list_sync

Distributed CSV Synchronization System

Babini Stefano: stefano.babini5@studio.unibo.it

Distributed Systems A.Y. 2024/25

Alma Mater Studiorum - Università di Bologna

Content

Problem & Motivation

Low-latency DB synchronization
Traditional batch pipelinesintroduce too much delay for modern services that need up to the second data.

Complexity of custom CDC
Rolling your own Change Data Capture logic for each RDBMS (MySQL binlog vs. PostgreSQL WAL) is error-prone and hard to maintain.

Multi-vendor environments
Organizations often run both MySQL and PostgreSQL. A single CDC solution must handle both seamlessly.
Scalability & fault-tolerance requirements
Need at-least-once delivery, replayable event logs, and the ability to add consumers on the fly without touching producers.

Real-time analytics & integration
Streaming DB changes into Kafka enables downstream services (analytics, caches, search) to react immediately to data mutations.

System Overview

End-to-End CDC Pipeline

Captures row-level changes from source DBs → serializes into JSON → publishes to Kafka topics → consumed and written to CSV sinks

Event-Driven Architecture

Loosely-coupled connectors (producers) and consumers communicate via Kafka, enabling real-time streaming, replayability, and at-least-once delivery guarantees

Polyglot DB Support

MySQL via row-based replication; PostgreSQL via wal2json logical decoding both feeding the same Kafka topics in a unified format

Functional Requirements

1. Capture database changes (inserts, updates, deletes) from MySQL and PostgreSQL
2. Stream changes to Kafka topics in real-time
3. Support multiple concurrent consumers
4. Output changes to CSV files
5. Support both batch processing and real-time streaming
6. The data produced to kafka need to use the same format from both databases
7. Support both databases concurrently

Architecture Diagram

Event-Driven Backbone: Loosely coupled connectors and consumers communicate via Apache Kafka, enabling real-time streaming, message persistence for replay, and at least once delivery guarantees.

Polyglot Connectors

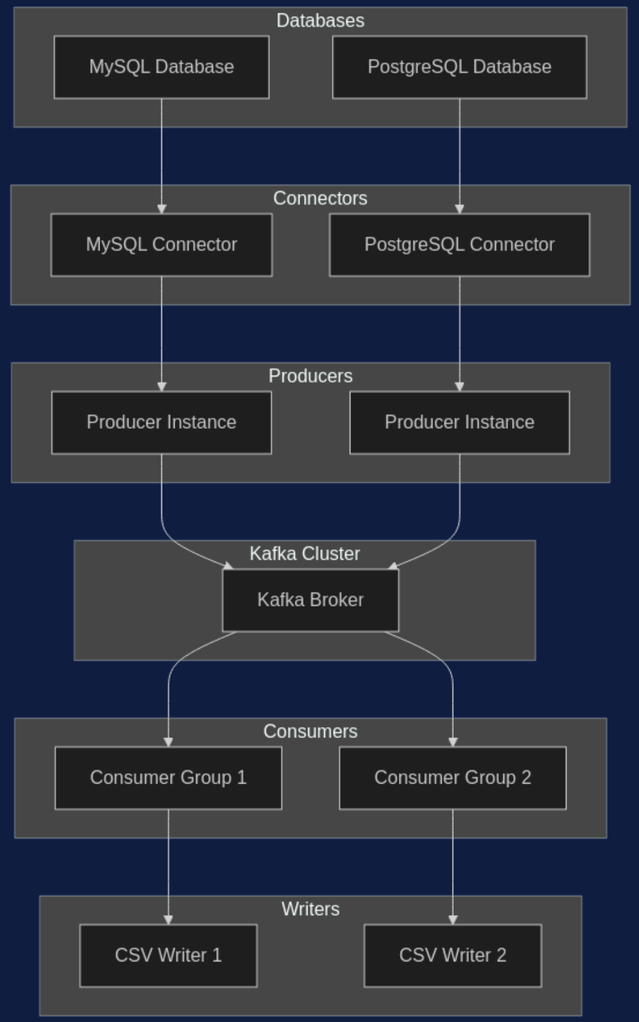
MySQL Connector: taps into row-based replication (binlog)

PostgreSQL Connector: uses wal2json for logical decoding.

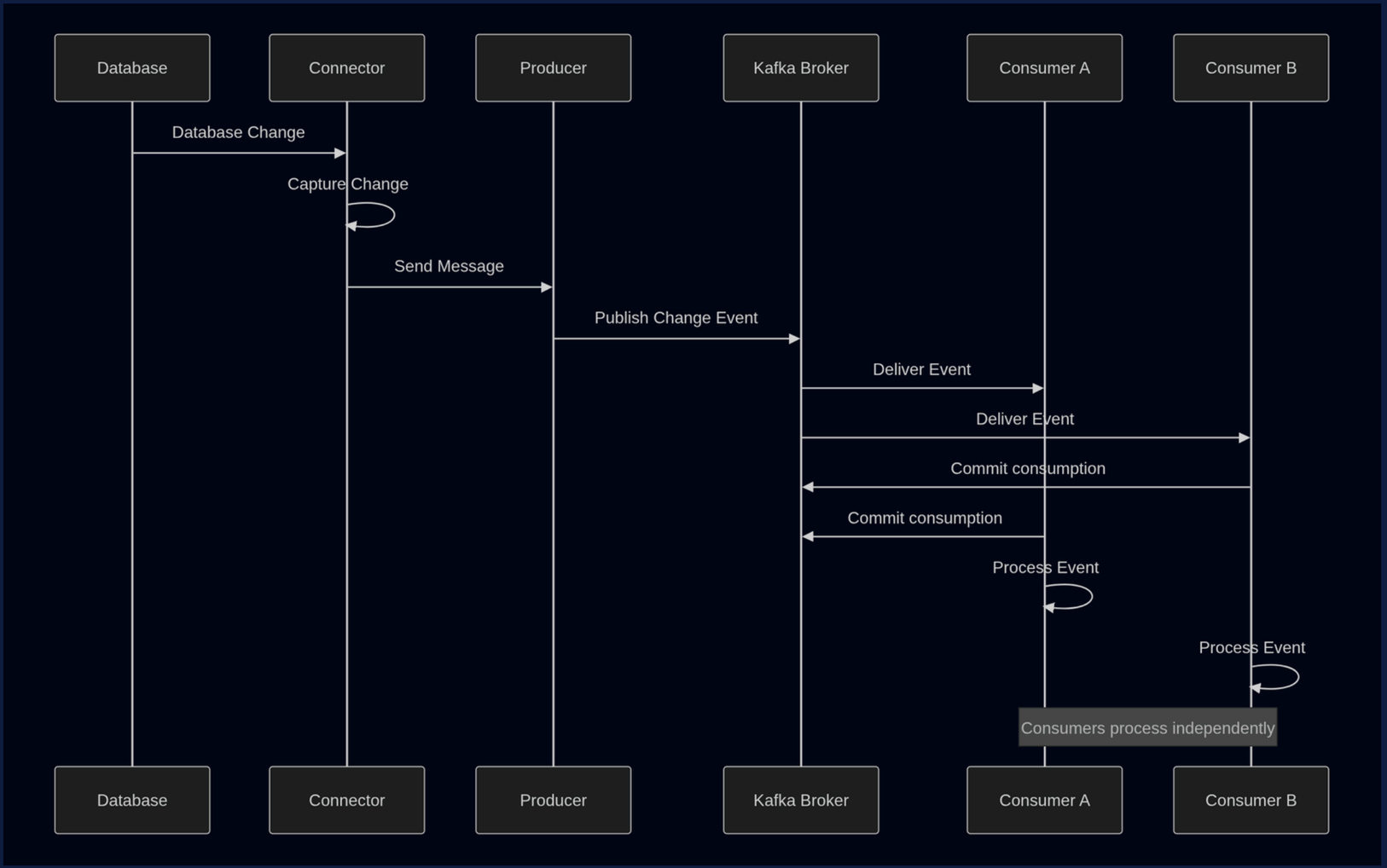
Kafka Cluster Configuration: Three broker nodes in Docker containers, each with RF=3 and 3 partitions ensuring fault tolerance and parallelism.

Consumer Groups & CSV Writers: Consumers pull batched messages, supporting independent scaling of downstream services .

Scalability & Resilience: Horizontal scaling possible at each layer (connectors, brokers, consumers).



Component Interaction



1. **Database change events**
2. **Connector normalization**
3. **Asynchronous production**
4. **Kafka persistence**
5. **Batch consumption**
6. **Offset commit & output**

Implementation Highlights

- **Python 3.12** library orchestrating A**pache Kafka** 3.9.0, deployed via **Docker** & Docker Compose
- **MySQL** row-based replication connector + **PostgreSQL wal2json** logical decoding connector, unified under one codebase
- Change events serialized to flexible **JSON**; keyed by primary key for partition based ordering guarantees
- Producers leverage **librdkafka**'s built-in retries and acks for at least once delivery

Performance & Testing Results

Tests

- Baseline End-to-End: processed ~500 k inserts + ~250 k updates & deletes in 43.06 s for seeding; full CDC pipeline (produce ➞ consume ➞ CSV) completed in 51.31 s without network constraints
- Network-Limited (20 ms RTT): Under a simulated 20 ms round-trip delay between Kafka brokers, seeding time rose to 51.78 s, and the full pipeline ran in 61.43 s

High-Load Limits

- At 100 concurrent consumers, test ran in 93.54 s
- performance bottlenecked by host CPU/RAM

Consumer Scalability (1 M Events)

- 48.48 s with 30 consumers
- 64.46 s with 50 consumers
- 78.60 s with 80 consumers (host CPU saturated)

Strengths & Weaknesses

Strengths

- **Polyglot CDC support:** Captures changes from both MySQL and PostgreSQL in a single codebase, eliminating duplicate effort.
- **Performance targets met:** Processes 1 million rows in under 3 minutes, validated across network conditions.
- **Scalable consumer architecture:** Leverages Kafka consumer groups and containerization for easy horizontal scaling.
- **Comprehensive test coverage:** Includes end-to-end and failure-mode tests under realistic network latencies.

Weaknesses

- **Limited configuration surface:** Only basic tuning options are exposed.
- **Minimal security by default:** TLS/SASL, authentication/authorization, and encryption must be manually enabled; not baked-in.
- **Sparse observability:** Lacks built-in metrics, dashboards, or alerts for monitoring throughput, lag, or error rates.

Conclusion & Future Work

Conclusion

- Unified, high-performance CDC: list_sync captures MySQL & PostgreSQL changes in real-time, streams them through Kafka
- Robust, scalable architecture: Leverages Kafka's replication, partitioning, and consumer groups for fault-tolerance, ordering, and horizontal scaling
- Thorough validation: Extensive tests under varied network latencies and consumer counts ensure reliability and predictability

Future Work

- Configurable tuning: Expose advanced parameters: e.g. retry/backoff policies, prefetch options
- Built-in security: Enable TLS/SASL, RBAC, and encryption by default across connectors, brokers, and consumers
- Enhanced observability: Integrate Prometheus/Grafana metrics, structured logging dashboards, and alerting for throughput, lag, and error rates
- Schema evolution & registry: Support Avro/Protobuf serialization with a Confluent Schema Registry for strong typing and version tracking
- Alternative sinks & formats: Add modules for Parquet, JDBC sinks, and REST API endpoints to broaden downstream integration

Thank you for attention!