Apache Kafka

실시간 데이터 스트리밍 플랫폼의 이해와 활용

분산 시스템 · 데이터 아키텍처 · 실시간 처리

CHAPTER 01

카프카 소개: 데이터 패러다임의 변화

데이터 패러다임 변화

기존의 저장 중심(Data-at-rest) 시스템에서 실시간 흐름 중심(Data-in-motion) 시스템으로의 근본적 전환

Legacy System		Apache Kafka
Data-at-rest	→	**Data-in-motion**
배치 처리, 지연 분석		이벤트 스트리밍, 실시간

분산 스트리밍 플랫폼 3대 기능

- **실시간 읽기/쓰기 (Publish & Subscribe)**
  메시지 큐와 유사하게 레코드 스트림을 읽고 씀
- **내결함성 저장 (Store)**
  레코드를 디스크에 복제 저장하여 장애 시에도 데이터 보존
- **실시간 처리 (Process)**
  데이터 발생 즉시 스트림 처리 애플리케이션으로 가공

Delivery Semantics (메시지 전달 보장)

At-most-once (최대 한 번)
메시지 유실 가능성 있음, 중복 없음.
속도가 중요한 로그 수집 등에 사용.

At-least-once (최소 한 번)
메시지 유실 없음, 중복 발생 가능.
대부분의 시스템 기본 설정 (멱등성 처리 필요).

Exactly-once (정확히 한 번) - 조건부
Idempotent Producer + Transaction 지원 시 가능.
Kafka Streams 등에서 강력한 보장 제공.

핵심 강점

- **Decoupling (결합도 감소)**
  Producer와 Consumer가 서로 독립적으로 동작하여 유연성 확보
- **Replay (데이터 재생)**
  이벤트를 로그로 보관하므로 과거 시점부터 재처리 가능

CHAPTER 02

내부 구조 Deep Dive

Append-only Log

디스크에 순차적으로만 기록.
랜덤 I/O를 최소화하여
높은 쓰기 처리량 달성.

Offset 기반 소비

컨슈머가 자신의 읽은 위치(Offset)를
관리. 서로 다른 속도로
독립적인 소비 가능.

읽기 ≠ 삭제

메시지 큐와 달리 읽어도
데이터가 사라지지 않음.
설정된 기간/용량까지 보존.

Replay (재생)

과거의 Offset으로 되돌아가
데이터를 다시 처리 가능.
로직 변경 시 유용.

아키텍처 및 설정

Partition (파티션)

병렬 처리와 순서 보장의 단위. Leader(쓰기/읽기)와 Follower(복제)로 구성. ISR(In-Sync Replicas)로 데이터 유실 방지.

KRaft (Kafka Raft Metadata) New Standard

Kafka 3.3부터 Production-ready. Kafka 4.0에서는 ZooKeeper 모드가 **완전 제거**됨. 별도 코디네이터 없이 자체 메타데이터 관리.

고성능 I/O 메커니즘

Page Cache 활용

JVM 힙 메모리 대신 OS의 페이지 캐시를 적극 사용. GC 오버헤드를 줄이고, 재시작 시에도 캐시 유지(Warm Cache).

Zero-Copy 전송

`sendfile()` / Java NIO `transferTo()` 사용. 커널 공간의 데이터를 유저 공간 복사 없이 NIC 버퍼로 직접 전송.

DATA FLOW

데이터 흐름: Producer → Broker → Consumer

Producer

이벤트 생성 및 전송

- App Server
- Web Service
- DB Change

Partitioner

Key가 있으면: 해시값으로 파티션 결정

Key 없으면: 라운드 로빈

Broker Cluster

Topic 및 Partition 저장소

Topic: order.events

Partition 0 (Leader)	Replica (Follower)
Partition 1 (Leader)	Replica (Follower)
Partition 2 (Leader)	Replica (Follower)

Consumer Group

Group ID: order-service

병렬 처리 (Parallel Processing)

- Consumer A
- Consumer B
- Consumer C

같은 Key → 같은 Partition → 순서 보장 (Ordering)

I/O 최적화 — Page Cache와 Zero-copy

Page Cache 활용

카프카는 JVM 힙 메모리 대신 **운영체제(OS)의 페이지 캐시**를 적극 활용합니다. 데이터가 디스크에 기록되기 전 메모리에 머무르며, 읽기 요청 시에도 디스크 접근 없이 메모리에서 바로 응답합니다. 프로세스 재시작 시에도 캐시가 유지(Warm Cache)되는 장점이 있습니다.

Sequential I/O

Zero-copy 전송

기존 방식(read/write)은 커널↔유저 공간 간 불필요한 데이터 복사를 유발합니다. 카프카는 리눅스의 `sendfile()` 및 Java NIO `transferTo()` 시스템 콜을 사용하여, 데이터를 **커널 영역에서 네트워크 인터페이스(NIC) 버퍼로 직접 전송**합니다. CPU 사용량을 줄이고 컨텍스트 스위칭을 최소화합니다.

sendfile() / transferTo()

Zero-copy 데이터 경로

1. Disk에서 OS Page Cache로 데이터 로드 (커널 공간)
2. **Zero-copy: 유저 공간 복사 없이 바로 소켓 버퍼로 이동**
3. NIC Buffer를 통해 네트워크 전송

* 기존 방식은 커널→유저→커널 복사 과정이 추가됨 (총 4회 복사 → 2회로 단축)

기대 효과

CPU와 메모리 대역폭 소모를 최소화하여, 디스크 속도가 아닌 **네트워크 대역폭 한계에 가까운 처리량**을 달성합니다.

제약 사항

SSL/TLS 암호화를 적용할 경우, 암복호화를 위해 데이터가 유저 공간(애플리케이션)을 거쳐야 하므로 Zero-copy를 사용할 수 없습니다.

Exactly-once 구현 원리 (조건부)

카프카의 "Exactly-once"는 마법이 아니라,**Idempotence(멱등성)**와**Transaction(트랜잭션)**메커니즘이 결합되어야 성립하는**조건부 보장**입니다.

1 Idempotent Producer

Producer ID(PID)와 Sequence Number를 통해 브로커가 중복 전송을 감지하고 제거합니다. 네트워크 재시도 시에도 중복이 발생하지 않습니다.

```
enable.idempotence=true
acks=all
max.in.flight... ≤ 5
```

2 Transactions (2PC)

여러 파티션에 걸친 쓰기 작업을 원자적으로 처리합니다. Transaction Coordinator가 2-Phase Commit 프로토콜을 수행하여 전체 커밋/롤백을 결정합니다.

```
transactional.id 설정
initTransactions()
commitTransaction()
```

3 Consumer Isolation

컨슈머는 트랜잭션이 성공적으로 커밋된 메시지만 읽습니다. 진행 중이거나 실패한(aborted) 트랜잭션의 메시지는 무시합니다.

```
isolation.level =
read_committed
(Default:
read_uncommitted)
```

4 External System

카프카 외부(DB, S3 등)로 나가는 경우, 카프카만으로는 Exactly-once를 보장할 수 없습니다. 타겟 시스템의 멱등성 처리나 2PC 지원이 필요합니다.

```
Connect Sink: Idempotent
Write
State Store: Checkpointing
```

Rebalancing 심화 & 운영 전략

Rebalance의 영향 (Risk)

컨슈머 그룹 멤버 변경 시 파티션 소유권을 재조정하는 과정입니다. 이 동안 **Stop-the-world (STW)**가 발생하여 일시적으로 메시지 처리가 중단될 수 있습니다.

> **주요 발생 원인**
> Consumer 추가/제거, 배포(Rolling Update), 일시적 네트워크 장애로 인한 Heartbeat 실패

Cooperative Rebalancing (증분)

전체 중단(Eager) 대신, 영향을 받는 파티션만 점진적으로 재할당하여 STW 시간을 최소화합니다.

구분	Eager (기존)	Cooperative (권장)
동작	모든 컨슈머가 일단 멈춤	필요한 파티션만 이동
영향	전체 처리 중단	중단 최소화

Static Membership (정적 멤버십)

컨슈머에 고유 ID를 부여하여, 재시작 시에도 그룹을 탈퇴하지 않고 멤버십을 유지합니다. Rolling Update 시 불필요한 리밸런싱을 방지합니다.

`group.instance.id = "consumer-host-1"` `session.timeout.ms = 30000+`

Offset 관리 & Lag 모니터링

Offset Commit 전략

Auto 편하지만 중복/유실 제어 어려움

Manual 정확하지만 복잡도 증가 (처리 후 커밋 권장)

Consumer Lag (지연)

Producer 속도 > Consumer 속도일 때 증가. 특정 파티션만 Lag가 튄다면 **Data Skew(데이터 쏠림)**나 컨슈머 로직 병목을 의심해야 합니다.

Partition·Key·Replay 전략

Partition 수 산정

기본 원칙

파티션 수는 컨슈머의 최대 병렬 처리 개수를 결정합니다. 목표 처리량을 기준으로 산정해야 합니다.

산정 공식 (예시)

```
목표 Throughput ÷ 파티션당 처리량
(100MB/s ÷ 10MB/s = 최소 10개)
```

고려 사항

레코드 크기, 압축률, 네트워크 대역폭, 디스크 I/O를 종합적으로 프로파일링해야 정확합니다.

Key 설계 전략

- **순서 보장 (Ordering)**
  동일한 Key는 항상 동일한 파티션으로 전송됩니다. 사용자 ID 등을 Key로 사용하면 해당 사용자의 이벤트 순서를 보장할 수 있습니다.
- **분산 (Distribution)**
  특정 Key에 데이터가 몰리면 **Hot Partition** 문제가 발생합니다. 이 경우 Key에 난수를 섞거나 Custom Partitioner를 고려해야 합니다.

CUSTOM PARTITIONER

특정 비즈니스 로직(VIP 고객 전용 파티션 등)이 필요할 때 구현합니다.

Replay & 장애 대응

OFFSET RESET

컨슈머 그룹의 Offset을 과거 시점으로 되돌려 데이터를 재처리합니다. (로직 버그 수정 후 배포 시 유용)

DLQ (DEAD LETTER QUEUE)

처리 실패한 메시지는 별도 토픽(DLQ)으로 빼서 메인 파티션의 지연(Lag)을 방지하고, 나중에 수동으로 재처리합니다.

IDEMPOTENT CONSUMER

재처리 시 중복이 발생하므로, 소비 로직은 반드시 **멱등성(Idempotency)**을 보장해야 안전합니다.

CHAPTER 03

프로젝트 설계 실전

설계 시 반드시 고려해야 할 5가지 질문

Q1	Q2	Q3	Q4	Q5
메시지 손실을 허용할 수 있는가?	어떤 Key 기준으로 묶을 것인가?	순서 보장이 필요한가?	전체 이력 vs 최신 상태 유지?	몇 개의 Consumer Group이 필요한가?

보장 수준 (Delivery Guarantee)

- **Throughput-first**: acks=0, retries=0 (손실 가능, 최대 속도)
- **At-least-once**: acks=1 or all, retries>0 (기본, 중복 가능)
- **Exactly-once**: enable.idempotence=true + Transaction

데이터 수명주기 (Retention)

- **설정**: retention.ms (시간), retention.bytes (용량)
- **Compaction**: cleanup.policy=compact (최신 상태만 유지)

* 이력이 중요한지, 최종 상태가 중요한지에 따라 정책 결정

순서 전략 (Ordering Strategy)

- **Same Key → Same Partition**: 엔티티 단위 순서 보장
- **No Key**: Round-robin / Sticky (순서 보장 X, 분산 최적화)

* 카프카는 파티션 내부 순서만 보장하며, 전역 순서는 보장하지 않음

파티션 수 산정 & 스케일링

- **산정 공식**: 목표 Throughput ÷ 파티션당 처리량

```
예) 100MB/s ÷ 10MB/s = 최소 10개 파티션
```

* 파티션은 늘리기는 쉽지만 줄이기는 어려우므로 초기 설계 시 여유 있게 고려

Kafka Connect & 확장 생태계

Kafka Connect

데이터 시스템 간의 연결을 위한 오픈소스 프레임워크입니다. 코드를 거의 작성하지 않고도 데이터를 가져오거나 내보낼 수 있습니다.

- ✓ **Source Connector:** DB, 파일 등에서 데이터 수집
- ✓ **Sink Connector:** S3, Elasticsearch 등으로 데이터 적재
- ✓ 운영 표준화 및 확장성 확보 용이

Kafka Streams

별도의 클러스터 없이 Java/Scala 애플리케이션 내에서 동작하는 경량 스트림 처리 라이브러리입니다.

- ✓ 상태 저장소(State Store) 내장으로 join, windowing 지원
- ✓ Exactly-once 처리 지원 (transactional.id)
- ✓ 애플리케이션 배포와 함께 스케일 아웃 가능

CDC: Change Data Capture

데이터베이스의 변경 사항을 실시간 이벤트 스트림으로 추출합니다. **Debezium**이 사실상 표준으로 사용됩니다.

`MySQL Binlog` `PostgreSQL WAL` `MongoDB Oplog`

> **활용 사례**
> 기존 레거시 DB를 수정하지 않고 MSA 환경으로 데이터를 실시간 동기화하거나, 캐시 갱신 트리거로 활용

ksqlDB

Kafka Streams를 SQL 문법으로 추상화한 스트리밍 데이터베이스입니다. 복잡한 코드 없이 실시간 쿼리가 가능합니다.

> **SQL 예시**
>
> ```
> CREATE STREAM fraud_alert AS SELECT * FROM payments WHERE amount > 10000;
> ```

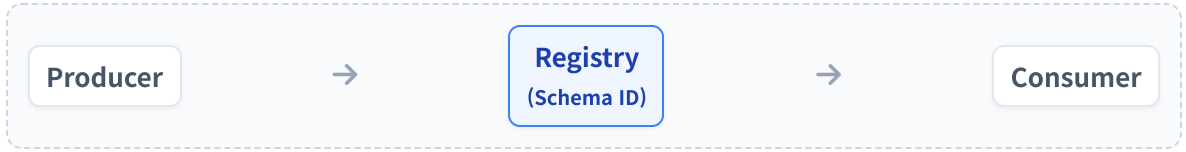
대시보드용 실시간 집계나 간단한 ETL 파이프라인 구축에 적합합니다.

CHAPTER 03

데이터 형식과 스키마 관리

구분	JSON (Schemaless)	Avro (Schema-based)
스키마 관리	없음 (데이터에 필드명 포함)	**Schema Registry로 중앙 관리**
용량 효율	**낮음** (필드명 반복으로 무거움)	**높음** (바이너리 포맷, 필드명 제외)
타입 안정성	약함 (파싱 시점 에러 발생)	**강함** (컴파일/직렬화 시점 검증)
가독성	**우수** (사람이 읽기 쉬움)	낮음 (바이너리라 도구 필요)

Schema Registry 역할



`BACKWARD` (Default)
Consumer 먼저 업그레이드. 새 스키마로 구 데이터 읽기 가능

`FORWARD`
Producer 먼저 업그레이드. 구 스키마로 신 데이터 읽기 가능

`FULL`
순서 상관없음. 양방향 호환성 보장

운영 권장사항

초기에는 JSON이 편할 수 있으나, 서비스 규모가 커지면 **Avro + Schema Registry** 조합을 강력 권장합니다.

Producer와 Consumer 간의 명확한 **데이터 계약(Contract)**을 통해, 필드 변경으로 인한 장애와 해석 오류를 원천 차단할 수 있습니다.

마무리

운영 가능한 이벤트 플랫폼 구축

로그 기반 구조
Append-only Log, Offset, Replay 가능

확장성 & 내구성
Partition 분산, 복제(Replica), ISR

높은 처리량
배치 전송, Zero-copy, Page Cache 활용

조건부 Exactly-once
Idempotence + Tx + read_committed

운영 필수 체크리스트

- Rebalancing 전략
- Lag 모니터링
- 파티션/Key 설계
- 스키마 관리 (Avro)

1. **PoC / 점진적 전환**
   Connect + CDC 활용
2. **최적화 (Calibration)**
   파티션 수, 처리량 검증
3. **관찰가능성 구축**
   메트릭, 알람, 대시보드
4. **가이드라인화**
   팀 표준 문서 작성

"단순한 메시지 도입을 넘어, 운영 가능한 데이터 플랫폼으로의 진화"