



X-Stream Lab

Streaming Lakehouse

Alex Campos, Solutions Engineer Ververica

Introduction

What Stream Processing is?

What is Stream Processing?

Stream processing is an architectural approach

designed to handle **data streams**

Rather than batch data

Event-driven decisions

as they are generated in real-time,

analyzing and acting as it arrives,

enabling immediate **insights and responses**.

Generate business value

Stream Processing - Key Characteristics



Continuous Processing

Data is processed immediately as it flows through the systems, without waiting for complete datasets to accumulate.



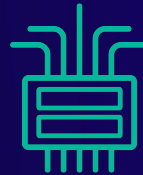
Low Latency

Results are produced within milliseconds to seconds of data arrival, enabling immediate responses to events.



Unbounded Data

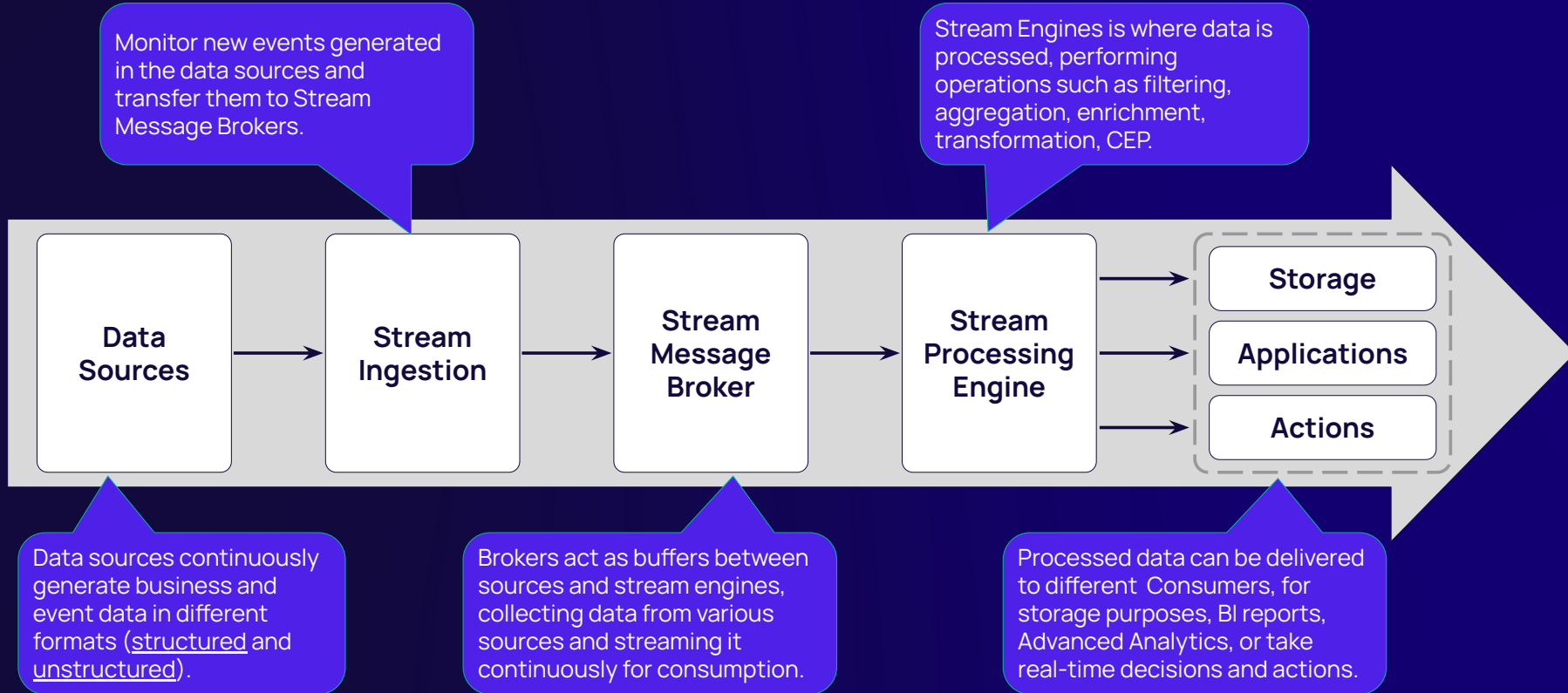
Handles potentially infinite streams of data that are generated continuously and don't have a predetermined end.



Event-Driven

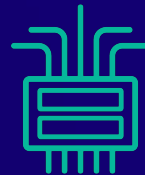
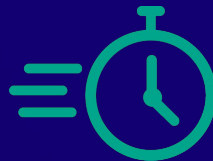
Processing is triggered by incoming data events, in real-time as soon as it arrives, rather than scheduled batch jobs.

Stream Processing - Architecture Components



In a Nutshell, Stream Processing is...

- Real-Time
- Continuous
- Unbounded
- Low Latency
- Event-Driven



Apache Flink Intro

What is Apache Flink®?

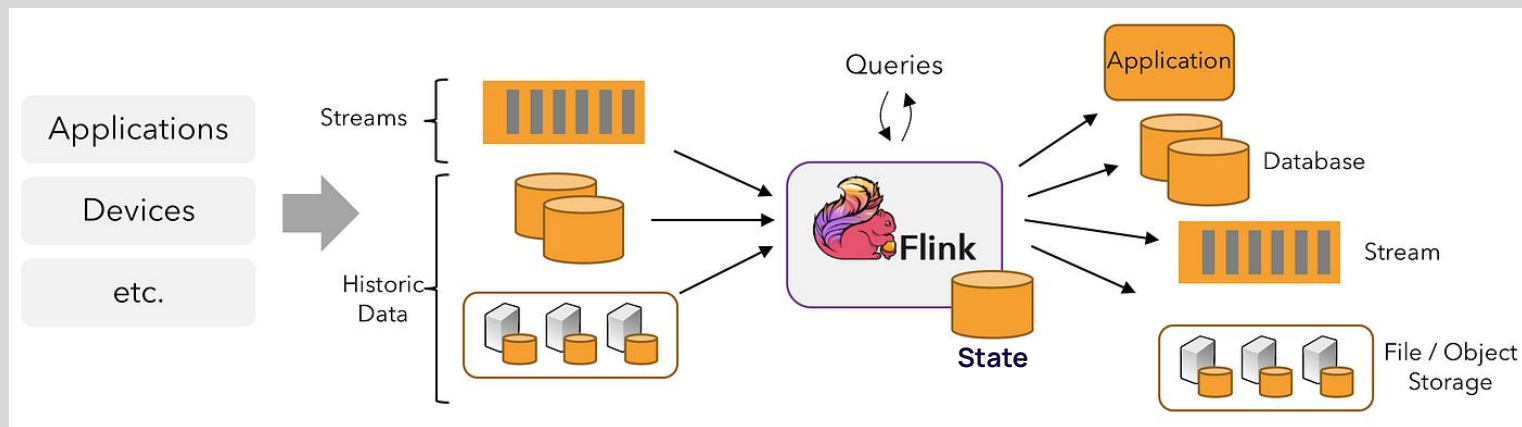
Apache Flink - Background



- Data Artisans (now Ververica)
- Donated to Apache Software Foundation in 2014
- Apache top-level project since 2015
- Contributions from Amazon, Alibaba, Ververica, Netflix, Uber and many other companies
- Multi-language support: Java, Python, SQL
- Integration with common data sources and sinks: Kafka, databases, file systems, and many others
- Millisecond-level latency (true low-latency streaming), making it ideal for real-time applications
- Batch and stream processing

What is Apache Flink?

A framework for stateful computations on unbounded and bounded data streams



Apache Flink & the API Stack

Different layers of abstraction cover various application scenarios

High-level
Analytics API

SQL / Table API (dynamic tables)

```
SELECT room, TUMBLE_END(rowtime, INTERVAL '1' HOUR), AVG(temp)
FROM sensors
GROUP BY TUMBLE(rowtime, INTERVAL '1' HOUR), room
```

Stream & Batch
Data Processing

DataStream API (streams, windows)

```
Val stats = stream
  .keyBy("sensor")
  .timeWindow(Time.seconds(5))
  .sum((a, B) -> a.add(B))
```

Stateful Event-
driven Applications

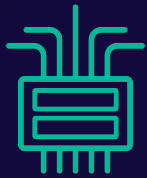
Process Function (events, state, time)

```
def processElement(event: MyEvent, ctx: Context, out: Collector[Result]) = {
  // work with event and state
  (event state.value) match { ... }

  out.collect(...) // emit events
  State.update(...) // modify state

  // schedule a timer callback
  ctx.timerService.register EventTime Timer(event.timestamp + 500)
}
```

Apache Flink - Key Concepts



Streaming

Process bounded and unbounded sequence of events leveraging a distributed architecture for resilience and parallelism.



Time

Flink provides mechanisms to manage event ordering, crucial to ensure analytics accuracy and proper late data handling.



State

Stateful streaming applications need to remember and maintain state over time across multiple events and processing cycles.



Snapshot

Snapshots are consistent, global and point-in-time state across all operators in a Flink job to provide fault-tolerance and recovery.

In a Nutshell, Apache Flink...

- **Unified Framework:** A single engine for both streaming and batch data processing.
- **Stateful Computations:** Maintains and updates application state with exactly-once consistency.
- **Layered APIs:** Offers different APIs for various use cases, including DataStream, DataSet, and Table/SQL.
- **Fault-Tolerant:** Recovers from failures using distributed checkpoints, ensuring no data loss.
- **High Performance:** Designed for high throughput, low latency, and scalable architecture.
- **Multiple connectors** for data source and sinks.



Apache Fluss Intro

What is Apache Fluss?

Introduction to Apache Fluss



Apache Fluss

**Streaming Storage for Next-Gen
Data Analytics and AI**

Unified Storage for Real-Time Analytics



Operational
Databases



Message
Queues



Streaming
Writes

Real-Time
Updates

Fluss Cluster

Server



Server



Server



Columnar
Storage

Stream-Table
Duality

Projection
Pushdown

Ad-hoc
Query

Sub-second
Latency

Data
Tiering



Streaming Reads

Batch Reads (Paimon API)

Lookup Joins

Point Queries

Fluss Architecture Components



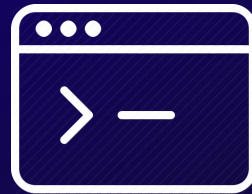
Tablet Server

Responsible for data storage, persistence, and serving data directly to users



Coordinator Server

Acts as the brain of the cluster, serving as the central control and management



Fluss Client

Fluss clients/SDKs support streaming reads/writes, batch reads/writes, DDL and point queries.

Introducing Stream as Table



- **Fluss uses tables**, not topics, data streams as schematized tables.
- **Schema requirement**, Fluss requires a schema to be defined before writing any data.
- **TTL support**, 7 days default retention
- Multi-field **Partitioned Tables**
- **Column pruning** -> *SELECT col1 FROM TABLE*
- **Predicate pushdown** -> *WHERE col1 = YESTERDAY*

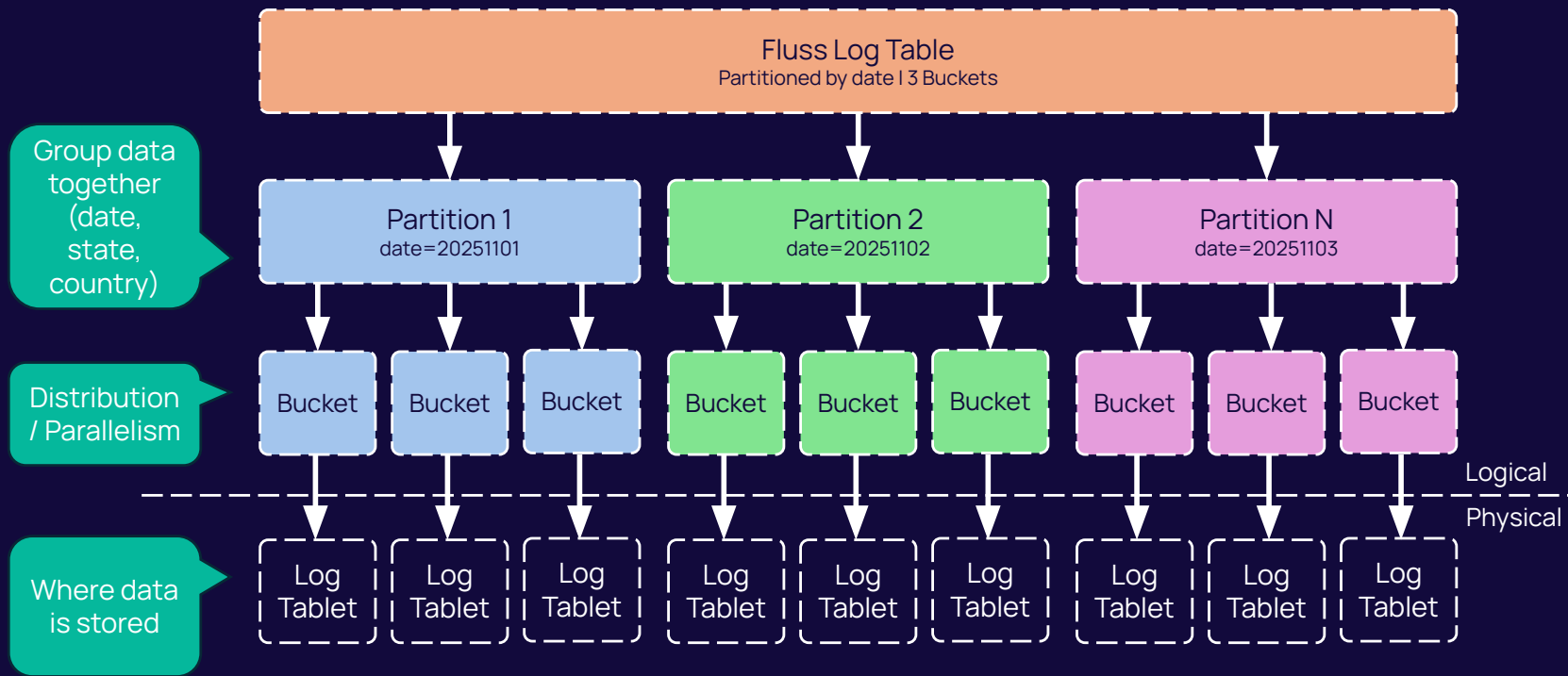
Log Table



```
CREATE TABLE log_table (  
  order_id BIGINT,  
  item_id BIGINT,  
  amount INT,  
  address STRING,  
  dt DATE  
)  
WITH ('bucket.num' = '3');
```

- Append-only, INSERT
- Kafka behaviour, no record mutation
- No Primary Key defined
- Columnar compression with Apache Arrow
- Predicate pushdown and column pruning
- Bucket assignment: sticky, round-robin, hash-based
- Common use cases: log ingestion, transaction logs, click streams, sensor readings

Log Table - Structure



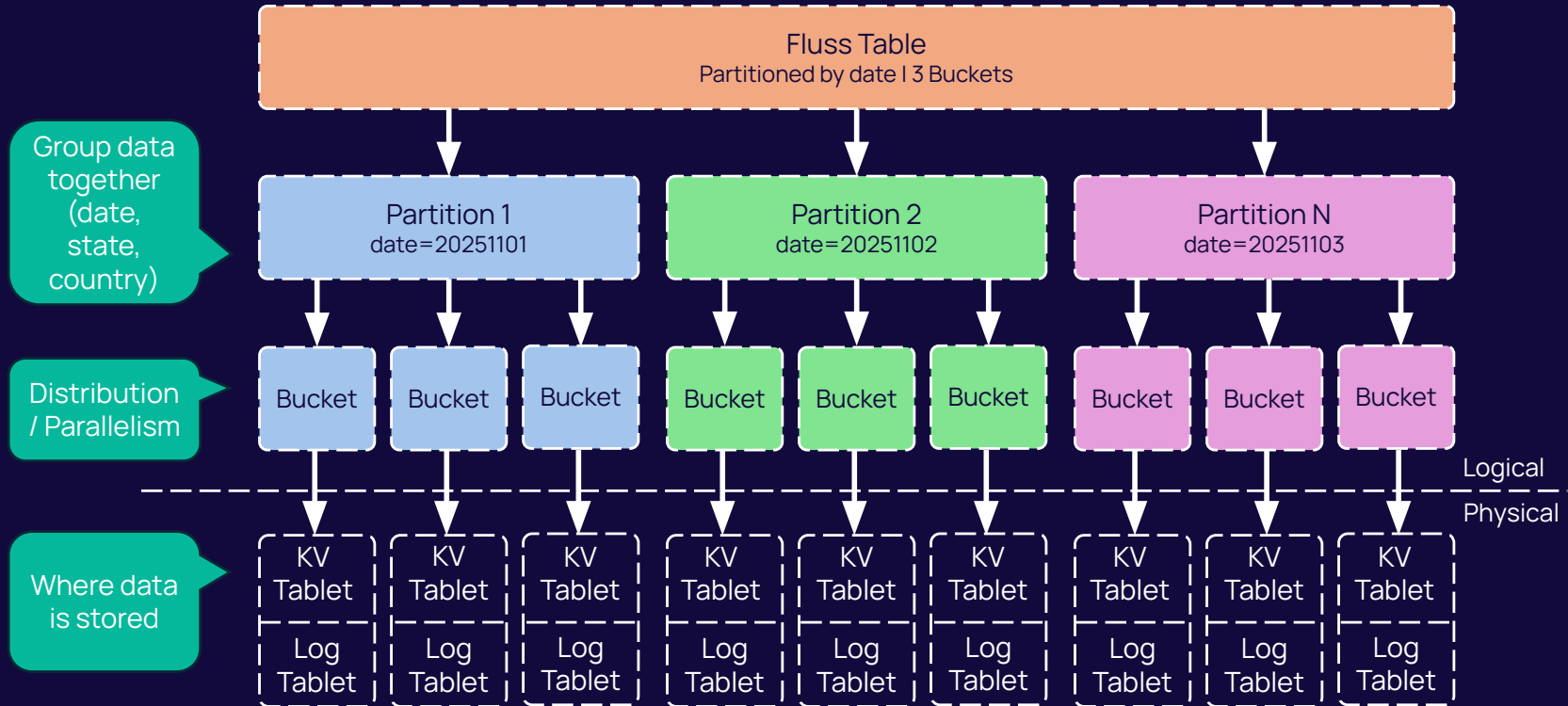
Primary Key Table



```
CREATE TABLE pk_table(  
  shop_id      BIGINT,  
  user_id      BIGINT,  
  num_orders   INT,  
  total_amount INT,  
  PRIMARY KEY (shop_id, user_id)  
)  
WITH ('bucket.num' = '4');
```

- Ensure uniqueness of a given record based on its primary key (mandatory)
- Supports INSERT, DELETE, UPDATE
- Changelog behaviour
- Records are stored in RocksDB tables
- Multiple merge engines (Last Row, First Row, Versioned)
- Lookup, Prefix Lookup and Changelog reads
- Common use cases: account balance, user profile, product catalog, shopping carts.

Fluss Table Hierarchy - Primary Key Table



Fluss Table Storage

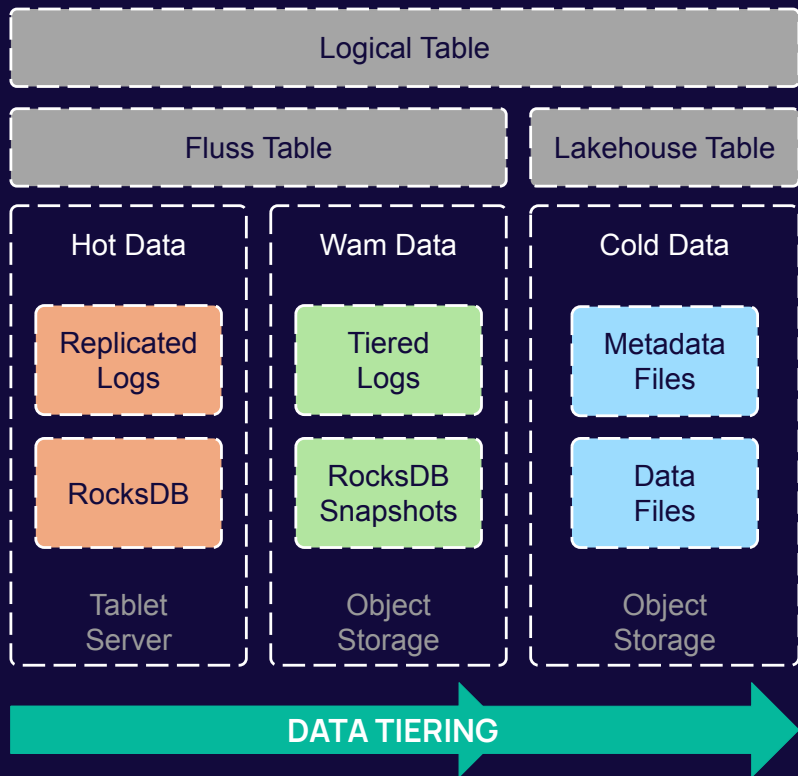


- **Log Tables** data is compressed using Arrow
- **Columnar Format**, optimized in-memory layout for analytics.
- **High-Performance Engine**, better CPU cache use and SIMD acceleration.
- **Compression decode**: NONE, LZ4_FRAME, ZSTD



- **Primary Key Table** records are stored in RocksDB
- High-performance **Key-Value** store
- Log-structured merge-tree (**LSM**) architecture ensures efficient write amplification and high throughput.
- **Highly tunable** for varied workloads.

Data Tiering Overview



- **Internal Tiering:** Copying hot data from Fluss tables to Object Storage (warm data).
- **Lakehouse Tiering:** Copying hot data from Fluss tables to Lakehouse (cold data). Paimon and Iceberg table format are currently supported.
- **Storage Unification:** Unifying historical Lakehouse data with real-time Fluss data

Apache Fluss Differentiators



Schema-First

Fluss adopts a design with structured tables, explicit schemas, and support for all kinds of data types, organizing data into tables with partitions and buckets.



Data Mutation

Fluss supports native record mutation (Primary Key Table), enabling changelog use cases, sub-second updates and simplifying overall architecture.



Storage Model

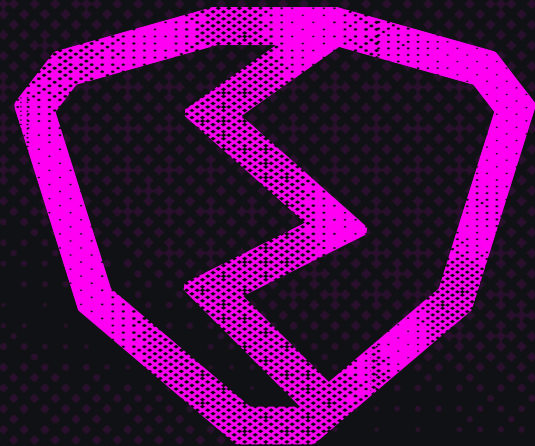
Fluss introduces Apache Arrow for columnar data compression, delivering filter pushdown, columnar pruning and storage cost optimization.



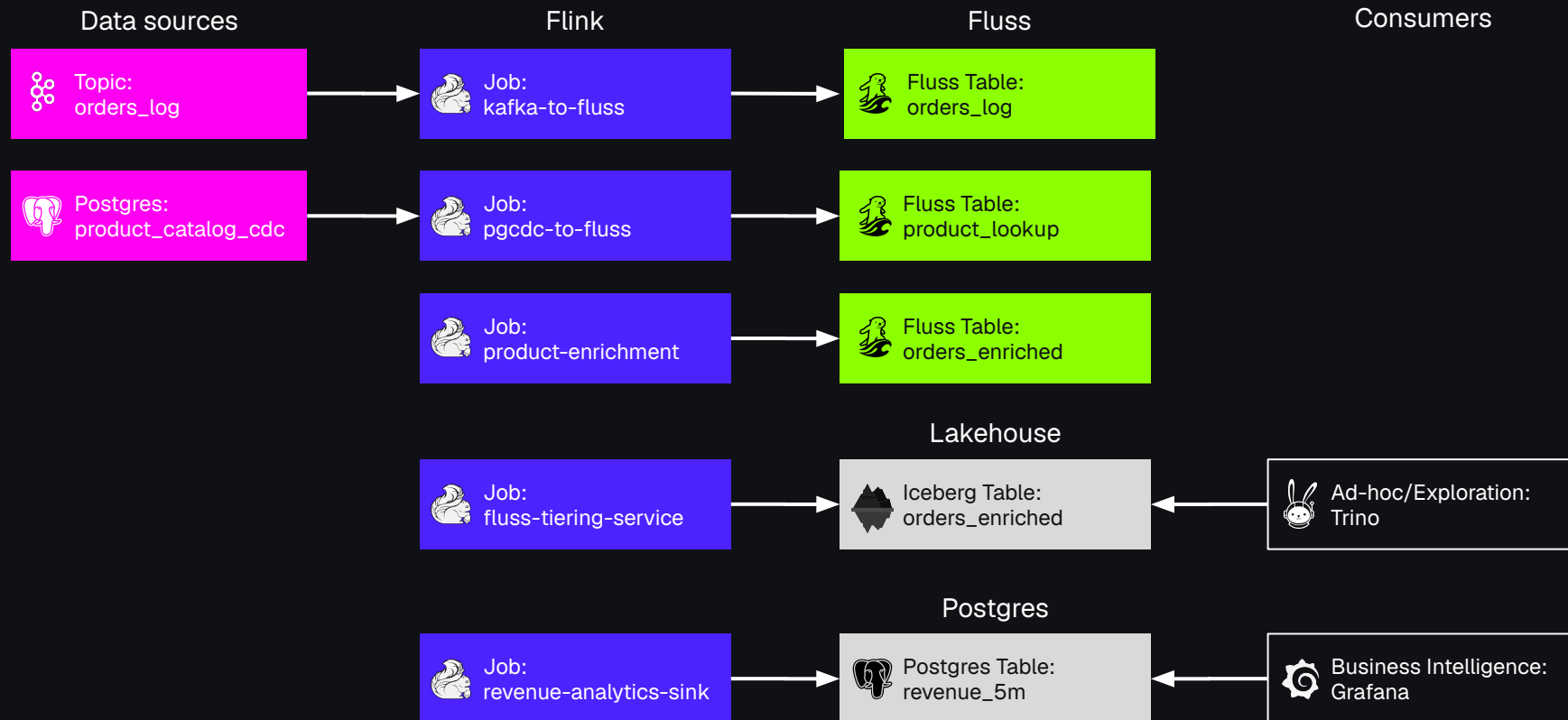
Data Tiering

Fluss provides internal and Lakehouse data tiering with zero-copy. Clients can query fresh and historical data with a union view from any engine.

Workshop Use Case



E-Commerce Order Analytics



Order events



Topic:
orders_log

Entity: sales orders

Kafka topic: orders_log

Format: JSON

Sample data:

```
{
  "order_id": "ORD-97572D1F",
  "customer_id": "CUST-1024",
  "product_id": "PRD-00303",
  "status": "SHIPPED",
  "amount": 34.71,
  "event_time": "2026-07-23 16:29:26.120"
}

{
  "order_id": "ORD-C7E81CCE",
  "customer_id": "CUST-1011",
  "product_id": "PRD-00324",
  "status": "PLACED",
  "amount": 15.6,
  "event_time": "2026-07-23 16:53:08.346"
}
```

Product Catalog



Postgres:
product_catalog_cdc

Entity: product catalog

Kafka topic: product_lookup_cdc

Format: tabular

Sample data:

Column	Record 1	Record 2	Record 3
product_id	PRD-00001	PRD-00002	PRD-00003
sku	SKU-OIVG1043	SKU-OGMB0013	SKU-ZAWZ3794
product_name	Yang USB-C Speaker	Gonzalez Wireless Earbuds	Wolfe USB-C Earbuds
category	Electronics	Electronics	Electronics
brand	Yang	Gonzalez	Wolfe
unit_price	16.13	19.14	103.1
cost	6.98	7.24	54.31
weight_kg	2.72	2.83	9.72
in_stock	TRUE	TRUE	FALSE
rating	4.4	4.1	4.6
created_at	2024-07-12 00:00:00	2025-01-03 00:00:00	2024-12-15 00:00:00

Labs

- Lab 1: Fluss Catalog & Kafka Ingestion
- Lab 2: Product Enrichment
- Lab 3: Enabling the Lakehouse — Tiering Fluss to Iceberg
- Lab 4: Querying the Lakehouse — Real-Time + Historical
- Lab 5: Freshness Dashboard in Grafana (Optional)
- Bonus Track Lab: Tiering an Upsert (PK) Table

Lab 1 - Fluss Catalog & Kafka Ingestion

- Register Fluss as a Flink catalog
- Create the orders_log Fluss Log table
- Register the Kafka orders_log topic as a Flink source table
- Deploy the first long-running job (Kafka → Fluss) and confirm events are flowing

Lab 2 - Product Enrichment

- Create a Fluss PK table (product_lookup) mirroring a Postgres dimension table
- Wire up a postgres-cdc source and deploy the CDC → Fluss sync job
- Create the enriched orders_enriched Log table
- Deploy a temporal lookup join job, enriching orders with product data in real time

Lab 3 - Enabling the Lakehouse Tiering

- Start the shared Fluss Lakehouse Tiering Service
- Enable tiering on orders_enriched with a single ALTER TABLE
- Verify tiered data landed in Iceberg (via Trino/Lakekeeper)
- Understand why dual-tier storage (fresh Fluss + historical Iceberg) matters

Lab 4 - Querying the Lakehouse

- See the "union read" — Flink transparently merging fresh + tiered data
- Query the same table directly from Trino, independent of Fluss/Flink
- Run a cheap historical batch analytics query over the full order history
- (Bonus) Use Iceberg time travel to query a past snapshot

Lab 5 - Freshness Dashboard in Grafana

- Build a windowed analytics job aggregating revenue into PostgreSQL
- Import and explore a live Grafana dashboard fed by that job

Bonus Track Lab - Tiering an Upsert (PK) Table

- Build order_status, a PK (upsert) table, and its state-aggregation job
- Apply the same tiering mechanic to a PK table instead of a Log table
- Verify upsert-table tiering works identically to Log-table tiering in Iceberg

THANK YOU

Alex Campos