Apache Kafka

CM1-2 박주원

Kafka의 등장

- 2011년 LinkedIn에서 “Kafka: a Distributed Messaging System for Log Processing” 라는 논문으로 처음 소개

Kafka: a Distributed Messaging System for Log Processing

Jay Kreps
LinkedIn Corp.
jkreps@linkedin.com

Neha Narkhede
LinkedIn Corp.
nnarkhede@linkedin.com

Jun Rao
LinkedIn Corp.
jrao@linkedin.com

ABSTRACT

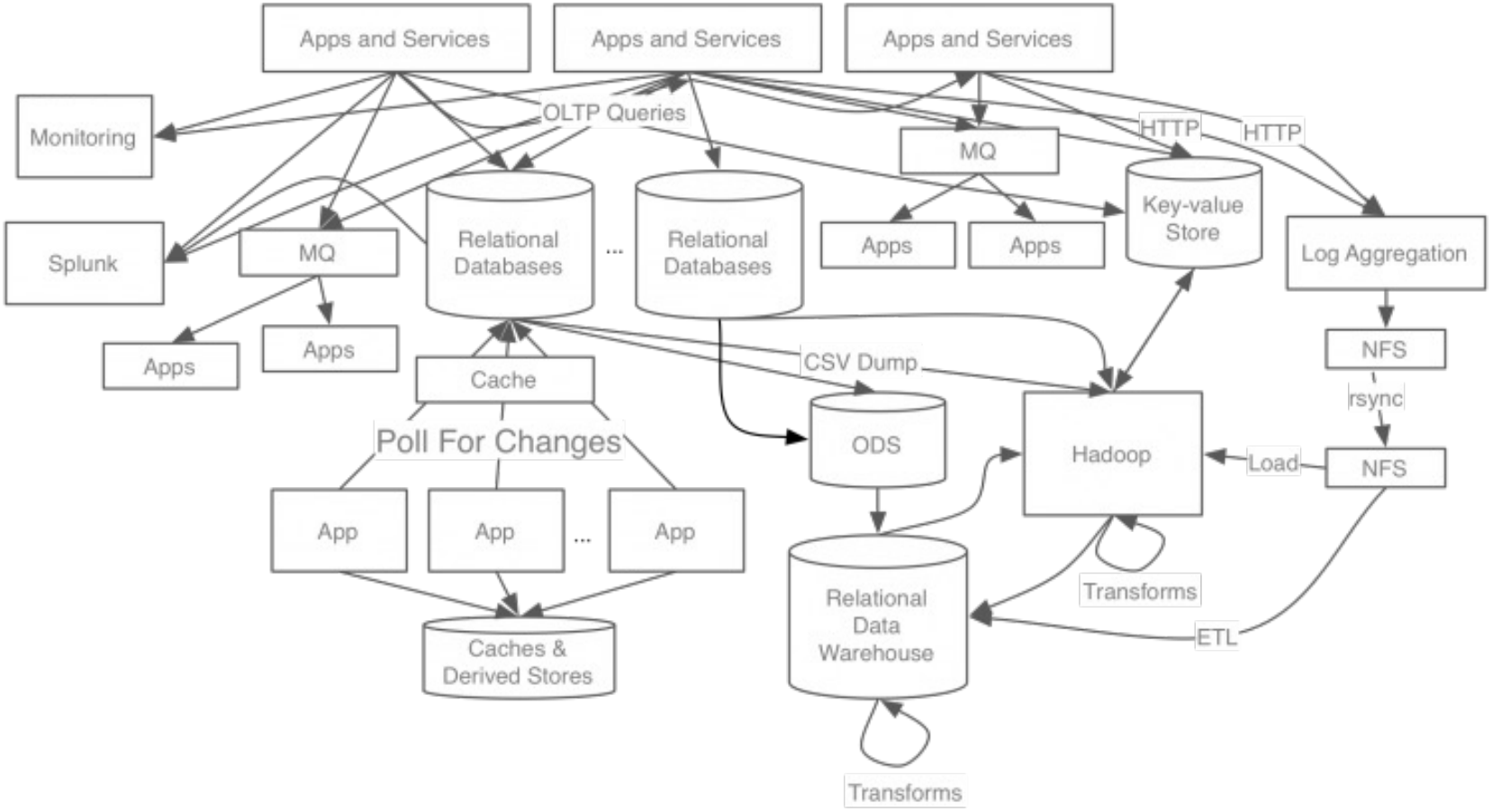
Log processing has become a critical component of the data pipeline for consumer internet companies. We introduce Kafka, a distributed messaging system that we developed for collecting and delivering high volumes of log data with low latency. Our system incorporates ideas from existing log aggregators and messaging systems, and is suitable for both offline and online message consumption. We made quite a few unconventional yet practical design choices in Kafka to make our system efficient and scalable. Our experimental results show that Kafka has superior performance when compared to two popular messaging systems. We have been using Kafka in production for some time and it is processing hundreds of gigabytes of new data each day.

granular click-through rates, which generate log records not only for every user click, but also for dozens of items on each page that are not clicked. Every day, China Mobile collects 5–8TB of phone call records [11] and Facebook gathers almost 6TB of various user activity events [12].

Many early systems for processing this kind of data relied on physically scraping log files off production servers for analysis. In recent years, several specialized distributed log aggregators have been built, including Facebook’s Scribe [6], Yahoo’s Data Highway [4], and Cloudera’s Flume [3]. Those systems are primarily designed for collecting and loading the log data into a data warehouse or Hadoop [8] for offline consumption. At LinkedIn (a social network site), we found that in addition to traditional offline analytics, we needed to support most of the

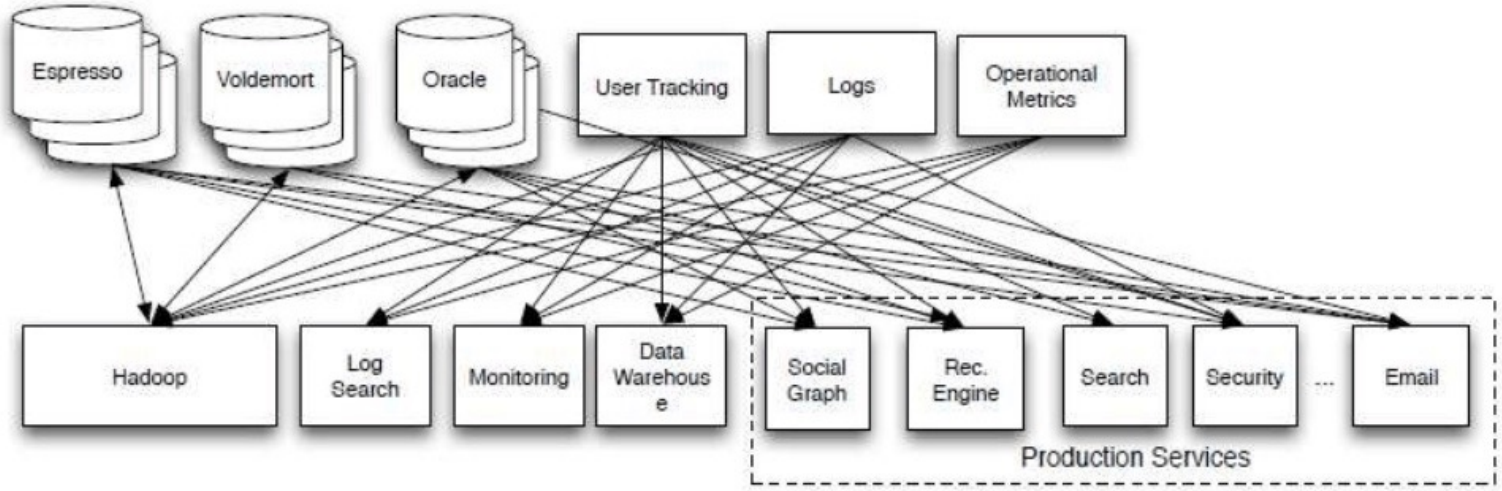
Kafka 등장 배경

- LinkedIn의 기존 시스템 구성도



Kafka 등장 배경

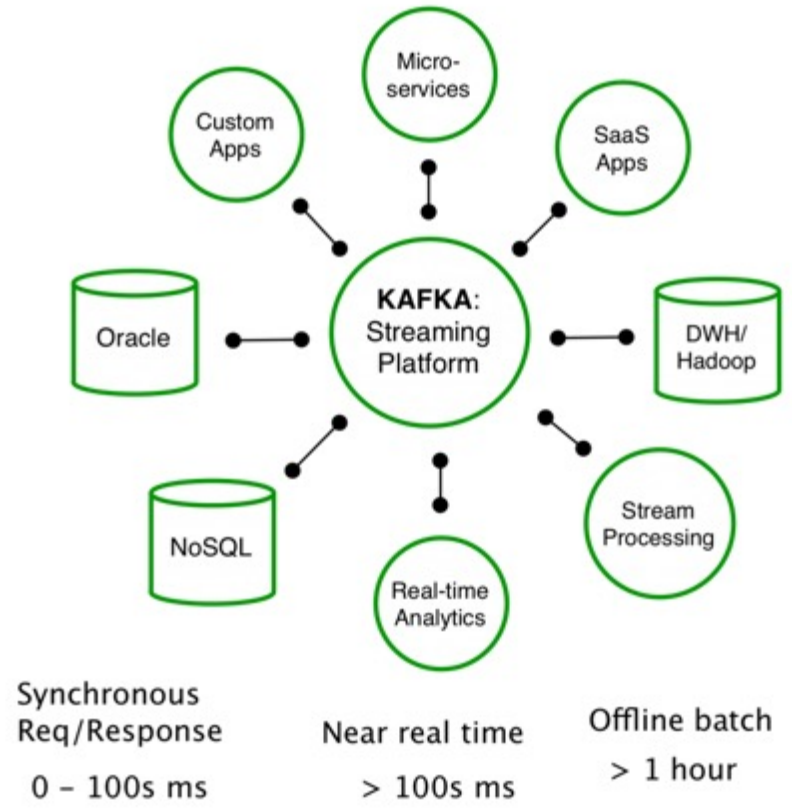
- 기존 Point to Point 구조
- 연결이 하나가 늘어날 경우 수많은 시스템과 1:1로 연결
  - 유지보수의 어려움
  - 성능에도 치명적

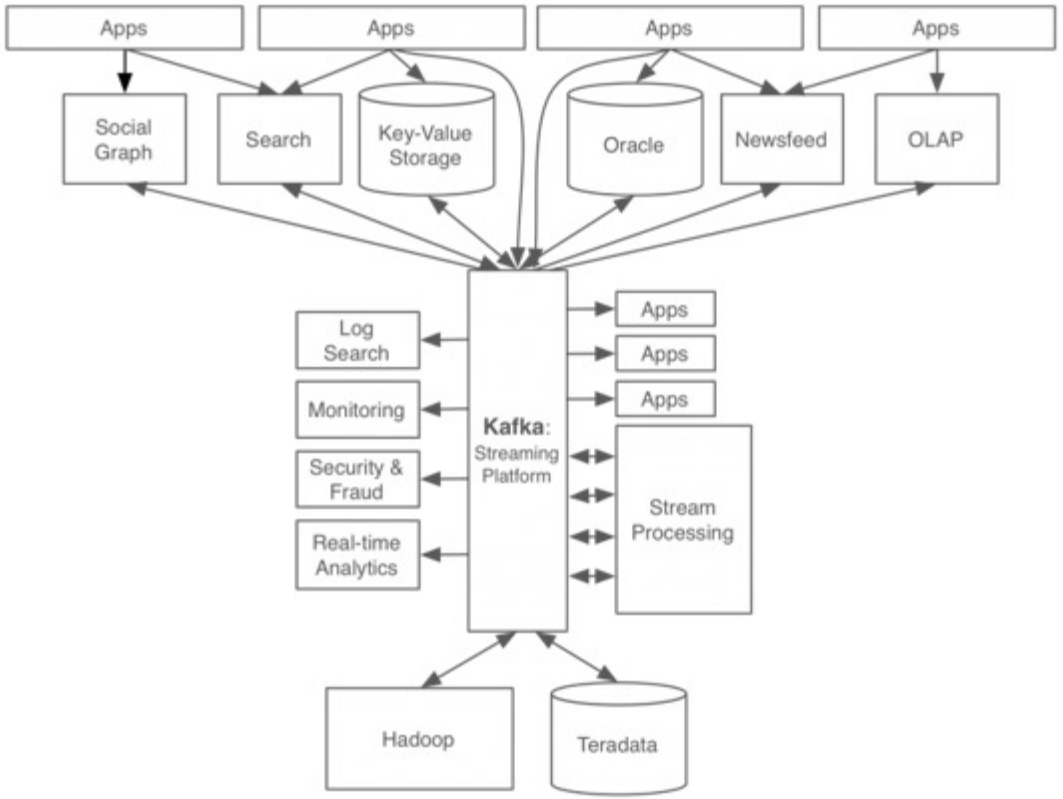


Kafka 자체 개발

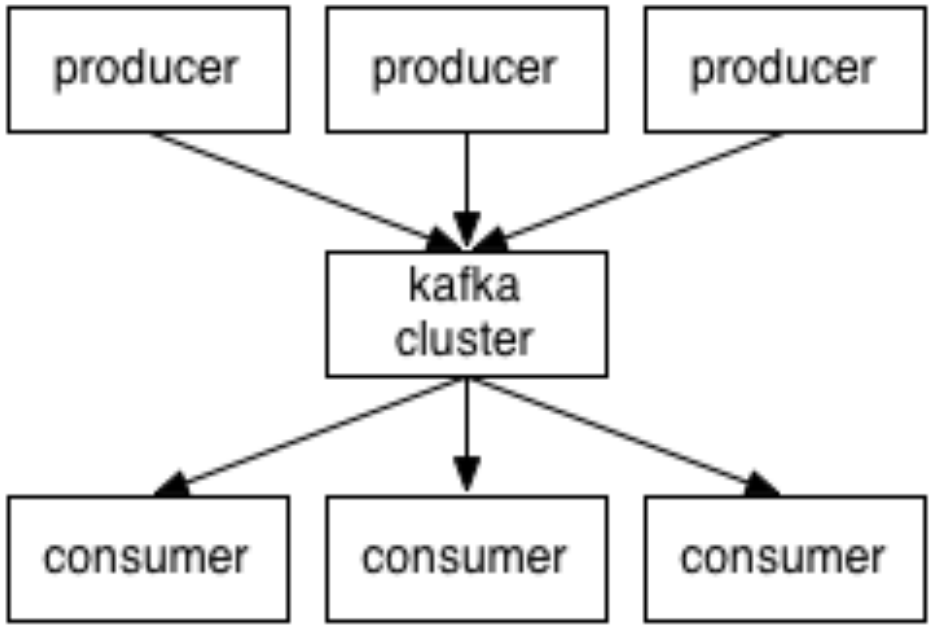
- 요구 사항
  - 높은 처리량으로 실시간 처리한다.
  - 임의의 타이밍에 데이터를 읽는다.
  - 다양한 제품과 시스템에 쉽게 연동한다.
  - 메시지를 잃지 않는다.
- 실현 수단
  - 메시징 모델과 스케일 아웃형 아키텍처
  - 디스크로의 데이터 영속화
  - 이해하기 쉬운 API 제공
  - 전달 보증

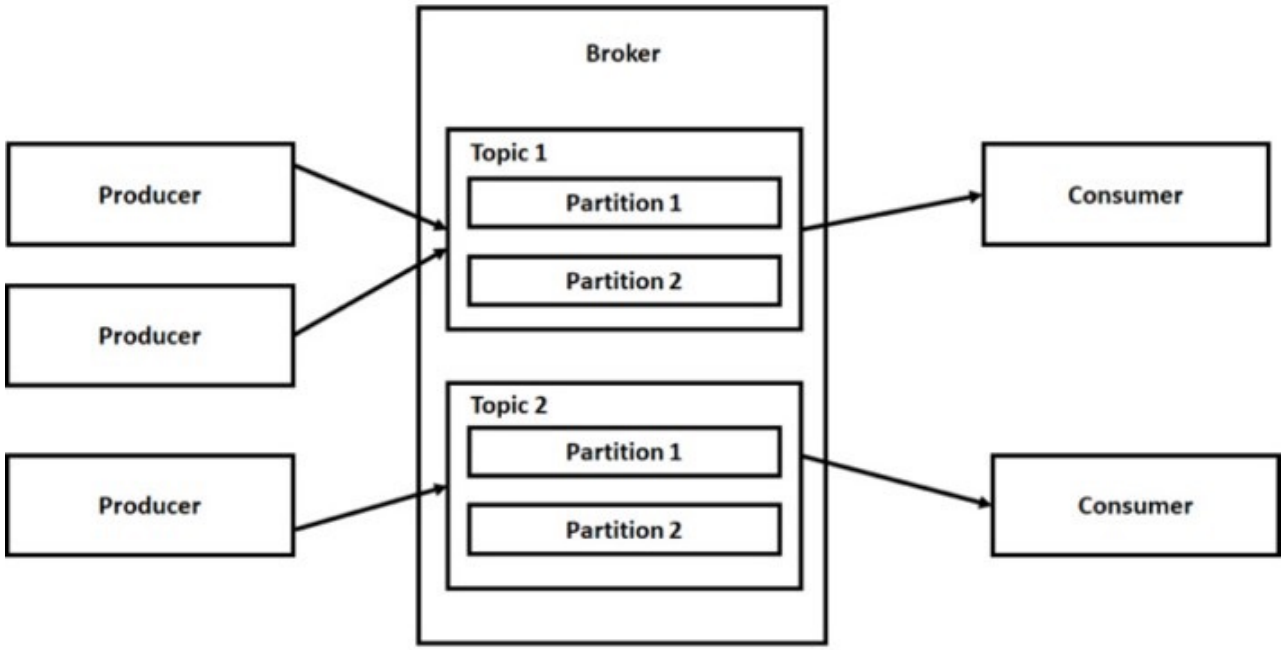
새로운 시스템 아키텍처 및 구성도





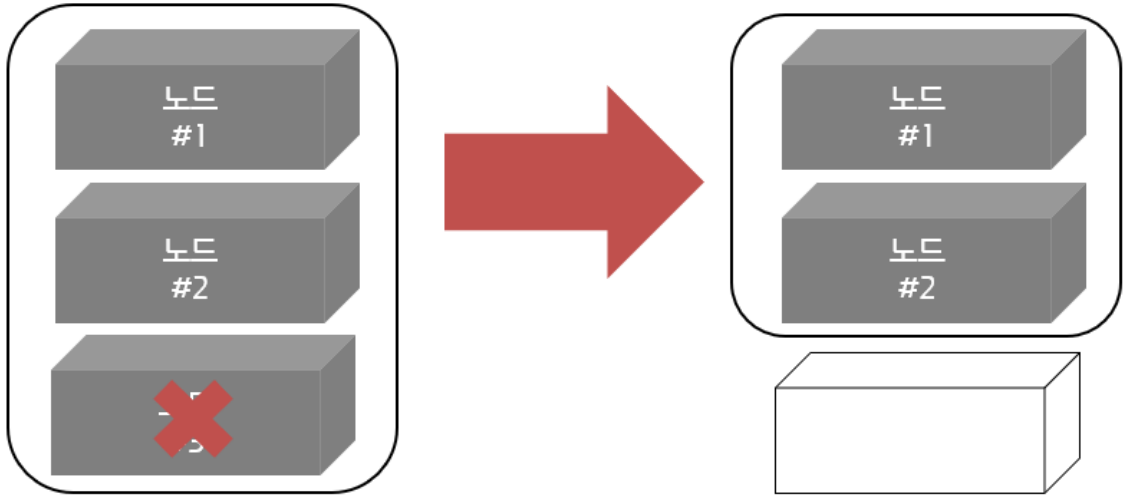
Pub/Sub 모델, Message Queueing 모델



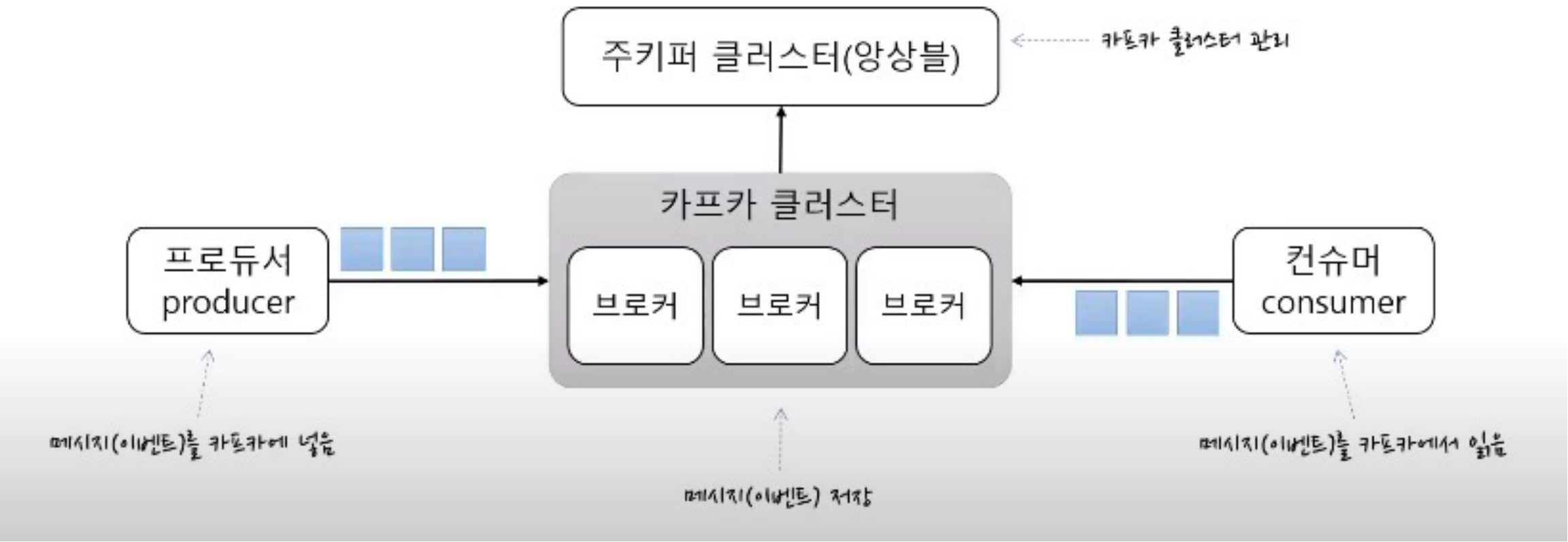


Kafka의 분산시스템

- 높은 성능, 처리량
- 안전성, Fault Tolerance
- 쉬운 시스템 변경

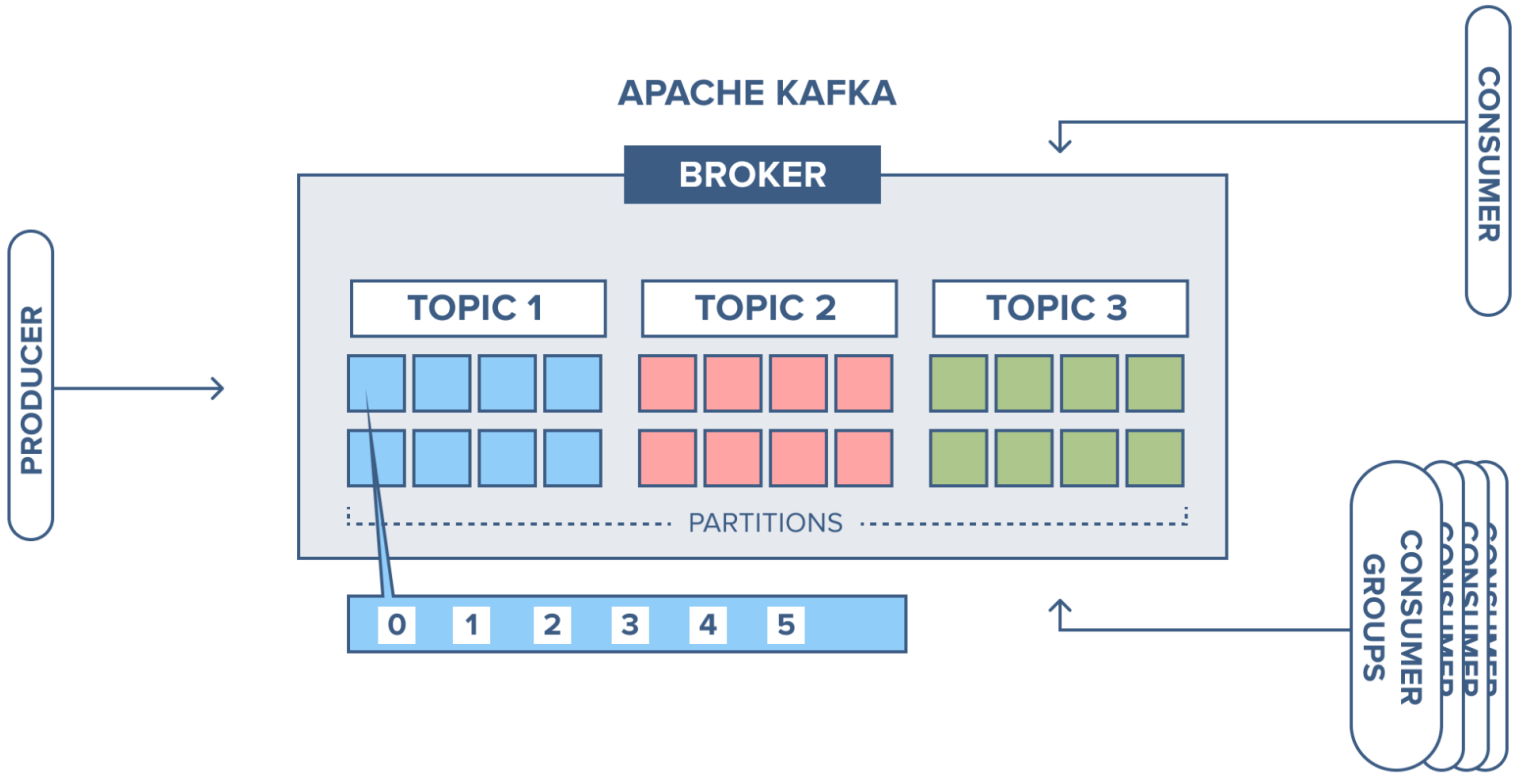


Kafka의 분산시스템



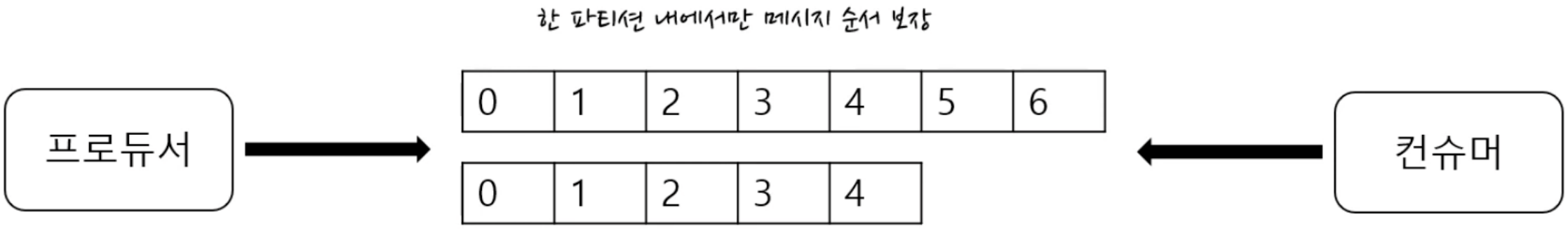
토픽과 파티션

- 토픽은 메시지를 구분하는 단위: 파일시스템의 폴더, DB 테이블과 유사
- 한 개의 토픽은 한 개 이상의 파티션으로 구성
  - 파티션은 메시지를 저장하는 물리적인 파일



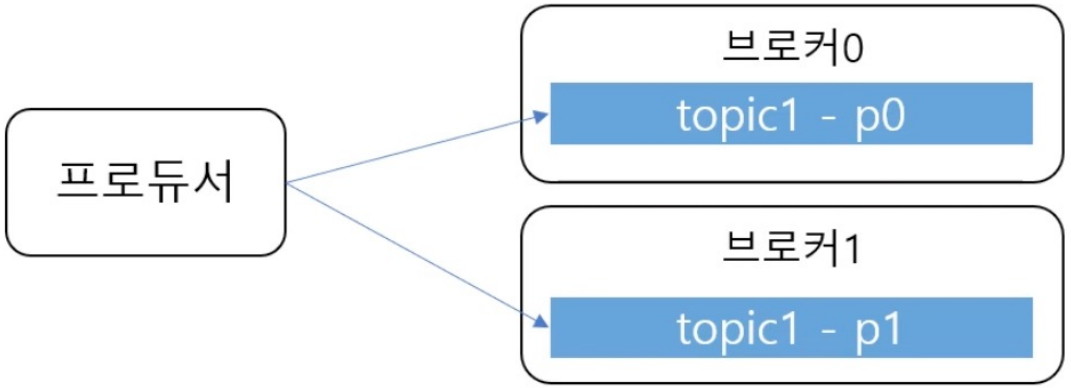
파티션과 오프셋, 메시지 순서

- 파티션은 추가만 가능한(append-only) 파일
  - 각 메시지 저장 위치를 오프셋(offset)이라고 함
  - 프로듀서가 넣은 메시지는 파티션의 맨 뒤에 추가
  - 컨슈머는 오프셋을 기준으로 메시지를 순서대로 읽음
  - 메시지는 삭제되지 않음 (설정에 따라 일정시간이 지난 뒤 삭제)



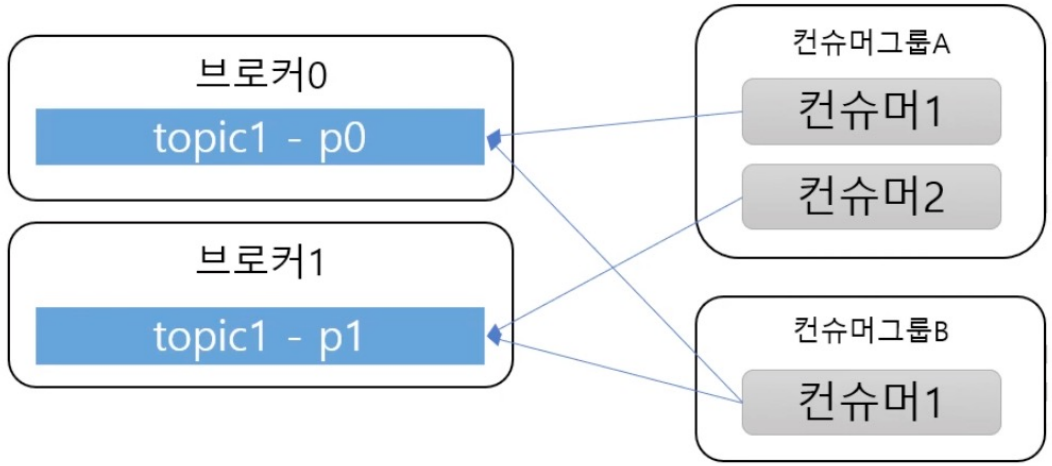
여러 파티션과 프로듀서

- 프로듀서는 라운드로빈 또는 키로 파티션 선택
  - 키 값이 null 이면 라운드로빈으로 파티션 선택
  - 같은 키를 갖는 메시지는 같은 파티션에 저장 -> 같은 키는 순서 유지



여러 파티션과 컨슈머

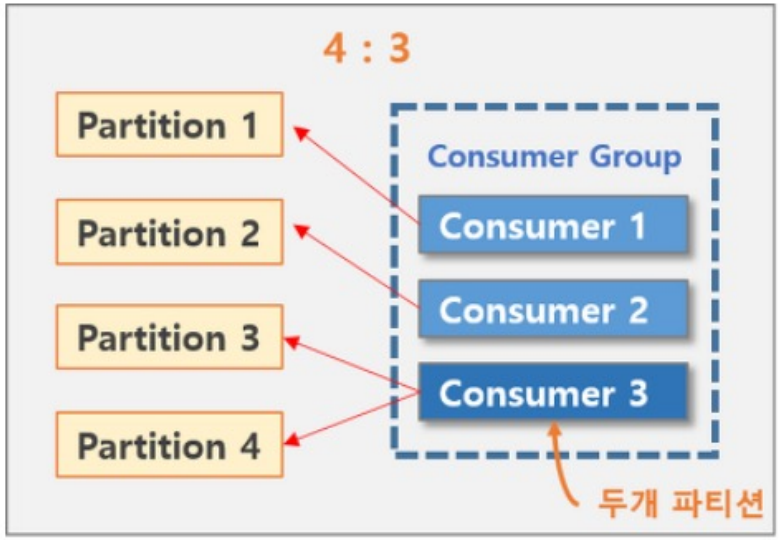
- 컨슈머는 컨슈머그룹에 속함
- 한 개 파티션은 컨슈머그룹의 한 개 컨슈머만 연결 가능
  - 즉 컨슈머그룹에 속한 컨슈머들은 한 파티션을 공유할 수 없음
  - 한 컨슈머그룹 기준으로 파티션의 메시지는 순서대로 처리

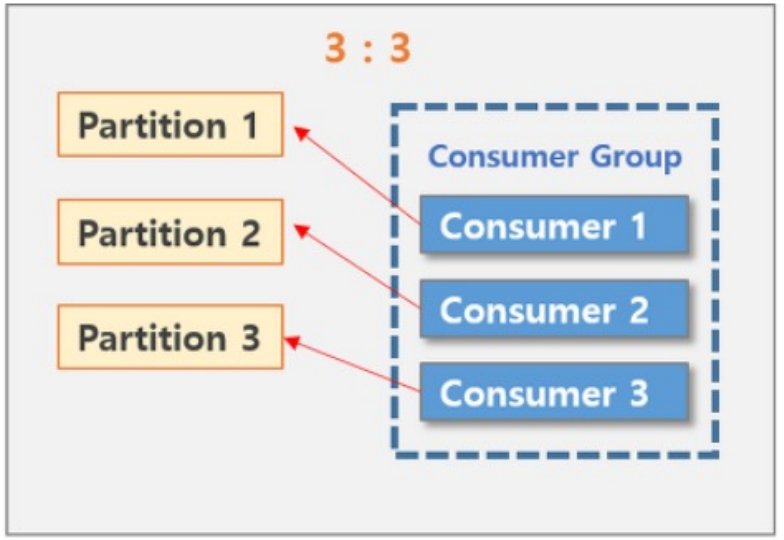


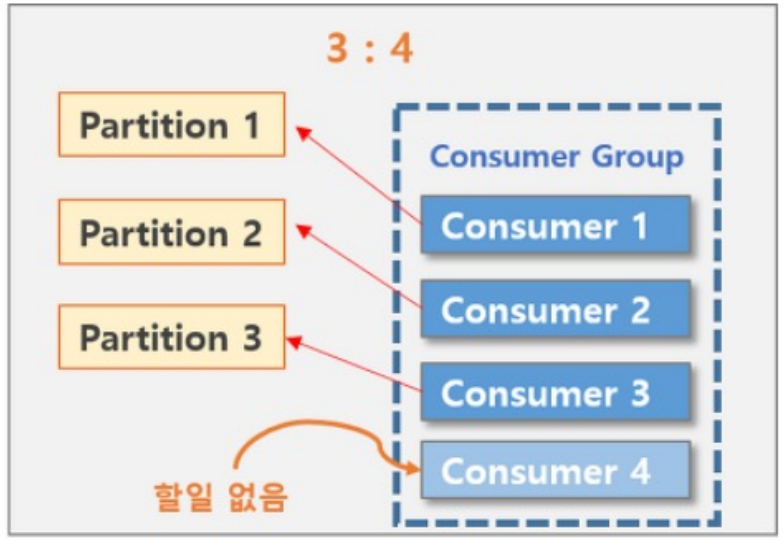
파티션은 많을 수록 좋은가?

1. 카프카는 모든 디렉토리의 파일들에 대한 핸들을 열기에 리소스 낭비 문제가 있음
2. 카프카의 replication(복제)은 파티션 기준으로 돌기에 파티션이 많을 경우 장애 복구 시간이 증가함

Consumer Group과 파티션 수의 관계

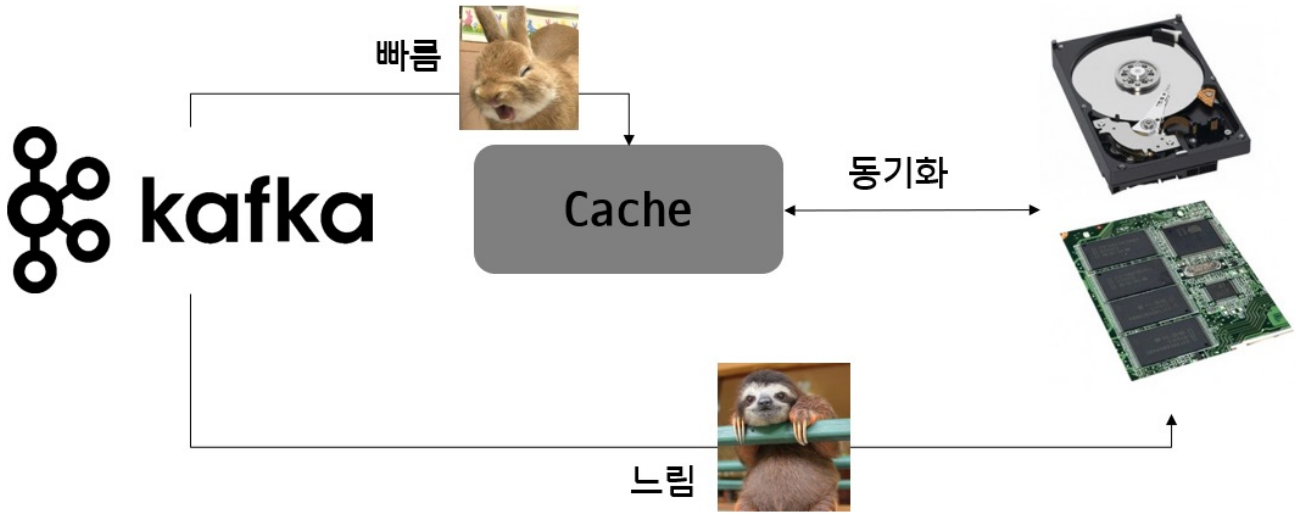






페이지 캐시와 zero copy

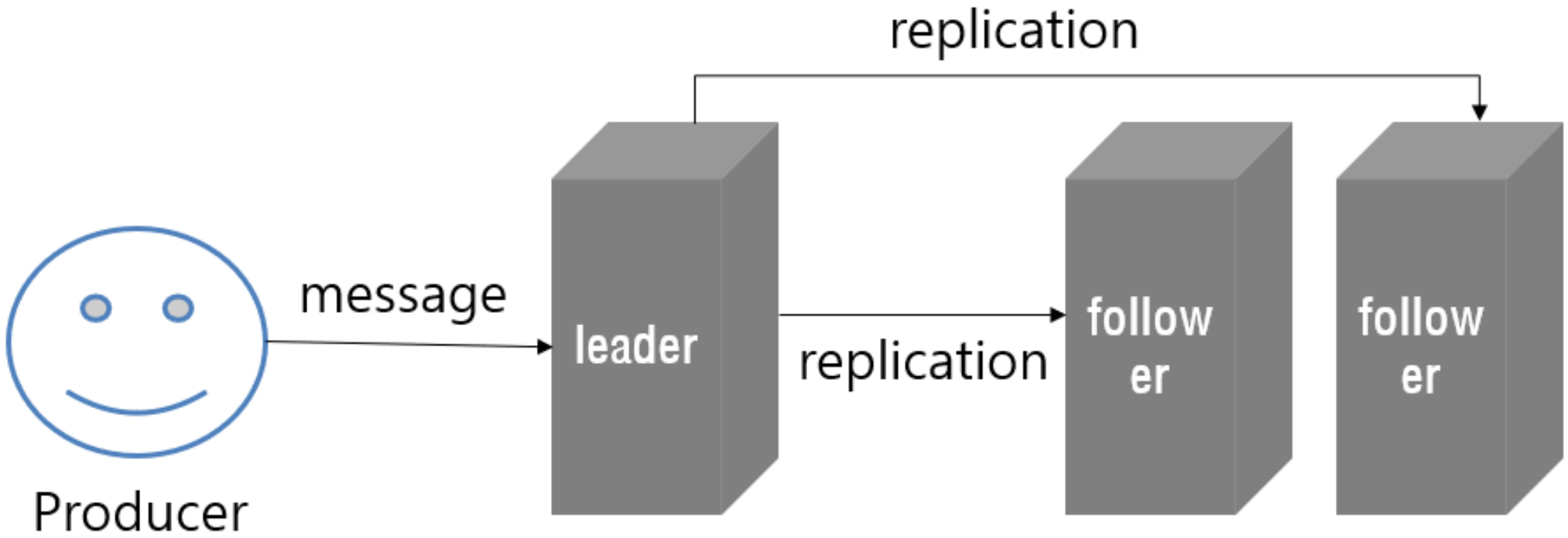
- 파티션 파일은 os 페이지캐시 사용
  - 파티션에 대한 파일 io를 메모리에서 처리
  - 서버에서 페이지캐시를 카프카만 사용해야 성능에 유리
- Zero Copy
  - 디스크 버퍼에서 네트워크 버퍼로 직접 데이터 복사



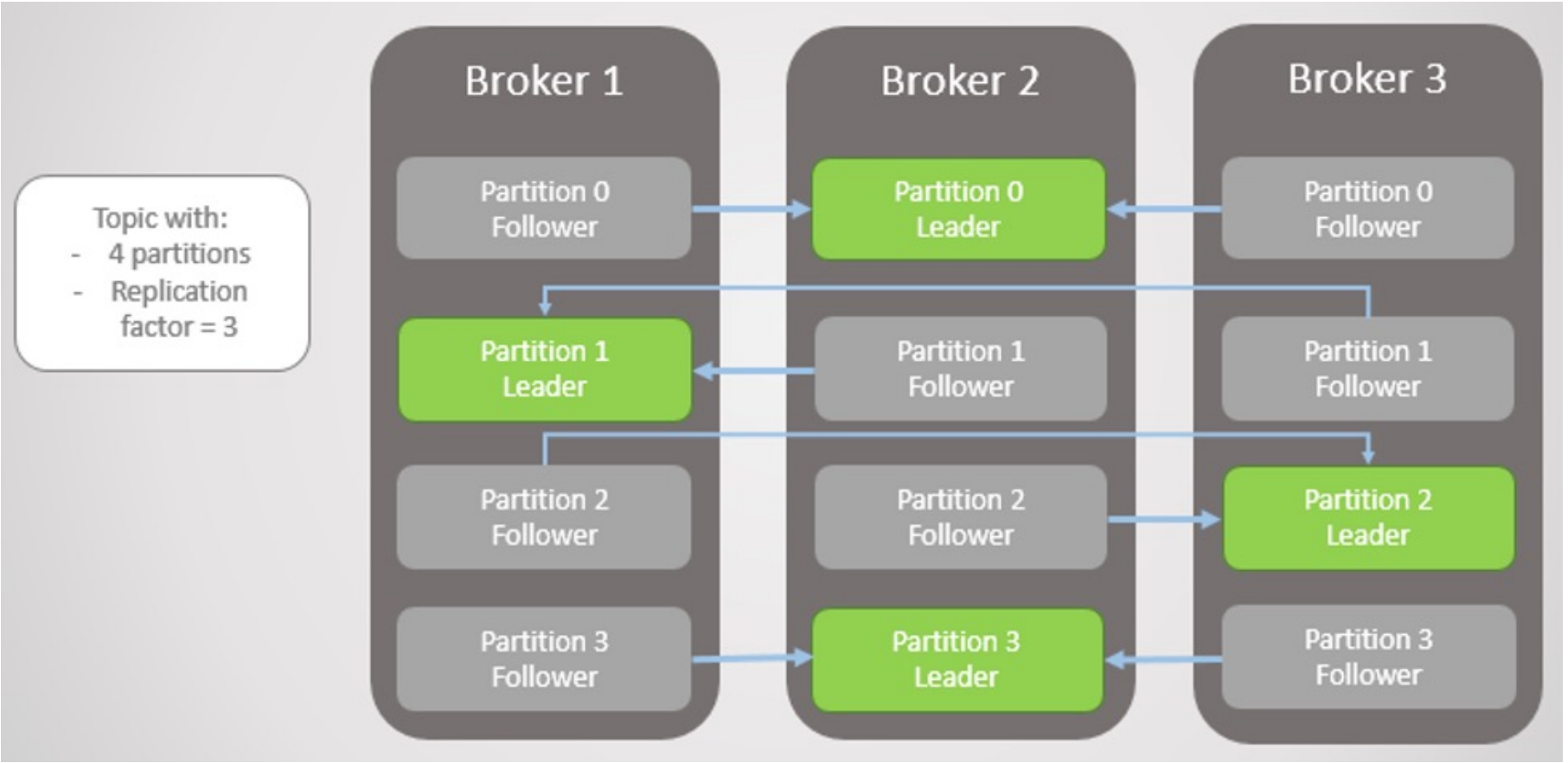
배치 처리

- 묶어서 보내기, 묶어서 받기 (batch)
- 낱개 처리보다 처리량 증가

Replication 팔로워 리더

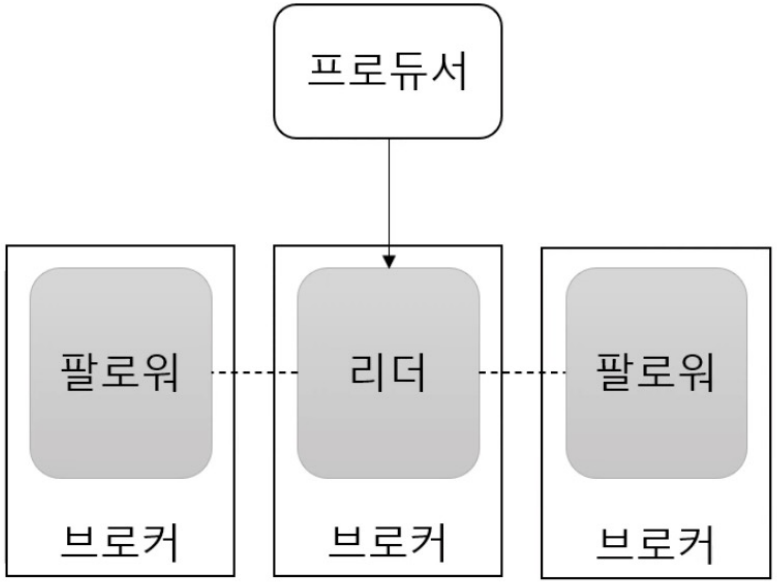


Replication 팔로워 리더



전송보장과 ack

- ack = 0
  - 서버 응답을 기다리지 않음
  - 전송 보장 x
- ack = 1
  - 파티션의 리더에 저장되면 응답 받음
  - 리더 장애 시 메시지 유실 가능
- ack = all (또는 -1)
  - 모든 레플리카에 저장되면 응답 받음
    - 브로커 min.insync.replicas 설정에 따라 달라짐



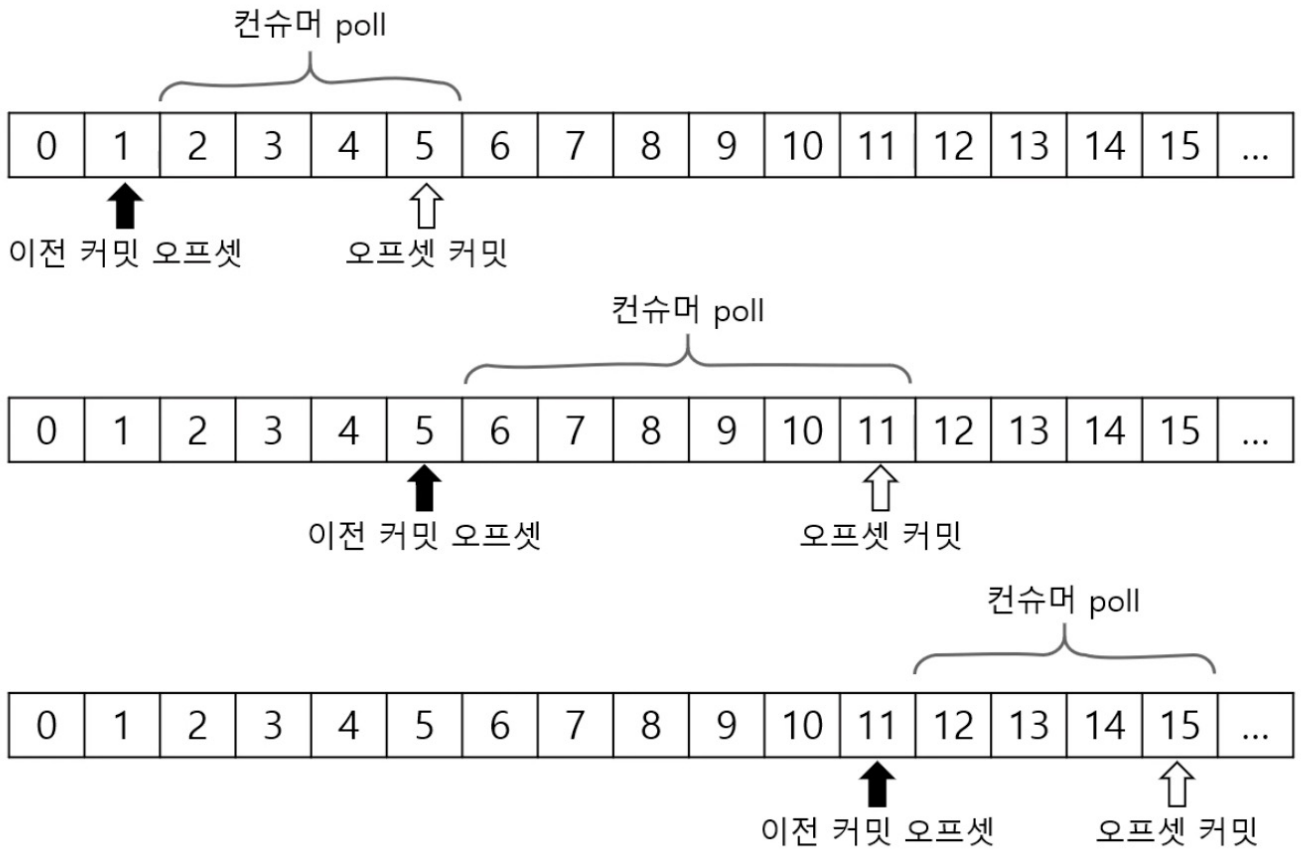
ack 와 min.insync.replicas

- min.insync.replicas (브로커 옵션)
  - 프로듀서 ack 옵션이 all일 때 저장에 성공했다고 응답할 수 있는 동기화된 최소 레플리카 개수
- 예1
  - 레플리카 개수 3, ack = all, min.insync.replicas = 2
  - 리더에 저장하고, 팔로워 중 한 개에 저장하면 성공 응답

ack 와 min.insync.replicas

- 예2
  - 레플리카 개수 3, ack = all, min.insync.replicas = 1
  - 리더에 저장되면 성공 응답
  - ack = 1과 동일 → 리더 장애 시 메시지 유실 가능
- 예3
  - 레플리카 개수 3, ack = all, min.insync.replicas = 3
  - 리더와 팔로워 2개에 저장되면 성공 응답
  - 팔로워 중 한 개라도 장애가 나면 레플리카 부족으로 저장에 실패

커밋과 오프셋


컨슈머 poll
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
이전 커밋 오프셋
오프셋 커밋
컨슈머 poll
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
이전 커밋 오프셋
오프셋 커밋
컨슈머 poll
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...
이전 커밋 오프셋
오프셋 커밋

재처리와 순서

- 동일 메시지 조회 가능성
  - 일시적 커밋 실패, 컨슈머 리밸런스 등에 의해 발생
- 컨슈머는 멱등성(idempotence)를 고려해야 함
- 데이터 특성에 따라 타임스탬프, 일련번호 등을 사용

세션 타임아웃, 하트비트, 최대 poll 간격

- 컨슈머는 하트비트를 전송해서 연결 유지
  - 브로커는 일정 시간 컨슈머로부터 하트비트가 없으면 컨슈머를 그룹에서 빼고 리밸런스 진행
- max.poll.intervals.ms : poll() 메서드의 최대 호출 간격
  - 이 시간이 지나도록 poll() 하지 않으면 컨슈머를 그룹에서 빼고 리밸런스 진행

Reference