Clase 213 — Streaming intro: Kafka, Kinesis

Parte: 5 — Ingeniería de Datos · Fuente: Kreps, Building a Real-Time Data Pipeline + Narkhede, Shapira, Palino Kafka: The Definitive Guide (O'Reilly, 2ª ed., 2021) + Reis & Housley cap. 7. Duración estimada: 85 min.

Clase 213 — Streaming intro: Kafka, Kinesis

> *Parte: 5 — Ingeniería de Datos · Fuente: Kreps, Building a Real-Time Data Pipeline + Narkhede, Shapira, Palino Kafka: The Definitive Guide (O'Reilly, 2ª ed., 2021) + Reis & Housley cap. 7. Duración estimada: 85 min.*

Objetivo

Entender el modelo streaming vs batch (Clase 208), producir y consumir mensajes en Kafka (con confluent-kafka o kafka-python), comparar contra AWS Kinesis Data Streams (managed equivalent), y reconocer los 3 problemas clásicos de streaming: exactly-once, out-of-order events, backpressure.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Diferenciar batch (datos llegan en bloques) de streaming (datos llegan continuos, baja latencia).
- Producir mensajes a un topic Kafka con keys (garantiza orden por key, distribuye load).
- Consumir con consumer groups: partitions distribuidas entre consumers, offset management.
- Diseñar para at-least-once (default razonable) y entender qué requiere exactly-once (transactional API).
- Decidir Kafka (self-hosted o Confluent Cloud) vs Kinesis (AWS) vs Pub/Sub (GCP) vs Event Hubs (Azure).

Temas

#	Tema	Por qué importa
1	Batch vs streaming — el espectro real	Spectrum: batch → micro-batch → streaming
2	Kafka model: topic, partition, offset	Vocabulario obligatorio.
3	Producer: keys, acks, idempotence	Garantías que querés desde el día 1.
4	Consumer groups + rebalancing	Cómo escalan los consumers.
5	Delivery semantics: at-most/at-least/exact	Trade-offs reales con código.
6	Kinesis comparison	Mismo modelo, vendor-specific.

Definiciones y características

- Topic: log durable, particionado, append-only. Análogo a una "tabla" en streaming.
- Partition: subset ordenado del topic. Cada partition tiene un único consumer dentro de un consumer group. Más partitions → más paralelismo.
- Offset: posición del consumer dentro de una partition. Persistido en Kafka (__consumer_offsets topic) o externamente.
- Producer: escribe mensajes a un topic. Puede especificar key (decide partition vía hash → mensajes con misma key van a la misma partition → orden garantizado por key).
- Consumer: lee mensajes de un topic en un consumer group. Si un consumer muere, otro toma sus partitions (rebalancing).
- Consumer group: lógicamente "un consumidor". 3 consumers en el mismo grupo se dividen las

partitions; 3 grupos distintos cada uno ve todos los mensajes.
- At-most-once: mensaje se pierde si falla algo. acks=0.
- At-least-once: mensaje puede llegar duplicado. acks=all + enable_auto_commit=False + commit después de procesar. Default razonable.
- Exactly-once: requiere idempotent producer (enable.idempotence=true) + transactional API (Kafka Streams o Flink). Caro, complejo, no siempre necesario.
- Backpressure: cuando consumer no procesa tan rápido como producer escribe → lag crece. Acción: escalar consumers, o aplicar throttling al producer.
- Kinesis Data Stream: AWS-managed. "Shard" ≈ partition, "iterator" ≈ offset, KCL (Kinesis Client Library) ≈ consumer group. Mismo modelo conceptual, distintos nombres.

Dataset / recursos

- Kafka local: docker-compose con Kafka (KRaft mode, sin Zookeeper) + Kafka UI.
- Stream sintético: producer genera "click events" cada 100 ms.
- Librerías: confluent-kafka>=2.5 (más robusta) o kafka-python>=2.0 (más simple), faker para data sintética.

Ejercicios

1. Setup local: docker-compose con Kafka + Kafka UI. Crear topic clicks con 4 partitions. Verificar con docker exec kafka kafka-topics --list ....
2. Producer: script Python que produce 1000 mensajes con key=user_id, value={"page":"/foo","ts":...}. Verificar en Kafka UI que mensajes con mismo user_id caen en la misma partition.
3. Consumer 1 instancia: consumer.subscribe(['clicks']) + loop for msg in consumer. Procesar = print(msg.value()). Commitear offset cada 100 mensajes.
4. Consumer group, 2 instancias: levantar 2 consumers con mismo group.id. Confirmar que se reparten las 4 partitions (2-2). Matar uno, observar rebalancing — el otro toma las 4.
5. At-least-once explícito: enable.auto.commit=False, procesar mensaje, consumer.commit(). Si crash entre procesar y commit → duplicate al reiniciar.

Homework verificable

Sistema con:

1. Producer Python que simula 100 eventos/s durante 1 min (sintéticos con faker).
2. 3 consumers en el mismo group consumiendo de un topic con 6 partitions.
3. Cada consumer procesa y escribe a una tabla DuckDB (idempotente: PK = (user_id, event_ts)).
4. Métrica de consumer lag monitoreada (chequear kafka-consumer-groups --describe).
5. Demonstración: matar 1 consumer durante la corrida y verificar rebalancing + cero pérdida de mensajes.
6. README comparando con un equivalente en Kinesis (snippet de código sin necesidad de cuenta AWS).

Criterio de aceptación: 6000 mensajes producidos → 6000 mensajes en DuckDB (sin duplicados gracias a PK), consumer lag estabiliza <1000, rebalancing funcionó.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
Topic does not exist aunque acabás de crea	Auto-create topics está deshabilitado y cr
Consumer lag crece infinito	Consumer no procesa rápido suficiente. Fix
Mensajes duplicados al reiniciar consumer	Falta commit del último offset procesado.
UnknownTopicOrPartitionError intermitente	Rebalancing en curso. Fix: catch y reinten
Performance horrible con enable.idempotenc	Idempotente requiere ordering por key, baj
Producer.flush() se cuelga	Network unreachable. Fix: agregar timeout,
Kafka UI no muestra topics	UI conectada a broker antes que esté ready

Preguntas frecuentes

¿Cuándo necesito streaming en vez de batch?

Cuando: (1) latencia importa (fraud detection: minutos), (2) volume es alto y constante (logs, telemetría), (3) data es continuamente generada (clickstream, IoT). Si tus datos llegan en cron diario y no urge: batch. La regla: batch primero, streaming cuando duela.

Kafka self-hosted o Confluent Cloud?

Self-hosted: control total, costo en EC2/operaciones. Confluent Cloud (managed): no operás brokers, pagás más. Para empezar: Confluent Cloud free tier. Para escala >100 MB/s sostenido: el TCO favorece self-hosted o Confluent Cloud Enterprise.

Kafka vs Kinesis vs Pub/Sub?

- Kafka: open-source, multi-cloud, ecosistema enorme (Kafka Connect, Kafka Streams, ksqlDB).
- Kinesis: AWS-native, integrado con Lambda/Firehose. Más limitado (shards manualmente escalados; Kinesis On-Demand mejora esto).
- Pub/Sub: GCP-native, simpler API (pull/push, no shards expuestos), gestionado al 100%.

Para multi-cloud o vendor-neutral: Kafka. Para AWS/GCP-native simple: Kinesis/Pub/Sub.

¿Cuándo necesito exactly-once?

Casi nunca. At-least-once + idempotent consumer (idempotency via PK en DB) resuelve el 95% de los casos. Exactly-once con transactional API tiene 20-30% overhead. Reservarlo para: pagos, billing.

¿Cómo manejo schemas evolucionando?

Schema Registry (Confluent, AWS Glue, Apicurio): producer registra schema Avro/Protobuf, consumer lo lee. Compatibilidad checks evita producer rompiendo consumers downstream.

¿Streaming SQL? ¿Flink, Spark Streaming, ksqlDB?

Para queries continuas (windowed aggregates, joins de streams):

- Kafka Streams: lib Java/Scala, embed en tu app.
- ksqlDB: SQL sobre Kafka topics.
- Apache Flink: streaming-first, stateful, exactly-once.
- Spark Structured Streaming: micro-batch (latencia ~seconds), reusa skills de Spark.

Para Python: PyFlink y Spark Structured Streaming son los más usados.

Referencias

- Narkhede, Shapira, Palino Kafka: The Definitive Guide (O'Reilly, 2ª ed., 2021).
- Kafka docs — Quickstart, Producer/Consumer configs.
- Confluent Cloud free tier.
- Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 7.
- Awesome Kafka.

Siguiente clase

Clase 214 — Formatos columnares: Parquet, Avro

Apéndice: notebook (primer bloque)

Primera celda ejecutable del notebook de la clase.

```
compose = '''\
services:
  kafka:
    image: bitnami/kafka:3.7
    ports: ["9092:9092"]
    environment:
      KAFKA_CFG_NODE_ID: 1
      KAFKA_CFG_PROCESS_ROLES: controller,broker
      KAFKA_CFG_LISTENERS: PLAINTEXT://:9092,CONTROLLER://:9093
      KAFKA_CFG_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_CFG_LISTENER_SECURITY_PROTOCOL_MAP: CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      KAFKA_CFG_CONTROLLER_LISTENER_NAMES: CONTROLLER
      KAFKA_CFG_CONTROLLER_QUORUM_VOTERS: 1@kafka:9093
      KAFKA_CFG_AUTO_CREATE_TOPICS_ENABLE: "true"
    healthcheck:
      test: kafka-topics.sh --bootstrap-server localhost:9092 --list
      interval: 5s

  kafka-ui:
    image: provectuslabs/kafka-ui:latest
    ports: ["8080:8080"]
    environment:
      KAFKA_CLUSTERS_0_NAME: local
      KAFKA_CLUSTERS_0_BOOTSTRAPSERVERS: kafka:9092
    depends_on: { kafka: { condition: service_healthy } }
'''
print(compose)
print('# levantar: docker-compose up -d')
print('# UI: http://localhost:8080')
```

Archivos complementarios

- notebook.ipynb