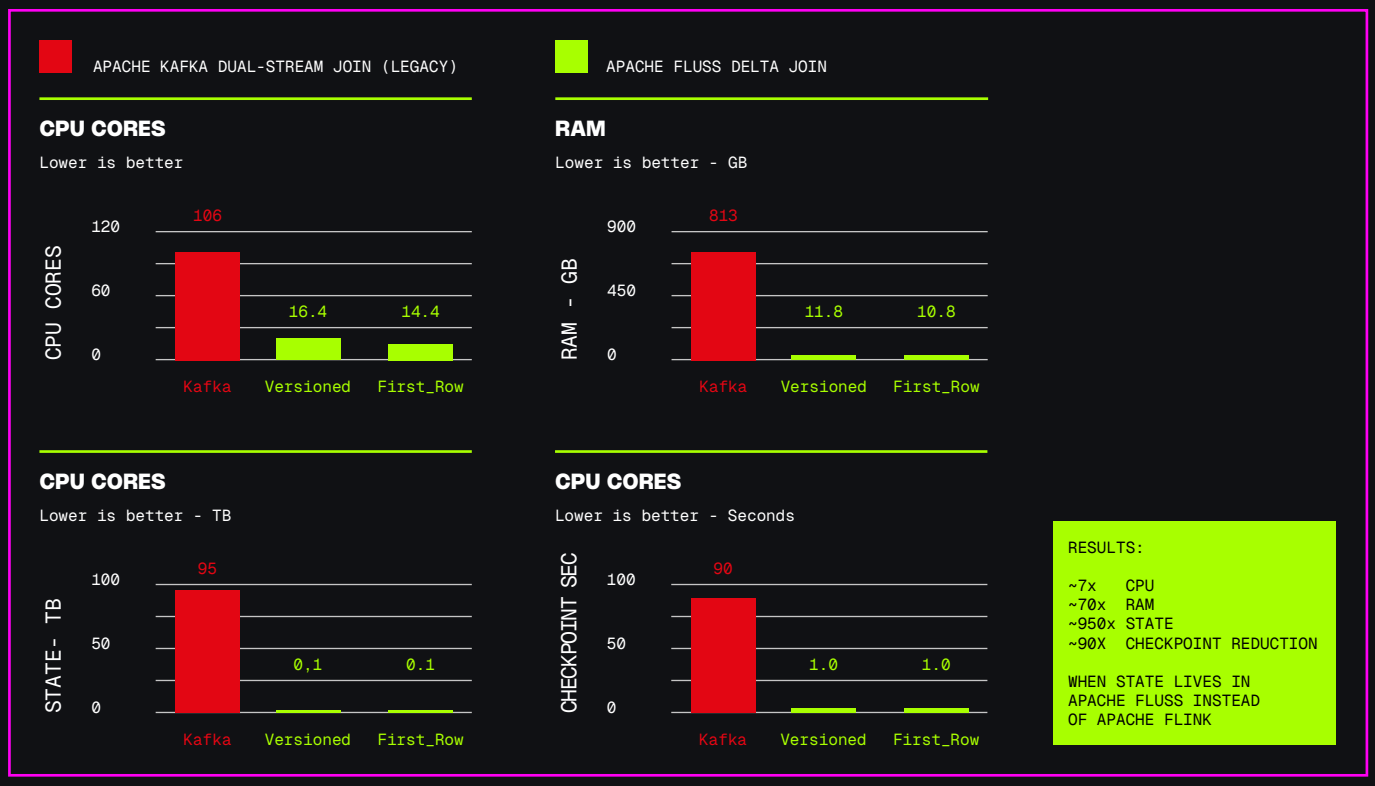


DIAGRAM 14: DELTA JOIN BENCHMARK, KAFKA BASELINE VS. FLUSS

The same dual-stream join expressed three ways. Kafka with stateful Flink (legacy baseline) consumes 106 cores, 813 GB of RAM, 95 TB of operator state, and ~90-second checkpoints. Fluss with the delta-join operator, running with either the versioned or first_row merge engine, runs the same join in ~14–16 cores, ~11 GB of RAM, ~0.1 TB of state, and ~1-second checkpoints. State leaves the operator and lives in Fluss PK Tables; the Flink job becomes a stateless lookup, and every operational dimension drops by one to three orders of magnitude.

THE AGGREGATION MERGE ENGINE

Aggregated features (rolling counts, sums, velocity signals) require stateful accumulation across many writes. Conventionally that work lives in a Flink operator, which carries the running aggregate in its own state and pays the cost of checkpointing it. The Aggregation Merge Engine fills that gap by moving the accumulation step into the leader: when a write arrives for a PK Table configured with aggregation semantics, the server performs a read-modify-write cycle. It reads the current row from RocksDB, applies the configured field-level aggregation logic (examples include `sum`, `max`, `min`, `listagg`, `bool_and`, `bool_or`, plus the bitmap aggregators discussed in Part VII, (see Appendix A.3 for the full reference), and writes the updated result back with a standard RocksDB Put. This happens at write time, not deferred to compaction time.

A Flink job computing 30-day transaction velocity writes only increments; the leader performs the merge on receipt and stores the updated accumulator immediately. Multi-producer aggregation becomes natural, multiple independent Flink jobs write partial increments to the same table concurrently, and the leader serializes and accumulates them correctly. The merged aggregate is always immediately available via sub-millisecond KV lookup. Because of this, no key value store or cache like Redis is required to serve the results.

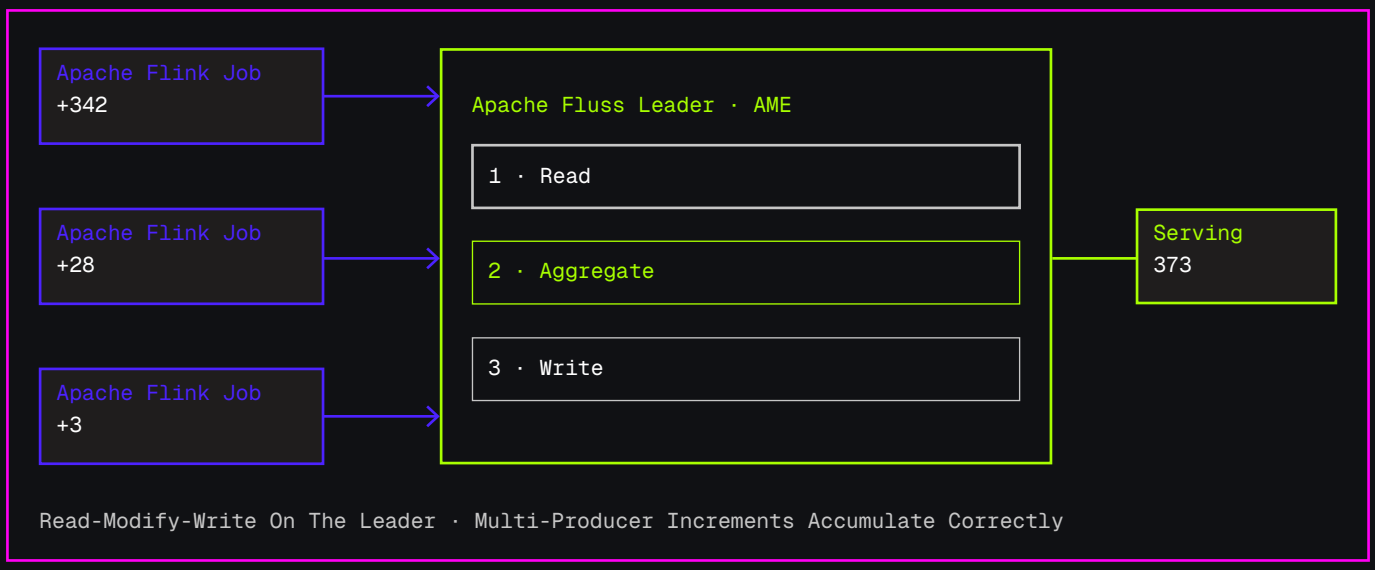


DIAGRAM 15: MULTI-PRODUCER AGGREGATION

Three independent Flink jobs emit increments to the same key. The leader's Aggregation Merge Engine performs a read-modify-write at write time, producing 342 + 28 + 3 = 373 . The result is served via sub-ms KV lookup without a secondary cache.

RECOVERY SEMANTICS: WHAT CHANGES WHEN FLINK STOPS CARRYING STATE

The stateless compute model fundamentally changes what recovery means for a Flink job. In the conventional stateful model, the job's correctness and its ability to resume are tied to its own internal state, the operator state held in each task slot, periodically snapshotted to a checkpoint in object storage. When the job fails, the framework downloads that checkpoint, rehydrates the state into operator memory, and replays events from the checkpointed offset forward. Two cost metrics follow directly: Recovery Point Objective (RPO) (how much data is at risk if the compute layer fails) and Recovery Time Objective (RTO), which is how long before the job resumes serving.

When the state lives in Fluss PK Tables rather than in Flink, the Flink job's recovery problem shrinks dramatically. It no longer needs to checkpoint the state itself, only its committed read offset. A failed Flink job restarts, reads its last committed offset, re-establishes its connection to Fluss, and resumes. There is no gigabyte-scale snapshot to download, no operator-state rehydration to perform, and no multi-minute warmup window during which the job is running but not yet serving correctly. Current values for every entity are already durably committed on the Fluss leader and served via sub-millisecond KV lookup the moment they are needed.

Two axes of recovery now move independently. Durability of records landed in Fluss is per-write, bounded by the ISR log quorum, and independent of Flink checkpoint cadence. Exactly-once visibility of derived Flink output downstream is still bounded by the Flink checkpoint interval, because committed output flips visible at checkpoint barriers. For sinks targeting PK Tables with the Aggregation Merge Engine, the connector additionally uses undo-based recovery to compensate backward on restart, since read-modify-write semantics make naïve replay non-idempotent.

As a result, RTO is no longer coupled to state-snapshot size, instead it tracks job restart plus Fluss reconnect, on the order of single-digit seconds. The practical consequence for platform design: the Flink checkpoint interval becomes a throughput-overhead knob, not simultaneously an RPO and RTO knob.

WHEN STATE LIVES IN FLUSS INSTEAD OF FLINK, RECOVERY SHRINKS TO AN OFFSET READ AND A RECONNECT, JOBS RESTART IN SECONDS, NOT MINUTES, AND CHECKPOINT INTERVAL BECOMES A THROUGHPUT KNOB, NOT A RECOVERY DEADLINE.

JOB EVOLUTION: SCHEMA CHANGES WITHOUT STATE REPLAY

Schema evolution is one of the worst pain points of conventional stateful Flink. Adding a new feature column or joining an additional dataset into an existing pipeline changes the operator-state schema, and the previously checkpointed state becomes incompatible with the new job graph.

Teams typically face one of two unappealing options: backfill the entire history through the new pipeline (hours to days, depending on retention) or stand up a parallel pipeline alongside the old one and cut over once it has caught up. Either path turns a one-line change into a multi-day operational project, and SQL state compatibility becomes the de facto governor on iteration speed.

When state lives in Fluss rather than in Flink, schema evolution becomes a metadata-only operation. Adding a feature column is an `ADD COLUMN` against the Fluss PK Table; the existing log retains historical records unchanged, and the new column is populated from the moment the updated job resumes writing. Joining additional input data follows the same pattern: register the new PK Table, update the SQL, restart. The Flink job's only checkpointed state is a read offset, so on restart it picks up from the last committed offset and forward progress resumes immediately. There is no historical backfill, no operator-state migration, no parallel pipeline to drain.

The practical consequence: the iteration cycle for business changes (a new ML feature, an additional join, a renamed column) collapses from days to minutes. Schema compatibility, which dominates the cost of evolving a stateful Flink topology, ceases to be a blocker. Teams ship feature changes the same afternoon they're requested.

WHEN SCHEMA COMPATIBILITY NO LONGER BLOCKS EVERY CHANGE, ITERATION SPEED TRANSFORMS. CHANGES THAT ONCE TOOK DAYS NOW SHIP THE SAME AFTERNOON.

DIAGRAM 16: JOB EVOLUTION, BEFORE VS. WITH FLUSS

Schema evolution stops being a project. When state is in the table rather than the operator, adding a column or a new join input is metadata work, the job restarts from its last offset and the checkpoint stays valid.

STREAMING LOOKUP JOINS

A lookup join is, by definition, stateless on the Flink side: the operator does not buffer the reference table at all, for each arriving event it issues a point-key request against the latest state of an external system and combines the result inline. The cost the conventional pattern pays is operational rather than state-related: an external KV store has to be stood up next to Flink, kept in sync with the source of truth via a separate pipeline, and sized for the read fan-out of every running job. Bootstrap, staleness, and memory pressure show up not in the operator but in the dedicated cache tier behind it.

Fluss collapses that cache tier away. With state already materialized in PK Tables on the bucket leaders, Flink's async lookup function can issue point-key requests directly against the Fluss leader and get sub-millisecond responses. The table is always current to the latest committed write; the lookup pipelines many concurrent in-flight requests per task slot; multi-stream lookups issue concurrent requests to multiple PK Tables simultaneously. There is no separate KV cache, no bootstrap, and no synchronization pipeline to maintain.

DIAGRAM 17: THE STATELESS COMPUTE MODEL

One boundary, two responsibilities. Compute belongs to Flink and is transient; state belongs to Fluss and is durable. Only writes, reads, and a kilobyte-sized read offset cross the line.

PARTIAL-UPDATES: 360 VIEW COMPOSITION WITHOUT JOINS

The fourth expression of state-in-Fluss is partial-updates, a writer-driven mechanism rather than a merge engine. A producer specifies a column subset on each write; on the leader, the PartialUpdater performs a read-modify-write that overwrites only the supplied columns and leaves the rest of the row untouched. Partial-updates are not a value of `table.merge-engine` and do not constitute a merge engine themselves; they are orthogonal to merge-engine selection. On a PK Table with no merge engine, the supplied columns simply overwrite the existing row. On a PK Table configured with the Aggregation Merge Engine, partial-updates compose with the merge logic; different Flink jobs can update different aggregation columns on the same row independently. Support for partial-updates with the `first_row` and `versioned` engines is not implemented today but is not architecturally precluded.

This addresses a problem that would otherwise require a large stateful Flink join: assembling a wide entity row whose columns arrive from multiple independent streams at different cadences. A user profile row with 300 columns might have its demographic fields populated by a CDC stream from the customer database, its transaction velocity fields by an Aggregation-Merge-Engine-powered stream, its credit-bureau fields by a periodic API pull, and its session fields by a clickstream processor. With partial-updates, each producer writes only its own columns to the same PK Table. No stateful join, no multi-stream coordination, no watermark alignment. The wide row assembles on the leader as producers arrive.

Partial-updates compose cleanly with the other primitives in this part: the row is still queryable via sub-millisecond KV lookup (the KV store reflects the merged state on write), still streams a correct changelog (each partial write emits a `-U/+U` pair capturing the before and after), and still tiers to the cold layer as a normal PK Table (the merged row is what gets snapshotted and committed). A downstream Flink job tailing the changelog sees each column update as a normal CDC event without needing to know which upstream producer wrote it.

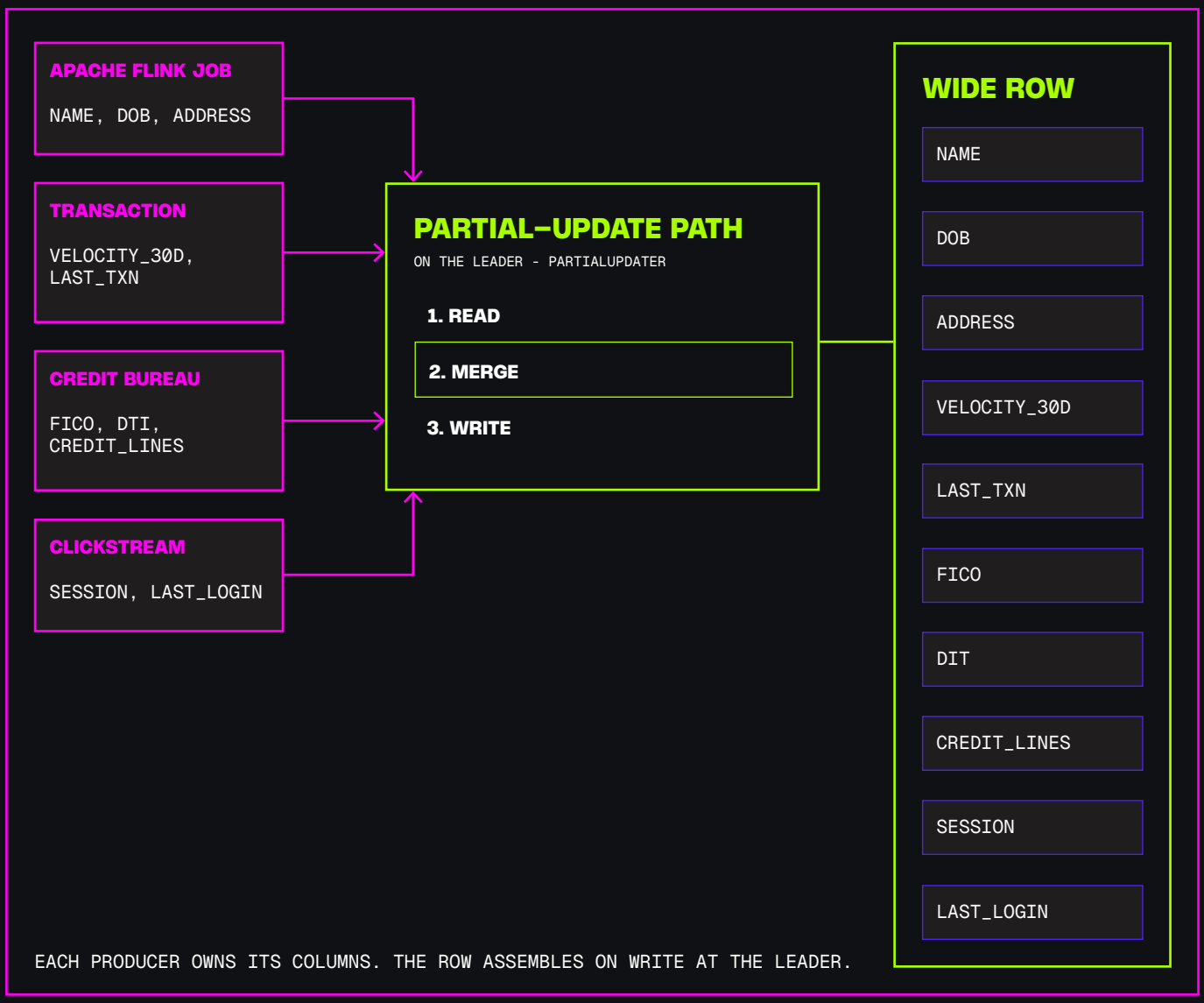


DIAGRAM 18: PARTIAL-UPDATES, FOUR PRODUCERS, ONE WIDE ROW

Four producers emit disjoint column subsets for the same key (via targetColumns). The Partial-Update Path on the leader (a writer-driven mechanism, not a merge engine) overwrites only the supplied columns, assembling a wide row with no stateful Flink join.

PART VII

THE AI DATA SUBSTRATE

ONE SUBSTRATE FOR FEATURES, CONTEXT, AND PROFILES

FLUSS AS AI INFRASTRUCTURE

Modern AI systems, including recommendation engines, fraud detection, RAG pipelines, and agentic workflows all depend on the same three primitives: live feature values, retrievable context, and entity profiles that stay current.

In conventional architectures, these primitives live in separate systems: an online feature store, a vector index, a graph database, an offline lake. They diverge over time, and every team pays the cost of reconciling them.

Fluss collapses that surface area. The same PK Tables, changelog, and KV substrate that power streaming pipelines also serve as the feature store, the context store, and the entity-profile store. The data is the same, only the view changes.

The content below walks through that consolidation: how training and serving read from the same table, how Fluss functions as a context store for LLM and agent systems, how virtual tables expose multiple views of the same state, and how a real-time entity profile assembles directly on the leader.

ELIMINATING TRAINING–SERVING SKEW

In conventional feature stores, training reads from one store (often Iceberg on Amazon S3) and serving reads from another (typically Redis). Features diverge over time, not because engineers are careless, but because structurally no single system is responsible for keeping them aligned. With Fluss, training and serving read from the same underlying store (PK Tables) accessed through different views. Serving reads the current KV state from the leader via KV lookup. Training reads the same table as a deterministic point-in-time snapshot. Because both paths read the same table, training-serving skew is structurally eliminated.

- **FRESHNESS** Features are current to the most recent ISR-acknowledged write applied to the leader. Once a write has cleared the pre-write buffer and been materialized into the leader's RocksDB, the new value is immediately visible to KV lookups.
- **WIDE-TABLE SEMANTICS** A single PK Table can hold hundreds of feature columns for each entity. Column projection at query time ensures models pay only for the features they consume.
- **POINT-IN-TIME CORRECTNESS** The changelog preserves every state transition, and the same table can be replayed deterministically as a point-in-time view, training labels join against the exact feature state that existed when each label event occurred.

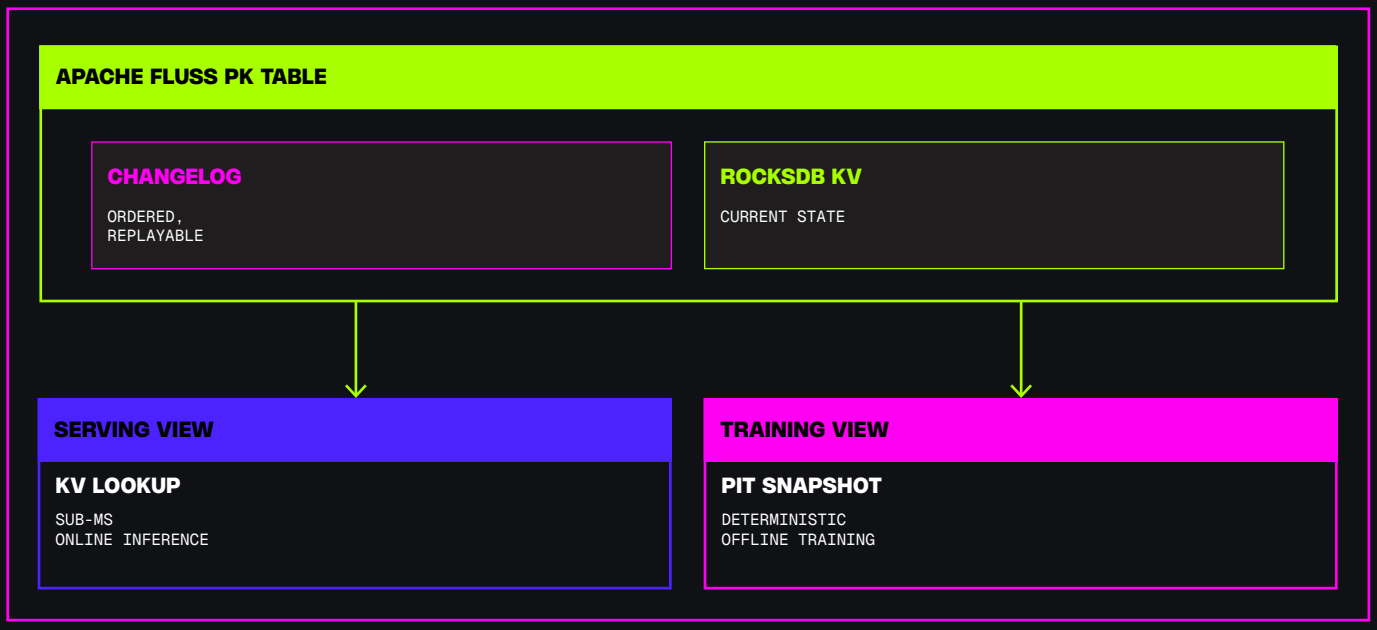


DIAGRAM 19: TRAINING AND SERVING FROM THE SAME TABLE

Serving reads the current RocksDB KV state via KV lookup. Training reads the same PK Table as a deterministic point-in-time snapshot. Because both paths read the same underlying storage, training-serving skew is structurally eliminated.

FLUSS AS A CONTEXT STORE FOR AI SYSTEMS

Context engineering for LLM and agentic systems reduces to the same set of primitives: durable event logs (session memory), consistent KV state (entity memory), streaming feature aggregates (behavioural signals), and vector search (semantic memory). Fluss provides the first three through a single storage substrate with a consistent schema surface, Log Tables for session memory, PK Tables for entity memory and aggregated features. Today, vector embeddings typically sit in a paired engine optimized for similarity search, Lance / LanceDB is a strong fit because it is also one of Fluss's cold-tier targets, so the same data substrate that holds entity state can hold the embedding columns alongside it; alternatives include Milvus®, Weaviate, and OpenSearch.

Note: Native vector support inside Fluss itself is a work in progress as of the time of this publication. The community is actively designing first-class vector columns and similarity-search operators on PK Tables, which would let semantic memory live in the same substrate as session, entity, and behavioural memory, removing the last external dependency in the context-assembly path. Until that ships, the pattern below pairs Fluss with an external vector index, with Lance / LanceDB the path of least resistance for teams already running Fluss with Lance as their cold tier.

An LLM-based loan advisory assistant assembles context for each turn as follows: session history is read from a Log Table; the applicant's live feature vector is read from a PK Table via sub-millisecond KV lookup; relevant policy documents are retrieved via vector similarity search against Lance / LanceDB (or another paired vector engine); aggregated behavioural signals are read from Aggregation-Merge-Engine-powered PK Tables. The Fluss reads happen against a single storage system with consistent schemas.

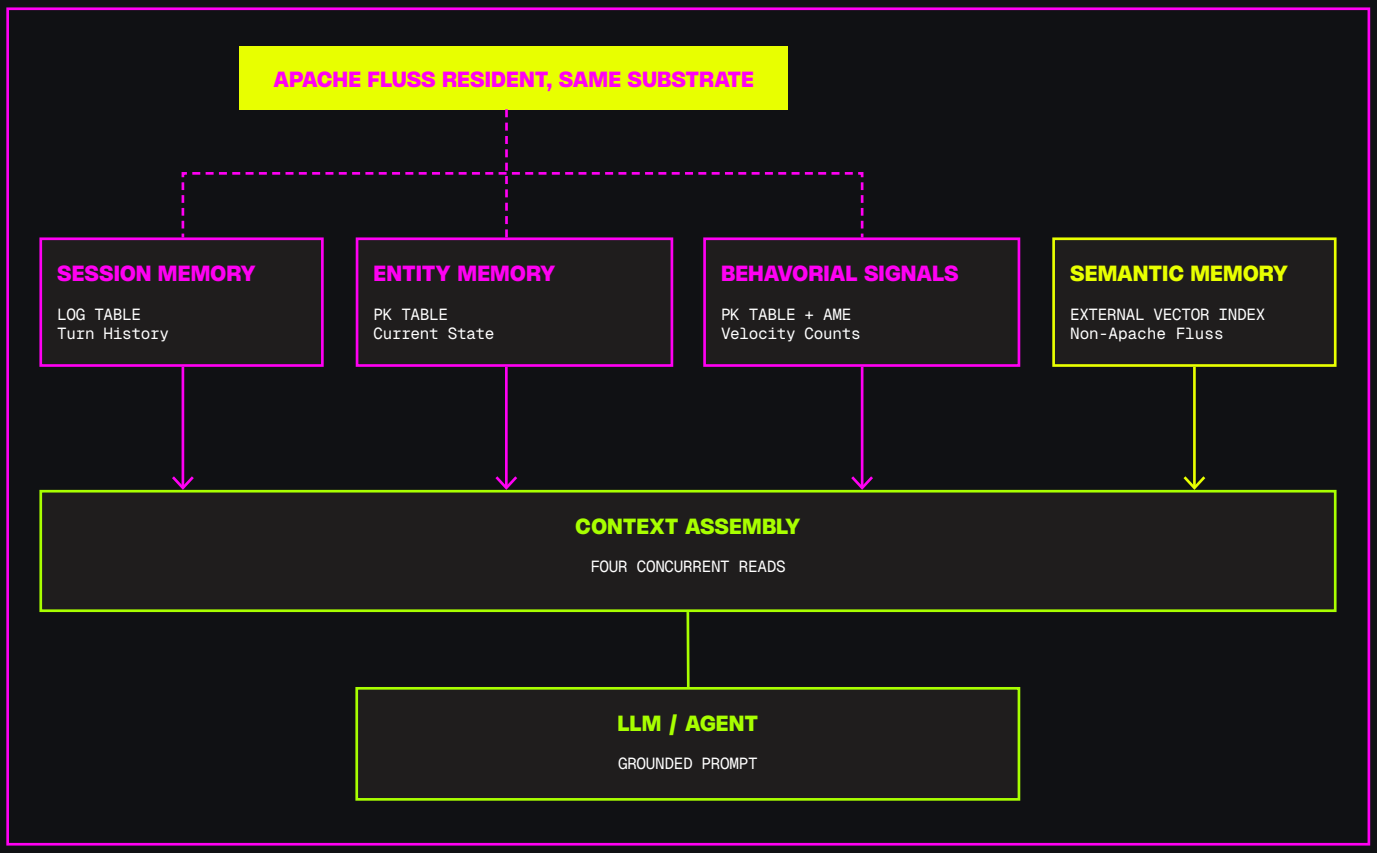


DIAGRAM 20: CONTEXT ASSEMBLY, FOUR MEMORIES, ONE SUBSTRATE

Context assembly for LLM systems. Four memory primitives: session, entity, behavioural, semantic, live in the same Fluss substrate under a consistent schema surface, read concurrently into a grounded prompt.

FLUSS AS A CONTEXT STORE FOR AI SYSTEMS

A single PK Table often needs to serve more than one consumer, each with its own column subset, filter, or merge semantics. The online serving path may want a narrow projection of the latest row, the training path may want the full schema as a deterministic point-in-time view, while the audit path may want the changelog at the raw record level.

Fluss exposes virtual tables as a way to project the same underlying physical PK Table into multiple logical surfaces, without copying state. The physical table holds one source of truth on the leader; each virtual table is a thin metadata-level view on top of it (its own name, its own column list, its own access pattern) backed by the same RocksDB rows and the same changelog underneath.

The practical effect is that the feature store, context store, and audit pipeline can each present a different surface to their callers while sharing one substrate beneath. There is no fan-out copy job, no schema-drift risk between views, and no second consistency model to keep aligned, and views are recomputed by definition on every read.

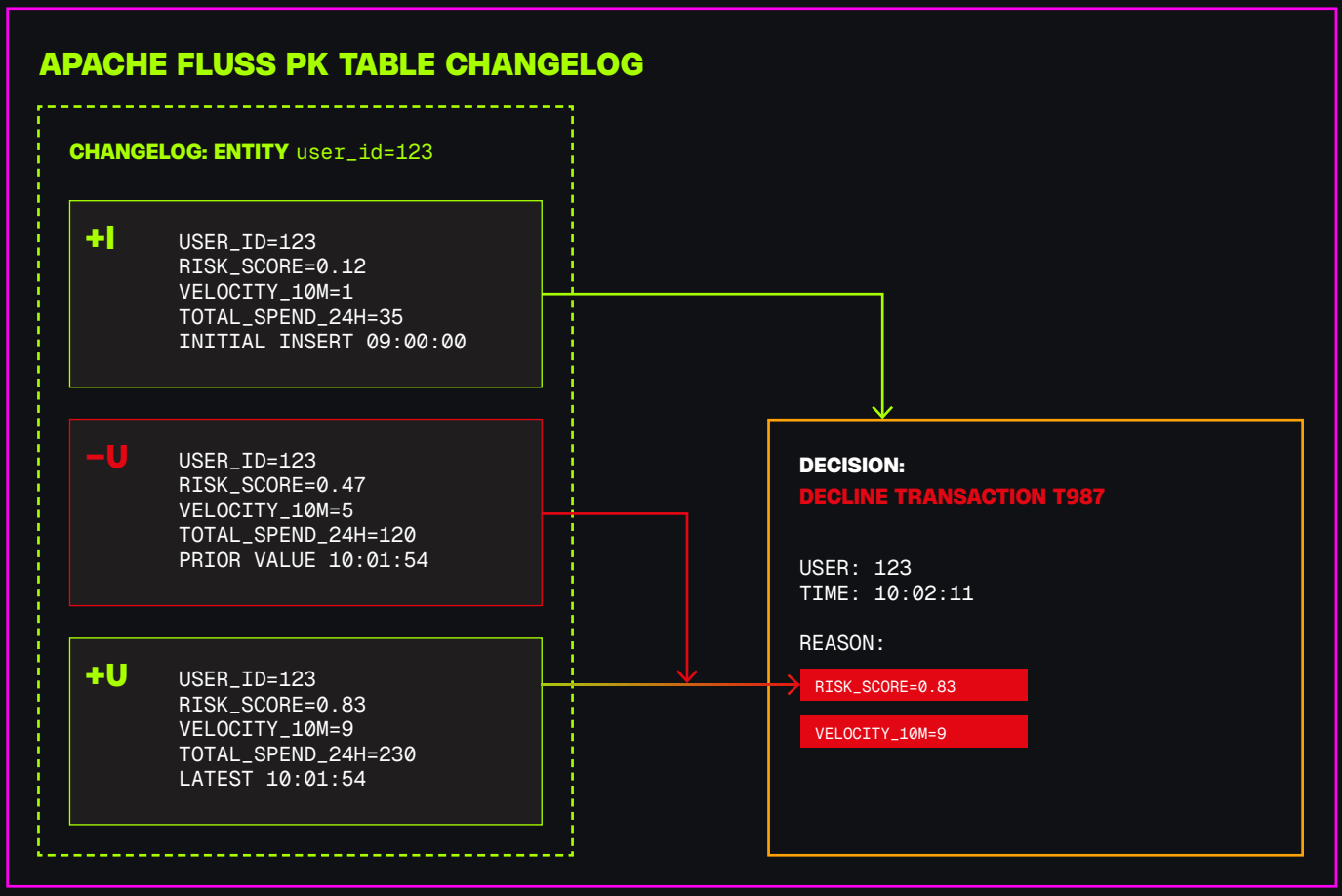


DIAGRAM 21: CHANGELOG AS AN AUDIT TRAIL: EVERY DECISION RECONSTRUCTABLE

Every state transition for an entity is preserved as a record in the changelog (+I , -U , +U , -D). When a real-time decision fires (here, declining transaction T987 for user_id 123) the exact values that drove it (risk_score=0.83 , velocity_10m=9) trace directly back to the specific changelog record. The decision is fully reconstructible from the log itself, without any secondary audit pipeline.

REAL–TIME ENTITY PROFILES ON THE SAME SUBSTRATE

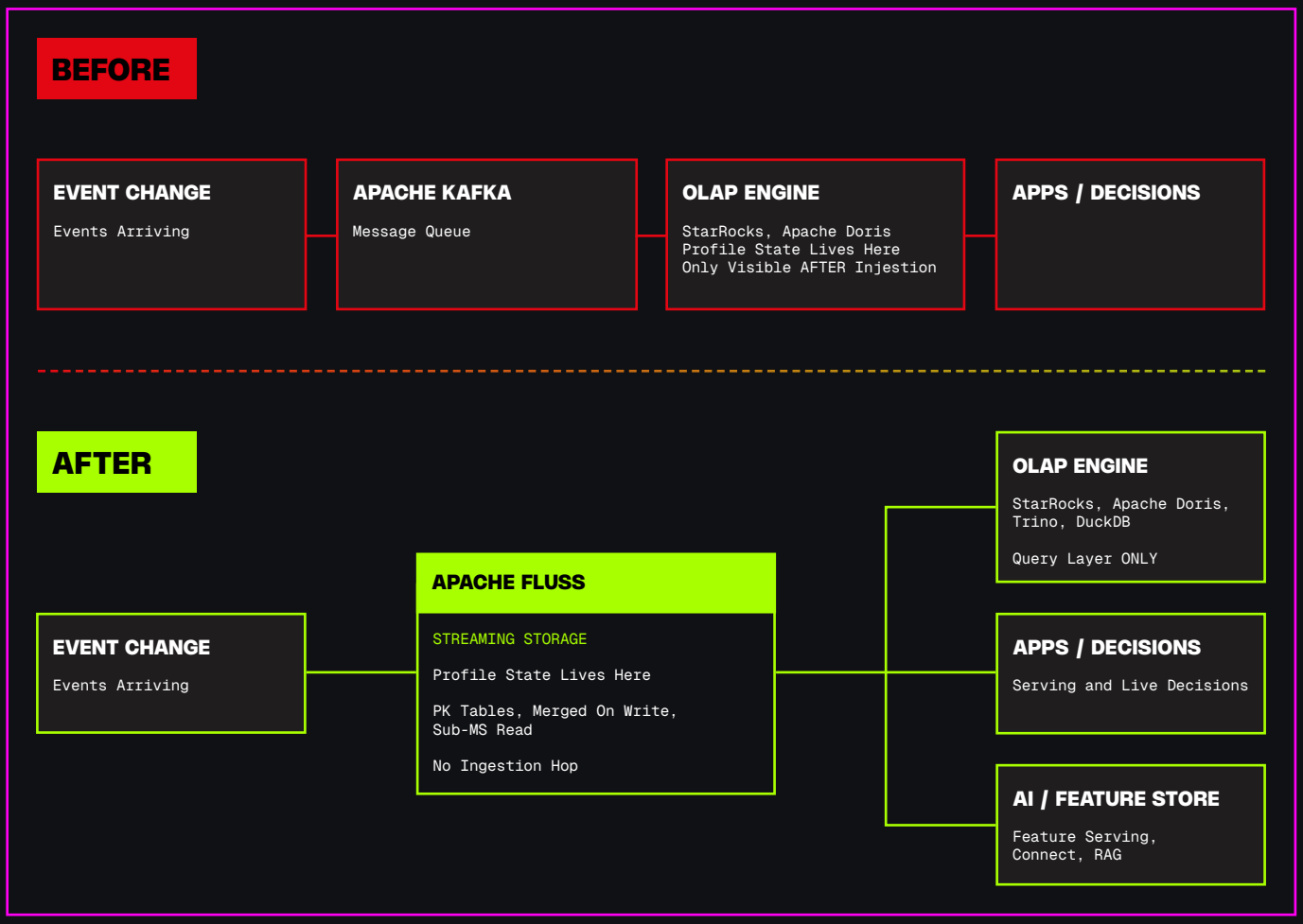


DIAGRAM 22: PROFILE STATE, BEFORE VS. WITH FLUSS

The profile is not a downstream copy. Conventional architectures push profile state into a secondary OLAP store, paying an ingestion hop and a second consistency model. With Fluss, the profile is the streaming storage; OLAP, apps, and feature stores read it directly.

Every system that makes live decisions about users, devices, sessions, or transactions ultimately answers the same underlying question: what do we currently know about this entity, and does it change what we should do next?

Until recently the conventional answer routed events into an OLAP engine (StarRocks, Doris, ClickHouse) and relied on its low-latency query surface. That leaves the profile state sitting in the query layer rather than the streaming layer. Every profile read becomes a query, every update has to travel through ingestion into the OLAP store before it becomes visible, and the stream and the profile end up as two systems with two consistency models.

Fluss collapses that separation by keeping the profile inside the streaming storage layer itself. The profile is maintained on write inside a PK Table via the Aggregation Merge Engine; the stream and the profile share one consistency model and one recovery guarantee. Any consumer that can read a Fluss table (an OLAP engine, a notebook, a feature store, an inference service) gets the profile for free, without owning the state or running an extract pipeline.

AUTO-INCREMENT COLUMNS Real-world identifiers arrive as strings, user IDs, device fingerprints, session tokens. Roaring Bitmaps operate on integers. Fluss auto-increment columns bridge this gap by mapping each new string identifier to a stable 32-bit or 64-bit integer in a dimension table and caching the mapping. The mapping table is a standard PK Table configured with auto-increment.fields and lookup.insert-if-not-exists; the cache is partial with configurable expiry so memory pressure stays bounded.

ROARING BITMAPS A compressed set representation that automatically selects an internal encoding (bitmap, array, or run) depending on density. Membership tests, unions, intersections, and differences resolve in nanoseconds. Group membership across millions of entities can be stored, queried, and intersected at sub-query latency, with exact cardinality counts. RoaringBitmap is markedly more performant on 32-bit values than on 64-bit, so production deployments standardise on the 32-bit variant: real-world string IDs are mapped through auto-increment columns to stable 32-bit integers and stored in `rbm32` bitmaps. `rbm64` is available for keyspaces that genuinely exceed 32-bit cardinality, but is the exception, not the default.

THE RBM32 AGGREGATOR Each Flink write carries a single-element BYTES bitmap produced by `TO_BITMAP32(entity_int32)`. Fluss merges that into the accumulated group bitmap at the storage layer, on the leader, at write time, via read-modify-write, the same Aggregation Merge Engine mechanism from Part VI of this paper (“The Aggregation Merge Engine”). The Flink job itself holds almost no state; its checkpoint footprint is effectively independent of how many entities are tracked. Column type must be BYTES; the aggregator factory validates this on table creation. The same mechanism powers `rbm64` when 64-bit cardinality is unavoidable, but `rbm32` is the recommended path. Replay correctness uses the undo-based recovery mechanism from Part II (“Write path, read path, replication”): on restart the connector compares bucket offsets against log end offsets, computes per-key inverse operations, and applies them in `OVERWRITE` mode before resuming forward progress.

This is deliberate. Profile rules reduce to set algebra, AND, OR, AND NOT over groups like “users who clicked in the last ten minutes”, “devices seen on more than one account”, “accounts that crossed a usage threshold today”. Roaring Bitmaps answer storage efficiency and combination speed; the Aggregation Merge Engine answers continuous update; auto-increment columns bridge string identifiers into the integer space the other two primitives require.

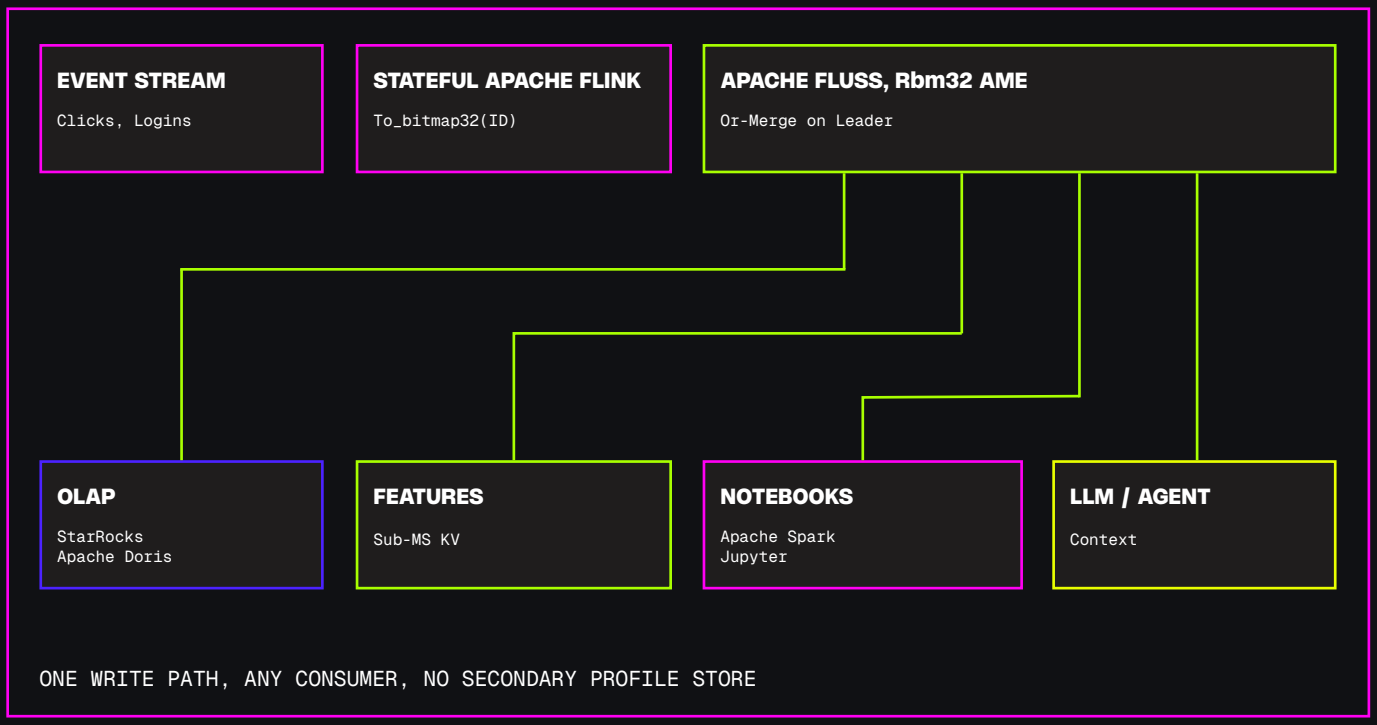


DIAGRAM 23: REAL-TIME PROVIDE COMPOSITION

Event streams flow through a stateless Flink transform into a Fluss PK Table with the rbm32 Aggregation Merge Engine. Any downstream consumer reads the same profile state without owning it or running an extract pipeline.

CLOSING THE LOOP

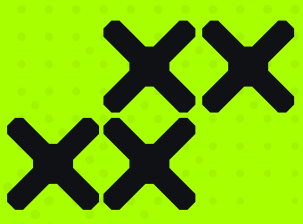
ONE TABLE, THREE VIEWS

The feature store, the context store, and the real-time profile are not three systems that need to be kept in sync. They are instead three views over one PK Table, maintained by a single write path and a single consistency model.

Every capability covered in this whitepaper from Parts II through VII, (including dual representation, ISR durability, the Aggregation Merge Engine, compound pruning, stateless compute, decoupled RPO and RTO, and Union Read) converges here into a single architectural statement: one substrate, many consumers, no secondary stores.

PART VIII

SUMMARY



VIII SUMMARY

CONCLUSION

Apache Fluss represents a single architectural bet: that a unified, lake-house-native storage primitive, one that natively speaks the language of event ingestion, KV serving, streaming computation, historical archival, and audit; eliminates the “multiple systems tax” that conventional real-time infrastructures accumulate across the common five stages of maturity.

The mechanisms described in Parts II through VII are not independent features. Each is a direct consequence of the same underlying storage design, and each maps cleanly to a concrete operational outcome. Together, they are the foundation on which Ververica Streamhouse is built: one copy of data, serving every workload, streaming ingestion, low-latency operational analytics, batch processing, machine learning features, and AI context, without the synchronization pipelines, secondary caches, and diverging consistency models that a multi-system stack demands.

As a result, the synchronization problem does not need to be solved. On a Fluss-centric substrate, it simply does not exist.

Explore the open source Apache Fluss project at:

fluss.apache.org

For teams interested in production-ready Flink, complete with managed infrastructure, enterprise support, and full Streamhouse™ capabilities, visit

ververica.com

APPENDIX

STORAGE ARCHITECTURE REFERENCE

A. THE THREE STORAGE TIERS: OPERATIONAL REFERENCE

A.1 THE THREE TIERS AT A GLANCE

Each tier owns a different durability profile, freshness window, and reader audience. Knowing which tier holds the bytes a given workload needs is the most useful operational mental model in the cluster.

FIGURE A.1: THREE TIERS, QUICK REFERENCE

A one-glance summary of what each tier holds, its freshness window, and who reads it. Full architecture and the mechanisms that move data between tiers is discussed in Part IV of this paper.

TIER	SUBSTRATE	HOLDS	READ BY
Tier 1, Hot	TabletServer local disc (leader RocksDB) pre-write buffer	Live log segments current KV state) in-flight batches	Fluss clients (FlinK) sub-ms KV lookups
Tier 2, Fluss-Native Remote	Object storage (S3, HDFS, OSS), Fluss-native binary	Tiered log segments and periodic KV snapshots (point-in-time RocksDB + recorded log offset)	Fluss only (leader-election failover, point-in-time reads, consumers reading offsets past local retention
Tier 3, Open Lakehouse	Object storage, open table format	The fluss log projected into Iceberg, Paimon, or Lance	Any engine (Spark) Trino (Athena) Duck DB (Flink batch) Fluss streaming consumers reading offsets past local retention

The Single Most Useful Distinction

Tier 2 and Tier 3 both sit on object storage but they hold different artifacts written by different processes for different consumers. Tier 2 is Fluss-native: tiered log segments (written by the internal log tiering subsystem) and periodic KV snapshots (written by the KV snapshot process), readable only by Fluss for failover, point-in-time reads, and consumers reading offsets past local retention. Tier 3 is the open-format projection of the log, written by the lake Tiering Service Flink job into Iceberg, Paimon, or Lance, readable by any external engine. The two remote tiers are not duplicates: Tier 2 carries Fluss-native log segments plus KV snapshots; Tier 3 carries the same log re-encoded in an open table format.

A.2 WRITE PIPELINE STAGES

A write traverses the cluster in five ordered stages. Tier 2 and Tier 3 fall outside the critical path, neither blocks acknowledgement.

1. CLIENT BATCH. The producer accumulates records into an ARROW batch, applies the bucket-key hash to determine the target bucket, and sends the batch to the current bucket leader.

2. PRE-WRITE BUFFER. The leader stages the batch in an in-memory pre-write buffer while it awaits ISR quorum acknowledgement. The buffer is consulted by subsequent writes on the same key (to compute correct UPDATE_BEFORE values for the changelog), but not by read paths: pending records may be rolled back, so they remain invisible to users. Reads (including KV lookups) bypass the buffer and query RocksDB directly.

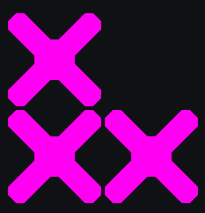
3. LOG APPEND + ISR REPLICATION. The leader appends the batch to the bucket's log and replicates to the in-sync replicas. The write is acknowledged once the configured ISR quorum has persisted the batch.

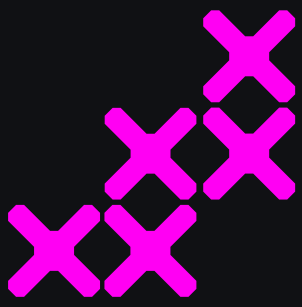
4. KV MATERIALIZATION. For PK Tables, the leader applies the batch to its local RocksDB instance (standard Put for append and replace; read-modify-write for AME, partial-update, and rbm32/rbm64 tables). Followers do not materialise the KV store.

5. TIER 2 / TIER 3 PROPAGATION. The background snapshot process and the Tiering Service operate independently of the write path. Neither is on the critical path of a write acknowledgement.

A.3 CONFIGURATION REFERENCE

PARAMETER	SCOPE	NOTES
table.log.format	Log Tables, PK Tables	Default arrow. PK Tables may be set to compacted for row-oriented log; columnar pruning does not apply.
table.kv.format	PK Tables	Governs how rows are encoded inside the leader's RocksDB store, independent of table.log.format. Defaults to compacted.
table.merge-engine	PK Tables	Options: first_row, versioned, aggregation. When unset, the table uses last-write-wins replace semantics. Mutually exclusive with the partial-update write path.
Partial-updates (writer-side targetColumns)	PK Tables (no merge engine)	A producer specifies a column subset on write; the leader merges via the PartialUpdater, overwriting only the supplied columns. Cannot be combined with a merge engine.
auto-increment.fields	PK Tables (dimension)	Declares columns populated by the auto-increment sequence. Used with lookup.insert-if-not-exists for string → int64 mapping tables.
lookup.insert-if-not-exists	Lookup joins	If a lookup key is absent, insert a new row (triggering auto-increment). Enables the bridge from string identifiers to int64 in profile pipelines.
Aggregator factory (per column)	AME tables	Examples include sum, product, max, min, last_value, last_value_ignore_nulls, first_value, first_value_ignore_nulls, listagg, string_agg, bool_and, bool_or, rbm32, rbm64. rbm64 and rbm32 require BYTES column type.





REFERENCES

>> [BLOG] INTRODUCING THE ERA OF "ZERO-STATE" STREAMING

READ MORE

>> [BLOG] FROM KAPPA ARCHITECTURE TO STREAMHOUSE: MAKING THE LAKEHOUSE REAL-TIME

READ MORE

>> [BLOG] FROM EVENTS TO REAL-TIME PROFILES ON APACHE FLUSS

READ MORE

>> [BLOG] TAOBAO INSTANT COMMERCE: REAL-TIME DECISIONS AT SCALE WITH APACHE FLUSS

READ MORE

>> [BLOG] WHAT DOES APACHE FLUSS MEAN IN THE CONTEXT OF AI?

READ MORE

>> [BLOG] HOW APACHE FLUSS ACHIEVES TRUE PRUNING IN STREAMING STORAGE

READ MORE

>> [BLOG] SETUP REAL-TIME MULTIMODAL AI ANALYTICS WITH APACHE FLUSS (INCUBATING) AND LANCE

READ MORE

>> [VIDEO] APACHE FLUSS AND THE SEVEN DEADLY SINS OF STREAMING

WATCH VIDEO

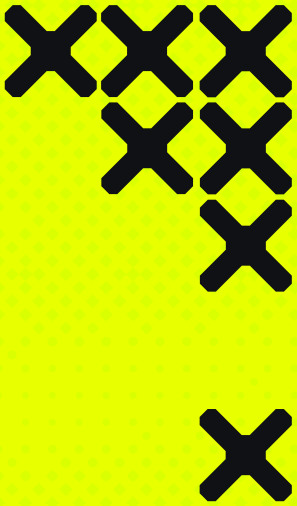
>> [VIDEO] APACHE FLUSS: MAKING YOUR LAKEHOUSE TRULY REAL TIME

WATCH VIDEO

ABOUT VERVERICA

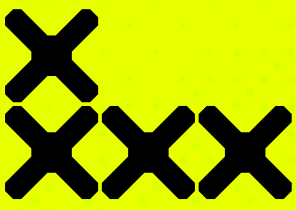
GOVERNED. ABSOLUTE. PRODUCTION-GRADE.

Ververica created Apache Flink®. Now we've perfected it. Our Unified Streaming Data Platform, powered by the VERA engine, runs stream processing at scale. Billions of events per second. Millisecond latency. 40% lower TCO. Production systems for Fortune 500 enterprises process, govern, and act on streaming data in real-time. Engineered in Europe. Sovereign by design. Public cloud. Private cloud. On-premise. Your infrastructure. Your control.



Discover more at

ververica.com



ververica