



Kafka Streams 이해



이벤트(event)는 무엇인가?



컴퓨팅에서 이벤트(event)란 프로그램에 의해 감지되고
처리될 수 있는 동작이나 사건을 말한다.

- wikipedia

‘이벤트’는 세상 어디에서나 존재한다



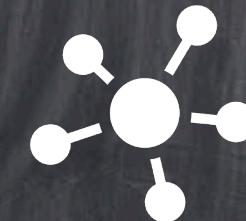
오픈마켓

- ✓ 주문
- ✓ 반품
- ✓ 배송



금융

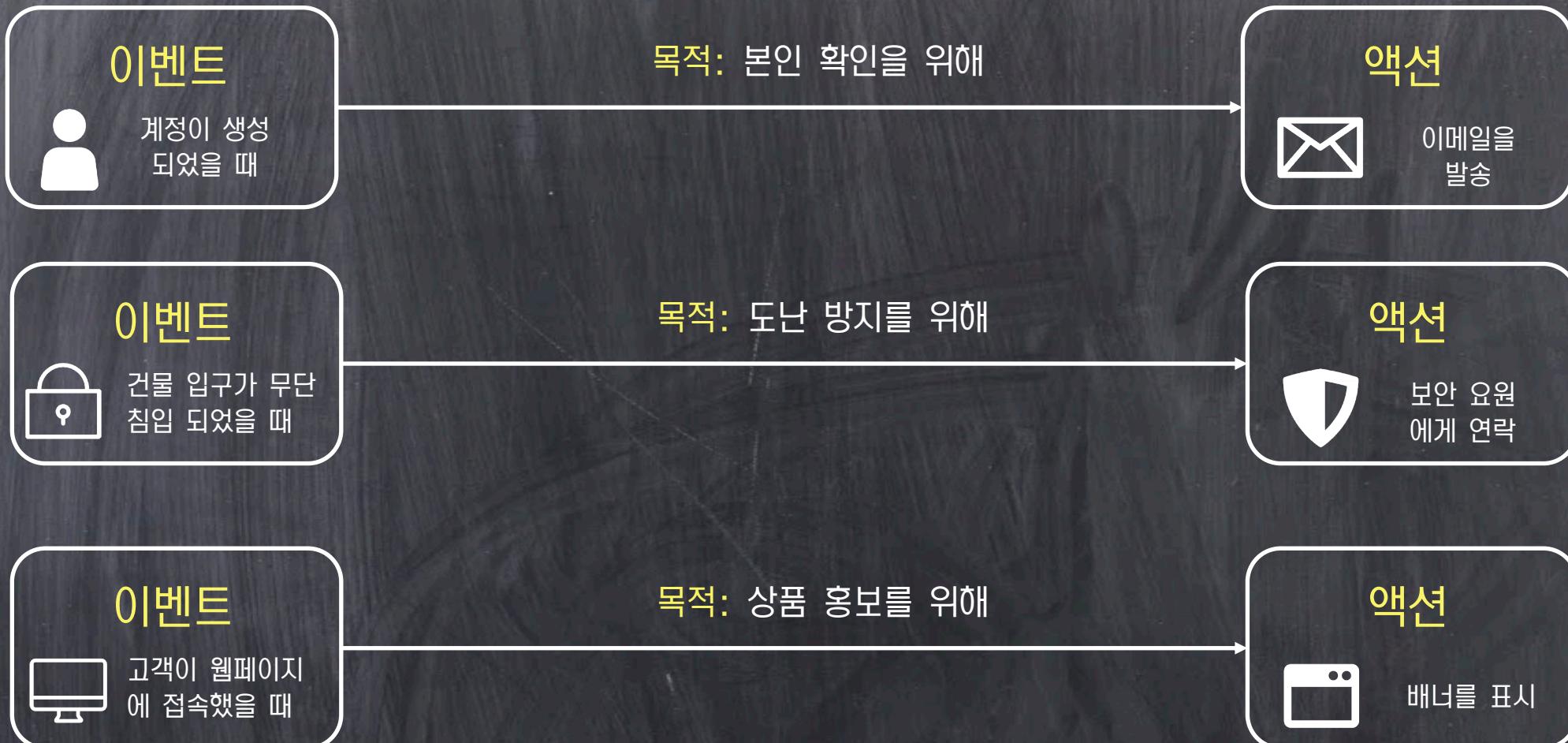
- ✓ 이체
- ✓ 출금
- ✓ 주식



포털

- ✓ 클릭 이벤트
- ✓ 광고 노출
- ✓ 광고 클릭

이벤트는 어떤 목적을 위해서 추가 액션을 발생시킨다.



그러면 이벤트를 어떻게 적용할 것인가?



이벤트 처리 단계는
아래와 같이 3가지로 나눌 수 있다

①



Modeling
Event

②



Transporting
Event

③



Processing
Event

① Modeling Events

```
{  
    "invoiceNumber" : 123456,  
    "createdTime": 1111111111,  
    "storeId": "S123",  
    "posId": "POS354",  
    "cashierId": "CAS123",  
    "customerCardNo": "3000124223",  
    "deliveryType": "home"  
}
```



Pos operator



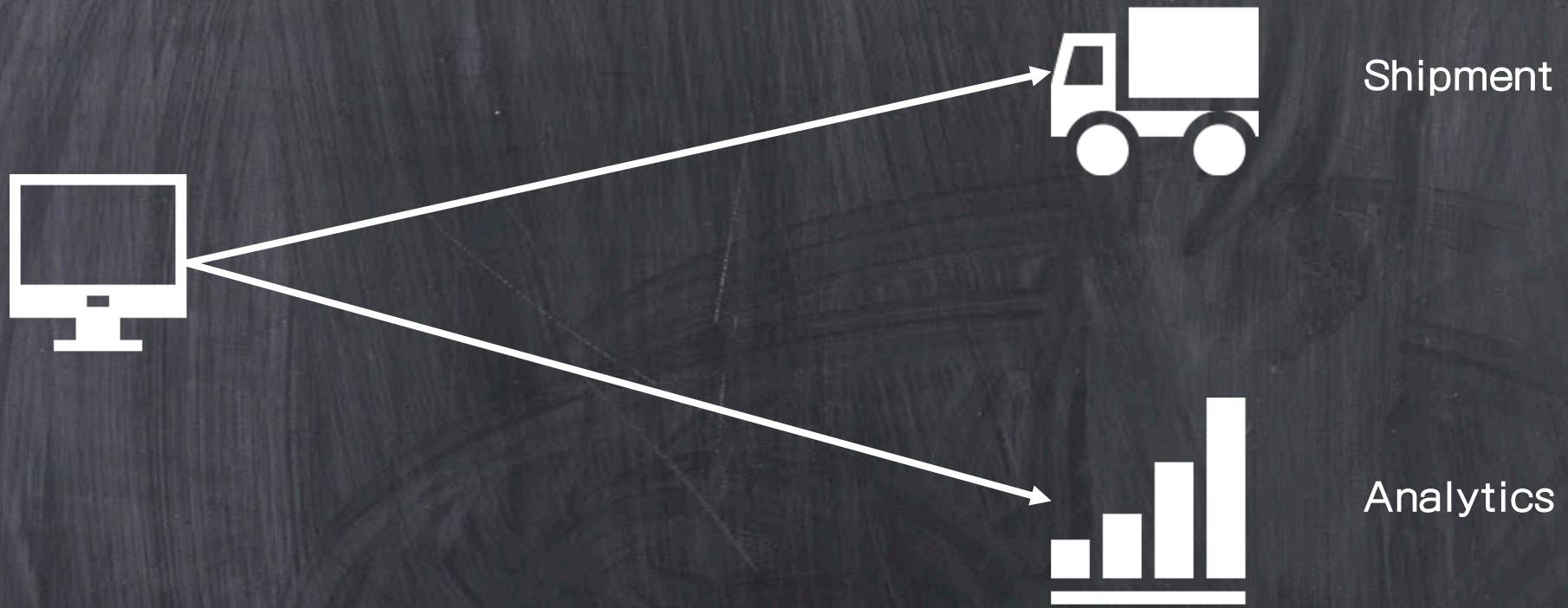
Create



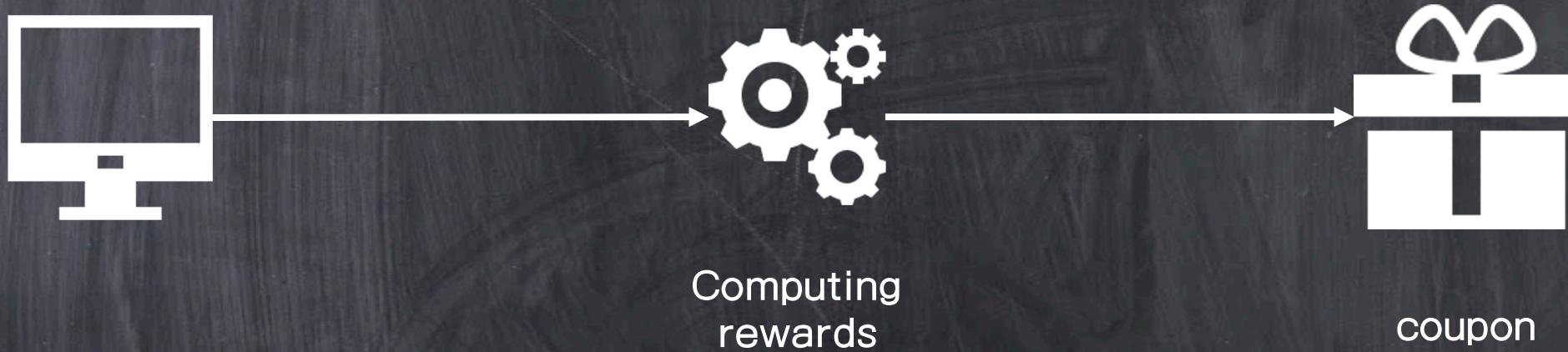
Invoice

②

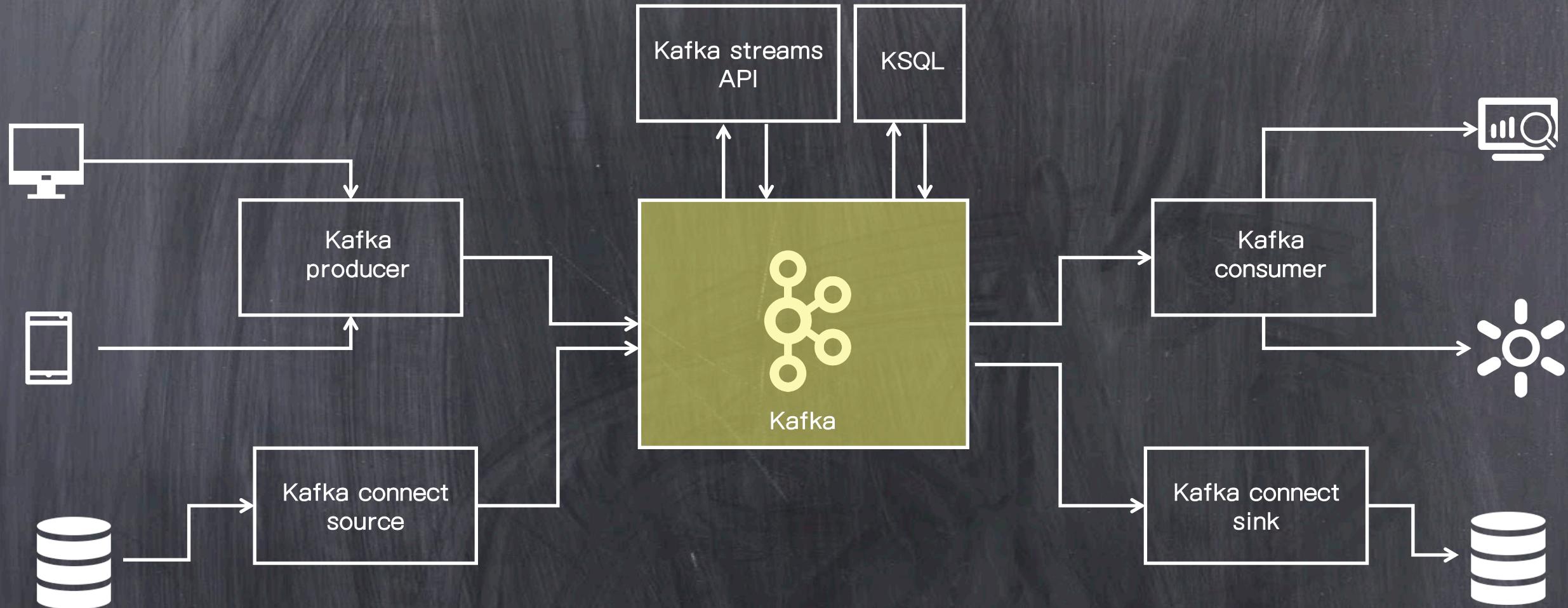
Transporting Events



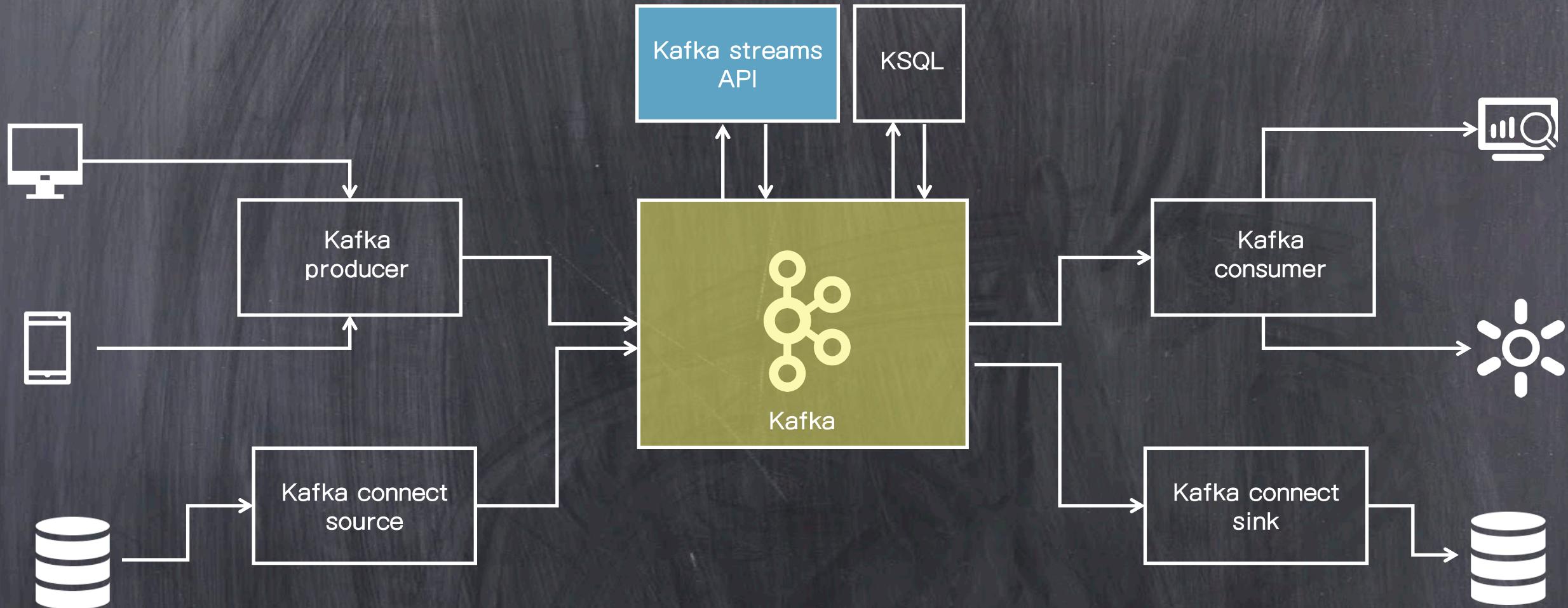
③ Processing Events



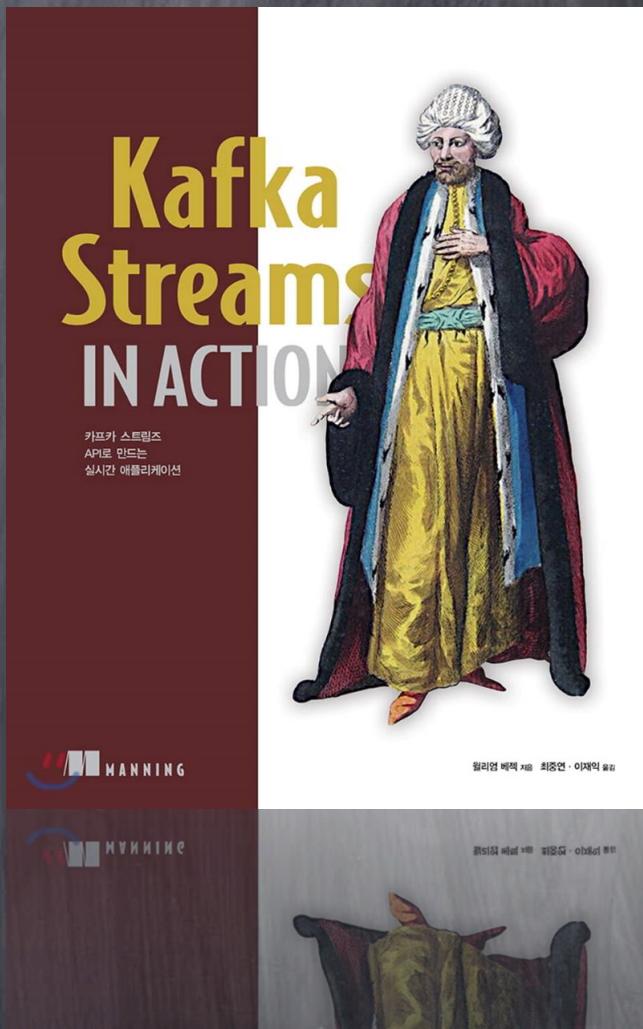
Kafka ecosystem은 아래와 같이 구성되어 있다.



여기에서는 Kafka Streams에 대해서 주로 다룬다.



또한 이 문서는
아래 도서를 기반으로 작성되었다.



Kafka Streams in Action 카프카 스트리밍 API로 만드는 실시간 애플리케이션
윌리엄 베젝 저/최충연, 이재익 역 | 에이콘출판사 | 2019년 07월 12일

이 책은 총 4부로 구성되어 있다.

1부 Kafka Streams 시작하기

- 1장 Kafka Streams에 오신 것을 환영합니다
- 2장 빠르게 살펴보는 Kafka

2부 Kafka Streams 개발

- 3장 Kafka Streams 개발
- 4장 스트림과 상태
- 5장 KTable API
- 6장 프로세서 API

3부 Kafka Streams 관리

- 7장 모니터링과 성능
- 8장 Kafka Streams 애플리케이션 테스트

4부 Kafka Streams 고급 개념

- 9장 Kafka Streams 고급 애플리케이션

Kafka의 기본지식과
Kafka Streams 개발을 위한
내용 위주로 정리하였다.



1

Part

Kafka Streams 시작하기

1. 카프카 스트림즈에 오신 것을 환영합니다
2. 빠르게 살펴보는 카프카

1. 카프카 스트림즈에 오신 것을 환영합니다

빅 데이터로의 전환, 그로 인한 프로그래밍 환경의 변화

- ✓ 최근 프로그래밍 환경은 기술과 함께 폭발적으로 증가했다. 특히 모바일 애플리케이션
- ✓ 매일 점점 더 많은 데이터를 처리해야 한다.
- ✓ 대량의 데이터를 벌크 처리(일괄 처리) 할 수 있는 능력으로 충분하지 않다.
- ✓ 많은 조직에서 실시간(스트림 처리)으로 처리할 필요성을 발견하고 있다.



데이터 증가



모바일 사용량



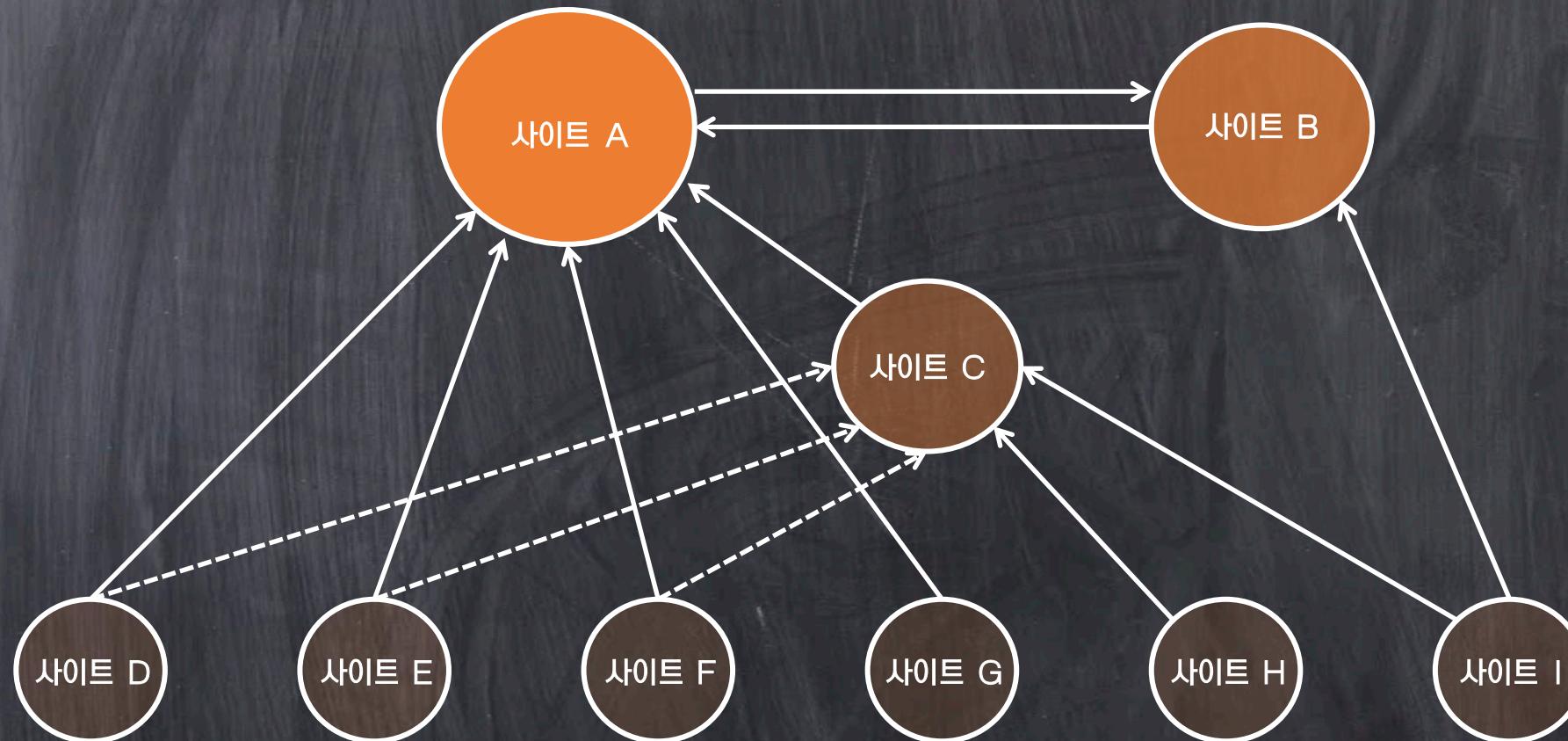
실시간 요구



분석 및 활용
필요성

빅 데이터의 기원

- ✓ 1998년 구글이 만든 때가 빅 데이터 시작이라고 할 수 있다.
- ✓ 검색을 위해 웹 페이지 순위를 매기는 새로운 방법인 페이지랭크 알고리즘을 개발했다.
- ✓ 웹 페이지가 더 중요하거나 관련성이 높을수록 더 많은 사이트에서 이를 참조한다는 가정이다.
- ✓ 사이트 중요성: 사이트 A > 사이트 B > 사이트 C > ...



맵리듀스의 중요 개념

- ✓ 맵리듀스의 핵심은 함수형 프로그래밍에 있다.
- ✓ 맵 함수는 원본값을 변경하지 않고 입력을 가져와 어떤 다른 값으로 매팅한다.

첫 번째 숫자에 시드값을 더한다.
 $0 + 1 = 1$

첫 번째 단계의 결과를 가져와 목록의 두 번째 숫자를 더한다.
 $1 + 2 = 3$

두 번째 단계의 합을 세 번째 숫자에 더한다.
 $3 + 3 = 6$

자바 8 람다를 사용하는 간단한 리듀스 함수

```
List<Integer> numbers = Arrays.asList(1, 2, 3);
int sum = numbers.stream().reduce(0, (i, j) -> i + j);
```

배치 처리로는 충분하지 않다

- ✓ 하둡/맵리듀스는 대량의 데이터를 수집하고 처리한 다음 나중에 사용할 수 있도록 출력을 저장하는 배치 프로세서다.
- ✓ 사용자 클릭을 실시간으로 보고 전체 인터넷에서 어떤 것이 가치 있는 것인지 결정할 수 없기 때문에 페이지랭크 같은 배치처리가 맞다.

지금 유행이 무엇인가?



지난 10분 동안 잘못된
로그인 시도가 얼마나 있었는가?



최근 출시된 기능은
어떻게 활용되고 있는가?



→ 사용자의 반응에 신속하게 대응할 필요성이 대두됐다. 그 솔루션은 스트림으로 나타났다

스트림 처리 소개

스트림 처리를 데이터가 시스템에 도착하는 대로 처리하는 것으로 정의한다.

즉 데이터를 수집하거나 저장할 필요없이 무한한 데이터 스트림을 유입하는 대로 연속으로 계산하는 능력



각 원은 특정 시점에 발생하는 정보나 이벤트를 나타낸다.

이벤트의 수는 제한이 없으며, 계속 왼쪽에서 오른쪽으로 이동한다.

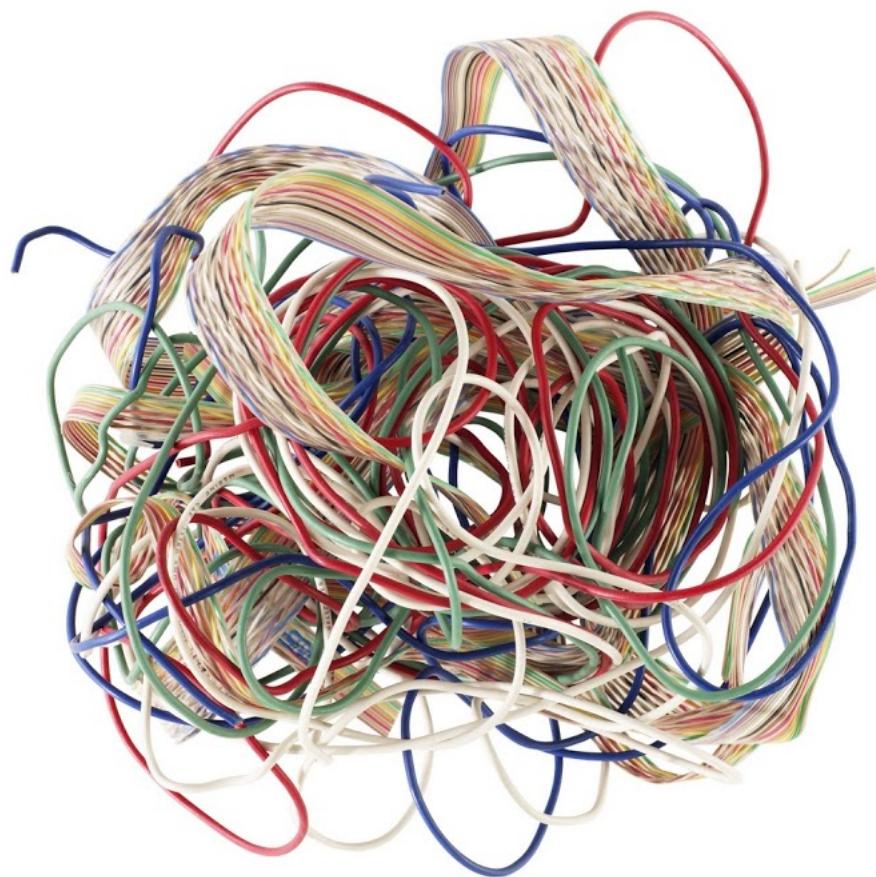
실시간 스트림을 설계할 때
고려해야 할 사항은?



1 Time Sensitivity

“시스템 간에 수초 내에
전달이 가능해야 해”



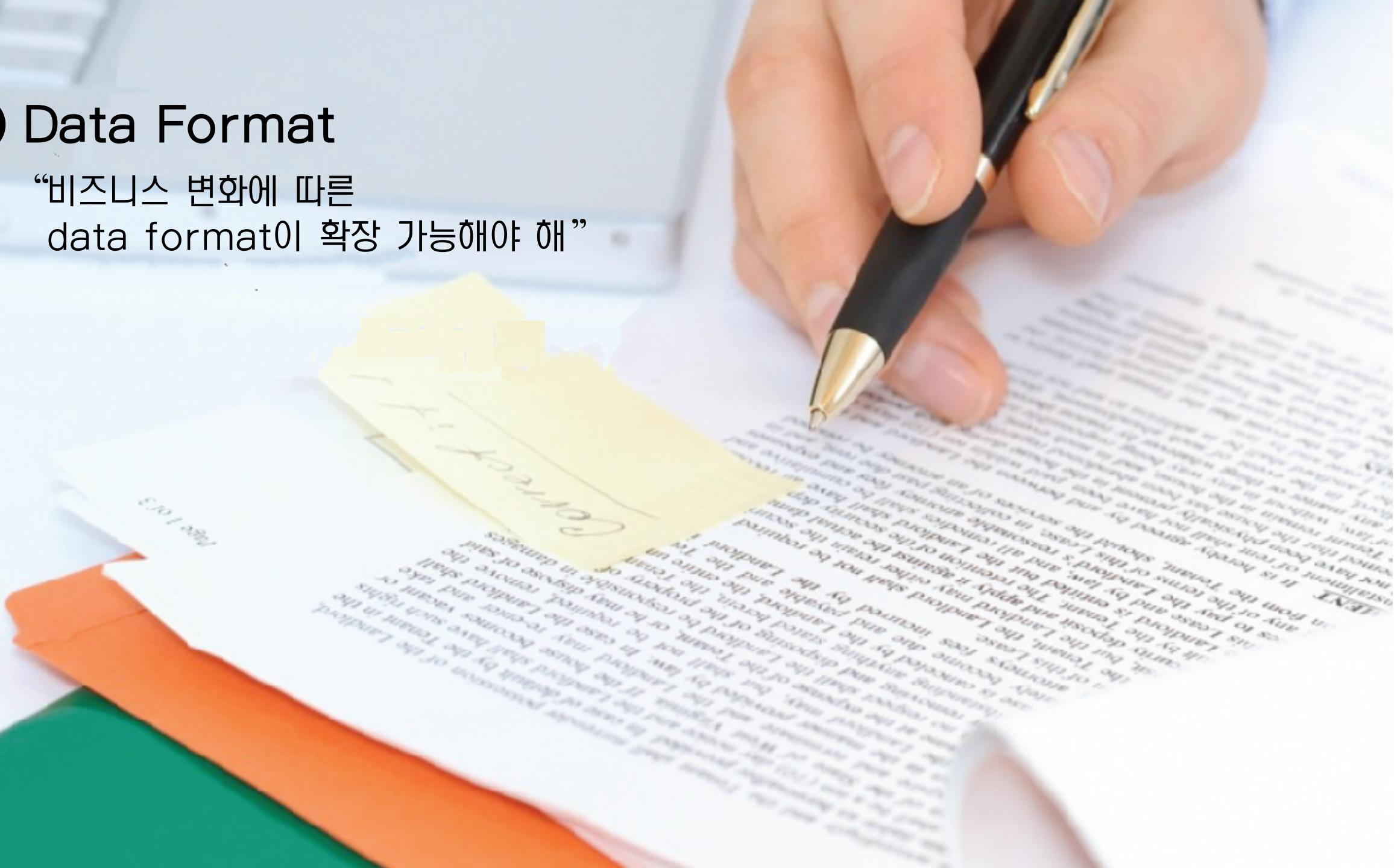


② Decoupling

“시스템의 변화에
영향이 적어야 해”

3 Data Format

“비즈니스 변화에 따른
data format이 확장 가능해야 해”



4 Reliability

“시스템 장애로 인해
데이터 손실이 있으면 안돼”



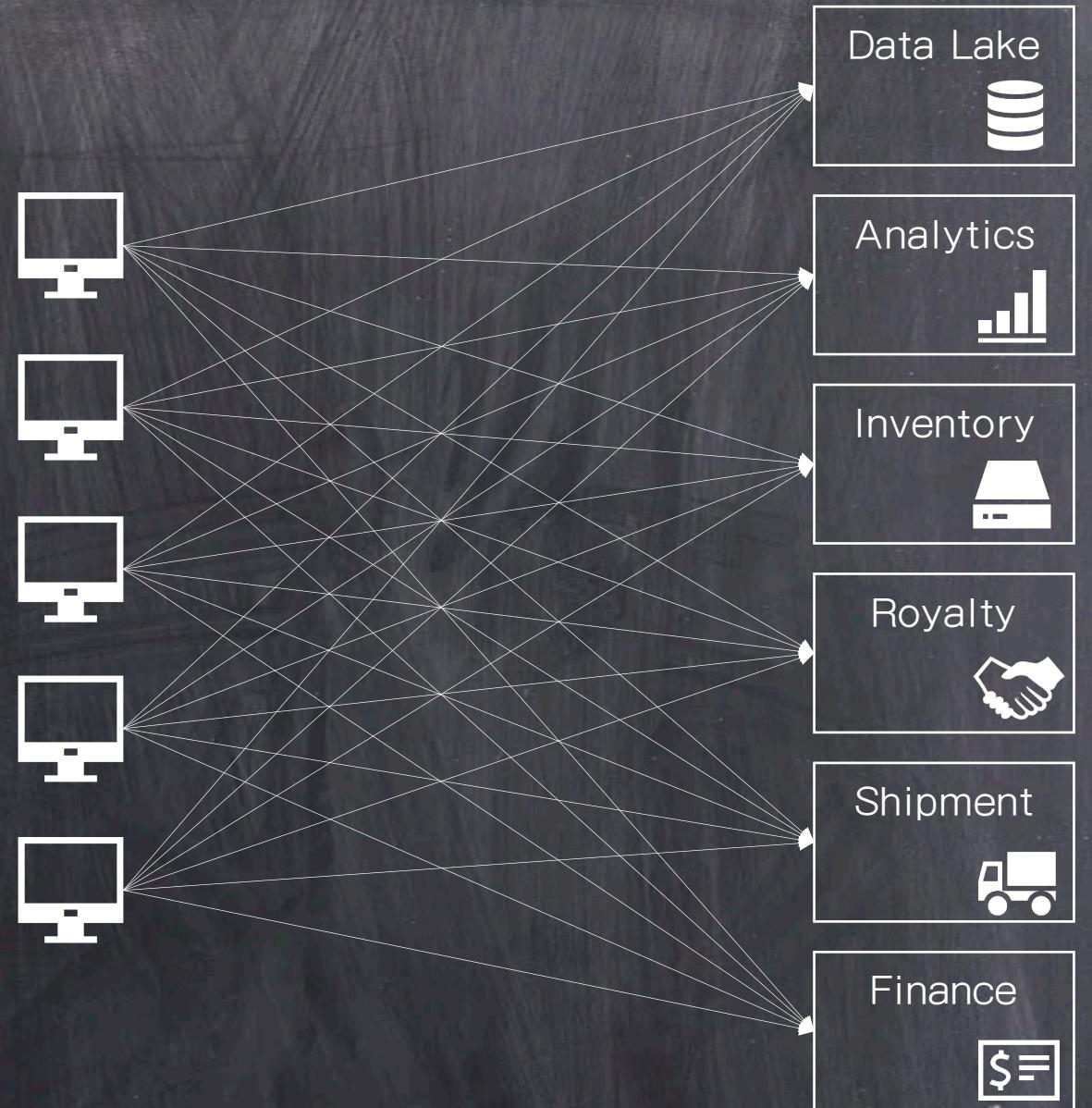
5 Scalability

“비즈니스 증가에 따른
수평 확장이 가능해야 해”



그럼 어떤 방식을 사용해야 할까?

- ① Shared Database
- ② RPC/RMI
- ③ File Transfer
- ④ Messaging



① Shared Database

1. Time sensitivity
2. Decoupling
3. Data format evolution
4. Reliability
5. Scalability

Event Producer



Event Consumer

Data format evolution, Reliability, Scalability는 지원하기 어렵다.

② RMI & RPC

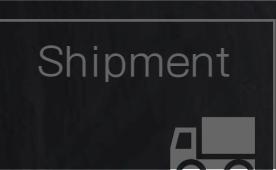
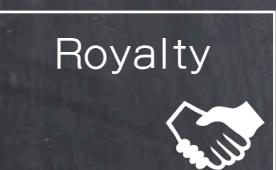
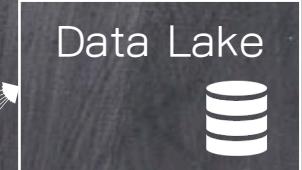
1. 카프카 스트림즈에 오신 것을 환영합니다

Event Consumer

1. Time sensitivity
2. Decoupling
3. Data format evolution
4. Reliability
5. Scalability

시스템간 강한 결합을 만들어내고 확장하기가 어렵다.

Event Producer

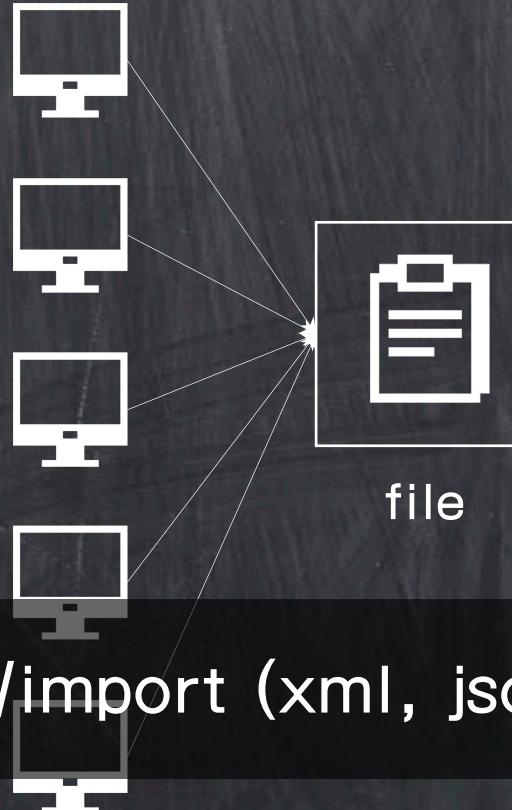


③ File Transfer

Event Consumer

1. Time sensitivity
2. Decoupling
3. Data format evolution
4. Reliability
5. Scalability

Event Producer



실시간 요구사항을 만족하지 못한다. export/import (xml, json, csv)

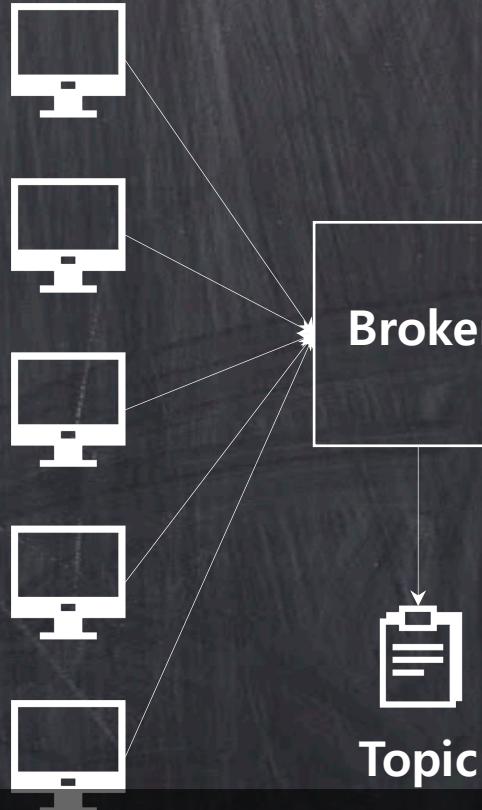
④ Messaging

1. Time sensitivity
2. Decoupling
3. Data format evolution
4. Reliability
5. Scalability

1. 카프카 스트리밍에 오신 것을 환영합니다

Event Consumer

Event Producer



5가지 항목을 모두 만족시킨다.

1. 카프카 스트림즈에 오신 것을 환영합니다

Broker로 Kafka를 사용한다.

Event Consumer

Event Producer



Kafka



Topic

Data Lake



Analytics



Inventory



Royalty



Shipment



Finance



스트림 처리를 사용해야 할 경우와 말아야 할 경우

스트림 처리도 모든 경우에 맞는 솔루션이 아니다.

들어오는 데이터에 신속하게 응답하거나 보고해야 하는 경우가 스트림 처리의 좋은 사례다.

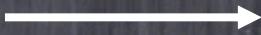
사용해야 할 경우

- ✓ 신용카드 사기
- ✓ 침입 탐지
- ✓ 뉴욕시 마라톤과 같은 대규모 경주
- ✓ 소비자가 언제 구매할 것인가에 대한 결정

사용하지 말아야 할 경우

- ✓ 경제 예측
- ✓ 학교 교과 과정 변경

구매 거래 처리 예제 (G-마트)



윤아는 퇴근길에
치약이 필요

G-마트를 들러 치약을
집어 계산대로 지불하려
간다

직불카드로
계산한다

가게에서 나오자 다음
방문 시 사용할 수 있는
할인 쿠폰이 발행

여러분이 G-마트 수석 개발자라면 어떻게 설계할 것인가?

- ✓ G-마트의 매출은 연간 10억달러 규모이다.
- ✓ 각 트랜잭션 데이터부터 도출된 사업적 기회를 창출해야 한다.
- ✓ 데이터를 수집 후 몇시간 기다려야 할 필요가 없다.

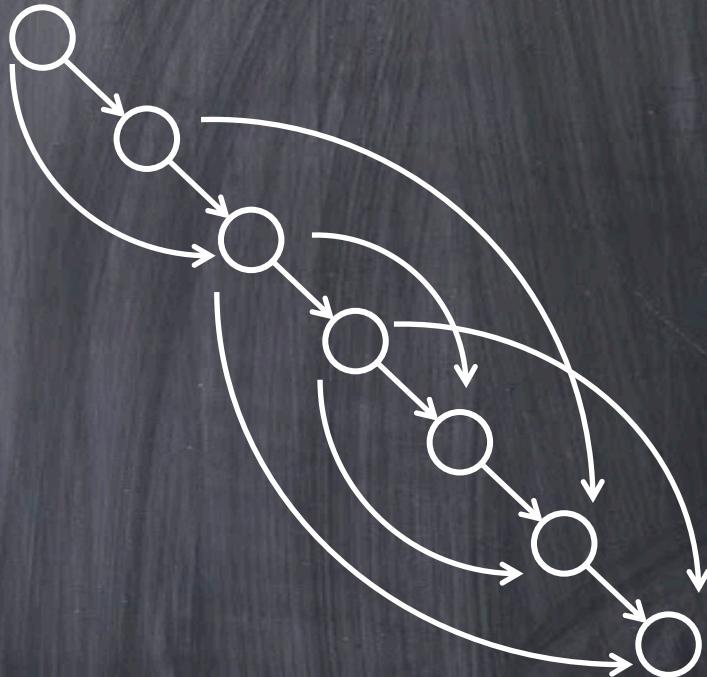


4가지 요구사항을 만족해야 한다.

- ① **프라이버시** 신용카드 정보 마스킹
- ② **고객 보상** 특정 품목의 지출에 보너스 포인트. 고객에게 알려주는 것
- ③ **판매 데이터** 특정 지역에서 어떤 항목이 더 인기가 있는지 파악
- ④ **스토리지** 모든 구매 기록이 스토리지에 저장

요구사항을 그래프로 분해

이전 요구사항을 방향성 비순환 그래프(DAG)로 빠르게 재구성할 수 있다.



방향성 비순환 그래프(DAG; Directed Acyclic Graph)란
개별 요소들이 특정한 방향을 향하고 있으며,
서로 순환하지 않는 구조로 짜여진 그래프를 말한다. - wikipedia

G-마트 요구사항을 DAG으로 표현해보자



- I. 프라이버시 신용카드 정보 마스킹
- II. 고객 보상 특정 품목의 지출에 보너스 포인트. 고객에게 알려주는 것
- III. 판매 데이터 특정 지역에서 어떤 항목이 더 인기가 있는지 파악
- IV. 스토리지 모든 구매 기록이 스토리지에 저장

소스 노드

구매

소스노드는 모든 트랜잭션이
시작되는 지점이야

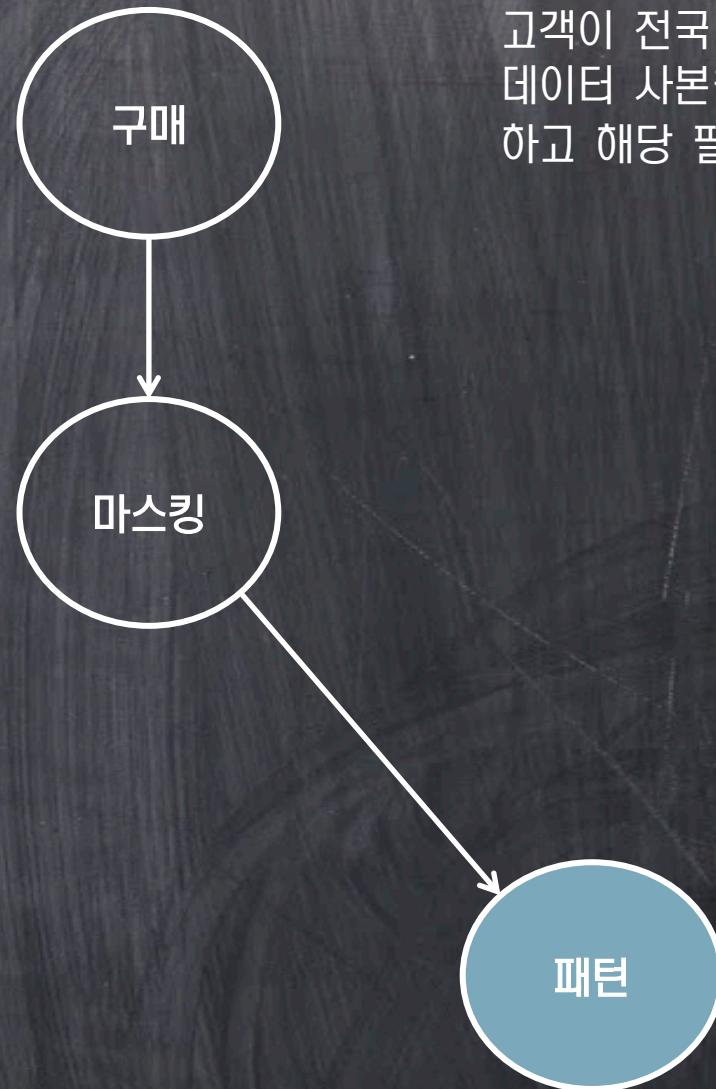
마스킹 노드



보안을 위해 신용카드 번호를
여기서 감춘다

신용카드 마지막 4자리를 제외한 신용카드 모든 자릿수를 x로 변환한다.
(xxxx-xxxx-xxxx-1122)

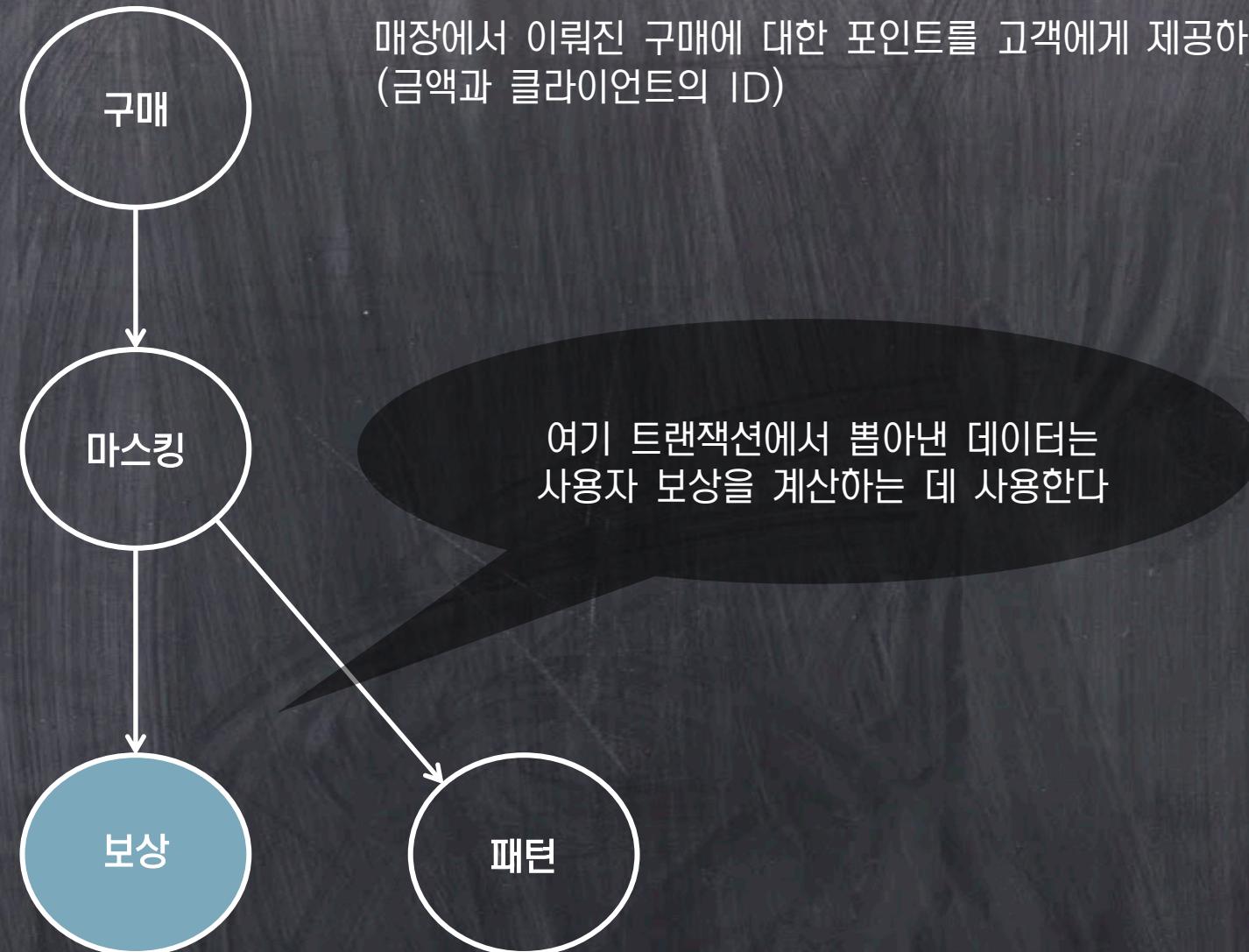
구매 패턴 노드



고객이 전국 어디에서 제품을 구매하는지 알아내기 위해 관련 정보를 추출한다. 데이터 사본을 만드는 대신 패턴 노드는 구매할 항목, 날짜 및 우편번호를 검색하고 해당 필드를 포함하는 새 객체를 만든다. (구매할 항목, 날짜 및 우편번호)

구매 패턴을 알아내기 위해
여기서 데이터를 추출한다

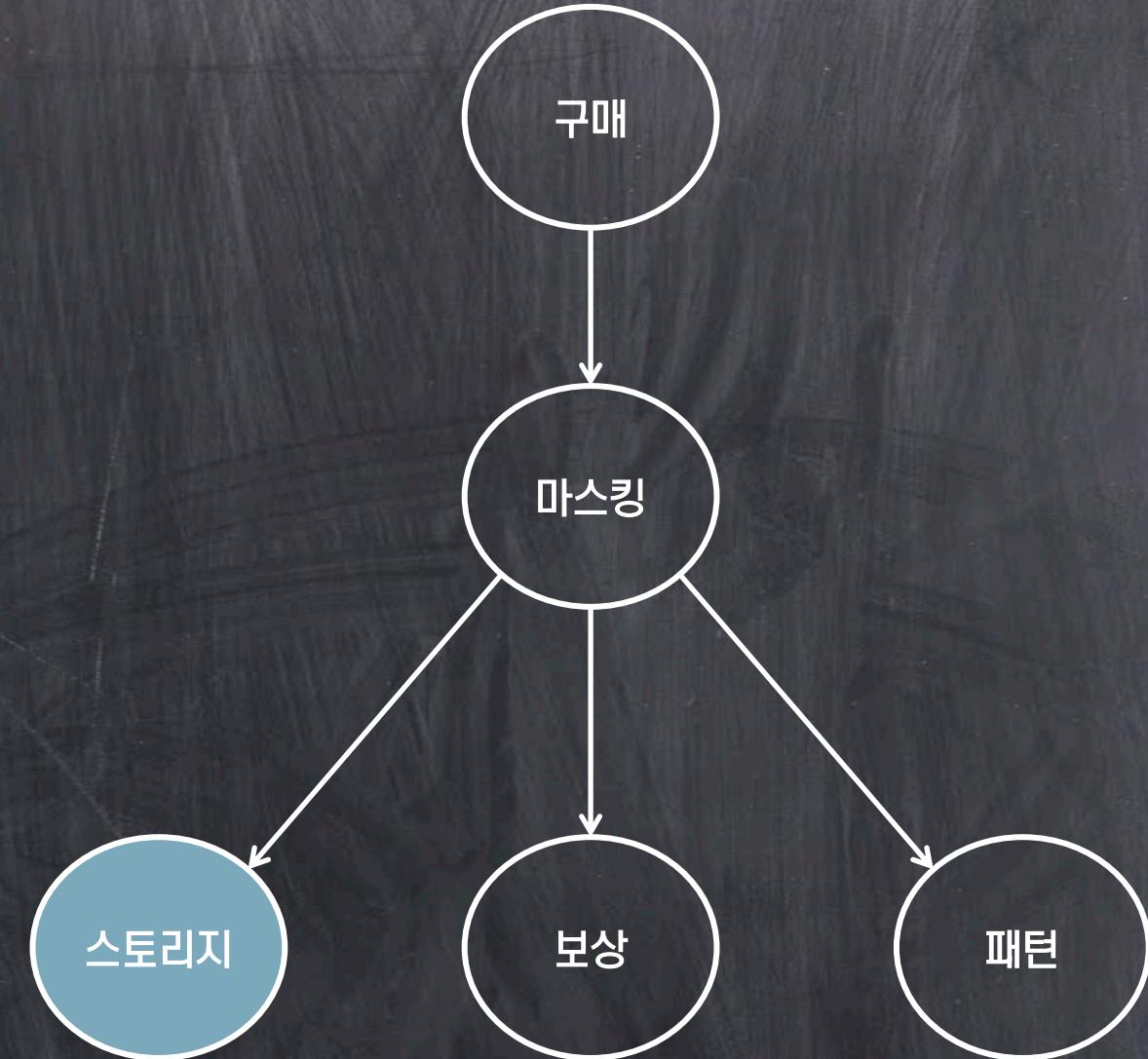
보상 노드



데이터 저장 노드

추가적인 분석에 사용하기 위해
구매를 여기 저장한다.

구매 데이터를 NoSQL 데이터 저장소에 기록한다.



2. 빠르게 살펴보는 카프카

빠르게 살펴보는 카프카

- ✓ Kafka Streams에도 Kafka에 대해서 설명하지 않고 Kafka Streams를 탐구하는 것은 불가능하다.
- ✓ 결국 Kafka Streams는 Kafka에서 실행하는 라이브러리다.



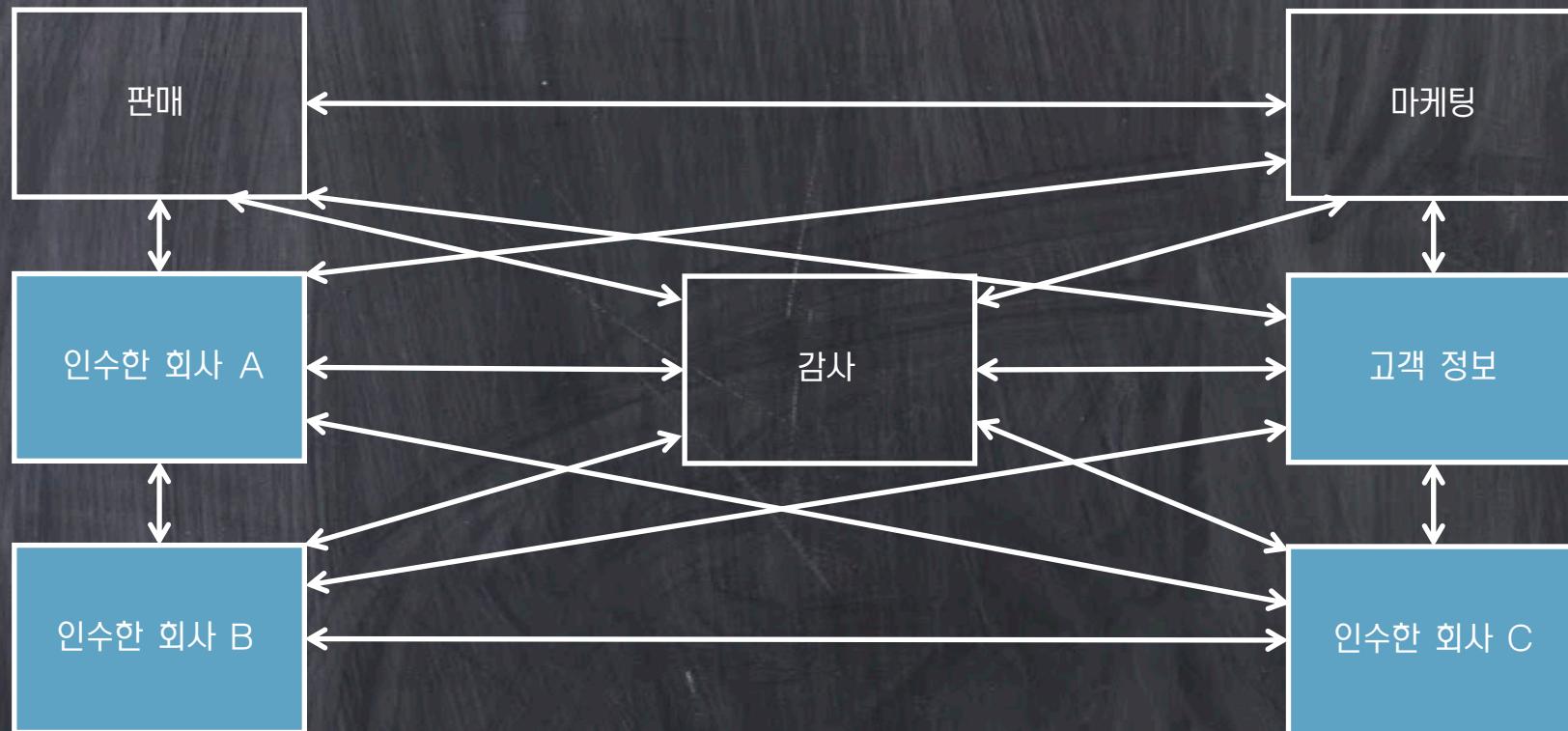
G-마트의 데이터 플랫폼

- ✓ 원래 G-마트는 소매 판매 데이터를 가진 소규모 회사였다.
- ✓ 한 부서의 판매 데이터는 해당 부서만의 관심사항이 아니다.
- ✓ 각 부서마다 중요한 내용과 원하는 데이터 구조가 다르다.



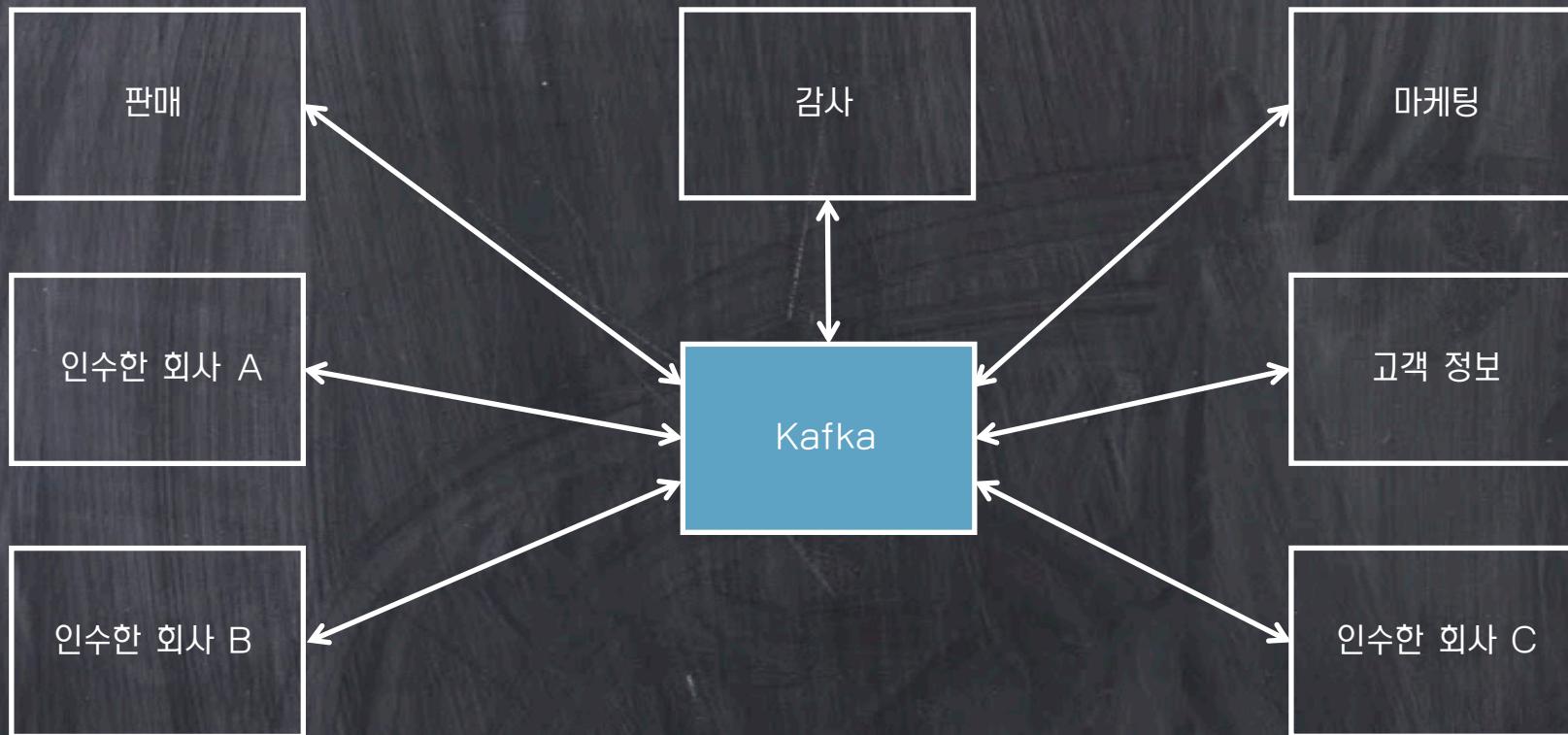
G-마트의 데이터 플랫폼

- ✓ 시간이 지나면서 G-마트는 다른 회사를 인수하고 기존 매장에서 상품을 늘려가면서 지속적으로 성장했다.
- ✓ 추가할 때마다 애플리케이션 간의 연결이 복잡해졌다.



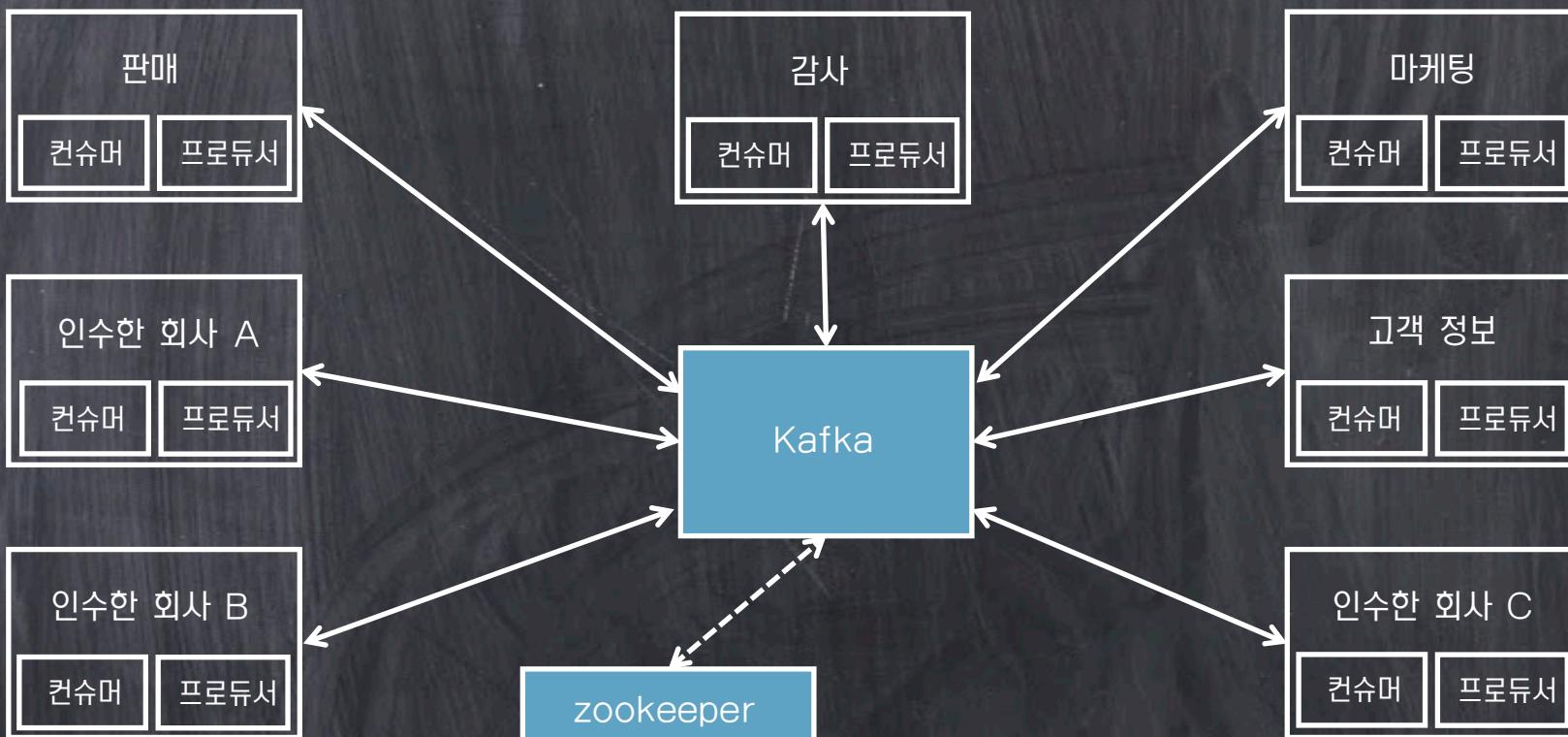
Kafka 판매 거래 데이터 허브

- ✓ G-마트 문제의 해결책은 모든 거래 데이터를 수용하는 하나의 유입 프로세스를 만드는 것이다.



Kafka는 메시지 브로커다

- ✓ 브로커는 상호 간 교환이나 거래를 위해 각자 반드시 알 필요가 없는 두 부분을 묶는 중개자다.
- ✓ 프로듀서와 컨슈머가 추가되어 서로 직접 통신을 하지 않는다.



주키퍼 노드 클러스터는 카프카와 통신해 토픽 정보를 유지하고
클러스터의 브로커를 추적한다.

카프카는 로그다

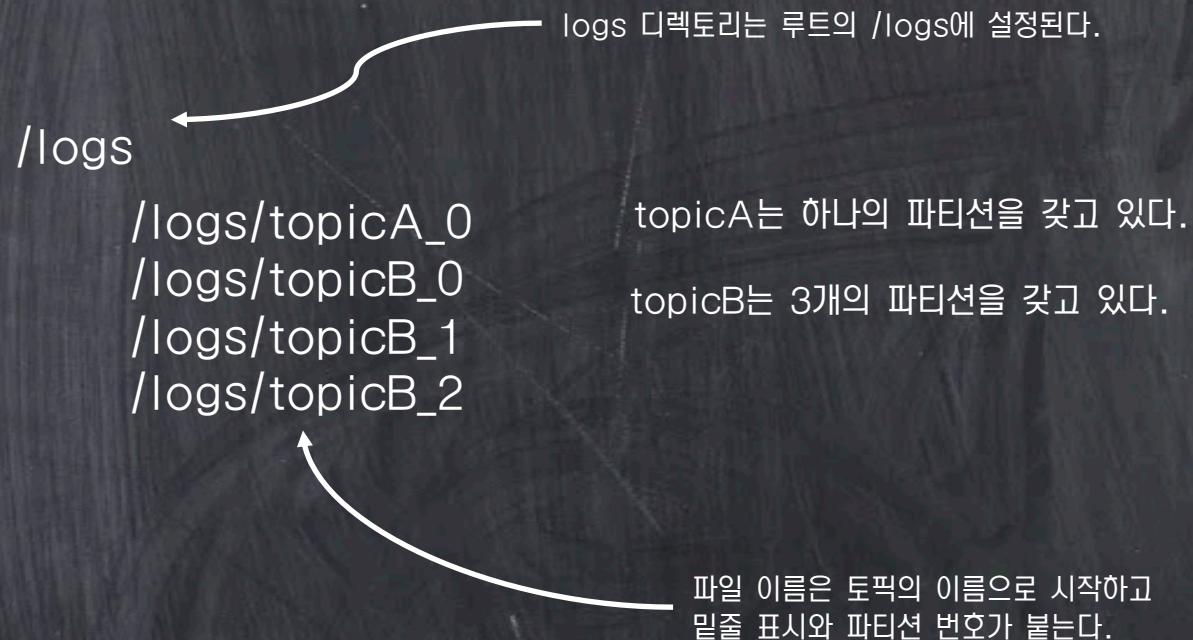
- ✓ 카프카의 기본 메커니즘은 로그(log)다.
- ✓ 토픽은 라벨이 붙은 로그라고 할 수 있다.

카프카의 맥락에서 로그는 “추가만 가능한(append-only) 시간 순으로 완전히 정렬된 레코드 시퀀스”다.



카프카에서 로그가 동작하는 방식

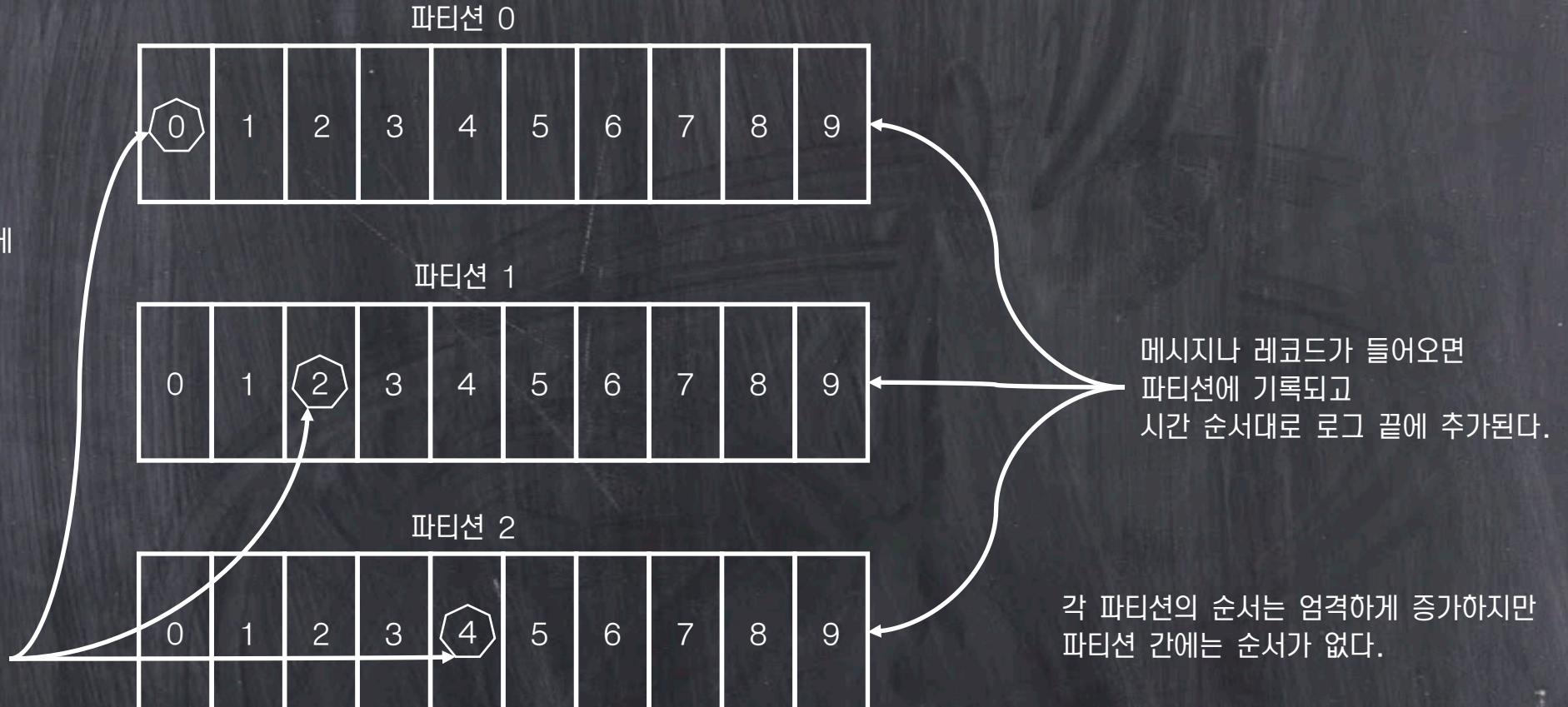
- ✓ 카프카를 설치할 때 설정 중 하나가 log.dir이며, 카프카가 로그 데이터를 저장하는 위치를 지정한다.
- ✓ 로그 파일이 특정 크기에 도달하거나 메시지 타임스탬프 간에 구성된 시간 차이에 도달하면 로그 파일을 교체하고, 카프카는 들어오는 메시지를 새 로그에 추가한다.



카프카와 파티션

- ✓ 파티션은 성능에 필수적이며, 같은 키를 가진 데이터가 동일한 컨슈머에게 순서대로 전송되도록 보장한다.
- ✓ 토픽을 파티션으로 분할하면 기본적으로 병렬 스트림에서 토픽에 전달되는 데이터가 분할되는데, 이는 카프카가 엄청난 처리량을 달성하는 비결이다.
- ✓ 파티션 간의 메시지 순서는 보장되지 않지만 각 파티션 내의 메시지 순서는 보장된다.

데이터는 단일 토픽으로 들어가지만 개별 파티션 (0, 1, 또는 2)에 배치된다.
이 메시지에는 키가 없기 때문에 라운드 로빈 방식으로 파티션이 할당된다.



키에 의한 그룹 데이터 분할

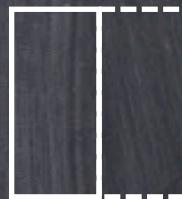
- ✓ 카프카는 키/값 쌍으로 데이터를 다룬다.
- ✓ 키가 널(null)이면 카프카 프로듀서는 라운드 로빈방식으로 선택된 파티션에 레코드를 쓴다.
- ✓ 널이 아닌 키로 파티션 할당이 어떻게 작동하는지 보여준다.

수신 메시지:

{foo, 메시지 데이터}
{bar, 메시지 데이터}

메시지 키는 메시지가 이동할 파티션을 결정하는 데 사용한다.
이러한 키는 널이 아니다.
키 바이트는 해시를 계산하는데 사용한다.

파티션 0



$\text{hashCode}(\text{fooBytes}) \% 2 = 0$

파티션 1



파티션이 결정되면 적절한 로그에 메시지가 추가된다.

$\text{hashCode}(\text{barBytes}) \% 2 = 1$

사용자 정의 파티셔너 작성하기

- ✓ 카프카에 구매 데이터가 유입되고 있고 키에 고객 ID와 거래 날짜라는 두 가지 값이 포함되어 있다고 가정하자.
- ✓ 고객 ID로 값을 그룹 지어야 하므로 고객 ID와 구매 날짜의 해시를 사용하면 작동하지 않을 것이다.
- ✓ 이 경우 사용자 정의 파티셔너를 작성해서 복합키의 어떤 부분이 어떤 파티션을 사용할지를 결정하는지 알아야 한다.

[키 변경] (고객 ID, 거래 날짜) → (고객 ID)

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {

    @Override
    public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
        Object newKey = null;
        if (key != null) {
            PurchaseKey purchaseKey = (PurchaseKey) key;
            newKey = purchaseKey.getCustomerId();
            keyBytes = ((String) newKey).getBytes(); ← 새로운 값으로 키의 바이트를 설정한다.
        }
        return super.partition(topic, newKey, keyBytes, value, valueBytes, cluster);
    }
}
```

새로운 값으로 키의 바이트를 설정한다.

슈퍼클래스에 위임하여 업데이트된 키로 파티션을 반환한다.

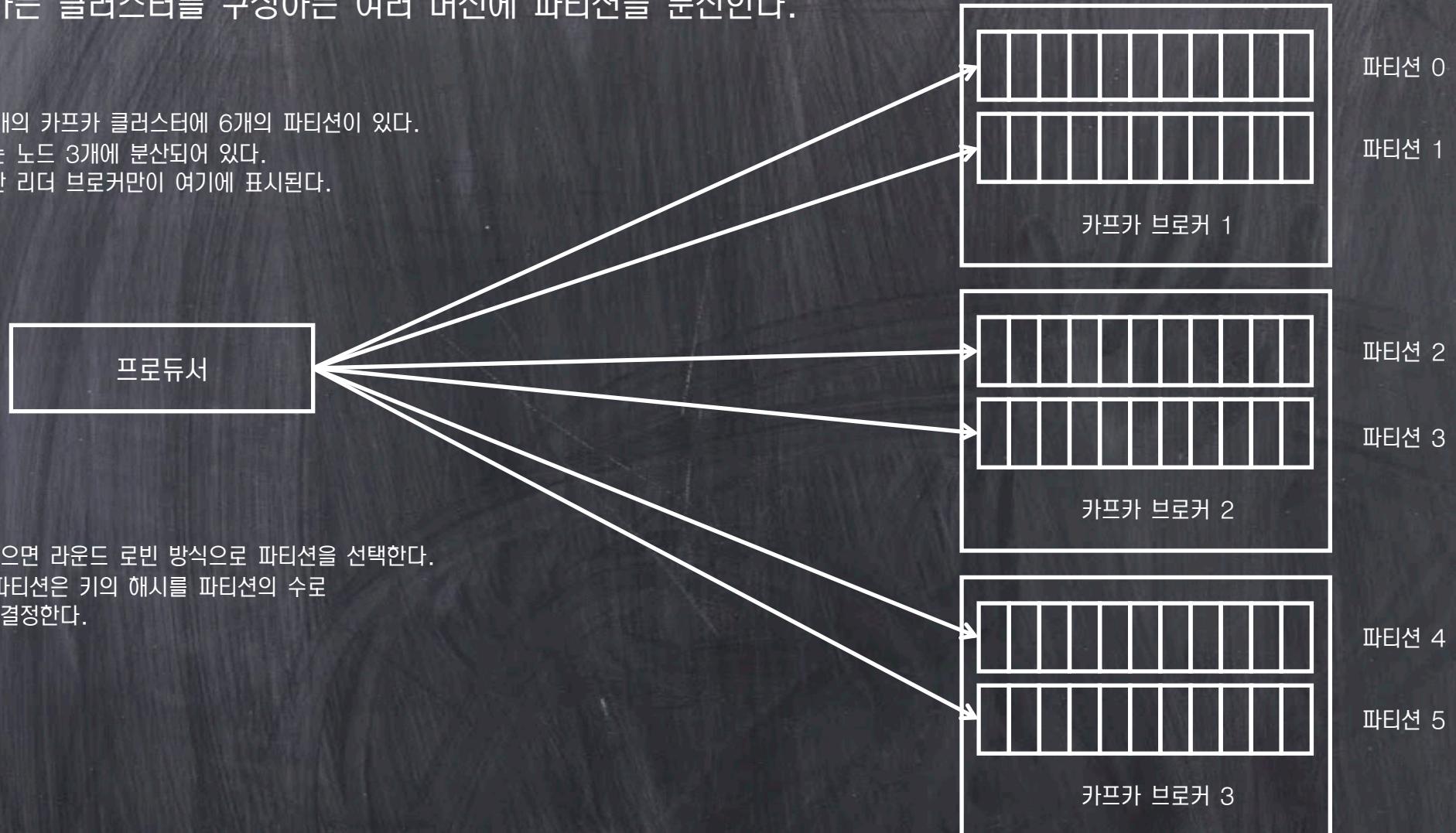
정확한 파티션 수 정하기

- ✓ 토픽을 만들 때 사용할 파티션 수를 선택할 때 고려사항 중 하나는 주어진 토픽에 들어오는 데이터 양이다.
- ✓ 데이터가 많을수록 처리량을 높이기 위해서는 더 많은 파티션이 필요하다.
- ✓ 그러나 파티션 수를 늘리면 TCP 연결 수와 열린 파일 핸들 수가 증가한다.
- ✓ 또한 컨슈머가 유입 레코드를 처리하는데 걸리는 시간도 처리량을 결정한다.
- ✓ 컨슈머가 대량의 데이터를 처리하는 경우에는 파티션을 추가하면 도움이 되지만, 궁극적으로 처리 속도가 느려지면 성능이 저하된다.

분산 로그

- ✓ 토픽이 분할되면 카프카는 하나의 머신에 모든 파티션을 할당하지 않는다.
- ✓ 카프카는 클러스터를 구성하는 여러 머신에 파티션을 분산한다.

이 토픽은 노드 3개의 카프카 클러스터에 6개의 파티션이 있다.
토픽에 대한 로그는 노드 3개에 분산되어 있다.
토픽 파티션에 대한 리더 브로커만이 여기에 표시된다.



리더, 팔로워, 복제

- ✓ 카프카는 리더와 팔로워 브로커라는 개념이 있다.
- ✓ 카프카에서 각 토픽 파티션별로 한 브로커가 다른 브로커(follower)의 리더(leader)로 선택된다.
- ✓ 리더의 주요 임무 중 하나는 팔로워 브로커에 토픽 파티션의 복제를 할당하는 것이다.

아파치 주키퍼

- ✓ 카프카 클러스터에서 브로커 중 하나는 컨트롤러로 선출된다.
- ✓ 메시지를 생성할 때 카프카는 레코드 파티션의 리더가 있는 브로커에게 레코드를 보낸다.



“주키퍼는 구성 정보를 유지 관리하고 이름을 지정하며 분산 동기화를 제공하고 그룹 서비스를 제공하는 중앙 집중식 서비스다”

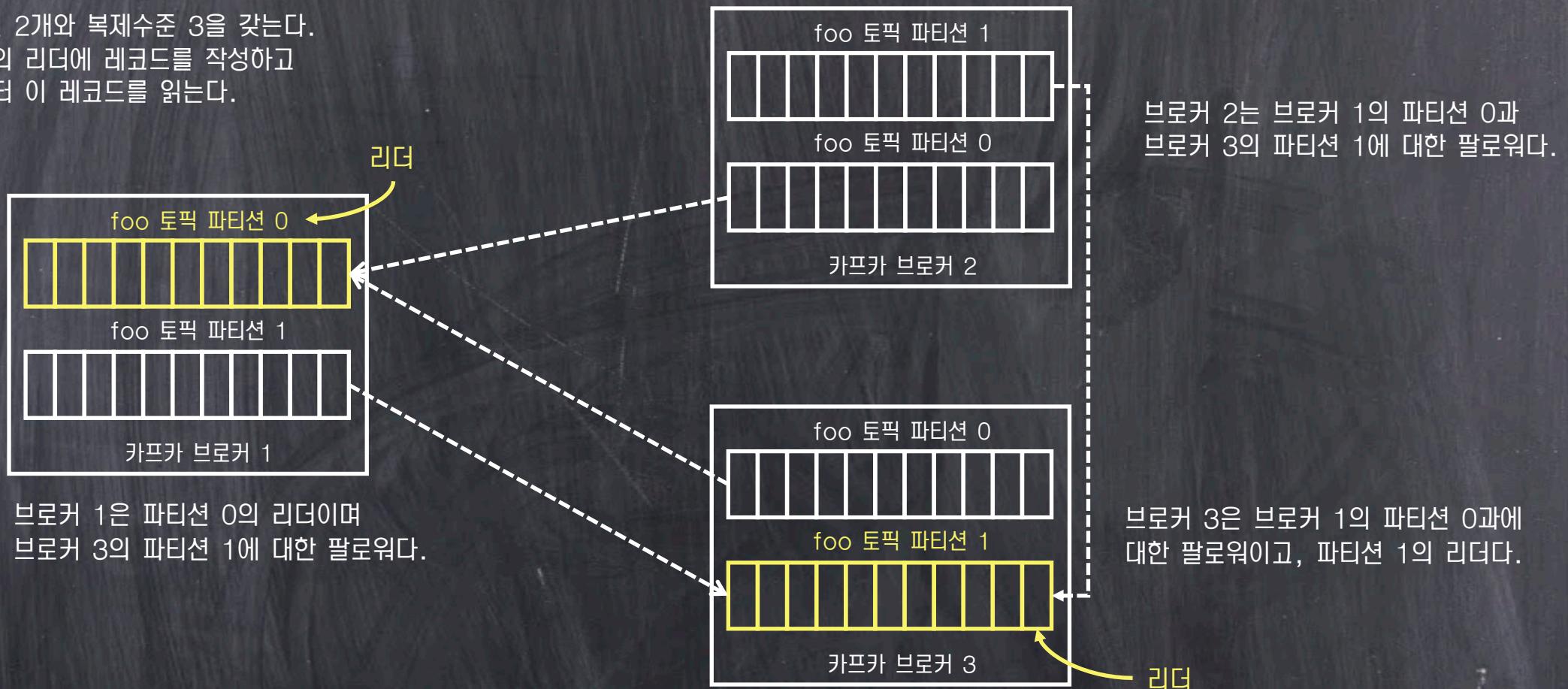
컨트롤러 선출

- ✓ 카프카는 주키퍼를 사용해 컨트롤러 브로커를 선출한다. (합의 알고리즘)
- ✓ 컨트롤러 브로커가 실패하거나 어떤 이유로든 사용할 수 없게 되면 주키퍼는 리더의 메시지 복제를 따라 잡은 것으로 간주되는 브로커 집합 ISR에서 새 컨트롤러를 선출한다.

복제

- ✓ 카프카는 클러스터의 브로커가 실패할 경우 데이터의 가용성을 보장하기 위해 브로커 간의 레코드를 복제한다.

토픽 foo는 파티션 2개와 복제수준 3을 갖는다.
프로듀서는 파티션의 리더에 레코드를 작성하고
팔로워는 리더로부터 이 레코드를 읽는다.



컨트롤러의 책임

- ✓ 컨트롤러 브로커는 토픽의 모든 파티션에 대한 리더/팔로워 관계를 설정한다.
- ✓ 카프카 노드가 죽거나 응답하지 않으면, 할당된 모든 파티션이 컨트롤러 브로커에 의해 재할당된다.

다음은 초기 리더와 팔로워다.

브로커 1은 리더 파티션 0, 팔로워 파티션 1

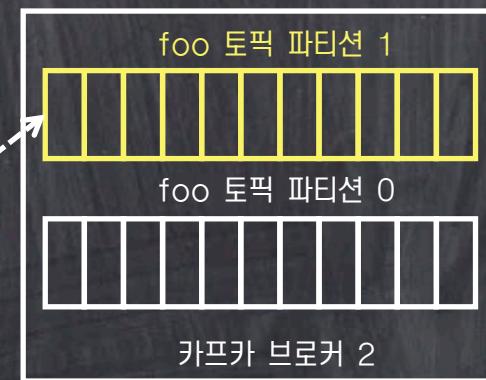
브로커 2는 팔로워 파티션 0, 팔로워 파티션 1

브로커 3은 팔로워 파티션 0, 리더 파티션 1



1단계:

리더로서 브로커 1은
브로커 3이 실패했음을 감지했다.



Fail



브로커 3이 응답하지 않는다.

2단계:

컨트롤러가 브로커 3에서 브로커 2로
파티션 1의 리더십을 재할당했다.
파티션 1에 기록할 레코드는 이제
브로커 2로 가고, 브로커 1은
파티션 1에 있는 메시지를 브로커 2에서
읽는다.

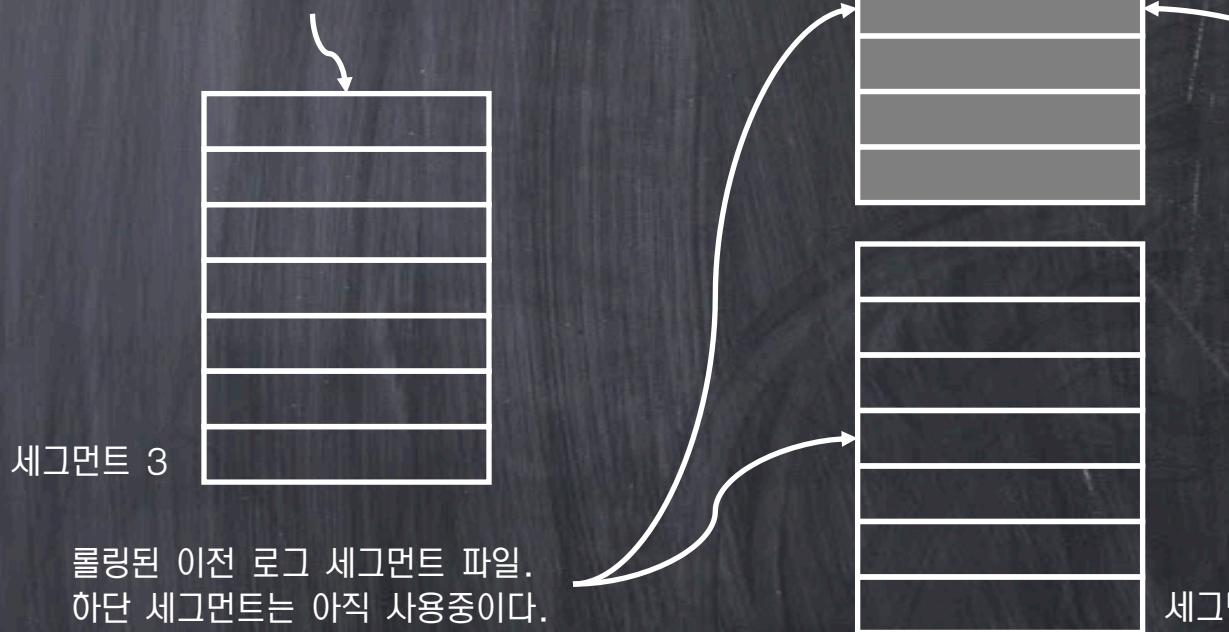
로그 관리

- ✓ 로그에 메시지가 추가된다고 언급했지만, 로그가 계속 커질 때 어떻게 관리되는지에 대해 언급하지 않았다.
- ✓ 클러스터에 있는 디스크의 공간은 한정된 자원이므로 카프카가 시간이 지남에 따라 메시지를 제거할 수 있어야 한다.
- ✓ 카프카에서 오래된 데이터를 제거할 때는 두가지 방식이 있는데, 전통적인 접근법인 **로그 삭제**, 그리고 **압축**이다.

로그 삭제

- ✓ 먼저 로그를 세그먼트로 나누어 가장 오래된 세그먼트를 삭제한다.
- ✓ 카프카는 새 메시지가 도착할 때의 타임스탬프가 해당 로그의 첫 번째 메시지의 타임스탬프와 log.roll.ms 설정값을 더한 값보다 크다면 로그를 분할하고 새로운 세그먼트를 새 활성 로그로 생성된다.
- ✓ 시간이 지남에 따라 세그먼트 수는 계속 증가할 것이고 오래된 세그먼트는 수신 데이터를 위한 여유 공간을 확보하기 위해 삭제돼야 한다.

레코드가 현재 로그에 추가된다.



세그먼트 1

이 세그먼트 로그가
삭제됐다

로그 툴링에는 두 가지 옵션이 있다.

- log.roll.ms: 주 설정이지만 기본값은 없다.
- log.roll.hours: 보조 설정이며 log.role.ms가 설정되지 않은 경우에만 사용한다. 기본값은 168시간(7일)이다.

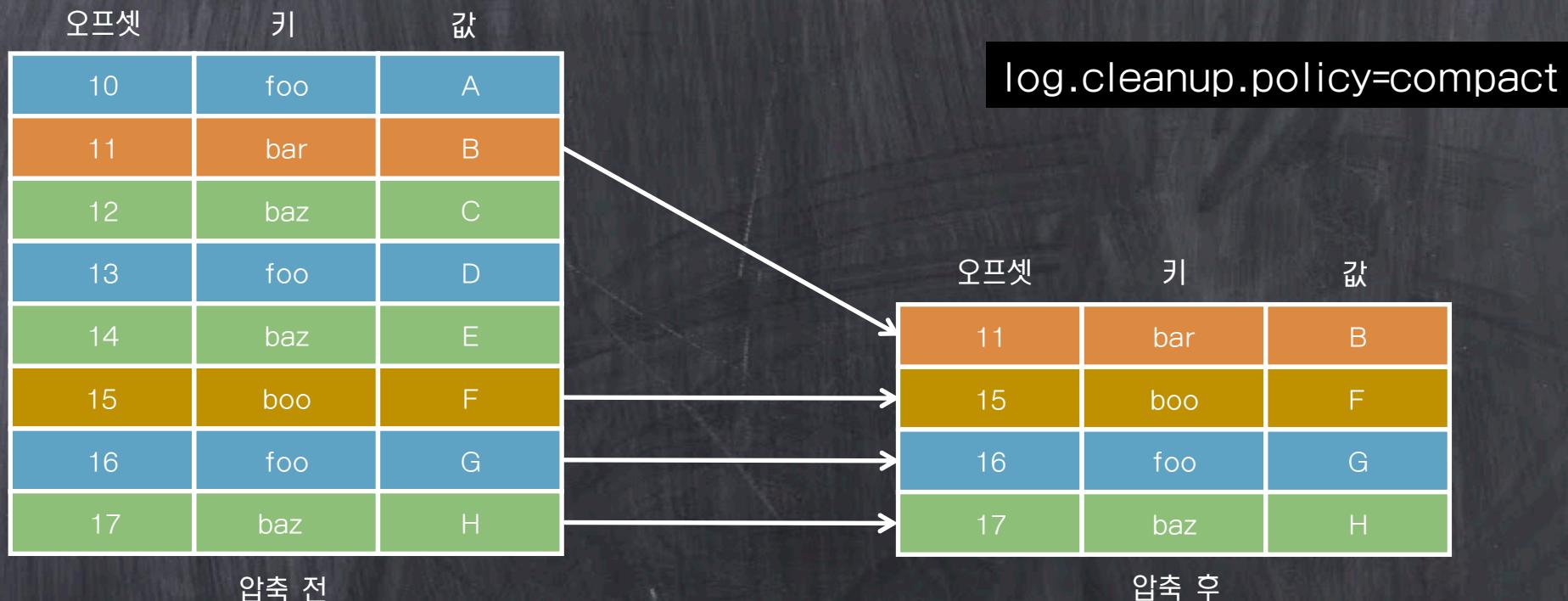
세그먼트 삭제도 메시지의 타임스탬프를 기반으로 한다.

- log.retention.ms: 로그 파일을 밀리초 단위로 보관하는 기간
- log.retention.minutes: 로그 파일을 분 단위로 보관하는 기간
- log.retention.hours: 시간 단위의 로그 파일을 보존 기간

(앞부분 설정이 뒷부분 설정 항목보다 우선한다)

로그 압축

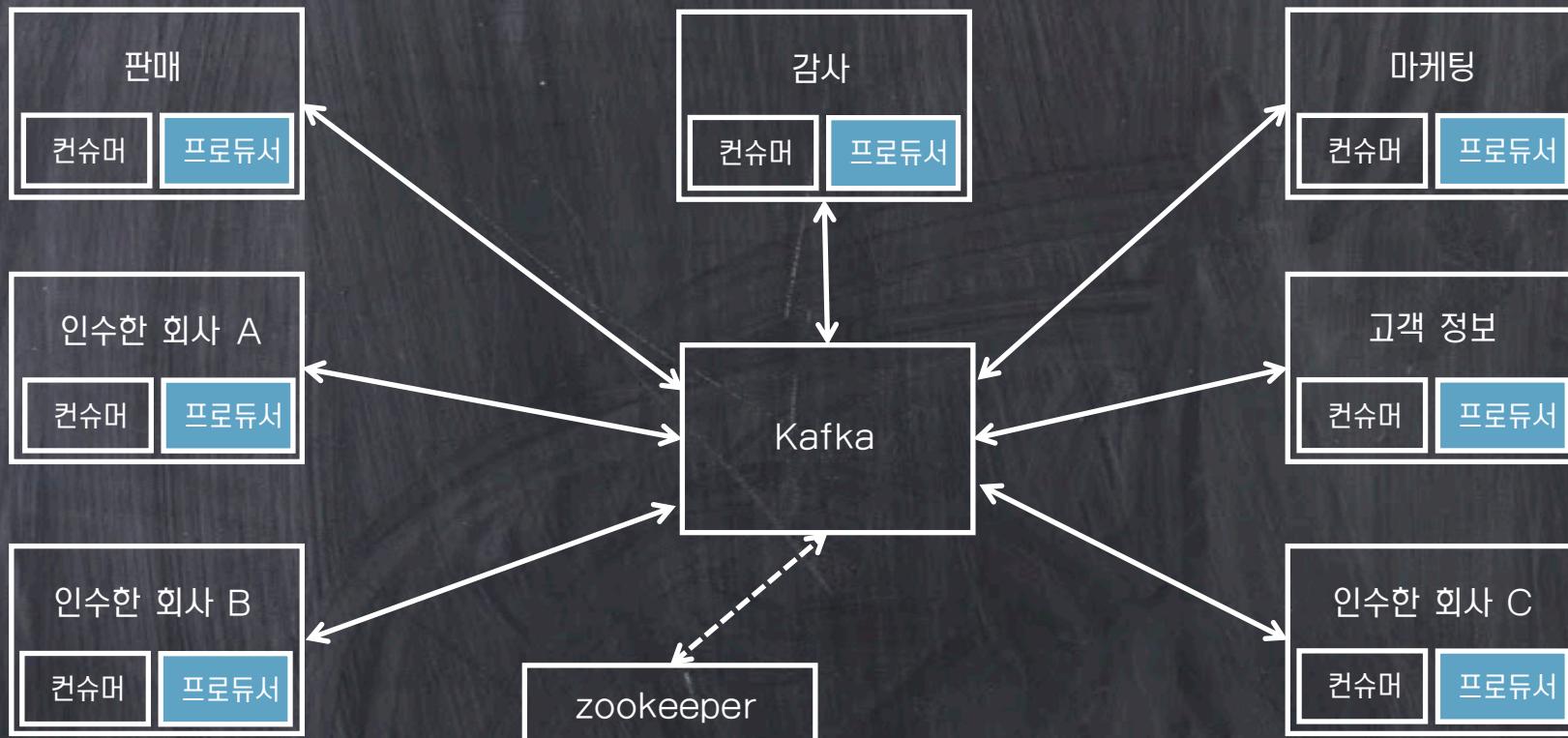
- ✓ 키를 가진 데이터가 있고, 시간이 지나면서 해당 데이터에 대한 업데이트를 받는 경우를 생각해보자.
- ✓ 동일한 키를 가진 새 레코드가 이전 값을 업데이트 한다는 것을 의미한다.
- ✓ 삭제 정책을 사용하면 마지막 업데이트와 애플리케이션의 크래시나 재시작 사이에 세그먼트가 제거됐을 수도 있다.
- ✓ 키로 레코드를 업데이트하는 것은 압축된 토퍽이 제공하는 동작이다.



이 방법은 주어진 키에 대한 마지막 레코드가 로그에 있음을 보장한다.

프로듀서로 메시지 보내기

- ✓ 프로듀서는 메시지를 보내는 데 사용하는 클라이언트다.
- ✓ 프로듀서는 어떤 컨슈머가 언제 메시지를 읽을지 모른다.
- ✓ 카프카로의 모든 전송은 비동기식이다.



프로듀서로 샘플 소스

SimpleProducer

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
properties.put("partitioner.class", PurchaseKeyPartitioner.class.getName());

PurchaseKey key = new PurchaseKey("12334568", new Date());

try(Producer<PurchaseKey, String> producer = new KafkaProducer<>(properties)) {
    ProducerRecord<PurchaseKey, String> record = new ProducerRecord<>("some-topic", key, "value");

    Callback callback = (metadata, exception) -> {
        if (exception != null) {
            exception.printStackTrace();
        }
    };
    Future<RecordMetadata> sendFuture = producer.send(record, callback);
}
```

프로듀서 속성

“serializer”

key.serializer와 value.serializer는 키와 값을 바이트 배열로 변환하는 방법을 카프카에 알려준다.

“bootstrap.servers”

쉼표로 구분된 호스트:포트 값의 리스트
프로듀서는 클러스터의 모든 브로커를 사용하며,
이 리스트는 처음에 클러스터에 연결하는 용도로만 사용한다.

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
properties.put("partitioner.class", PurchaseKeyPartitioner.class.getName());
```

“acks”

acks는 레코드 전송이 완료됐다고 생각하기 전에
프로듀서가 브로커로부터 기다리는 확인 수를 지정한다.
유효한 값은 all, 0, 1이다.

- all: 모든 팔로워가 레코드를 커밋할 때 까지 대기한다.
- 1: 브로커는 레코드를 로그에 기록하지만 팔로워의 레코드 커밋에 대한 확인 응답을 기다리지 않는다.
- 0: 프로듀서가 어떤 확인 응답도 기다리지 않음을 의미한다.

“retries”

배치 결과가 실패하는 경우 재전송 시도 횟수를 지정한다.
max.in.flight.requests.per.connection을 1로 설정해 실패한 레코드가 재전송되기 전에 두번째 배치가 성공적으로 보내지는 시나리오를 방지해야 한다.

“compression.type”

적용할 알고리즘이 있으면 지정한다.
있으면 메시지를 보내기 전에 압축하도록 프로듀서에
지시한다. 개별 레코드가 아닌 배치 단위로 압축한다.

“partitioner.class”

Partitioner 인터페이스를 구현하는
클래스 이름을 지정한다.

카프카의 타임스탬프

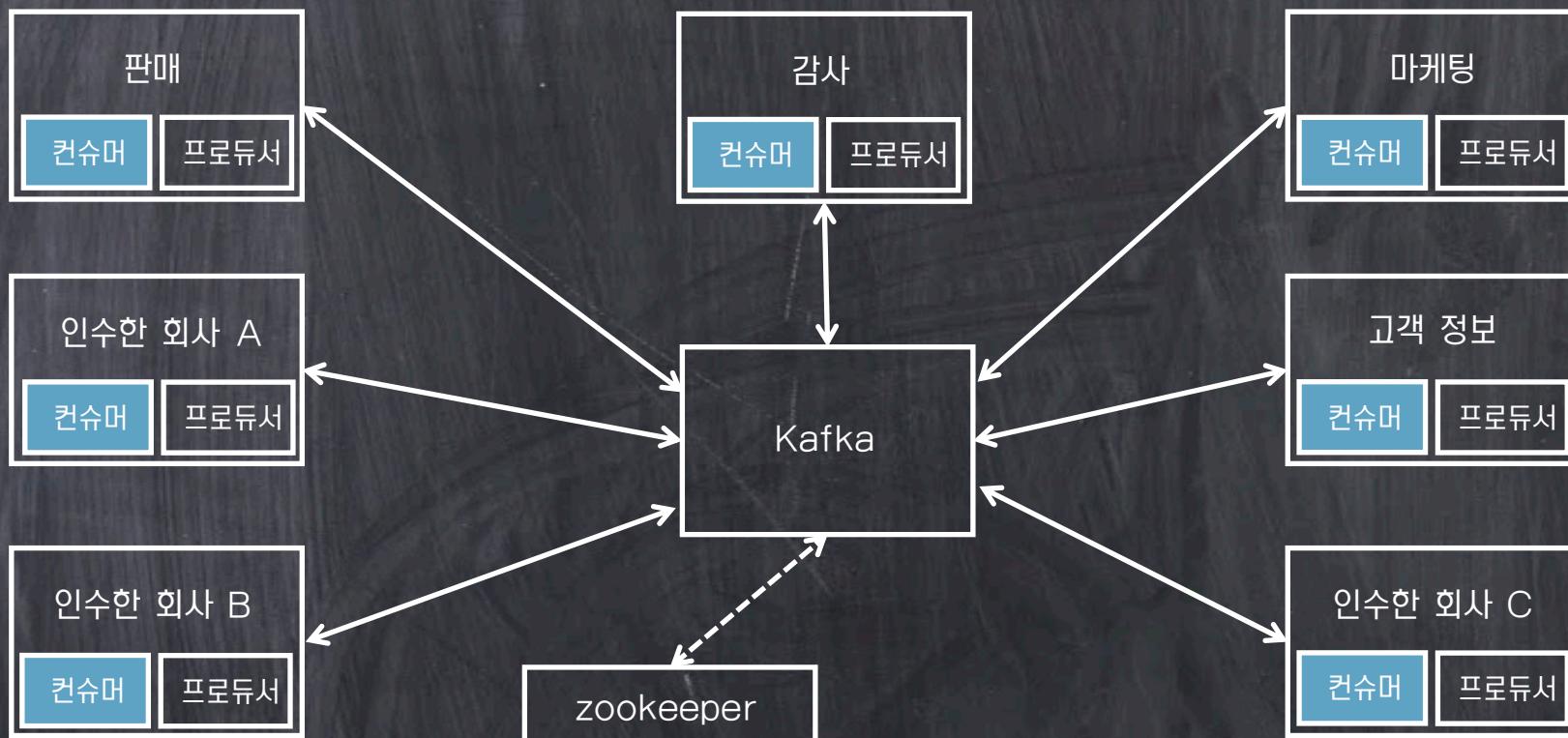
- ✓ 카프카 0.10버전부터 레코드에 타임스탬프를 추가했다.

```
ProducerRecord(String topic, Integer partition, Long timestamp, K key, V value)
```

- ✓ 타임스탬프를 설정하지 않으면 프로듀서가 레코드를 카프카 브로커에 보내기 전에 시간을 설정할 것이다.
- ✓ 타임스탬프는 log.message.timestamp.type 브로커 설정의 영향도 받는다.
- ✓ 이 설정은 CreateTime(기본값) 또는 LogAppendTime으로 설정할 수 있다.
- ✓ 토픽을 만들 때 해당 토픽에 다른 값을 지정할 수 있다.
- ✓ LogAppendTime은 ‘처리 시간’으로 간주되며 CreateTime은 ‘이벤트 시간’으로 간주한다.

컨슈머로 메시지 읽기

- ✓ 컨슈머는 카프카 메시지를 소비하는 데 사용할 클라이언트다.



오프셋 관리

- ✓ 오프셋 커밋은 컨슈머에 있어서 두 가지 의미가 있다.
 - 커밋한다는 것은 컨슈머가 메시지를 완전히 처리했음을 의미한다.
 - 커밋은 실패나 재시작 시 해당 컨슈머의 시작 지점도 나타낸다.

- ✓ `auto.offset.reset= "earliest"` : 사용 가능한 가장 이른 오프셋부터 시작해 메시지를 가져올 것이다.
- ✓ `auto.offset.reset= "latest"` : 가장 최신 오프셋에서 메시지를 읽어서 기본적으로 컨슈머가 클러스터에 합류한 지점부터 유입된 메시지만 소비된다.
- ✓ `auto.offset.reset= "none"` : 재설정 전략을 지정하지 않았다. 브로커가 컨슈머에게 예외를 발생시킨다.

`earliest` 설정은 오프셋 0에서 시작하는 메시지가 컨슈머에게 전송됨을 의미한다.



`latest`이면 컨슈머는 로그에 메시지가 추가될 때 다음 메시지를 받게 된다.

오프셋 커밋 - 자동

- ✓ 자동 오프셋 커밋 방식이 기본이며, `enable.auto.commit` 프로퍼티로 설정할 수 있다.
- ✓ 짹을 이루는 설정 옵션은 `auto.commit.interval.ms`인데 컨슈머가 오프셋을 커밋하는 주기를 지정한다. (기본 5초)
- ✓ 너무 작으면 네트워크 트래픽을 증가시키고, 너무 크면 실패 시 재시작 이벤트에서 컨슈머가 이미 받았던 데이터를 다시 받게 될 수 있다.

오프셋 커밋

- ✓ 수동 오프셋은 동기식 및 비동기식의 두 가지 유형이 있다.
- ✓ 인수가 없는 commitSync() 메소드는 마지막 검색에서 반환된 모든 오프셋이 성공할 때까지 블로킹한다.

```
consumer.commitSync()  
consumer.commitSync(Map<TopicPartition, OffsetAndMetadata>)
```

- ✓ commitAsync() 메소드는 완전 비동기식이고 즉시 반환된다.

```
consumer.commitAsync()
```

컨슈머와 파티션

- ✓ 토픽의 각 파티션마다 하나씩 여러 컨슈머 인스턴스가 필요할 것이다.
- ✓ 한 컨슈머가 여러 파티션에서 읽도록 할 수 있지만, 파티션 수만큼 스레드 풀에 스레드가 있는 컨슈머가 하나의 파티션에 배정되는 것이 일반적이지는 않다.
- ✓ 파티션당 컨슈머 패턴은 처리량을 최대화하지만, 여러 애플리케이션이나 머신이 컨슈머를 분산하는 경우 모든 인스턴스의 총 스레드 수는 해당 토픽의 총 파티션 수를 넘지 않아야 한다. 전체 파티션 수를 초과하는 스레드는 유휴 상태가 되기 때문이다.
- ✓ 리더 브로커는 동일 group.id를 가진 사용 가능한 모든 컨슈머에게 토픽 파티션을 할당한다.
- ✓ group.id는 컨슈머를 컨슈머 그룹에 속하도록 식별하는 설정이다.
- ✓ 이렇게 하면 컨슈머는 동일한 머신에 있을 필요가 없다. 하나의 머신이 실패할 경우, 리더 브로커는 상태가 좋은 머신의 컨슈머에게 토픽 파티션을 할당할 수 있다.

리밸런싱

- ✓ 컨슈머에게 토픽-파티션 할당을 추가 및 제거하는 프로세스를 리밸런싱이라고 한다.
- ✓ 동일한 그룹 ID를 가진 컨슈머를 추가하면 현재 토픽-파티션 할당 중 일부를 활성화 상태의 기존 컨슈머에서 가져와 새로운 컨슈머에게 준다.
- ✓ 이 재할당 프로세스는 모든 파티션이 데이터를 읽는 컨슈머에게 할당될 때까지 계속된다.
- ✓ 리밸런싱이 끝난 다음, 추가 컨슈머는 유휴 상태로 남을 것이다.
- ✓ 컨슈머가 어떤 이유로든 그룹을 떠난다면, 토픽-파티션 할당이 다른 컨슈머에게 재할당된다.

컨슈머 예제

```
private Runnable getConsumerThread(Properties properties) {  
    return () -> {  
        Consumer<String, String> consumer = null;  
        try {  
            consumer = new KafkaConsumer<>(properties);  
            consumer.subscribe(Collections.singletonList("test-topic")); ← 토픽을 구독  
            while (!doneConsuming) {  
                ConsumerRecords<String, String> records = consumer.poll(5000); ← 5초 동안 폴링  
                for (ConsumerRecord<String, String> record : records) {  
                    String message = String.format("Consumed: key = %s value = %s with offset = %d partition = %d",  
                        record.key(), record.value(), record.offset(), record.partition());  
                    System.out.println(message);  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        } finally {  
            if (consumer != null) {  
                consumer.close();  
            }  
        }  
    };  
}
```

토픽 생성하기

- ✓ 토픽을 생성하려면 kafka-topics.sh 스크립트를 실행해야 한다.

```
bin/kafka-topics.sh --create --topic first-topic --replication-factor 1 --partitions 1 --zookeeper localhost:2181
```

- ✓ replication-factor: 리더 브로커가 클러스터에 분산하는 메시지의 복사본 수를 결정한다.
1이면 복사본이 만들어지지 않는다.
- ✓ partitions: 토픽이 사용할 파티션 수를 지정한다. 부하가 큰 경우 더 많은 파티션이 필요할 것이다.

메시지 보내기

- ✓ 터미널 창에서 메시지를 보낼 수 있는 kafka-console-producer 스크립트를 실행한다.

```
bin/kafka-console-producer.sh --topic first-topic --broker-list localhost:9092
```

메시지 읽기

- ✓ 터미널 창에서 메시지를 읽을 수 있는 kafka-console-consumer 스크립트를 실행한다.

```
bin/kafka-console-consumer.sh --topic first-topic --bootstrap-server localhost:9092 --from-beginning
```

- ✓ from-beginning 매개변수는 해당 토픽에서 삭제되지 않은 메시지를 수신하도록 지정한다.
- ✓ from-beginning이 없으면 콘솔 컨슈머가 시작된 후에 보낸 메시지만 받는다.

Part 2

Kafka Streams 개발

- 3. Kafka Streams 개발
- 4. 스트림과 상태
- 5. KTable API
- 6. Processor API

3. Kafka Streams 개발

1. Streams API 소개
2. Streams Hello World
3. G-마트 Streams 애플리케이션

Streams API 소개

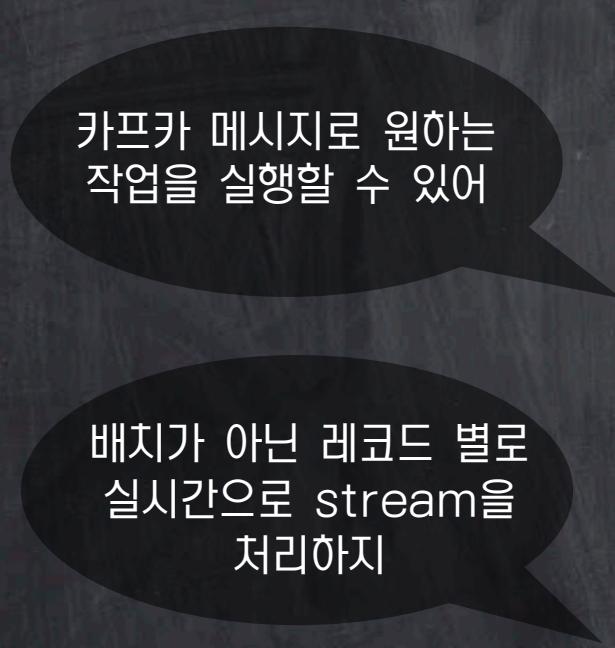
- ✓ Kafka Streams 애플리케이션을 신속하게 만들 수 있게 해주는 고수준 API이다.
- ✓ 고수준 API의 핵심은 키/값 쌍 레코드 스트림을 나타내는 KStream 객체이다.
- ✓ 이 외에 더 많은 제어가 가능한 저수준 API인 Processor API가 있다.



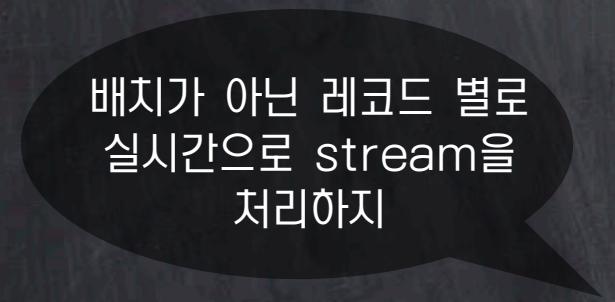
Streams API가
뭐지?



그냥 애플리케이션에서
실행되는 라이브러리야

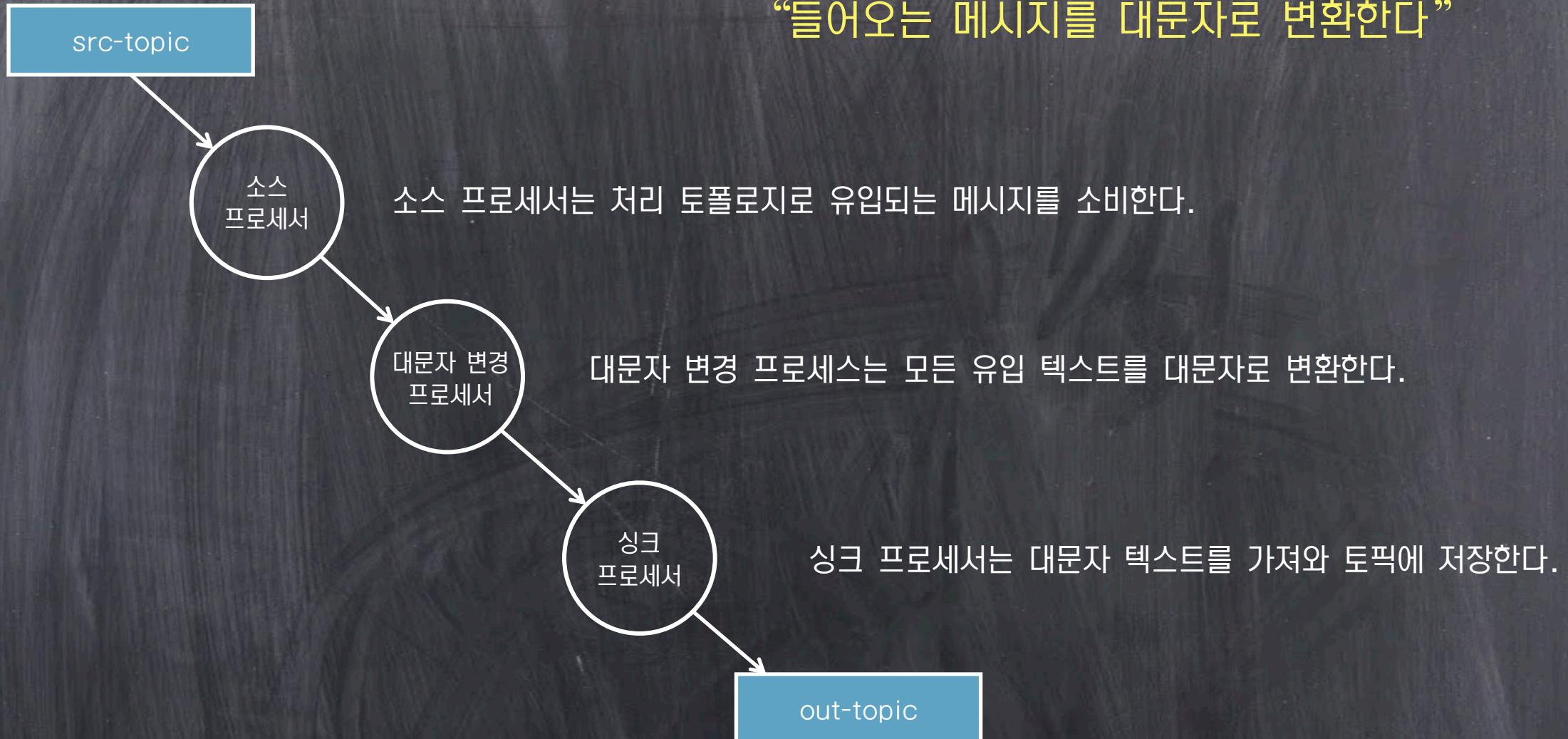


카프카 메시지로 원하는
작업을 실행할 수 있어

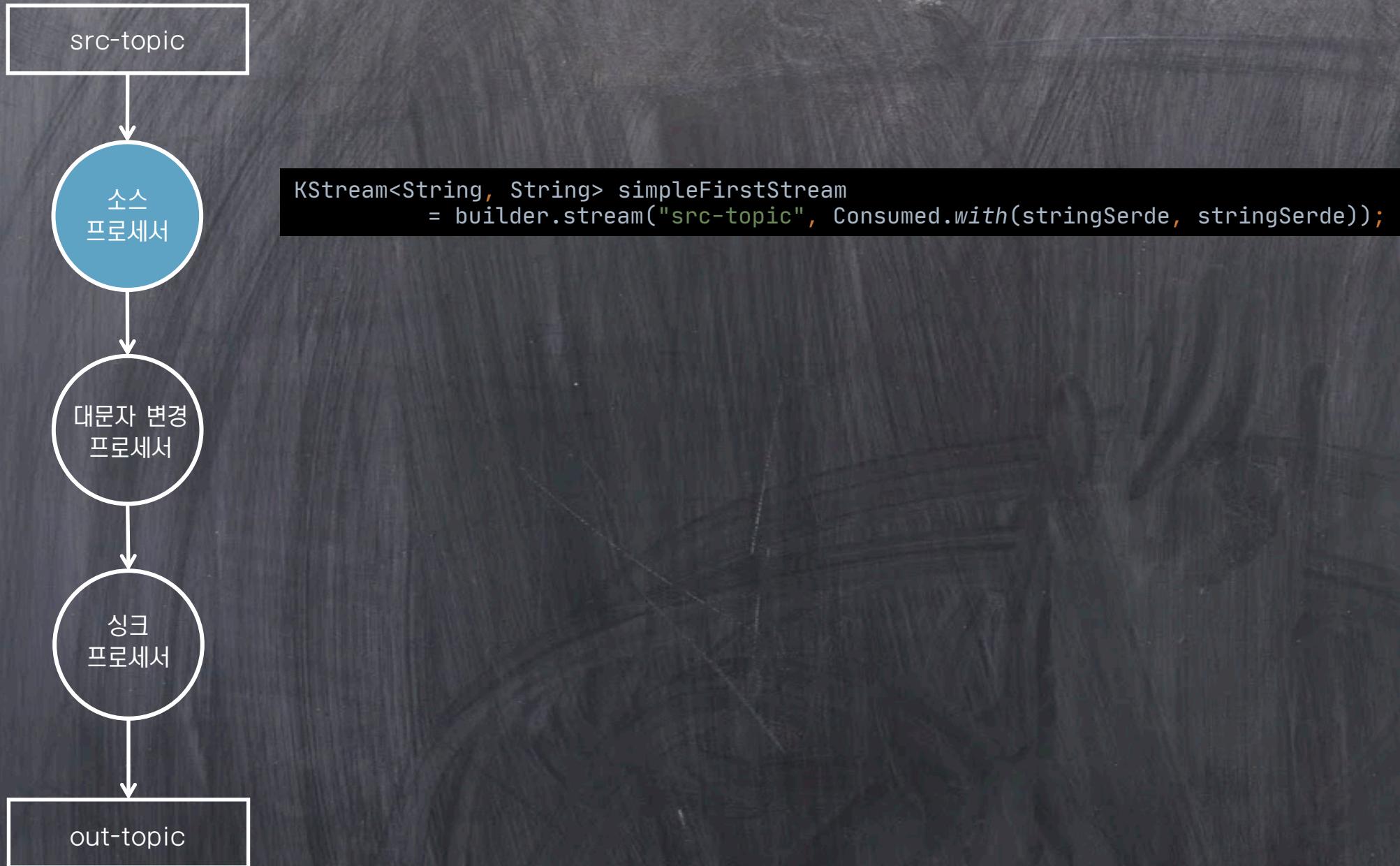


배치가 아닌 레코드 별로
실시간으로 stream을
처리하지

Hello World 애플리케이션

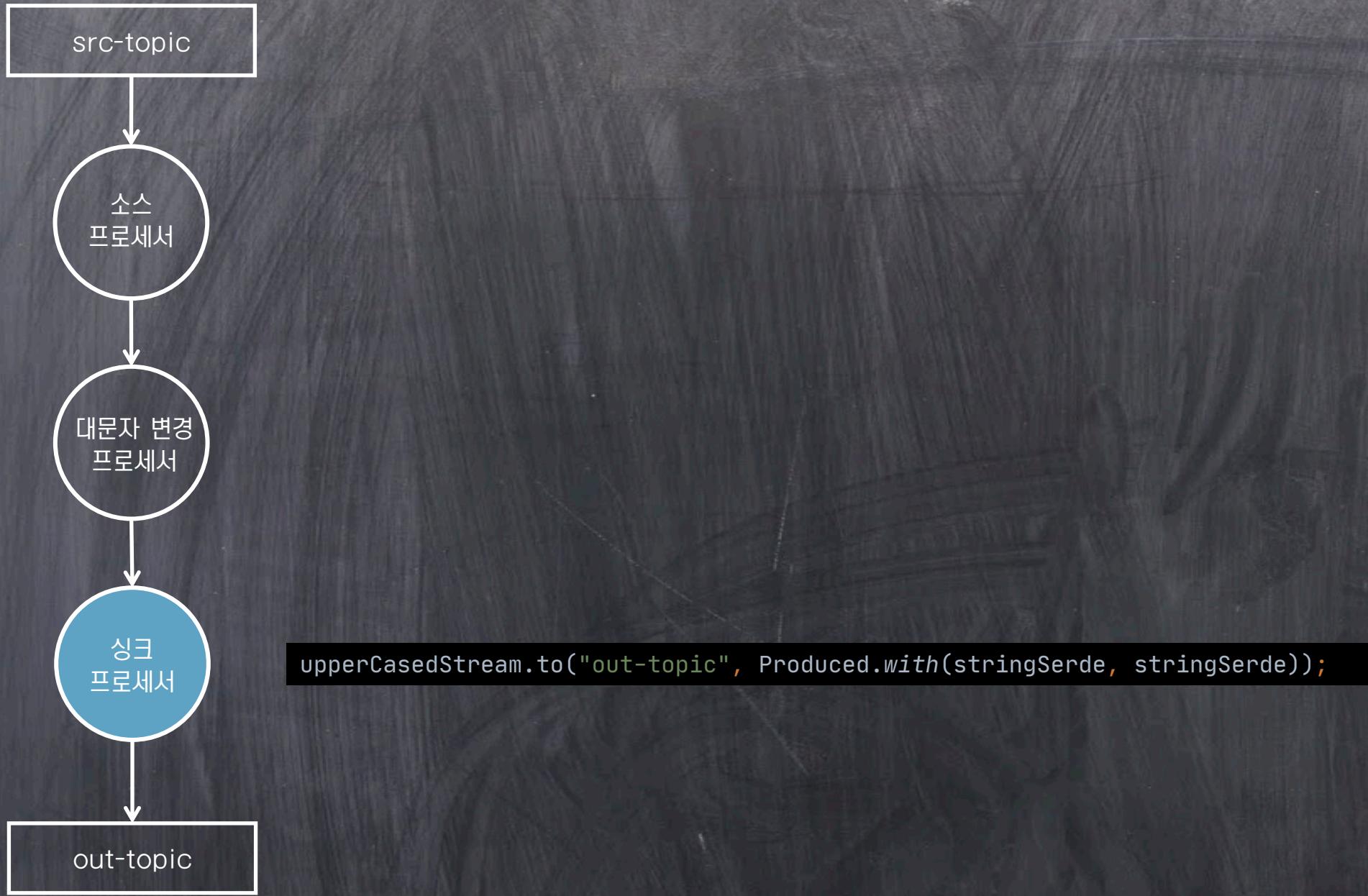


3. Kafka Streams 개발 > 2. Streams Hello World





```
KStream<String, String> upperCasedStream = simpleFirstStream.mapValues(String::toUpperCase);
```



Kafka Streams DSL은
이와 같은 형식이다

```
public class KafkaStreamsYellingApp {  
    public static void main(String[] args) throws Exception {  
        Properties props = new Properties();  
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");  
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
  
        StreamsConfig streamsConfig = new StreamsConfig(props);  
  
        Serde<String> stringSerde = Serdes.String();  
  
        StreamsBuilder builder = new StreamsBuilder();  
        KStream<String, String> simpleFirstStream = builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));  
        KStream<String, String> upperCasedStream = simpleFirstStream.mapValues(String::toUpperCase);  
  
        upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));  
        upperCasedStream.print(Printed.<String, String>toSysOut().withLabel("Yelling App"));  
  
        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);  
        kafkaStreams.start();  
  
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
            System.err.println("Shutting down stream");  
            kafkaStreams.close();  
        }));  
    }  
}
```

Kafka Stream 개발 4단계

- ① StreamsConfig 인스턴스 생성
- ② Serde 객체를 생성
- ③ 처리 토플로지를 구성
- ④ Kafka Streams 프로그램 시작

KafkaStreamsYellingApp

```

public class KafkaStreamsYellingApp {
    public static void main(String[] args) throws Exception {
        Properties props = new Properties();
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
        props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

        StreamsConfig streamsConfig = new StreamsConfig(props);           ← StreamsConfig 인스턴스 생성

        Serde<String> stringSerde = Serdes.String();                      ← Serde 객체를 생성

        StreamsBuilder builder = new StreamsBuilder();
        KStream<String, String> simpleFirstStream = builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));
        KStream<String, String> upperCasedStream = simpleFirstStream.mapValues(String::toUpperCase);          ← 처리 토플로지를 구성

        upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
        upperCasedStream.print(Printed.<String, String>toSysOut().withLabel("Yelling App"));

        KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
        kafkaStreams.start();

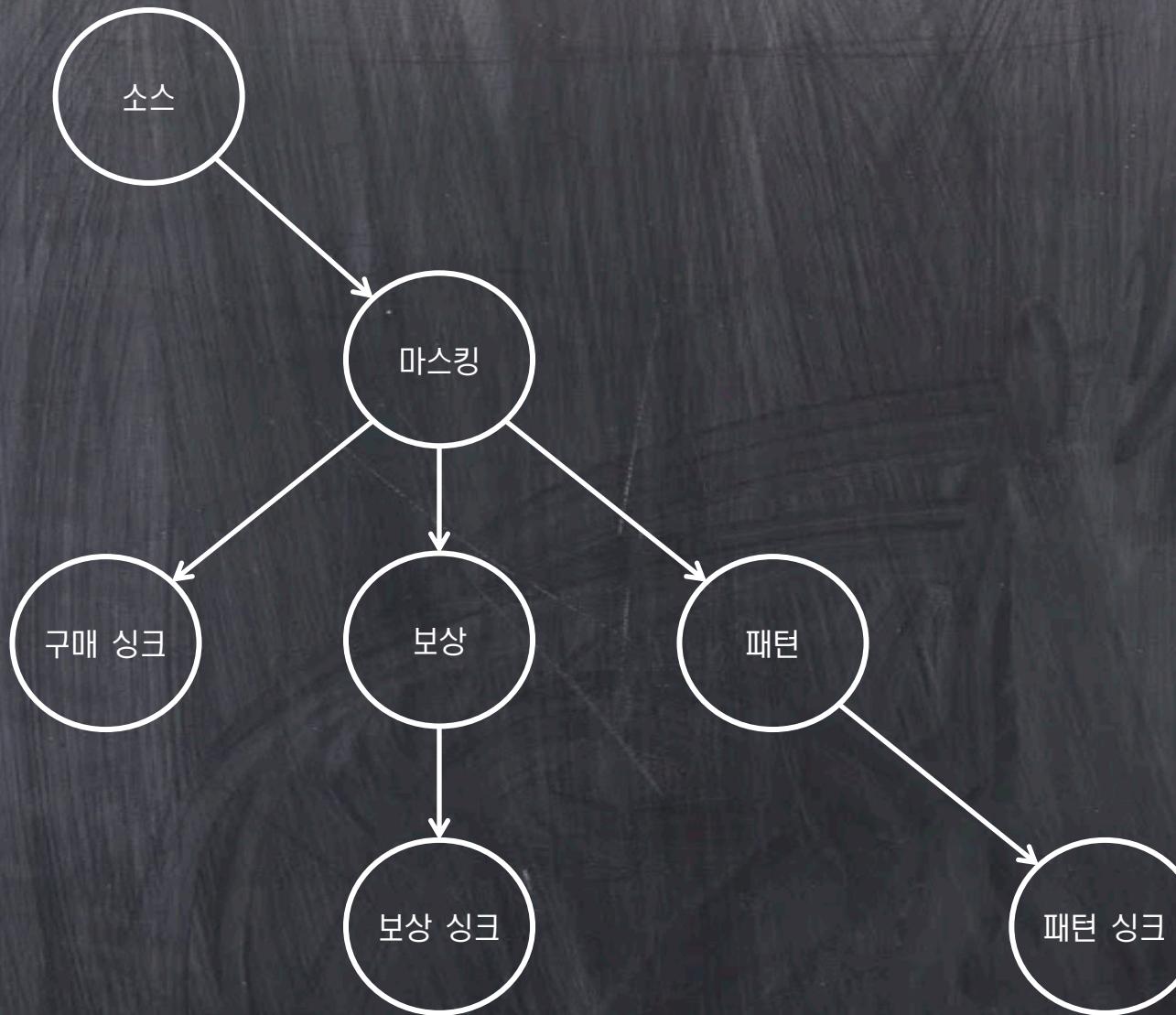
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {
            System.err.println("Shutting down stream");
            kafkaStreams.close();
        }));
    }
}

```

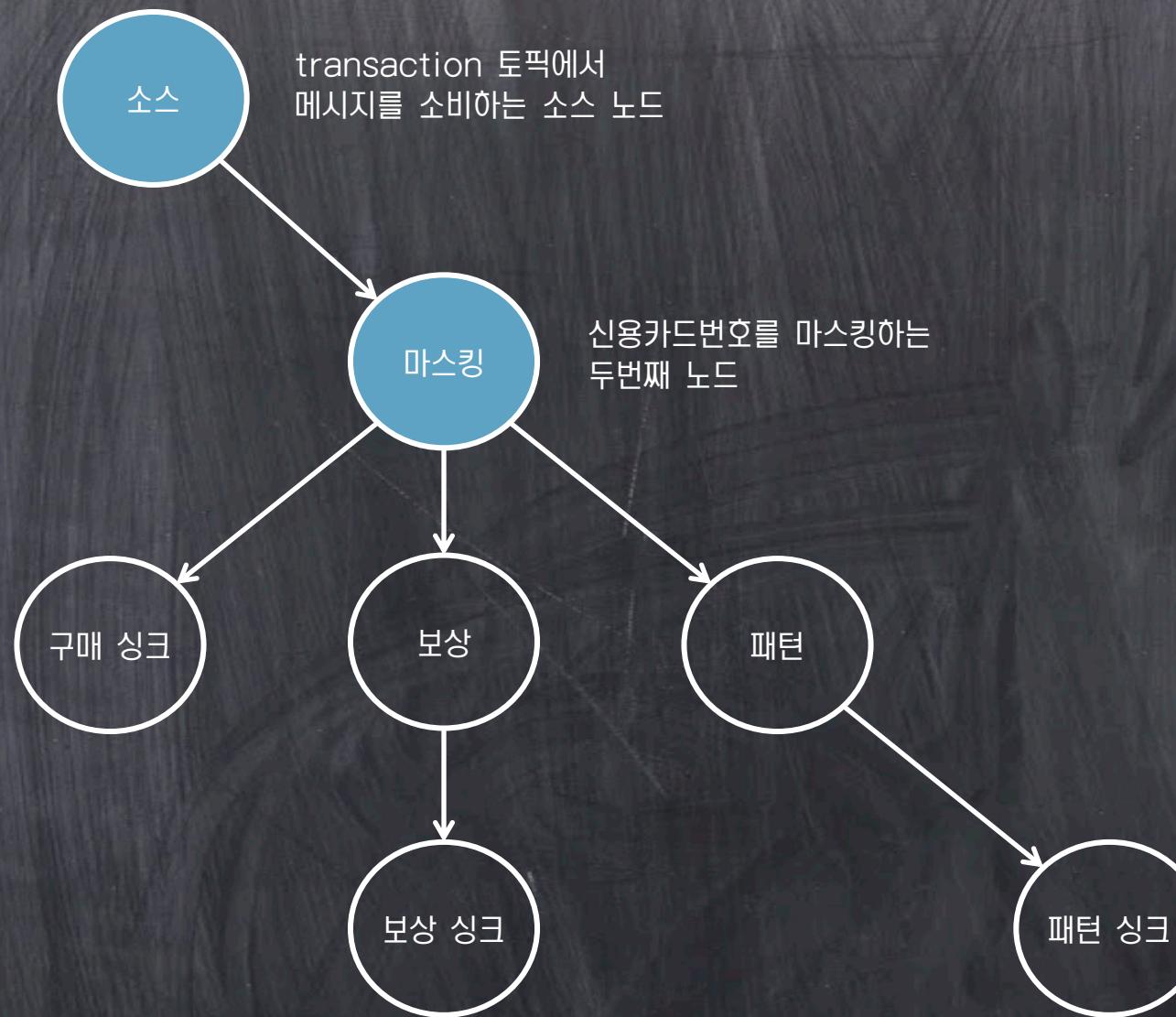
G-마트 요구사항

- ✓ 모든 기록은 보호된 신용카드 번호를 가져야 하며, 이 경우 처음 12자리를 마스킹해야 한다.
- ✓ 구매 패턴을 결정하려면 구입한 품목과 우편번호를 추출해야 한다. 이 데이터는 토픽에 기록할 것이다.
- ✓ 고객의 G-마트 회원 번호와 지출한 금액을 캡처해 이 정보를 토픽에 기록해야 한다.
토픽의 컨슈머는 이 데이터를 사용해 보상을 결정한다.
- ✓ 전체 트랜잭션을 토픽에 기록해야 하며, 임의 분석을 위해 스토리지 엔진(NoSQL)에서 사용한다.

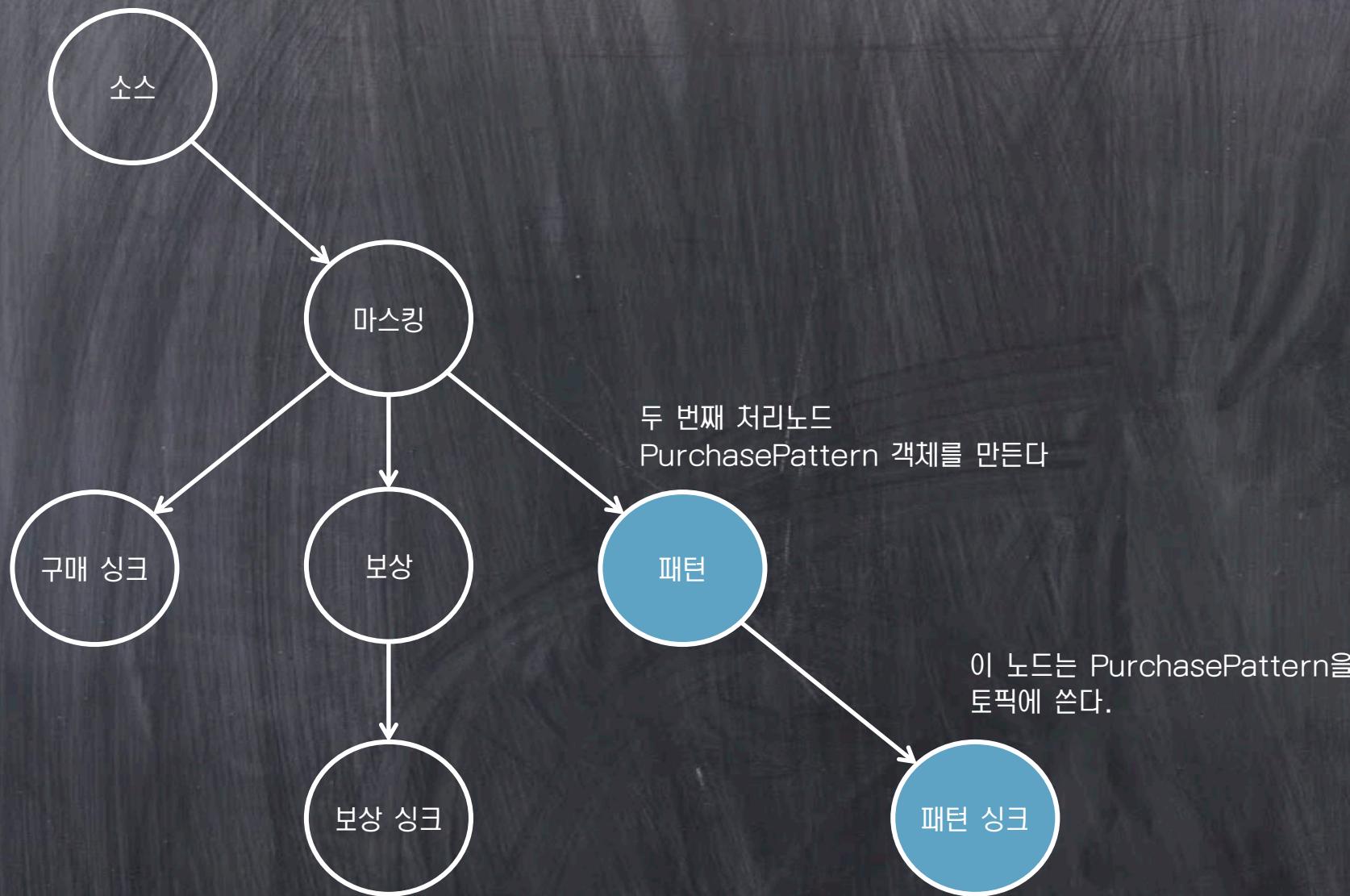
G-마트 예제 살펴보기



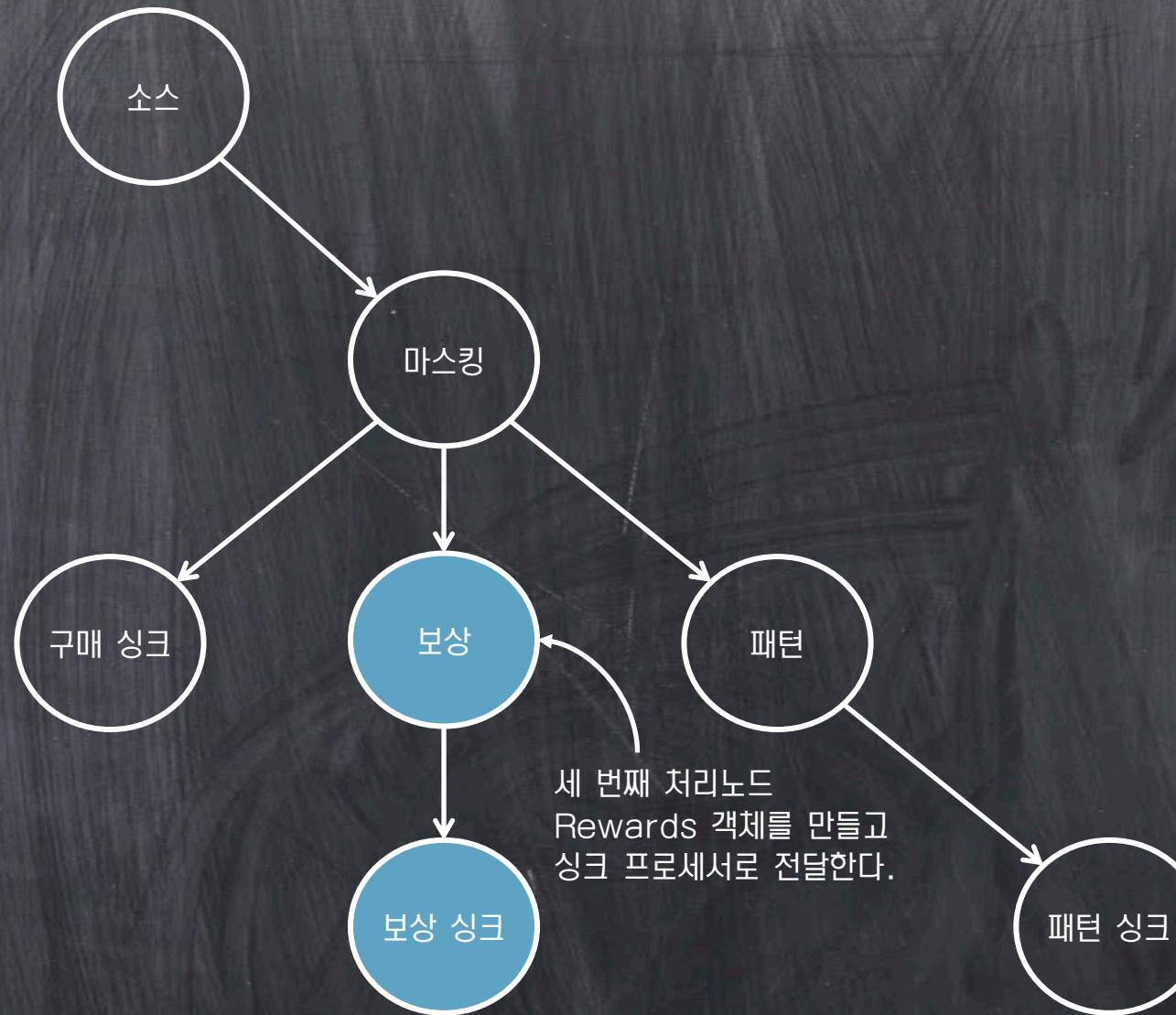
소스 노드 만들기



두 번째 프로세서 만들기



세 번째 프로세서 만들기



마지막 프로세서 만들기



```
public class GMartStream {  
    public static void main(String[] args) {  
        Properties config = new Properties();  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "gmart-stream-app");  
        config.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");  
  
        Serde<Purchase> purchaseSerde = StreamsSerdes.PurchaseSerde();  
        Serde<PurchasePattern> purchasePatternSerde = StreamsSerdes.PurchasePatternSerde();  
        Serde<RewardAccumulator> rewardAccumulatorSerde = StreamsSerdes.RewardAccumulatorSerde();  
        Serde<String> stringSerde = Serdes.String();  
  
        StreamsBuilder streamsBuilder = new StreamsBuilder();  
        KStream<String, Purchase> purchaseKStream  
            = streamsBuilder.stream("transactions", Consumed.with(stringSerde, purchaseSerde))  
                .mapValues(p -> Purchase.builder(p).maskCreditCard().build());  
  
        KStream<String, PurchasePattern> patternKStream  
            = purchaseKStream.mapValues(purchase -> PurchasePattern.builder(purchase).build());  
        patternKStream.to("patterns", Produced.with(stringSerde, purchasePatternSerde));  
  
        KStream<String, RewardAccumulator> rewardsKStream  
            = purchaseKStream.mapValues(purchase -> RewardAccumulator.builder(purchase).build());  
        rewardsKStream.to("rewards", Produced.with(stringSerde, rewardAccumulatorSerde));  
        purchaseKStream.to("purchases", Produced.with(stringSerde, purchaseSerde));  
  
        KafkaStreams streams = new KafkaStreams(builder.build(), config);  
        streams.start();  
  
        Runtime.getRuntime().addShutdownHook(new Thread(() -> {  
            streams.close();  
        }));  
    }  
}
```

새로운 요구사항

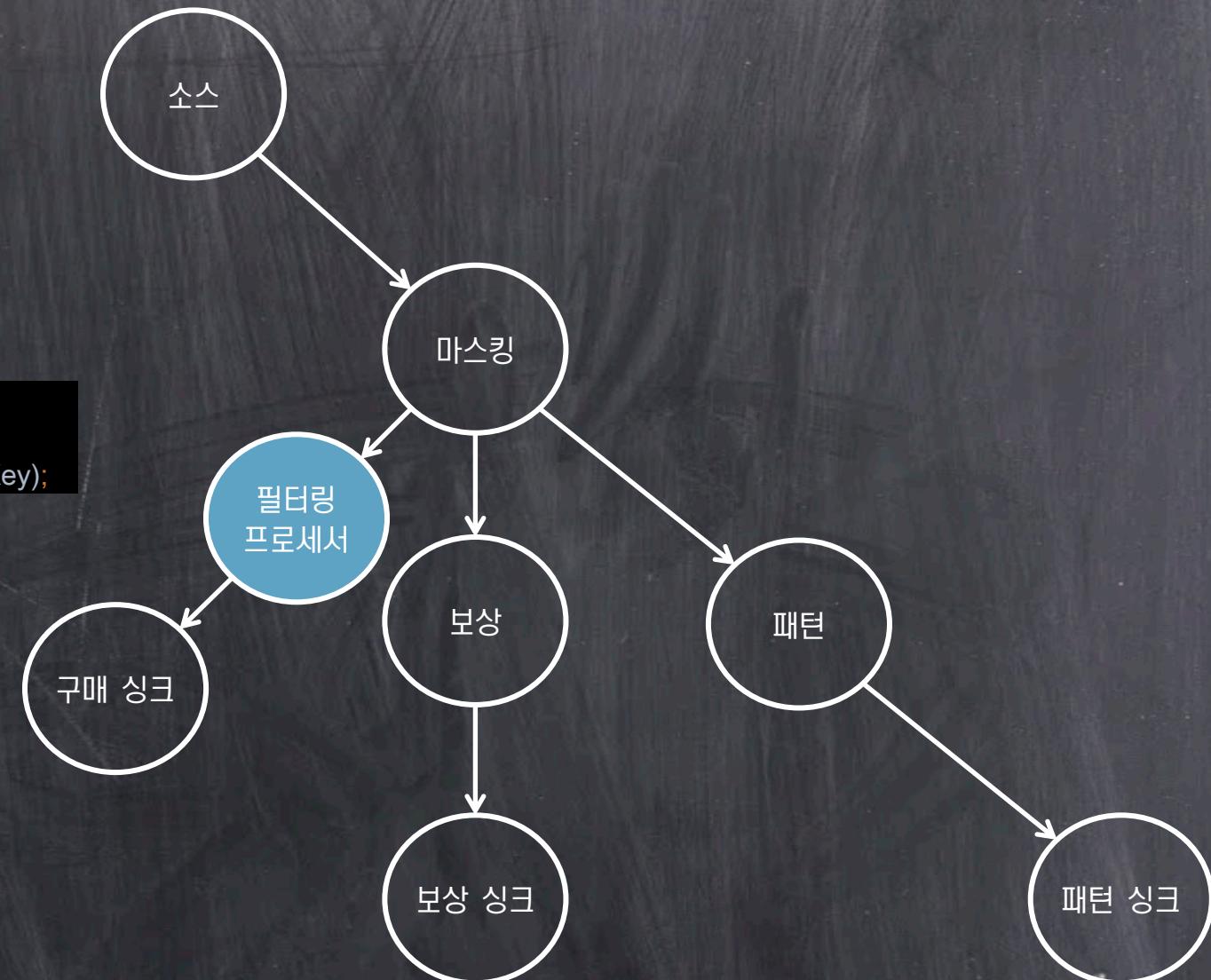
- ✓ 특정 액수 미만의 구매는 걸러낼 필요가 있다.
상위 관리자는 일반적인 소량의 일일 물품 구매에는 별로 관심이 없다.
 - ✓ G-마트가 확장되어 전자제품 체인과 인기 있는 커피 하우스 체인을 샀다.
새 상점에서 구입한 모든 항목은 구축한 스트리밍 애플리케이션을 통해 전달된다.
 - ✓ 선택한 NoSQL 솔루션은 항목을 키/값 형식으로 저장한다.
카프카 클러스터에 들어오는 레코드에는 키가 정의되어 있지 않다.
- 

구매 필터링

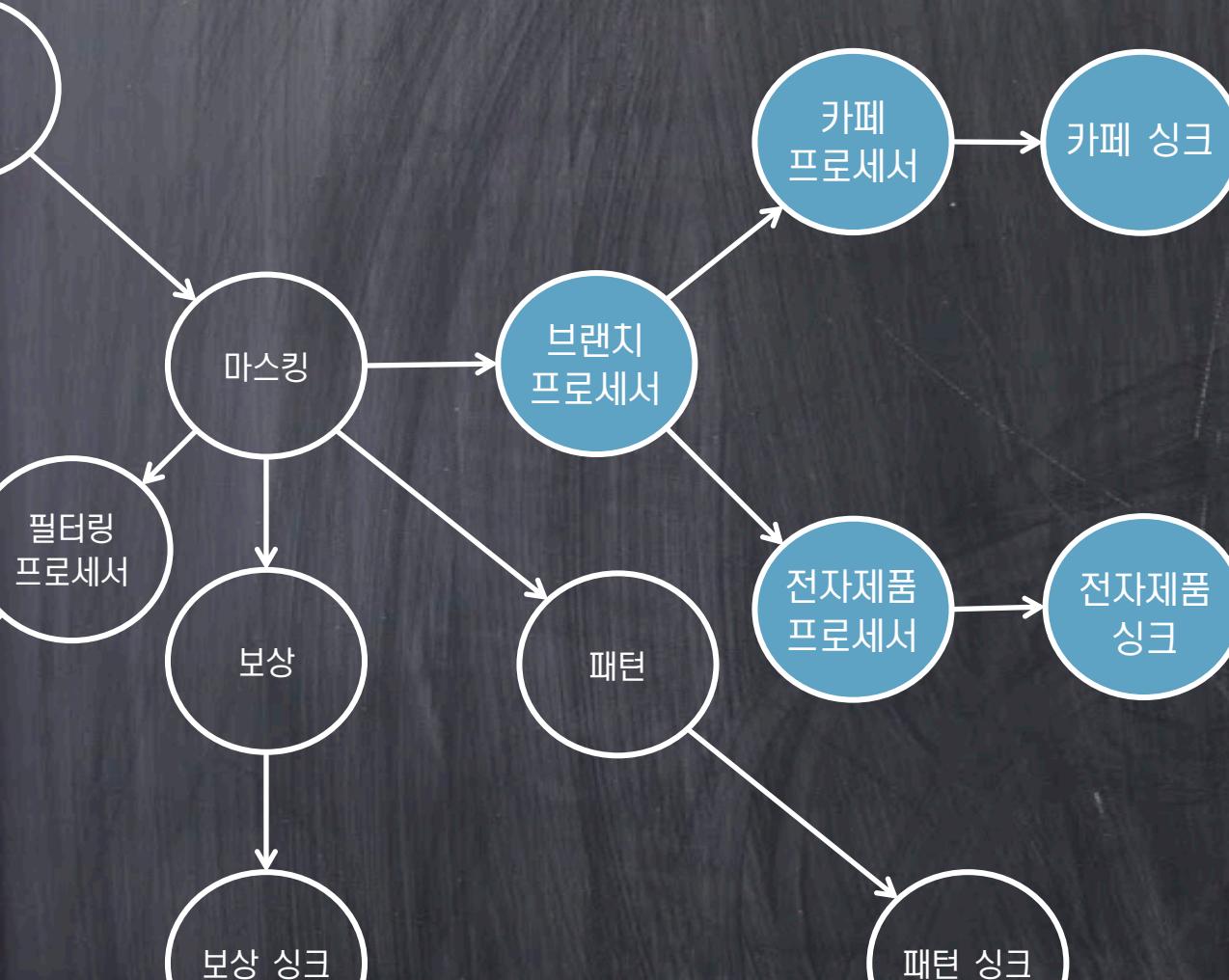
금액이 \$5 이상으로 필터링

```
KStream<Long, Purchase> filteredKStream  
= purchaseKStream.filter((key, purchase) ->  
    purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);
```

금액이 \$50이상인 데이터만 가져옴



스트림 나누기



판매 부서가 coffee와 electronics 인것만 선택

```

Predicate<String, Purchase> isCoffee = (key, purchase) ->
    purchase.getDepartment().equalsIgnoreCase("coffee");
Predicate<String, Purchase> isElectronics = (key, purchase) ->
    purchase.getDepartment().equalsIgnoreCase("electronics");
  
```

Topic을 2개로 쪼개기

```

KStream<String, Purchase>[] kstreamByDept =
    purchaseKStream.branch(isCoffee, isElectronics);

kstreamByDept[coffee].to("coffee",
    Produced.with(stringSerde, purchaseSerde));

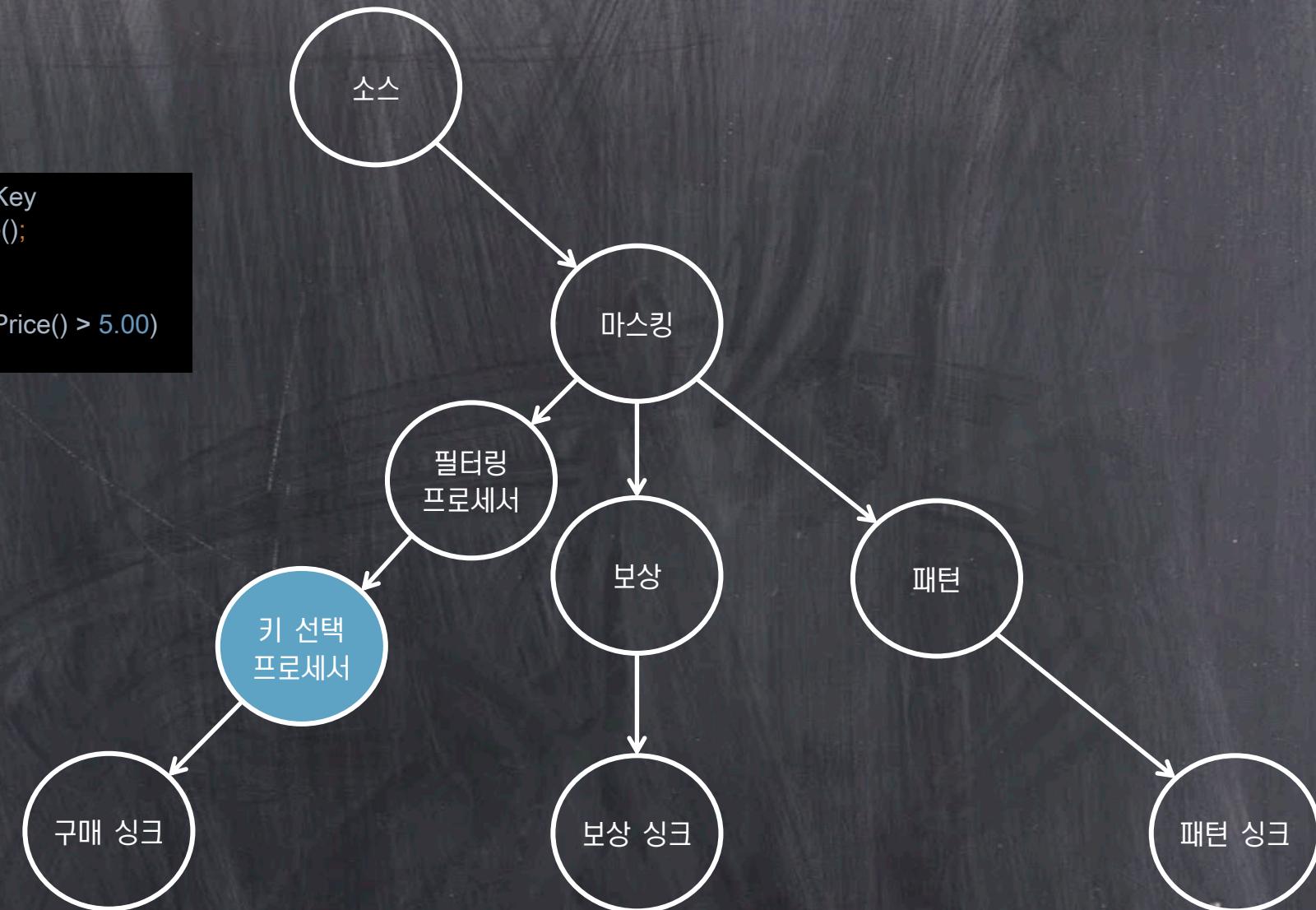
kstreamByDept[electronics].to("electronics",
    Produced.with(stringSerde, purchaseSerde));
  
```

키 생성하기

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey  
= (key, purchase) -> purchase.getPurchaseDate().getTime();
```

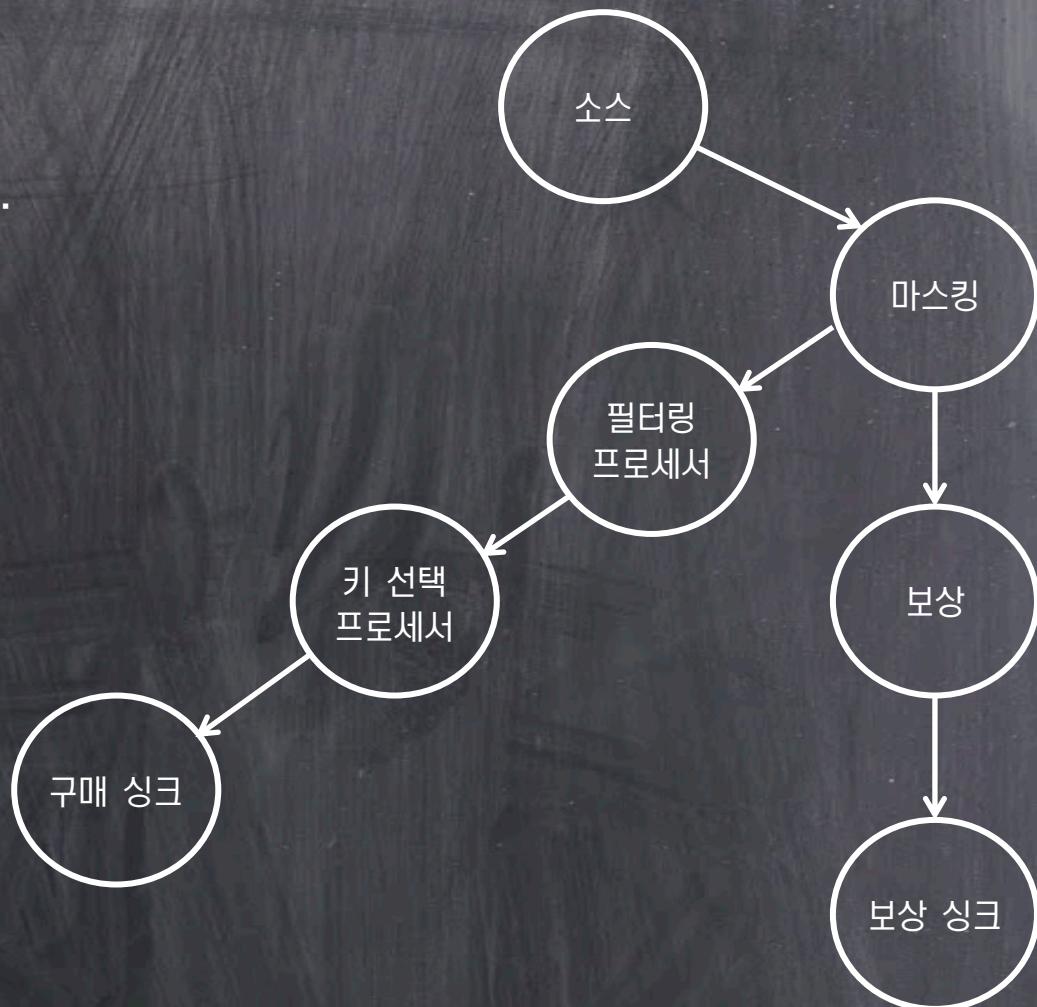
```
KStream<Long, Purchase> filteredKStream  
= purchaseKStream.filter((key, purchase) -> purchase.getPrice() > 5.00)  
.selectKey(purchaseDateAsKey);
```

구매 타임스탬프를 키로 사용하기 위해
selectKey 사용



카프카 외부에 레코드 기록하기

- ✓ G-마트 매장 중 한곳에 사기혐의가 있다.
- ✓ 특정 점장이 유효하지 않은 할인 코드를 입력하고 있다는 보고가 있다.
- ✓ 보안 전문가는 이 정보가 토퍽으로 전달되는 것을 원하지 않는다.

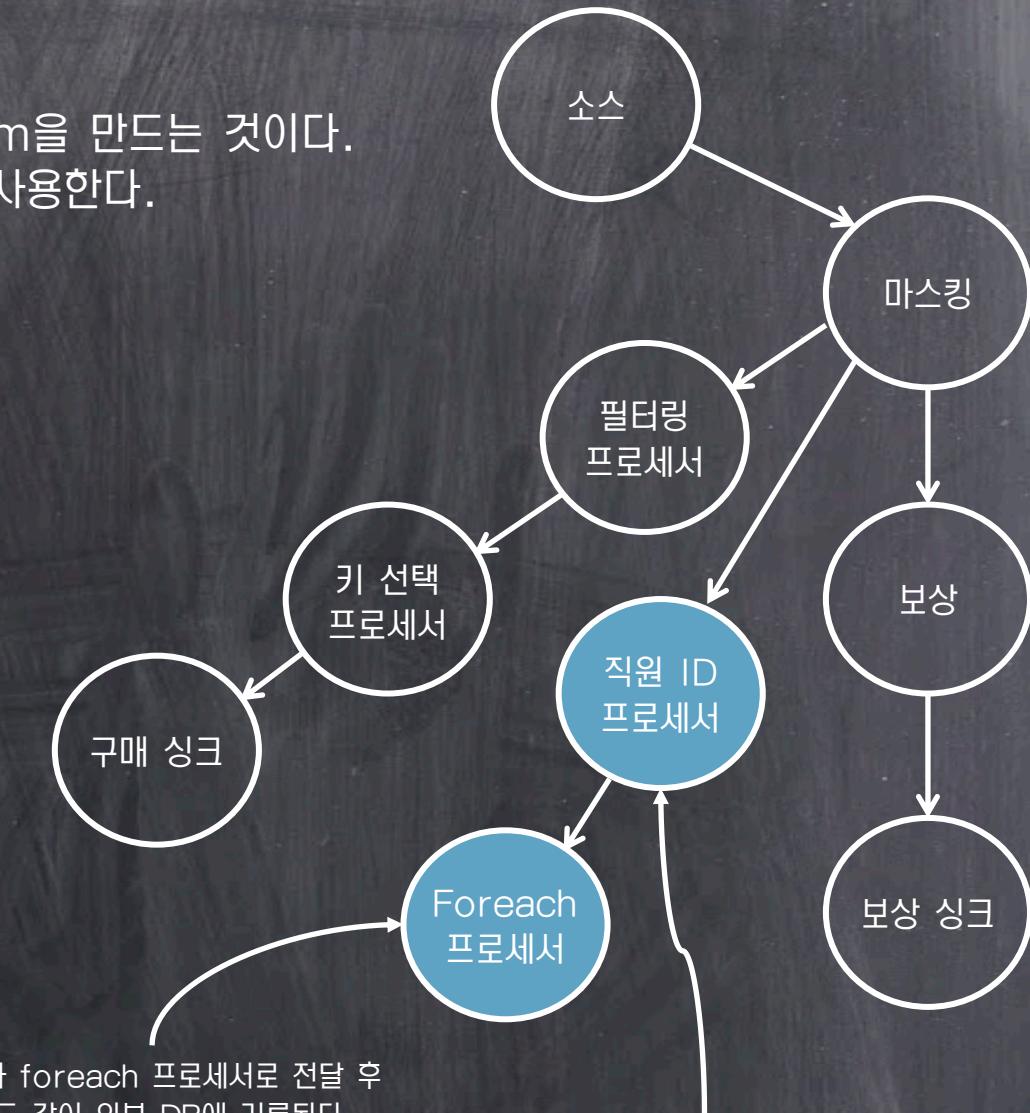


Foreach 액션

- ✓ 가장 먼저 해야할 일은 단일 직원 ID로 필터링 하는 새로운 KStream을 만드는 것이다.
- ✓ 특정 직원 ID와 일치하는지 보는 predicate와 함께 KStream을 사용한다.

```

ForeachAction<String, Purchase> purchaseForeachAction = (key, purchase) ->
    SecurityDBService.saveRecord(purchase.getPurchaseDate(),
purchase.getEmployeeId(), purchase.getItemPurchased());
purchaseKStream
    .filter((key, purchase) -> purchase.getEmployeeId().equals("000000"))
    .foreach(purchaseForeachAction);
  
```



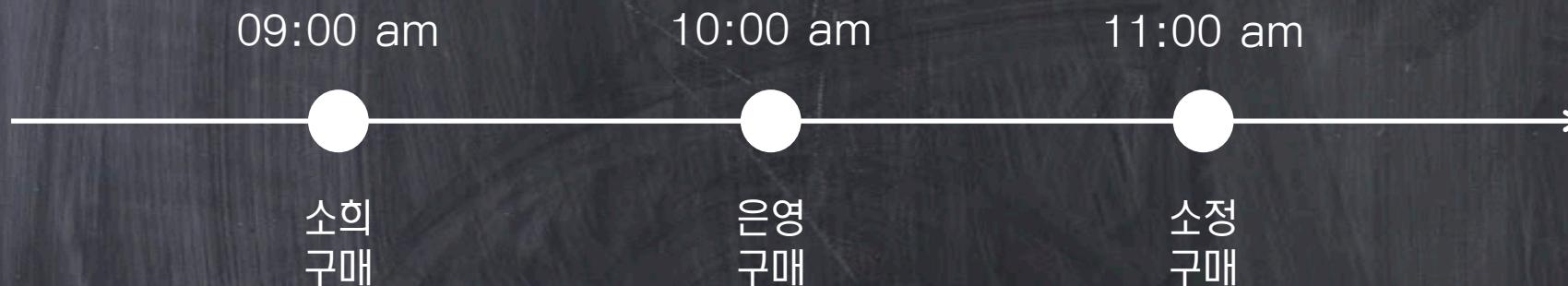
이 필터는 직원 ID가 주어진 predicate와 일치하는 레코드만 전달한다.

4. 스트림과 상태

1. Streams에 상태를 가진 작업 적용
2. 조회를 위해 상태 저장소를 사용
3. 스트림 조인
4. 시간과 타임스탬프의 역할

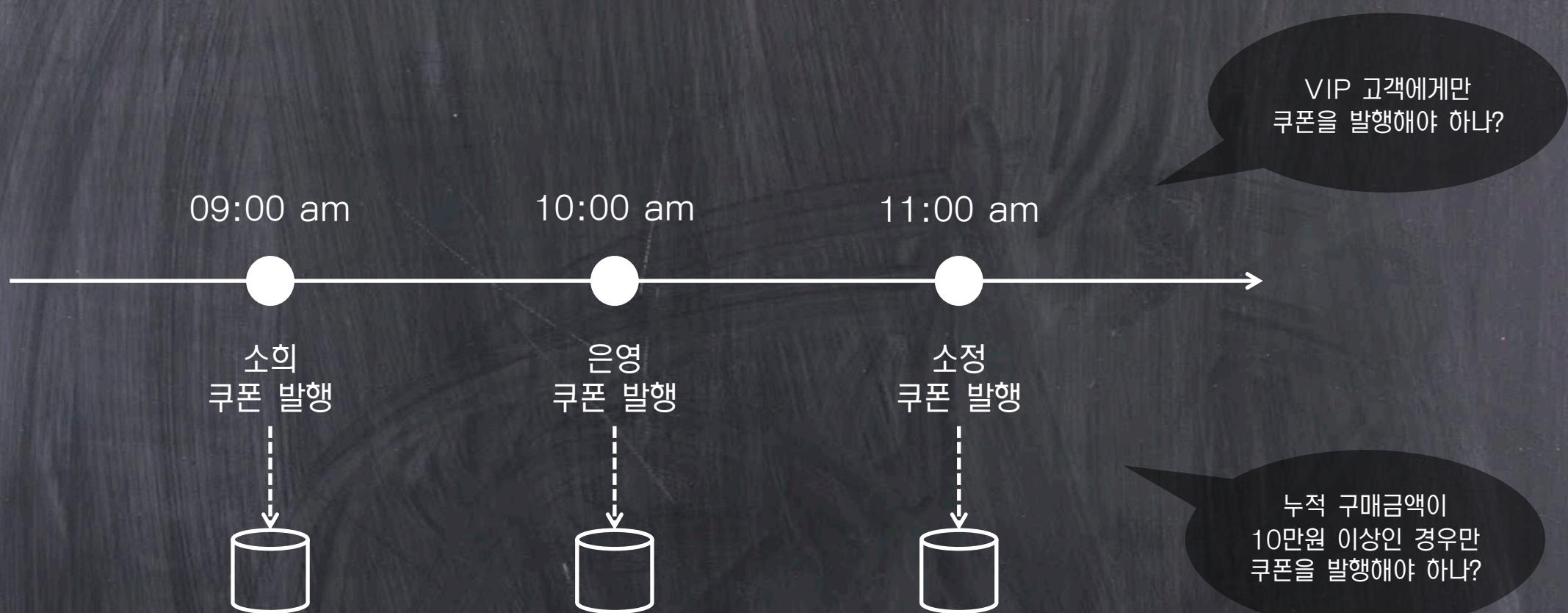
이벤트에 상태가 필요할까?

- ✓ 이벤트는 때로는 추가 정보나 문맥이 필요하지 않다.
- ✓ 앞에서 본 내용은 변환과 작업 모두 상태가 없는 1차원이었다.
즉, 트랜잭션 전후에 동시 또는 특정 시간 범위 내에서 발생하는 이벤트를 고려하지 않고 별개로 생각했다.

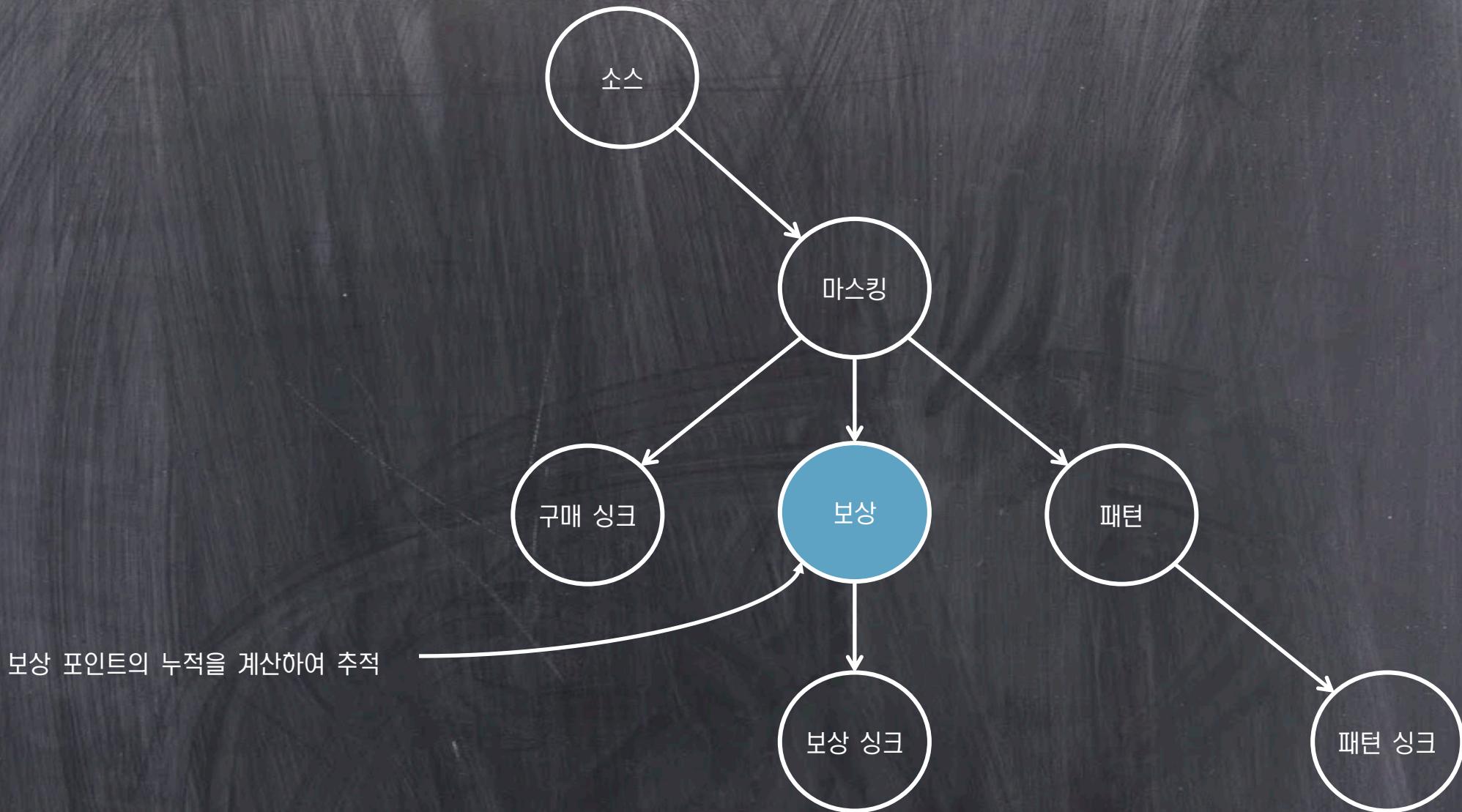


때로는 상태가 필요하다

- ✓ 이벤트를 처리할 때 때로는 상태정보를 조회해서 처리해야 할 필요성이 있다.
- ✓ 스트림 처리에서는 추가된 맵을 **상태(state)**라고 부른다.
- ✓ 상태의 개념은 데이터베이스 테이블 같은 정적 리소스의 이미지를 떠올릴 수 있다.



G-마트 보상 포인트 계산



transformValues 프로세서

가장 기본적인 상태 유지(stateful) 함수는 KStream.transformValues이다

상점에서 고객 구매를
나타내는 거래 객체 (Purchase)

- ✓ 날짜
- ✓ 구매 상품
- ✓ 고객 ID
- ✓ 매장 ID

키로 상태를 검색하고
이전에 본 데이터를 사용해
객체를 업데이트한다.

로컬 상태

인메모리
키/값 저장소

현재 데이터와 이전 데이터를 가져와
키에 의해 저장된 업데이트된
상태를 생성한다.

transformValues는 변환된 객체를
스트림의 다음 프로세서에 전달한다.
(Reward Accumulator)

transformValues
프로세서

transformValues는 변환을
수행하기 위해 로컬 상태를
사용한다.

- ✓ 고객 ID
- ✓ 총 보상 포인트
- ✓ 마지막 구매 후 경과일수

Purchase 객체를 RewardAccumulator에 매핑하기

Purchase

```
public class Purchase {

    private String firstName;
    private String lastName;
    private String customerId;
    private String creditCardNumber;
    private String itemPurchased;
    private String department;
    private String employeeId;
    private int quantity;
    private double price;
    private Date purchaseDate;
    private String zipCode;
    private String storeId;
}
```



RewardAccumulator

```
public class RewardAccumulator {
    private String customerId; // 고객 아이디
    private double purchaseTotal; // 총 구입 금액
    private int totalRewardPoints; // 총 보상 포인트
    private int currentRewardPoints; // 현재 보상 포인트
    private int daysFromLastPurchase; // 최종 구입 일자
}
```

구매 트랜잭션에 대한
Row data

고객의 누적 구매 집계 정보

PurchaseRewardTransformer

```

public class PurchaseRewardTransformer implements ValueTransformer<Purchase, RewardAccumulator> {

    private KeyValueStore<String, Integer> stateStore;
    private final String storeName;
    private ProcessorContext context;           ← 인스턴스 변수

    public PurchaseRewardTransformer(String storeName) {
        this.storeName = storeName;
    }

    @Override
    public void init(ProcessorContext context) {
        this.context = context;
        stateStore = (KeyValueStore) this.context.getStateStore(storeName);
    }

    @Override
    public RewardAccumulator transform(Purchase value) {
        RewardAccumulator rewardAccumulator = RewardAccumulator.builder(value).build();
        Integer accumulatedSoFar = stateStore.get(rewardAccumulator.getCustomerId());           ← Purchase에서 RewardAccumulator 만들기
                                                    ← 고객 ID로 최신 누적 보상 가져오기

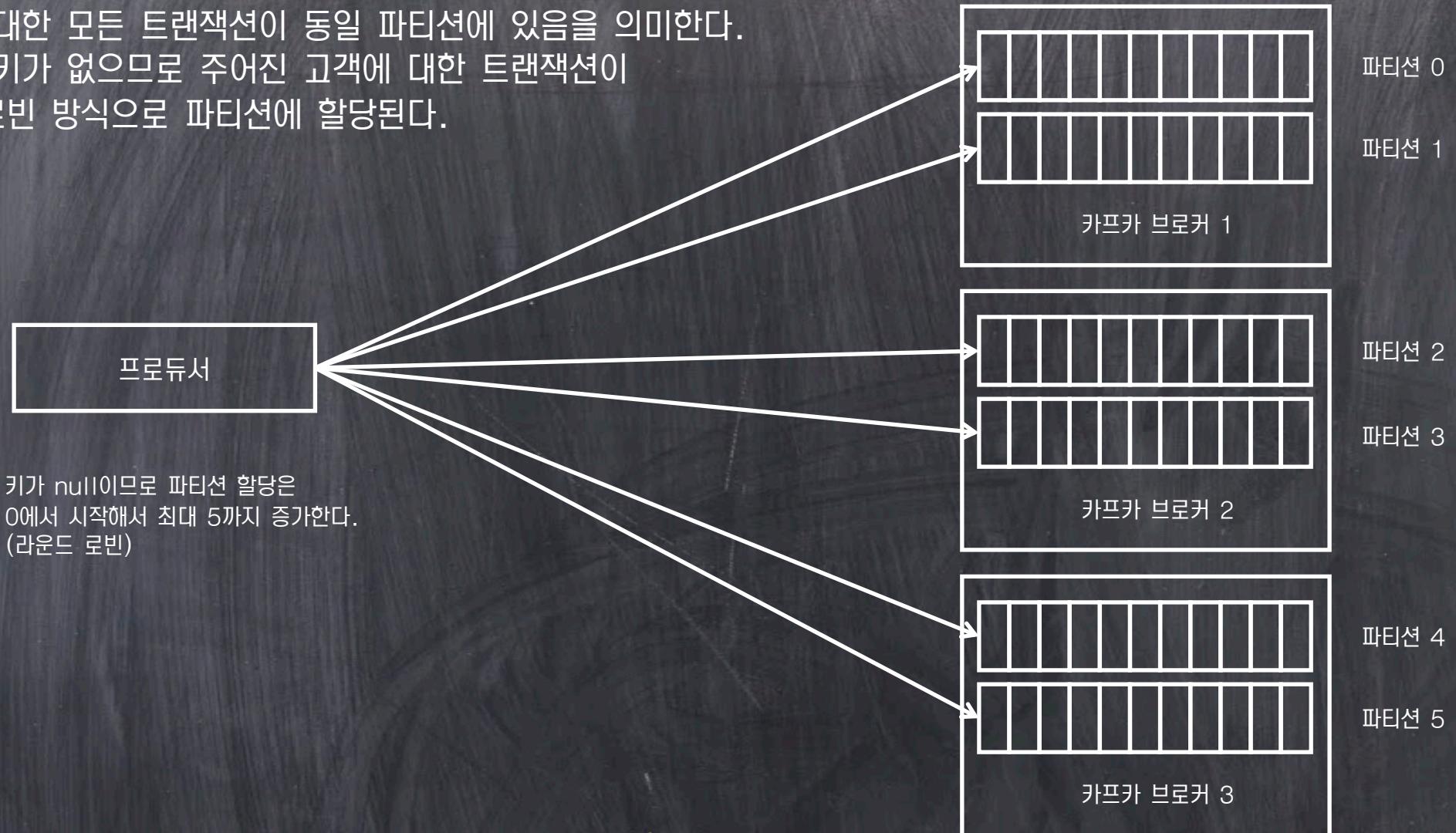
        if (accumulatedSoFar != null) {
            rewardAccumulator.addRewardPoints(accumulatedSoFar);                            ← 누적된 숫자가 있으면 현재 합계에 추가한다.
        }
        stateStore.put(rewardAccumulator.getCustomerId(), rewardAccumulator.getTotalRewardPoints());   ← 새로운 누적 보상 포인트를
                                                    ← stateStore에 저장한다.

        return rewardAccumulator;
    }
}

```

파티션이 여러 개이면?

고객에 대한 모든 트랜잭션이 동일 파티션에 있음을 의미한다.
그러나 키가 없으므로 주어진 고객에 대한 트랜잭션이
라운드로빈 방식으로 파티션에 할당된다.



이 문제를 해결하는 방법은 고객ID로 데이터를 다시 분할하는 것이다.

데이터 리파티셔닝

일반적으로 리파티셔닝은 원본 레코드의 키를 변경하거나 바꾼 다음 레코드를 새로운 토픽에 쓴다.

원본 토픽

파티션 0

```
{null, { "id": "5", "info": "123" }}
```

```
{null, { "id": "4", "info": "abc" }}
```

파티션 1

```
{null, { "id": "5", "info": "456" }}
```

```
{null, { "id": "4", "info": "def" }}
```

라파티션 토픽

파티션 0

```
{ "4", { "id": "4", "info": "def" } }
```

```
{ "4", { "id": "4", "info": "abc" } }
```

파티션 1

```
{ "5", { "id": "5", "info": "456" } }
```

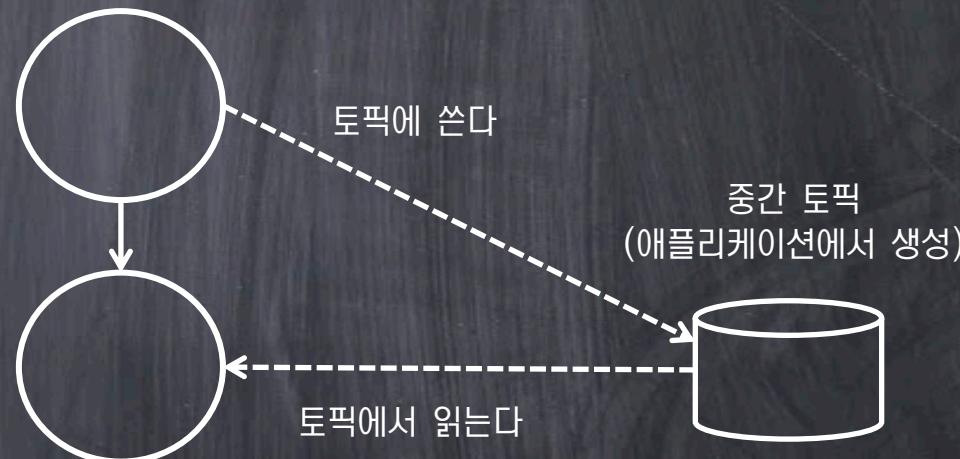
```
{ "5", { "id": "5", "info": "123" } }
```



Kafka Streams 리파티셔닝

Kafka Streams에서 리파티셔닝은 KStream.through()를 사용해 쉽게 수행할 수 있다.

‘through’ 출력을 하는
원본 KStream 노드



```

RewardsStreamPartitioner streamPartitioner = new RewardsStreamPartitioner();
KStream<String, Purchase> transByCustomerStream
    = purchaseKStream.through( "customer_transactions",
        Produced.with(stringSerde, purchaseSerde, streamPartitioner));
  
```

KStream.through로
KStream을 생성한다

StreamPartitioner 사용하기

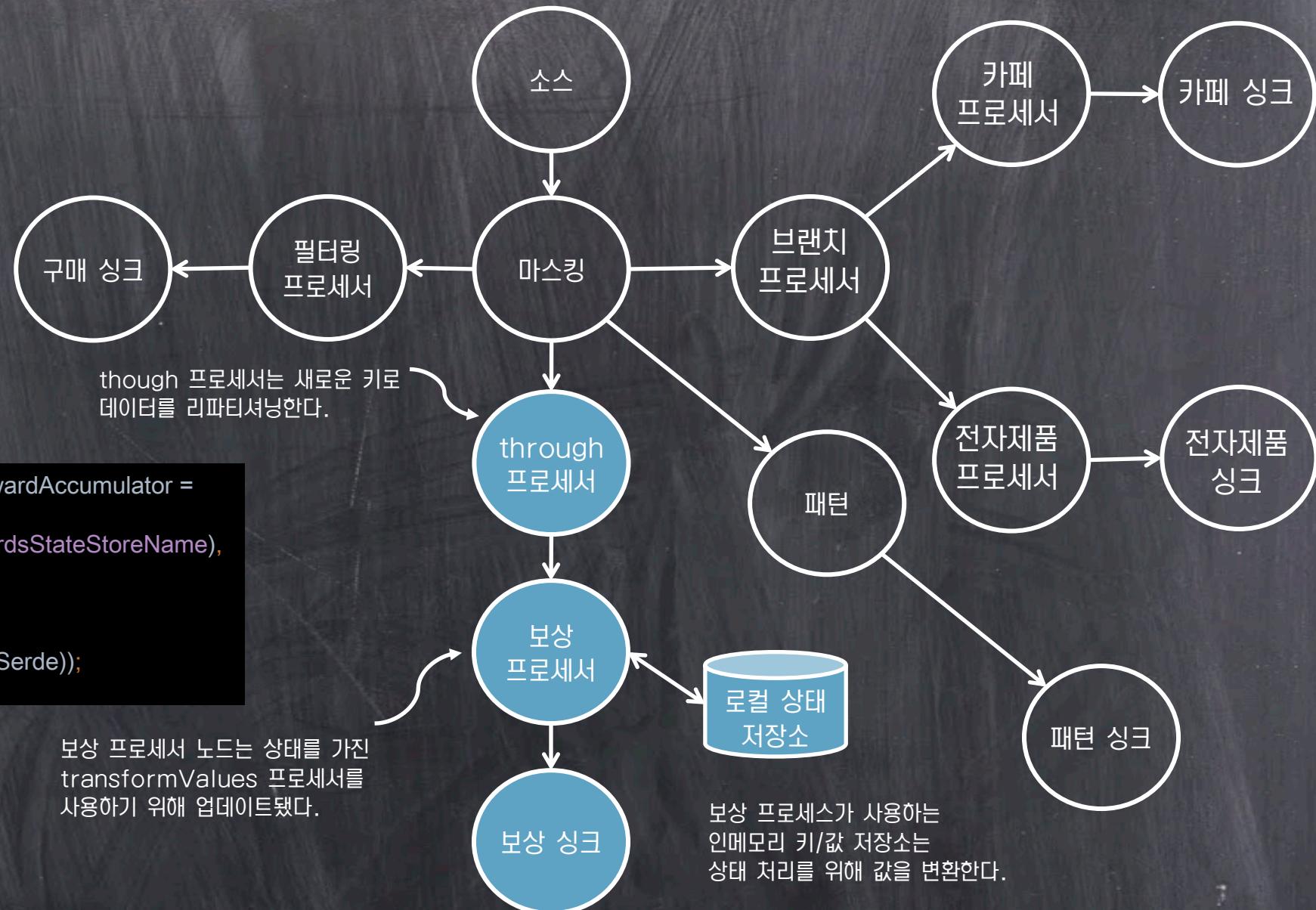
고객 ID를 사용해 특정 고객의 모든 데이터가 동일한 상태 저장소에 저장되게 함

RewardStreamPartitioner

```
public class RewardsStreamPartitioner implements StreamPartitioner<String, Purchase> {  
    @Override  
    public Integer partition(String key, Purchase value, int numPartitions) {  
        return value.getCustomerId().hashCode() % numPartitions;  
    }  
}
```

고객 ID로 파티션을 결정한다.

보상 프로세서 업데이트



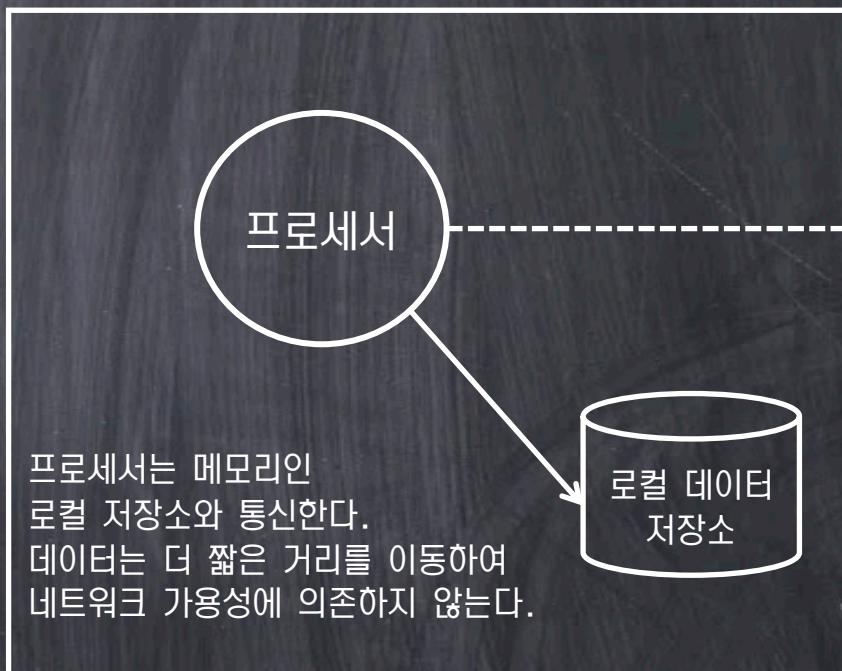
상태의 중요한 속성

1. 데이터 지역성 (data locality)
2. 실패 복구 (failure recovery)

데이터 지역성

- ✓ 데이터 지역성은 성능에 매우 중요하다.
- ✓ 키 조회는 일반적으로 매우 빠르지만 원격 저장소를 사용하면서 발생하는 대기시간(latency)은 병목이 될 수 있다.

서버



프로세서는 원격 저장소와 통신해야 한다.
데이터는 더 멀리 이동해야 하며
네트워크 가용성이 필요하다.

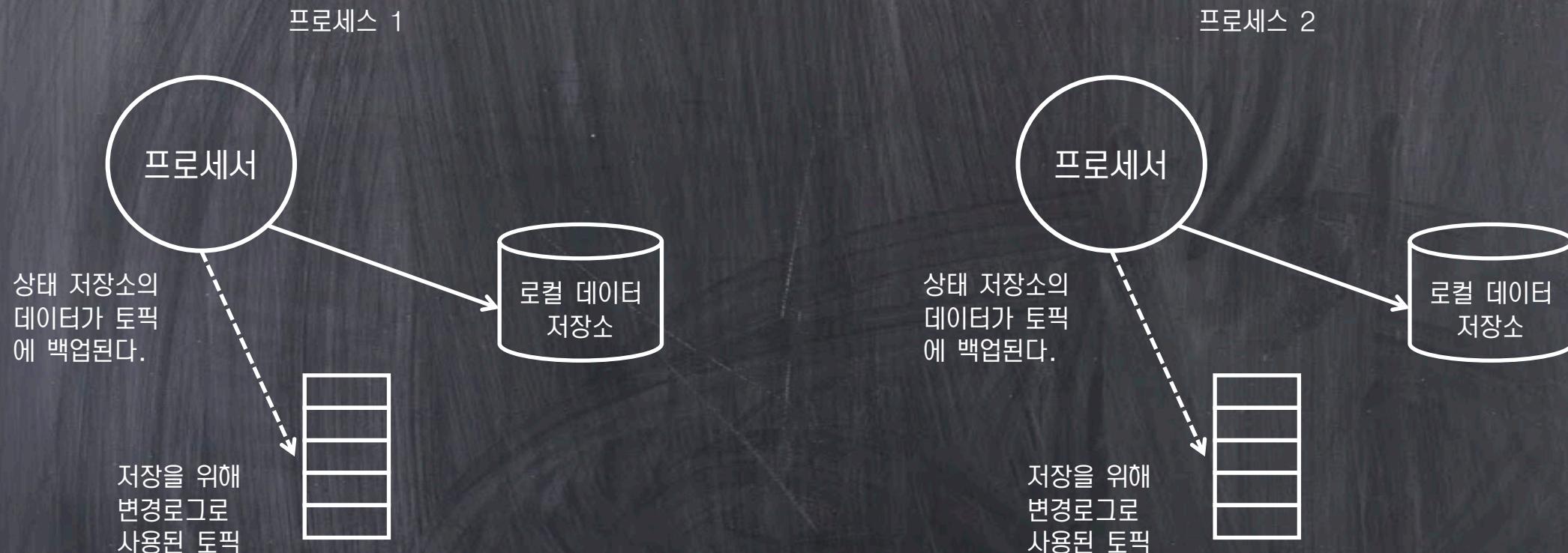
원격환경에서는 수백만 또는 수십억 개의
레코드를 처리할 경우
네트워크 지연에 큰 영향을 받는다.



데이터 지역성은 스트림 처리에 필요하다.
애플리케이션의 각 서버나 노드는 개별 데이터 저장소가 있어야 한다.

실패 복구와 내결함성

- ✓ 분산 애플리케이션에서 장애는 불가피하다.
- ✓ 실패를 예방하는 대신 실패나 재시작에서 신속하게 복구하는 데 중점을 둘 필요가 있다.



프로세스는 자체 로컬 상태 저장소와 비공유 아키텍처가 있기 때문에 두 프로세스 중 하나가 실패하면 영향을 받지 않을 것이다. 또한 각 저장소는 토픽에 복제된 키/값을 가지며 **프로세서가 실패하거나 다시 시작할 때 읽어버린 값을 복구하는 데 사용한다.**

Kafka Streams에서 상태 저장소 사용하기

- ✓ 상태 저장소를 추가하기 위한 두 가지 클래스
- ✓ Materialized와 StoreBuilder
- ✓ 고수준 DSL인 경우 Materialized 클래스 사용
- ✓ 저수준 프로세서 API는 StoreBuilder를 사용

```
String rewardsStateStoreName = "rewardsPointsStore";
KeyValueBytesStoreSupplier storeSupplier = Stores.inMemoryKeyValueStore(rewardsStateStoreName);

StoreBuilder<KeyValueStore<String, Integer>> storeBuilder =
    Stores.keyValueStoreBuilder(storeSupplier, Serdes.String(), Serdes.Integer());

builder.addStateStore(storeBuilder);
```

상태 저장소를 토플로지에 추가한다.

StateStore 공급자를 생성한다.

StoreBuilder를 생성하고
키와 값의 타입을 명시한다.

변경로그 토픽 설정하기

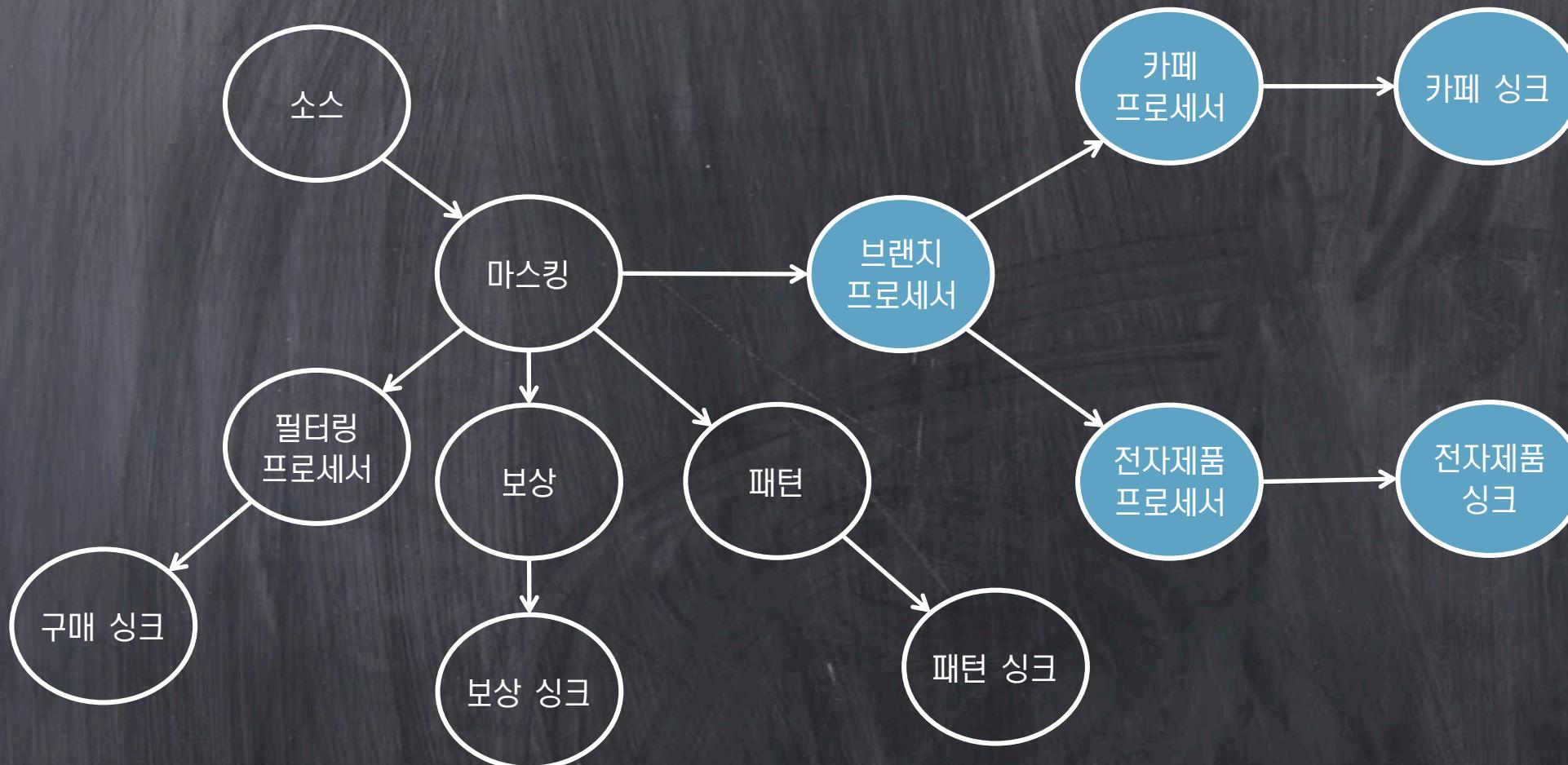
- ✓ 모든 StateStoreSupplier 타입은 기본적으로 로깅이 활성화되어 있다.
- ✓ 로깅은 저장소의 값을 백업하고 내결함성을 제공하기 위한 변경로그로 사용되는 카프카 토픽을 의미한다.
- ✓ 상태 저장소에 대한 변경로그는 withLoggingEnabled<Map<String, String> config) 메소드를 통해 설정 가능하다.
- ✓ 카프카 토픽을 사용하면 기본 설정은 일주일이고 크기는 무제한이다.

스트림 조인하기

- ✓ 스트림은 이벤트가 독립적이지 않을 때 상태가 필요하다.
- ✓ 또는 새로운 이벤트를 만들기 위해 동일한 키를 이용해 스트림 2개에서 각기 다른 이벤트를 가져와서 결합한다. ← 조인

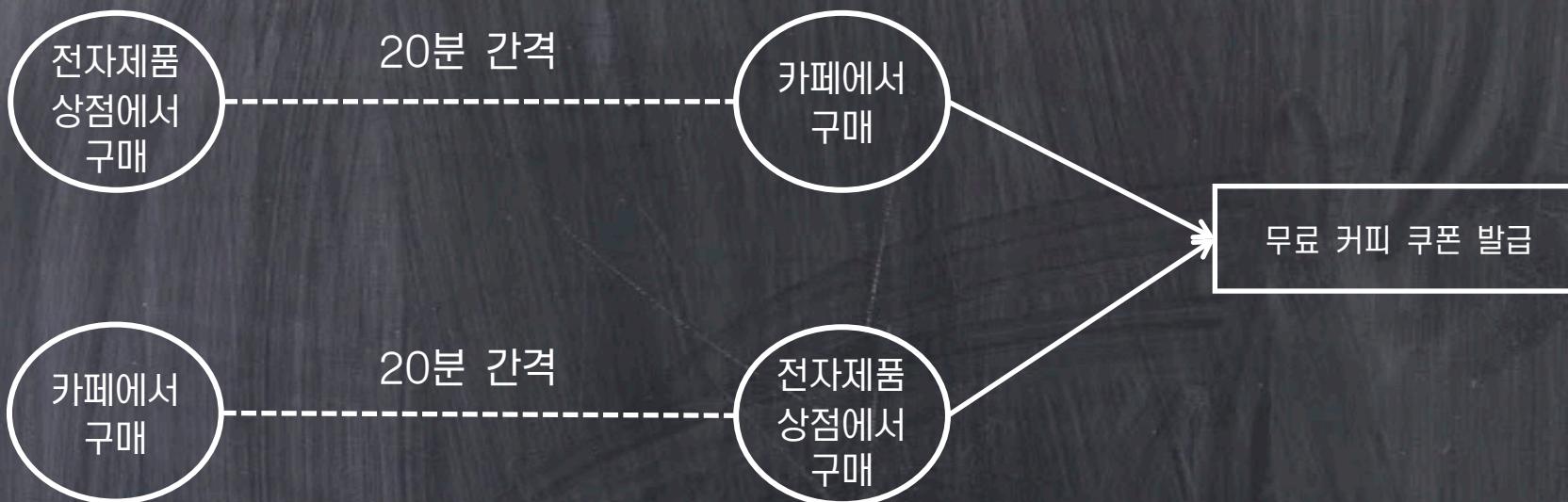
스트림 조인 예제

G-마트는 전자제품 매장을 열고 각 매장에 카페를 설치했다.
구매 트랜잭션을 각기 다른 스트림 2개로 분리하라는 요청을 받았다.



쿠폰 발행

G-마트는 커피를 사고 전자제품 상점에서 구매한 고객을 확인해서 두 번째 거래 직후 쿠폰을 지급하기 원한다. 쿠폰 발행 시기를 결정하려면 전자제품 상점의 판매와 카페의 판매를 조인해야 한다.



데이터 설정

- ✓ 브랜치 프로세서는 두 가지 스트림을 생성해 이를 처리한다.
- ✓ 하나는 카페 구매를 포함하고, 다른 하나는 전자제품 구매를 포함한다.

이 프로세서는 predicate 배열을 포함하고
주어진 predicate와 일치하는 레코드만 허용한다.



2개의 스트림 생성

레코드 매치를 위한 Predicate 정의하기

```
Predicate<String, Purchase> coffeePurchase  
    = (key, purchase) -> purchase.getDepartment().equalsIgnoreCase("coffee");  
Predicate<String, Purchase> electronicPurchase  
    = (key, purchase) -> purchase.getDepartment().equalsIgnoreCase("electronics");  
  
int COFFEE_PURCHASE = 0;  
int ELECTRONICS_PURCHASE = 1;  
  
KStream<String, Purchase>[] branchesStream = transactionStream.branch(coffeePurchase, electronicPurchase);
```

분기된 스트림 생성하기

고객 ID를 포함한 키 생성

조인을 수행하기 위해 고객 ID를 포함한 키 생성하기

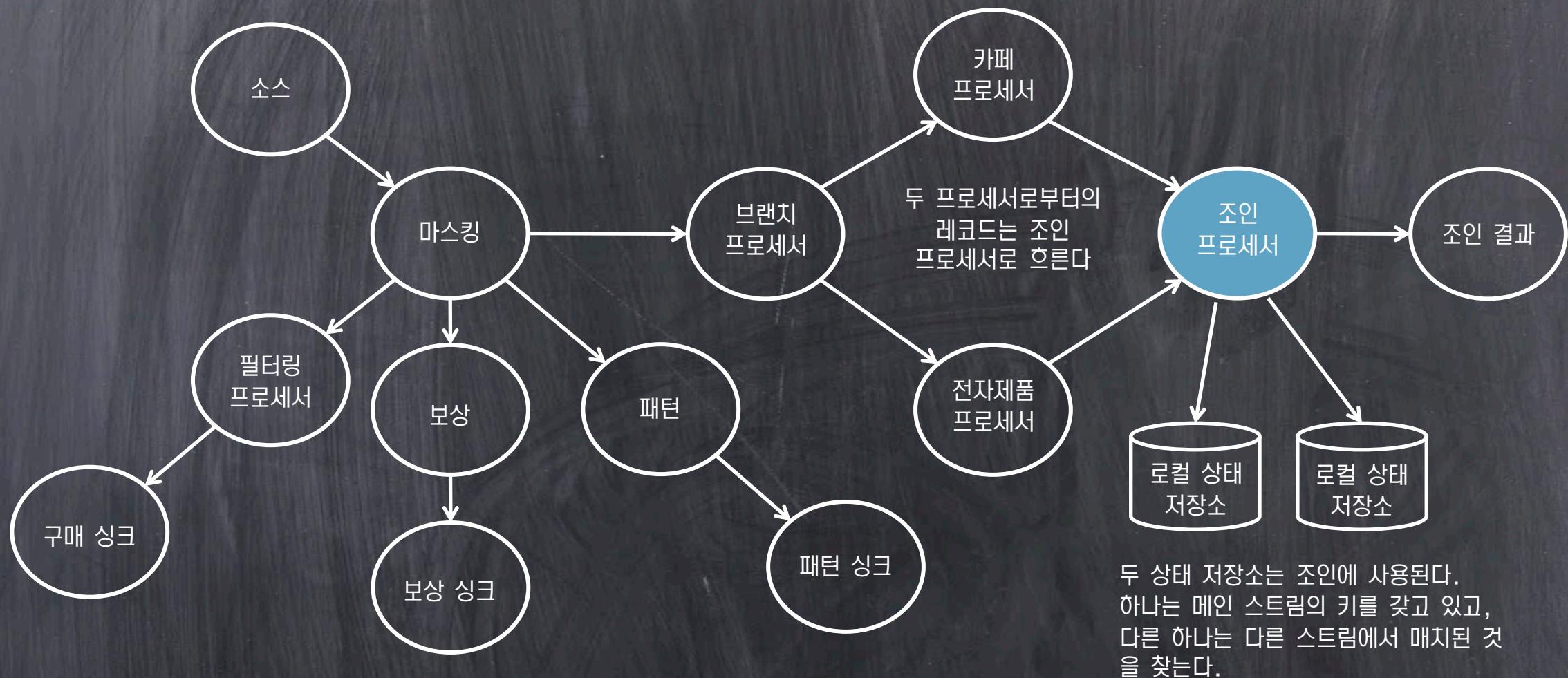
```
KStream<String, Purchase>[] branchesStream  
    = transactionStream.selectKey((k, v)-> v.getCustomerId()).branch(coffeePurchase, electronicPurchase);
```



selectKey 처리노드를 삽입한다.
selectKey를 사용하면 자동으로 리파티셔닝을 처리한다.

조인 구성하기

- ✓ 다음 단계는 분기된 스트림 2개를 가져와서 KStream.join() 메소드로 조인한다.



구매 레코드 조인하기

- ✓ 조인된 레코드를 만들려면 ValueJoiner의 인스턴스를 생성해야 한다.

PurchaseJoiner

```
public class PurchaseJoiner implements ValueJoiner<Purchase, Purchase, CorrelatedPurchase> {
    public CorrelatedPurchase apply(Purchase purchase, Purchase otherPurchase) {
        CorrelatedPurchase.Builder builder = CorrelatedPurchase.newBuilder();

        Date purchaseDate = purchase != null ? purchase.getPurchaseDate() : null;
        Double price = purchase != null ? purchase.getPrice() : 0.0;
        String itemPurchased = purchase != null ? purchase.getItemPurchased() : null;

        Date otherPurchaseDate = otherPurchase != null ? otherPurchase.getPurchaseDate() : null;
        Double otherPrice = otherPurchase != null ? otherPurchase.getPrice() : 0.0;
        String otherItemPurchased = otherPurchase != null ? otherPurchase.getItemPurchased() : null;

        List<String> purchasedItems = new ArrayList<>();
        if (itemPurchased != null) {
            purchasedItems.add(itemPurchased);
        }

        if (otherItemPurchased != null) {
            purchasedItems.add(otherItemPurchased);
        }

        String customerId = purchase != null ? purchase.getCustomerId() : null;
        String otherCustomerId = otherPurchase != null ? otherPurchase.getCustomerId() : null;

        builder.withCustomerId(customerId != null ? customerId : otherCustomerId)
            .withFirstPurchaseDate(purchaseDate)
            .withSecondPurchaseDate(otherPurchaseDate)
            .withItemsPurchased(purchasedItems)
            .withTotalAmount(price + otherPrice);
        return builder.build();
    }
}
```

조인 구현

- ✓ 구매가 서로 20분 이내 이뤄져야한다.
- ✓ 순서는 포함되지 않는다.

```
KStream<String, Purchase> coffeeStream = branchesStream[COFFEE_PURCHASE];
KStream<String, Purchase> electronicsStream = branchesStream[ELECTRONICS_PURCHASE];

ValueJoiner<Purchase, Purchase, CorrelatedPurchase> purchaseJoiner = new PurchaseJoiner();
JoinWindows twentyMinuteWindow = JoinWindows.of(60 * 1000 * 20);

KStream<String, CorrelatedPurchase> joinedKStream = coffeeStream.join(electronicsStream,
    purchaseJoiner,
    twentyMinuteWindow,
    Joined.with(stringSerde,
        purchaseSerde,
        purchaseSerde));
    
```

ValueJoiner 구현

조인할 전자 구매 스트림

타임스탬프가 서로 20분 이내에 있어야 한다.

```
joinedKStream.print(Printed.<String, CorrelatedPurchase>toSysOut().withLabel("joined KStream"));
```

코파티셔닝

- ✓ 조인을 수행하려면 모든 조인 참가자가 코파티셔닝(co-partitioning)되어 있음을 보장해야 한다.
- ✓ 이는 같은 수의 참가자가 있고 같은 타입의 키가 있음을 의미한다.
- ✓ 조인과 관련된 토픽이 동일한 수의 파티션을 갖는지 확인한다. 불일치가 발견되면 TopologyBuilderException이 발생한다.
- ✓ 코파티셔닝은 소스 토픽에 기록할 때 모든 카프카 프로듀서가 동일한 파티셔닝 클래스를 사용하도록 요구한다.

외부 조인

- ✓ 외부 조인(outer join)은 항상 레코드를 출력하며, 조인에서 명시한 두 이벤트가 모두 포함되지 않을 수 있다.

outerJoin

`coffeeStream.outerJoin(electronicsStream, ...)`

스트림의 이벤트만 타임 윈도에서 사용 가능해서, 포함된 유일한 레코드이다.



두 스트림의 이벤트 모두 타임 윈도에서 사용 가능해서, 둘 다 조인 레코드에 포함된다.



다른 스트림의 이벤트만 타임 윈도에서 사용 가능해서, 포함된 유일한 레코드이다.

왼쪽 외부 조인

- ✓ 조인 원도에서 다른 스트림에서만 사용 가능한 이벤트가 발생하면 출력이 없다.

leftJoin

`coffeeStream.leftJoin(electronicsStream, ...)`

호출한 스트림의 이벤트만 타임 윈도에서 사용 가능해서, 포함된 유일한 레코드이다.

→ (커피구매, null)



두 스트림의 이벤트 모두 타임 윈도에서 사용 가능해서, 둘 다 조인 레코드에 포함된다.

→ (커피구매, 전자제품 구매)



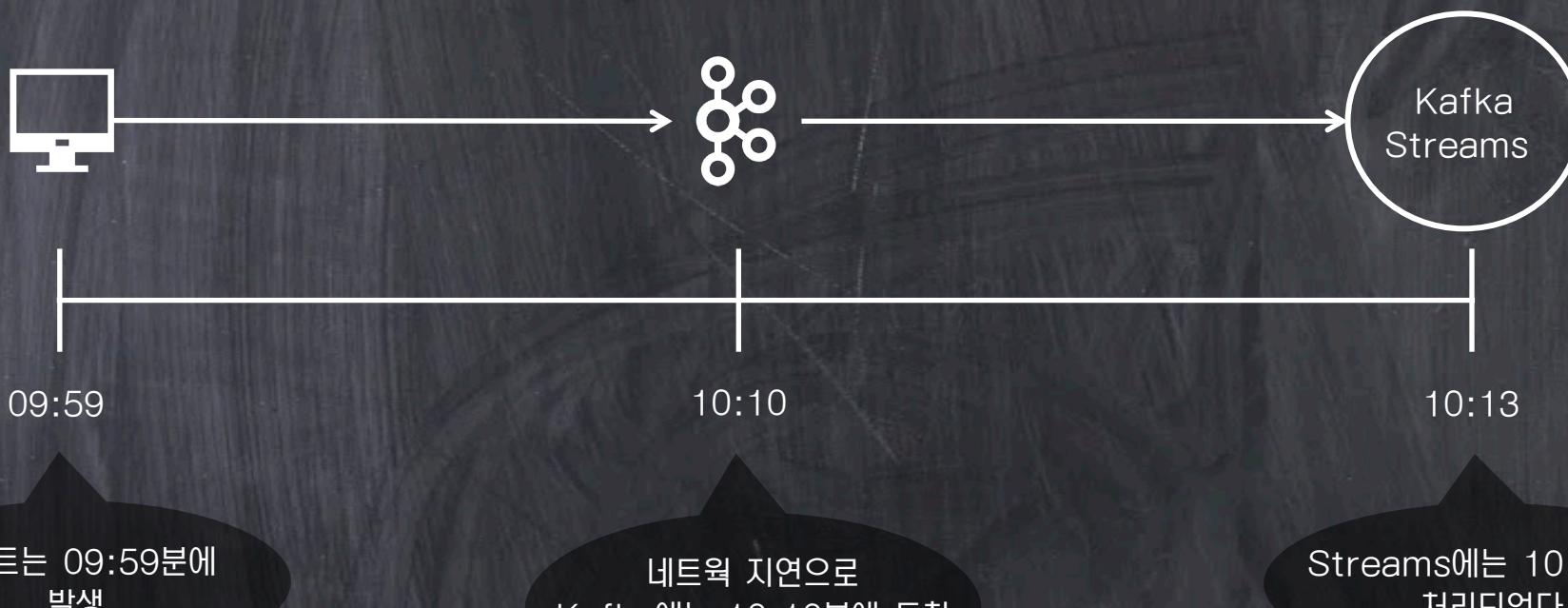
다른 스트림의 이벤트만 타임 윈도에서 사용 가능해서, 다운스트림에 아무것도 보내지 않는다.

→ 출력 없음

Kafka Stream의 타임스탬프

- ✓ 이전 장의 조인에서 사용한 20분은 어떤 시간인가?
- ✓ 토픽이 이벤트 시간? 처리시간?

어느 시간으로 계산이
되어야 하나?



Kafka Stream의 타임스탬프

✓ 이벤트 시간(event time)

- 이벤트가 발생했을 때 설정한 타임스탬프
- 보통 이벤트를 나타내는 데 사용된 객체에 포함된다.

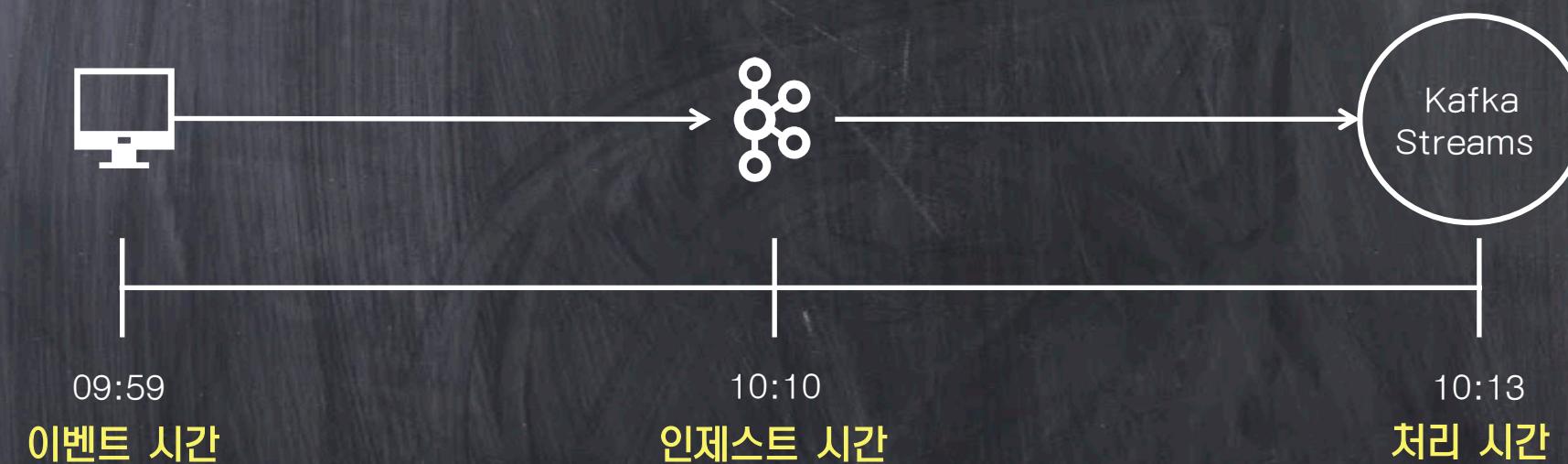
스트림 처리에서 타임스탬프를 세 가지 범주로 나눌 수 있다.

✓ 인제스트 시간(ingestion time)

- 데이터가 처음 데이터 처리 파이프라인에 들어갈 때 설정되는 타임스탬프
- 카프카 브로커가 설정한 타임스탬프를 인제스트 시간으로 생각할 수 있다.

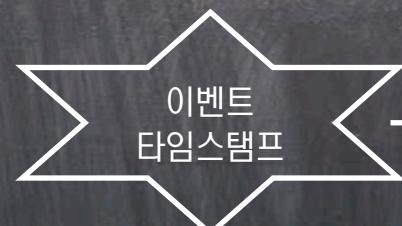
✓ 처리 시간(processing time)

- 데이터나 이벤트가 처음 처리 파이프라인을 통과하기 시작할 때 설정된 타임스탬프

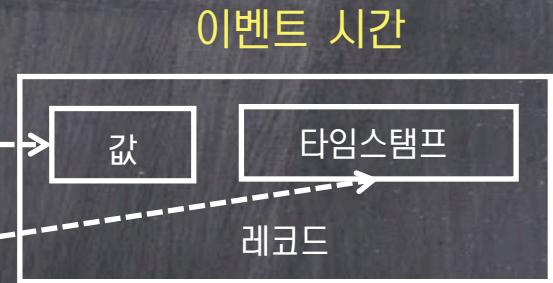


4. 스트림과 상태 > 4. 시간과 타임스탬프의 역할

이벤트 발생 시 데이터 객체에 포함된
타임스탬프 또는 카프카 프로듀서가
ProducerRecord에 설정한 타임스탬프



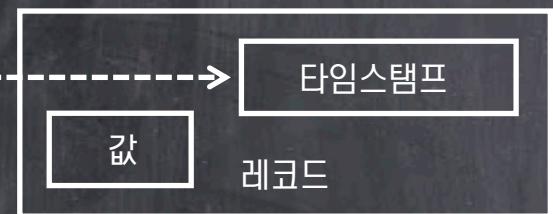
또는



타임 레코드에 설정된 타임스탬프는
로그(토픽)에 추가된다.



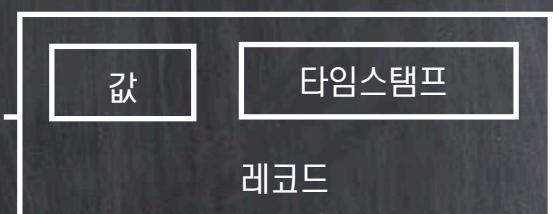
인제스트 시간



레코드가 소비될 때 생성된 타임스탬프 데이터 객체와 ConsumerRecord에 포함된 타임스탬프는 무시한다.



처리 시간

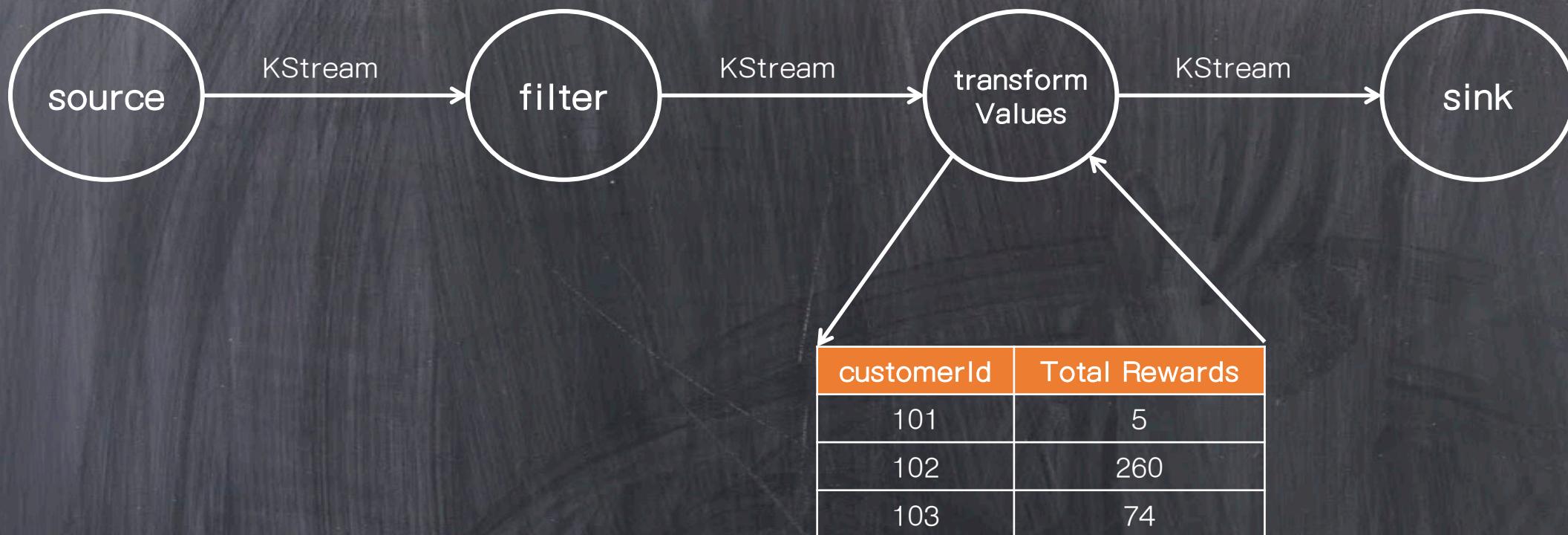


레코드가 소비될 때 생성된 타임스탬프
벽 시간(wall-clock time)

5. KTable API

1. 스트림과 테이블 간의 관계 정의
2. 레코드 업데이트와 KTable 결합
3. 집계와 윈도 작업
4. GlobalKTable

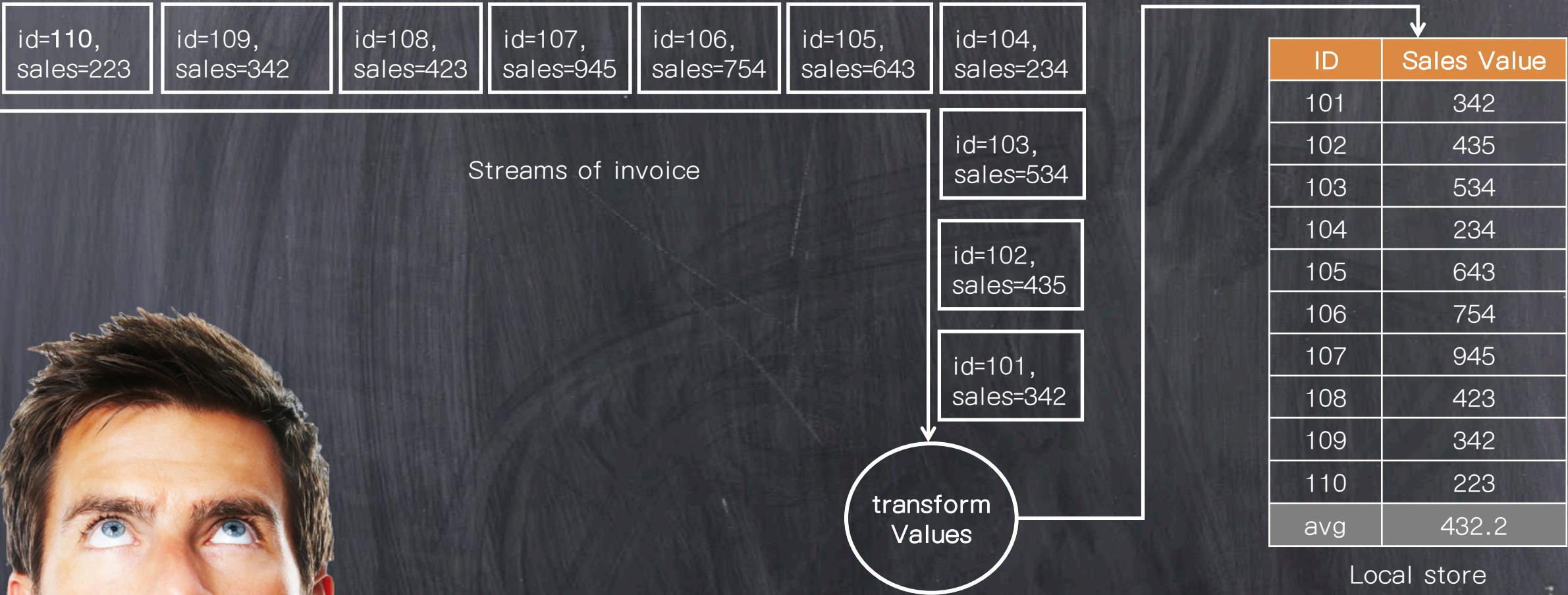
앞장에서 스트림 처리 시 이전 상태를 가져오려면 state를 사용했다



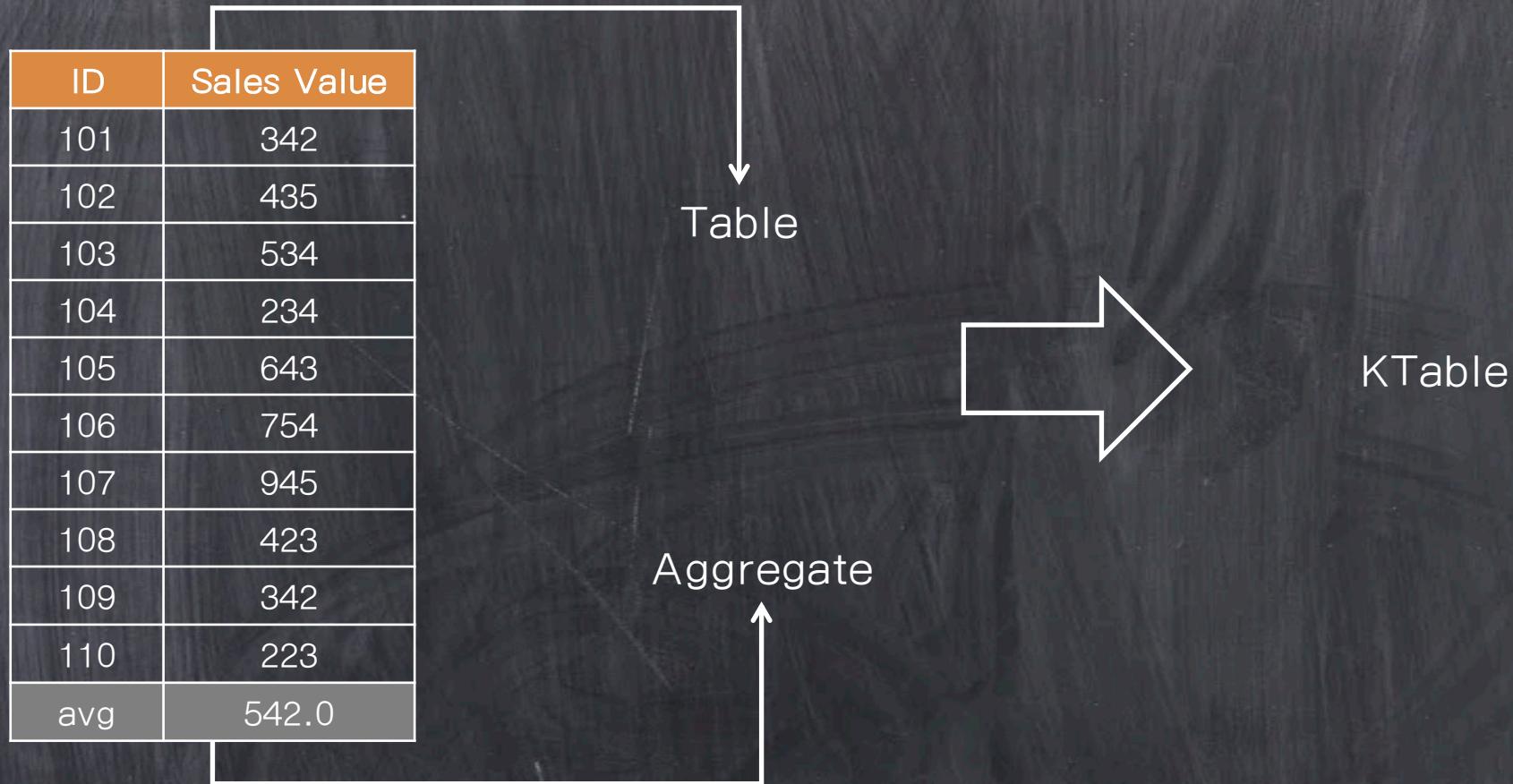
그럼 한시간의 동안 판매 평균 금액은 어떻게 구할까?

✓ 시나리오

- 9시~10시까지 총 10개의 판매 송장이 있다.
- 1시간 동안의 평균 판매 금액은 얼마인가?



이를 위해 필요한 것은 시간으로 데이터를 grouping하는 것이다.
Table과 Aggregate의 역할을 하는 것이 KTable이다.



KTable API

- ✓ KTable API는 레코드 업데이트 작업을 위해 설계됐다.
- ✓ 집계 카운트 같은 기능을 위해 업데이트 작업은 필요하다.
- ✓ 윈도는 주어진 기간에 대해 버킷 데이터를 처리한다.
예를 들어, 10분마다 최근 1시간 동안의 구매량을 집계할 경우다.

레코드 스트림

일련의 주가 업데이트를 보려고 한다. 각 주가 시세는 개별 이벤트이고 서로 관련이 없다는 사실을 알 수 있다. 이 이벤트 뷰는 KStream이 작동하는 방식인 레코드의 스트림이다.



스트림과 테이블의 관계

두 회사의 주식이긴 하지만 이 스트림의 각 항목을 하나의 이벤트로 생각하기 때문에 4개의 이벤트로 취급한다.
이로 인해, 각 이벤트는 삽입하고 테이블에 입력할 때마다 키 값이 증가한다.



키	Stock_ID	타임스탬프	주가
---	----------	-------	----

1	naver	148907726274	₩ 298,500
2	kakao	148907726274	₩ 369,000
3	naver	148907726474	₩ 300,000
4	kakao	148907726774	₩ 400,000

레코드 및 변경로그 업데이트

동일 고객 거래 스트림을 가져와보자.

변경 로그에서 유입 레코드는 같은 키를 가진 이전의 레코드를 덮어쓴다. 이 레코드 스트림은 총 4개의 이벤트가 있었지만 업데이트나 변경로그의 경우에는 2개만 남는다.



Stock_ID	타임스탬프	주가
naver	148907726274	₩ 298,500
kakao	148907726274	₩ 369,000
naver	148907726474	₩ 300,000
kakao	148907726774	₩ 400,000

이전 레코드는 업데이트로 덮어썼다.

이벤트 스트림의 최신 레코드

이벤트 스트림과 업데이트 스트림 비교

여기에서 KStream
이벤트/레코드가
3건씩 있다.

여기에서 KTable의
마지막 업데이트
레코드가 있다.

```

INITIALIZING THE PRODUCER
Producer initialized
KTable vs KStream output started
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.25, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.19, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=91.97, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=105.74, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=53.78, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.53, symbol='NDLE'}
Stock updates sent
[Stocks-KStream]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KStream]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KStream]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
[Stocks-KTable]: YERB , StockTickerData{price=106.67, symbol='YERB'}
[Stocks-KTable]: AUNA , StockTickerData{price=54.4, symbol='AUNA'}
[Stocks-KTable]: NDLE , StockTickerData{price=92.77, symbol='NDLE'}
Shutting down the Kafka Streams Application now
Shutting down data generation

```

예상처럼 마지막 KStream
이벤트와 KTable 업데이트 값은 같다.

주식 알림 프로그램

```
KTable<String, StockTickerData> stockTickerTable = builder.table(STOCK_TICKER_TABLE_TOPIC);  
KStream<String, StockTickerData> stockTickerStream = builder.stream(STOCK_TICKER_STREAM_TOPIC);  
  
stockTickerTable.toStream().print(Printed.<String, StockTickerData>toSysOut().withLabel("Stocks-KTable"));  
stockTickerStream.print(Printed.<String, StockTickerData>toSysOut().withLabel( "Stocks-KStream"));
```

KTabe 인스턴스 생성
KStream 인스턴스 생성

레코드 업데이트와 KTable 구성

KTable 기능을 이해하기 위해 다음과 같은 두 가지 질문을 할 수 있다.

- ✓ 레코드를 어디에 저장하는가?
- ✓ KTable은 레코드를 내보내는(emit) 결정을 어떻게 하는가?

Q1. 레코드를 어디에 저장하는가?

```
builder.table(STOCK_TICKER_TABLE_TOPIC);
```

streamBuilder는 KTable 인스턴스를 만들고
동시에 그 내부에 스트림 상태를 추적하는 상태 저장소(StateStore)를 만든다.

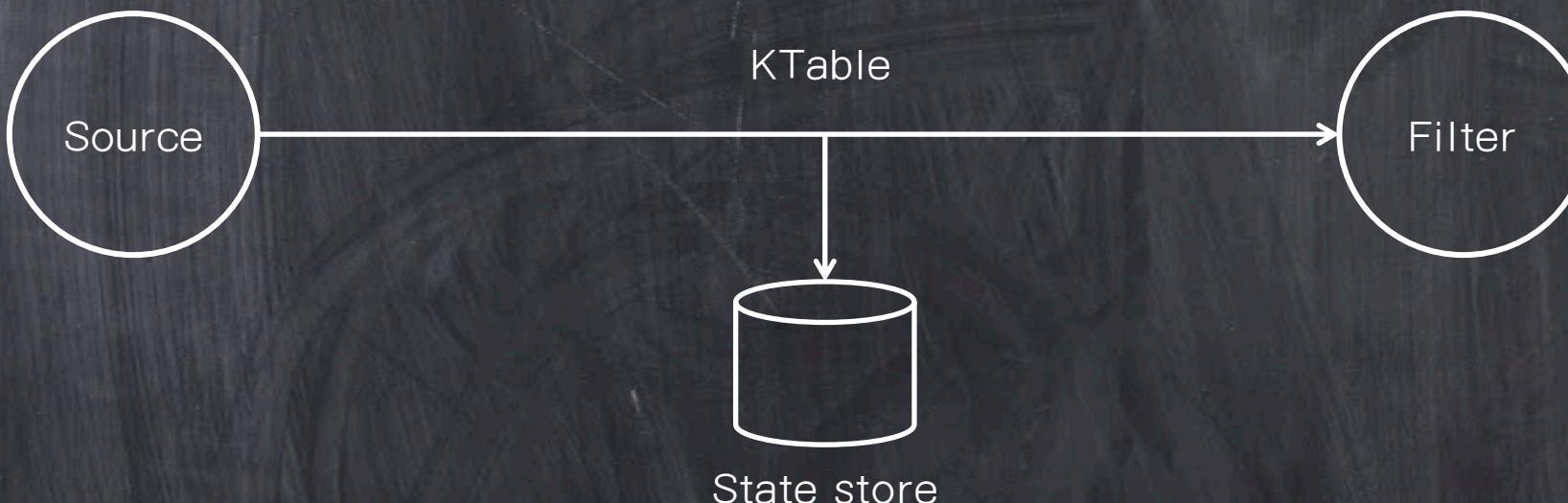
레코드를 어디에 저장하는가?

→ 첫 질문에 대한 답은 KTable은 Kafka Streams와 통합된 로컬 상태 저장소를 저장 공간으로 사용한다는 것이다.

Q2. KTable은 레코드를 내보내는 결정은 어떻게 하는가?

이 질문에 대한 답을 찾기 위해 다음과 같은 사항을 고려해봐야 한다.

- ✓ 애플리케이션에 유입되는 레코드 수. 높은 데이터 유입률은 업데이트된 레코드를 내보내는 비율을 높일 수 있다.
- ✓ 데이터에 구별되는 키가 얼마나 많은가? 구별되는 키가 많다면 다운스트림에 더 많은 업데이트를 보내게 된다.
- ✓ cache.max.bytes.buffering과 commit.interval.ms 구성 매개변수



캐시 버퍼 크기 설정하기

- ✓ KTable 캐시는 같은 키가 있는 레코드의 중복을 의미한다.
- ✓ 캐시를 활성화하면 모든 레코드 업데이트가 다운스트림에 전달되지 않는다.
- ✓ 큰 캐시는 내보낼 업데이트 수를 줄여줄 것이다.
- ✓ 캐시 크기는 cache.max.bytes.buffering 설정으로 레코드 캐시에 할당할 메모리 총량을 제어한다.

유입되는 주식 시세 레코드

kakao	400,000
-------	---------

레코드가 들어오면, 캐시에 있던
기존 레코드를 새 레코드로 교체한다.

캐시

kakao	400,000
naver	300,000
samsung	60,000

커밋 주기 설정하기

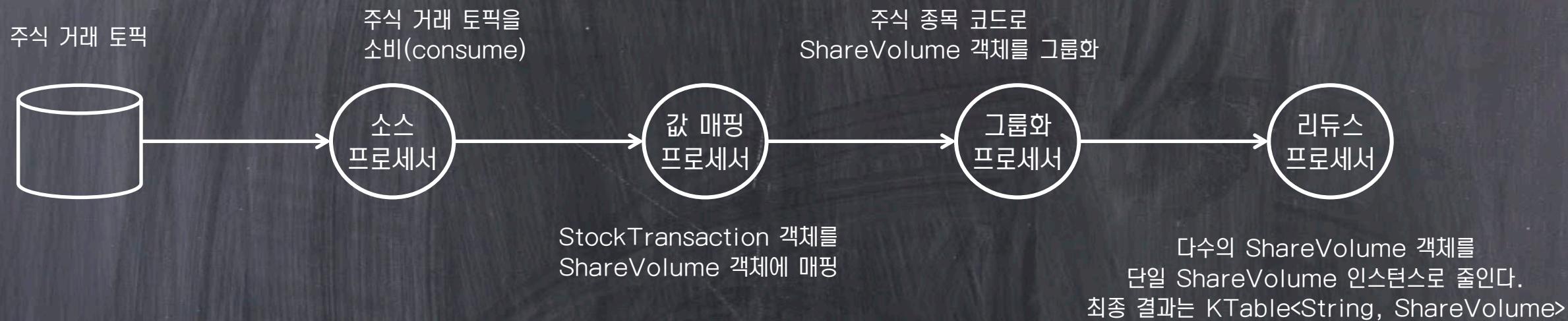
- ✓ 프로세서 상태를 얼마나 자주 저장할 지 지정한다. (commit.interval.ms)
- ✓ 프로세서 상태를 저장할 때, 캐시를 강제로 비우고 중복 제거된 마지막 업데이트 레코드를 다운스트림에 전송한다.

집계와 윈도 작업

스트리밍 데이터가 유입될 때 가끔은 단일 레코드를 처리하는 것만으로 충분하지 않으며, 통찰력을 얻기 위해서는 정렬과 그룹화가 필요할 것이다.

업계별 거래량 집계

1. 원시 주식 거래 정보가 입력된 토픽으로부터 소스를 만든다.
2. 종목코드로 ShareVolume그룹을 만든다.
이 코드로 그룹을 만들면, 전체 주식 거래량에서 그룹별 거래량으로 데이터를 줄일 수 있다.



주식 거래 맵리듀스 소스 코드

```
KTable<String, ShareVolume> shareVolume = builder.stream(STOCK_TRANSACTIONS_TOPIC,  
    Consumed.with(stringSerde, stockTransactionSerde)  
        .withOffsetResetPolicy(EARLIEST))  
    .mapValues(st -> ShareVolume.newBuilder(st).build())  
    .groupBy((k, v) -> v.getSymbol(), Serialized.with(stringSerde, shareVolumeSerde))  
    .reduce(ShareVolume::sum);
```

거래량을 툴링 집계하기 위한 ShareVolume 객체 리듀스

StockTransaction 객체를 ShareVolume 객체로 매팅

주식 종목 코드에 따른
ShareVolume 객체를 그룹화

윈도(windowing) 연산

어떨 때는 계속되는 집계와 리덕션이 필요하지만, 또 어떨 때는 주어진 시간 범위에 대해서만 작업을 수행할 필요도 있다.

예를 들어, 최근 10분 동안 특정 회사와 관련된 주식 거래가 얼마나 발생했는가?

최근 15분 동안 새 광고를 클릭한 사용자가 얼마나 되는가?

윈도 유형

Kafka Streams에서는 세 가지 유형의 윈도를 사용할 수 있다.

- ✓ 세션(session) 윈도
- ✓ 텀블링(tumbling) 윈도
- ✓ 슬라이딩(sliding) 또는 호핑(hopping) 윈도

세션 윈도

- ✓ 세션 윈도는 시간에 엄격하게 제한 받지 않고 사용자 활동과 관련이 있다.
- ✓ 만약 들어오는 레코드가 비활성화 간격을 넘어선다면 새 세션을 만든다.



여기는 비활성화 구간이 짧기 때문에
더 큰 1개의 세션으로 병합될 수 있을 것이다.

비활성화 구간이 넓기 때문에
새 이벤트는 분할된 세션이 된다.

세션 윈도

주식거래 포착을 위한 세션 윈도

```

Serde<String> stringSerde = Serdes.String();
Serde<StockTransaction> transactionSerde = StreamsSerdes.StockTransactionSerde();
Serde<TransactionSummary> transactionKeySerde = StreamsSerdes.TransactionSummarySerde();

StreamsBuilder builder = new StreamsBuilder();
long twentySeconds = 1000 * 20;
long fifteenMinutes = 1000 * 60 * 15;
long fiveSeconds = 1000 * 5;
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
    builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
transactionSerde).withOffsetResetPolicy(LATEST))
        .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
Serialized.with(transactionKeySerde, transactionSerde))
        .windowedBy(SessionWindows.with(twentySeconds).until(fifteenMinutes)).count();
customerTransactionCounts.toStream()
    .print(Printed.<Windowed<TransactionSummary>, Long>toSysOut().withLabel("Customer Transactions Counts"));

```

groupBy와 count호출로 생성된 KTable

TransactionSummary 객체에 저장된 고객 ID와 주식 종목으로 레코드를 그룹화한다.

비활성화 시간 20초, 유지 시간 15분의 SessionWindow로 그룹을 윈도 처리한 다음, count()와 같은 집계를 수행한다.

20초 비활성 간격으로 테이블 세션 처리

이 with 호출은 20초의 비활성 간격을 만든다.

이 until 메소드는 15분의 유지 기간을 만든다.
이 시간이 지나면 삭제된다.

```
SessionWindows.with(twentySeconds).until(fifteenMinutes))
```

도착 순서	키	타입 스템프	
1	abc	00:00:00	00:00:00
2	abc	00:00:15	00:00:00 ~ 00:00:15
3	abc	00:00:50	00:00:50
4	abc	00:00:05	00:00:00 ~ 00:00:15

레코드 1이 도착하면 새 세션 생성

레코드 2가 도착하면 1과 병합

레코드 3이 도착하면 새로운 세션 생성

레코드 4가 도착하면 1,2,4 병합

웹 사이트 사용자 클릭 count 예제 (5분)

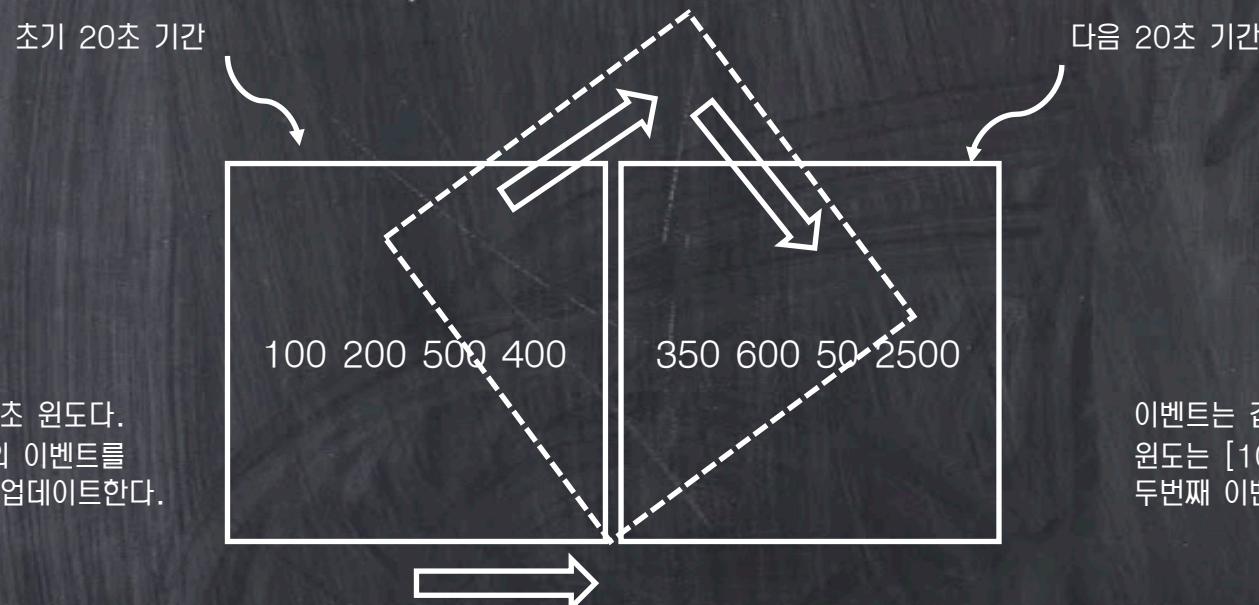
userId	createdTime	currentLink	nextLink	
rudaks	2020-10-10T10:01:00.00Z	NULL	Home	10:01 ~ 10:01, count: 1
rudaks	2020-10-10T10:04:00.00Z	Home	Books	10:01 ~ 10:04, count: 2
rudaks	2020-10-10T10:10:00.00Z	Books	Kafka	10:10 ~ 10:10, count: 1
rudaks	2020-10-10T10:07:00.00Z	Kafka	Preview	10:01 ~ 10:10, count: 4
rudaks	2020-10-10T10:19:00.00Z	Preview	Buy	10:19 ~ 10:19, count: 1

```
SessionWindows.with(5 * 60 * 1000)
```

- ✓ 세션 생성은 첫 클릭시 생성이 되고 5분 동안 idle이면 세션이 종료된다.

텀블링 윈도

- ✓ 지정한 기간 내의 이벤트를 추적한다.
- ✓ 예를 들면 특정 회사의 전체 주식 거래를 20초마다 추적해야 하고, 그 시간 동안 모든 이벤트를 수집한다고 하자.
- ✓ 20초가 경과하면 윈도는 다음 20초 감시 주기로 ‘텀블링’ 한다.

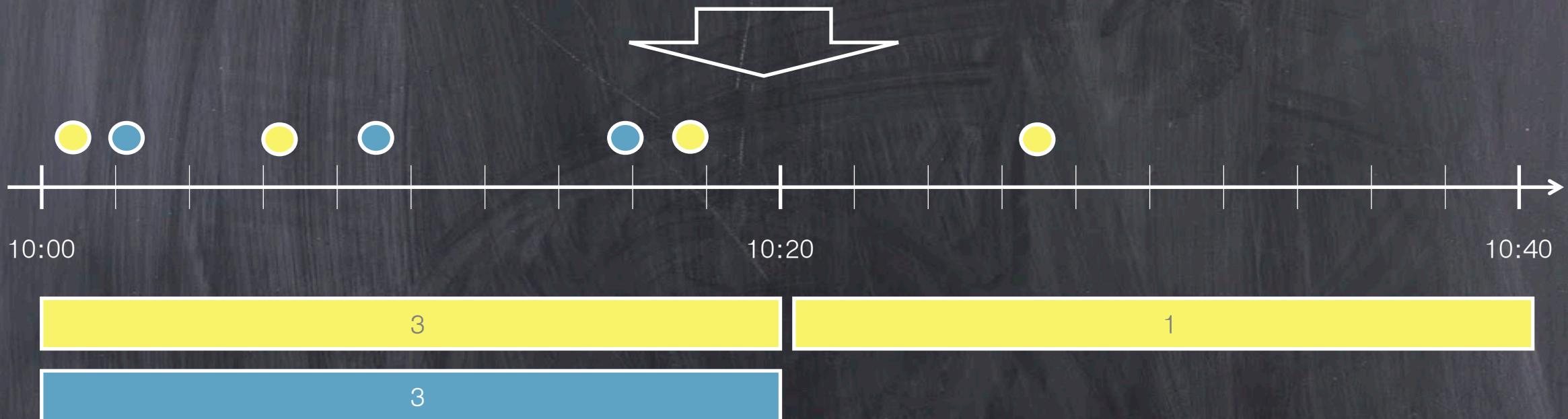


왼쪽에 있는 상자는 첫 20초 윈도다.
20초 후에 새 20초 기간의 이벤트를
추적하기 위해 넘어가거나 업데이트한다.

텀블링 윈도는 아래와 같은 형태이다.

- ✓ 텀블링 윈도는 고정 크기이며 갭(gap)이 없는 window이다.

Stock ID	Window ID	Window Start	Window End	Count
kakao	20546102012	2020-10-10T10:00Z	2020-10-10T10:20Z	3
kakao	20546102234	2020-10-10T10:20Z	2020-10-10T10:40Z	1
naver	20546102897	2020-10-10T10:00Z	2020-10-10T10:20Z	3

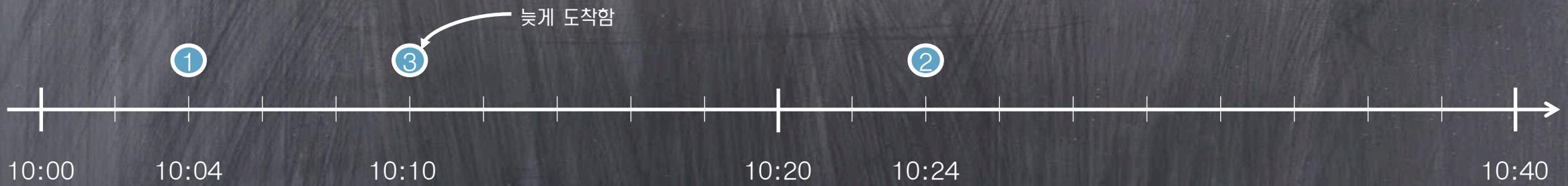


사용자 거래를 카운트 하기 위한 텀블링 윈도

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
    builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
transactionSerde).withOffsetResetPolicy(LATEST))  
        .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),  
            Serialized.with(transactionKeySerde, transactionSerde))  
        .windowedBy(TimeWindows.of(twentySeconds))  
        .count();
```

20초 텀블링 윈도 지정

이벤트가 늦게 도착하는 경우는 어떡하나?



①	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1	max stream time: 10:04
②	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1	
②	naver	2020-10-10T10:20Z	2020-10-10T10:40Z	1	max stream time: 10:24
③	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1	
③	naver	2020-10-10T10:20Z	2020-10-10T10:40Z	1	이전 시간에 집계가 되어야 할까? 아님 무시해야 할까?

grace 연산자를 통해 늦게 도착한 event를 포함할지 결정할 수 있다.

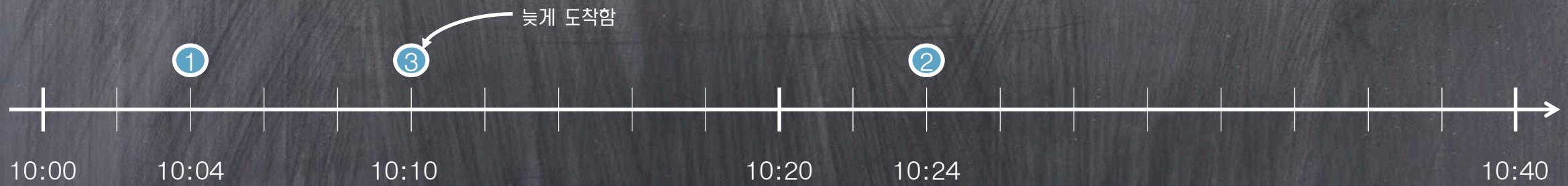
```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
    builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
transactionSerde).withOffsetResetPolicy(LATEST))  
        .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),  
            Serialized.with(transactionKeySerde, transactionSerde))  
        .windowedBy(TimeWindows.of(twentySeconds))  
        .grace(Duration.ofMinutes(2))  
        .count();
```

grace 연산자 추가

Grace period

- ✓ Kafka window는 늦게 도착한 메시지에 대해서 적용될 수 있다.
- ✓ Grace period는 늦게 도착한 메시지를 해당 window에 포함할지 결정할 수 있다.
- ✓ 기본값은 24 시간이다.

window end + grace time 안에 도착하면 포함



①	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1
---	-------	-------------------	-------------------	---

max stream time: 10:04

②	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1
	naver	2020-10-10T10:20Z	2020-10-10T10:40Z	1

max stream time: 10:24

③	naver	2020-10-10T10:00Z	2020-10-10T10:20Z	1
	naver	2020-10-10T10:20Z	2020-10-10T10:40Z	1

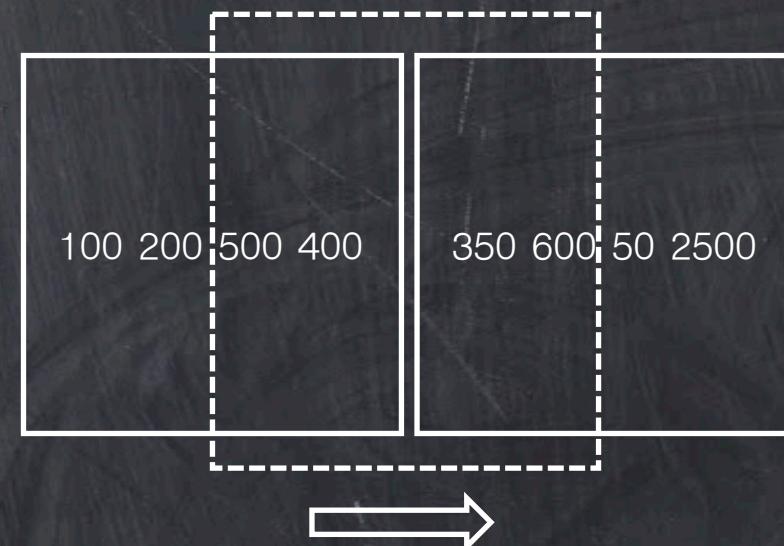
.grace(Duration.ofMinutes(2))으로
되어있기 때문에 10:22분 이전에 도착한
이벤트만 집계가 된다.

슬라이딩(호핑) 윈도

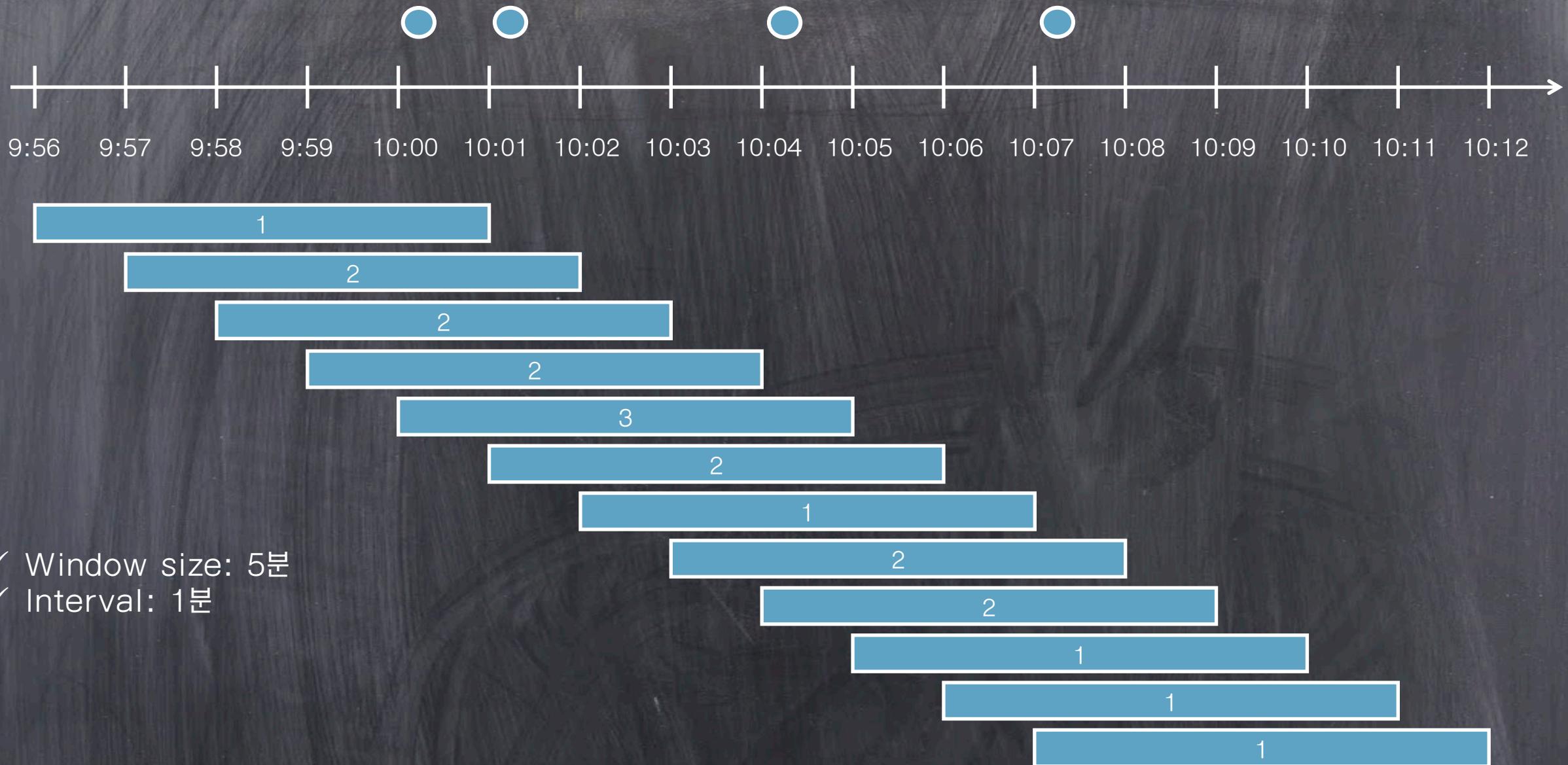
슬라이딩 윈도는 텀블링 윈도와 비슷하지만 작은 차이가 있다.

슬라이딩 윈도는 최근 이벤트를 처리할 새 윈도를 시작하기 전에 그 윈도 전체 시간을 기다리지 않는다.

왼쪽 박스는 첫 20초 윈도인데,
이 윈도는 새 윈도를 시작하기 위해
5초 뒤에 슬라이드하거나 업데이트한다.
여기서는 겹쳐지는 이벤트를 보게 된다.



윈도 1은 [100,200,500,400],
윈도 2는 [500,400,350,600],
윈도 3은 [350,600,50,2500]이다.



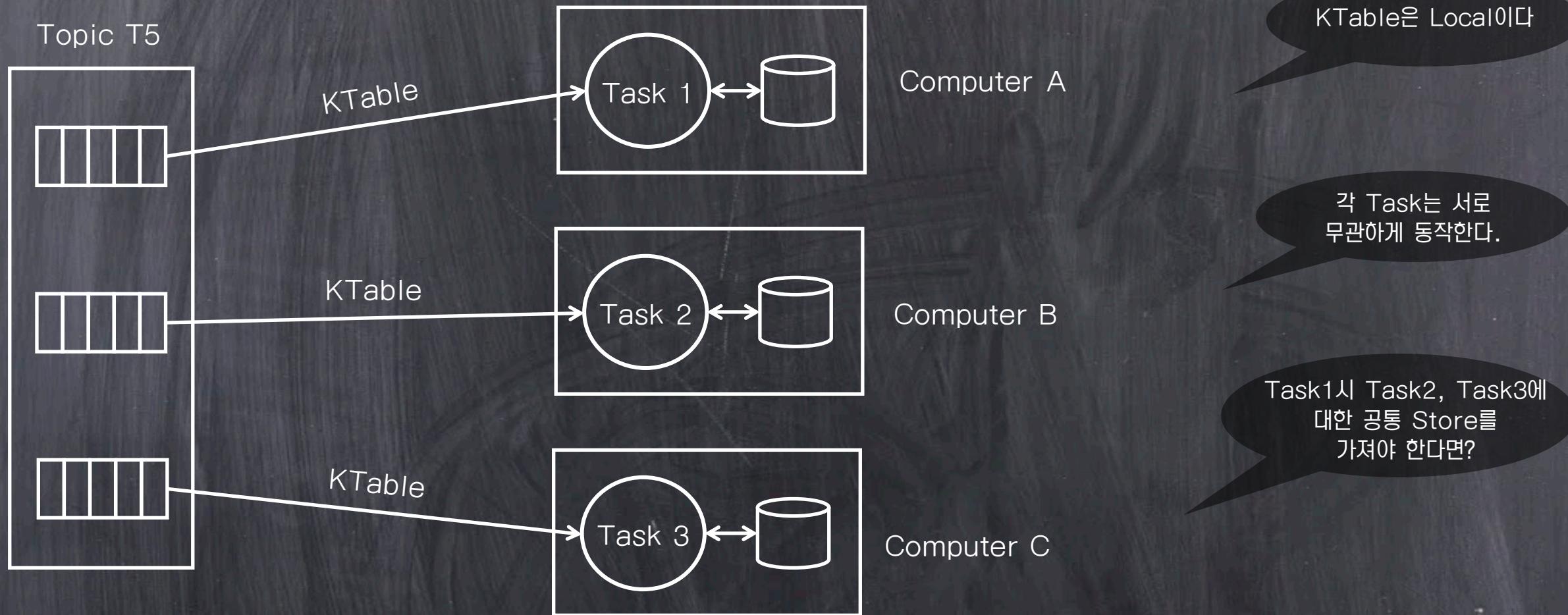
슬라이딩 윈도

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =  
    builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,  
transactionSerde).withOffsetResetPolicy(LATEST))  
        .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),  
            Serialized.with(transactionKeySerde, transactionSerde))  
        .windowedBy(TimeWindows.of(twentySeconds))  
        .advanceBy(fiveSeconds)  
        .count();
```



5초마다 이동하는
20초 간격의 오픈 윈도 사용

만일 KTable Task가 다른 store에 접근해야 한다면?



GlobalKTable

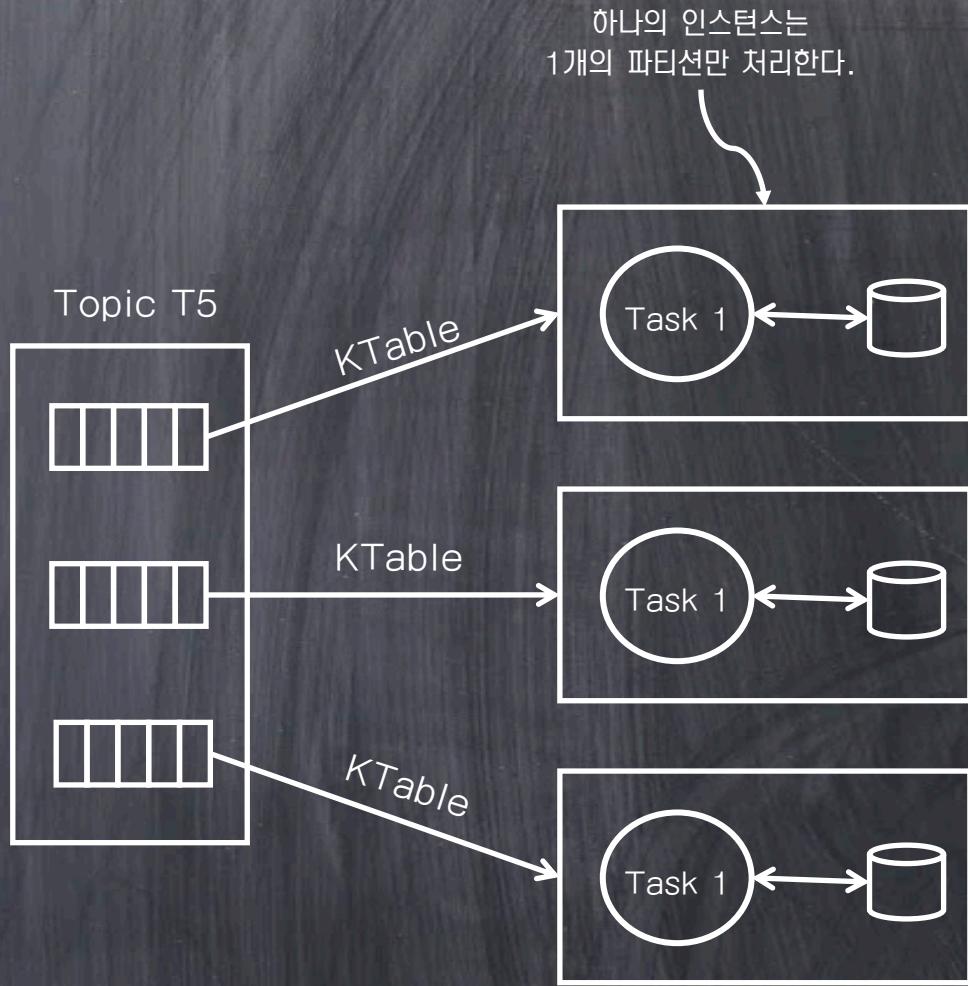
- ✓ 조인하려는 루프데이터는 비교적 작아서 이 조회 데이터 전체 사본을 개별 노드의 로컬에 배치할 수 있을 것이다.
- ✓ 조회 데이터가 상당히 작을 경우 카프카 스트림즈는 GlobalKTable을 사용할 수 있다.
- ✓ 애플리케이션이 모든 데이터를 각 노드에 동일하게 복제하기 때문에 GlobalKTable은 모두 같은 데이터를 갖는다.
- ✓ 전체 데이터가 개별 노드에 있기 때문에 데이터 키로 파티셔닝할 필요가 없다.

```
KStream<String, TransactionSummary> countStream =
    builder.stream( STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde, transactionSerde).withOffsetResetPolicy(LATEST))
        .groupBy((noKey, transaction) -> TransactionSummary.from(transaction), Serialized.with(transactionSummarySerde, transactionSerde))
        .windowedBy(SessionWindows.with(twentySeconds)).count()
        .toStream().map(transactionMapper);

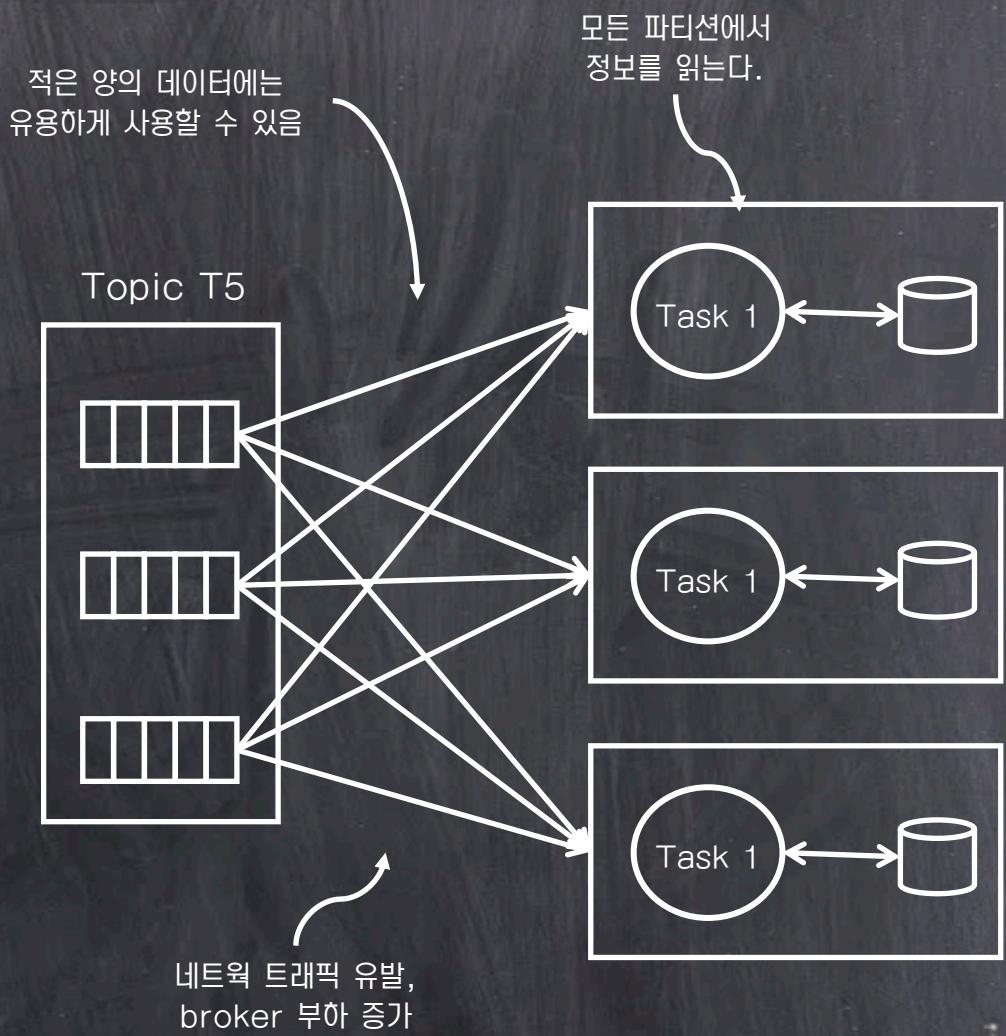
GlobalKTable<String, String> publicCompanies = builder.globalTable(COMPANIES.topicName());
GlobalKTable<String, String> clients = builder.globalTable(CLIENTS.topicName());

countStream.leftJoin(publicCompanies, (key, txn) -> txn.getStockTicker(), TransactionSummary::withCompanyName)
    .leftJoin(clients, (key, txn) -> txn.getCustomerId(), TransactionSummary::withCustomerName)
    .print(Printed.<String, TransactionSummary>toSysOut().withLabel("Resolved Transaction Summaries"));
```

KTable



GlobalKTable



6. Processor API

1. 트레이드 오프
2. 토플로지 작업
3. 프로세서 API 더 깊이 파고들기

Kafka Streams API는 최소한의 코드로 애플리케이션을 만들기 위한 DSL이다.



그런데, Kafka Streams DSL로
할 수 없는 것은 어떻게 해야 할까?

원하는 커스트마이징 로직을 추가하고 싶다.

Processor API

- ✓ Kafka Streams API는 개발자가 최소한의 코드로 견고한 애플리케이션을 만들기 위한 DSL이다.
- ✓ 그러나 특정 상황에서는 고수준 추상화로 불가능한 어떤 코드를 사용해야 한다.
- ✓ Processor API는 Kafka Streams DSL로 쉽게 할 수 없는 방식으로 스트림 처리를 해야 할 때 사용한다.

더 높은 수준의 추상화 vs 더 많은 제어 사이의 트레이드 오프

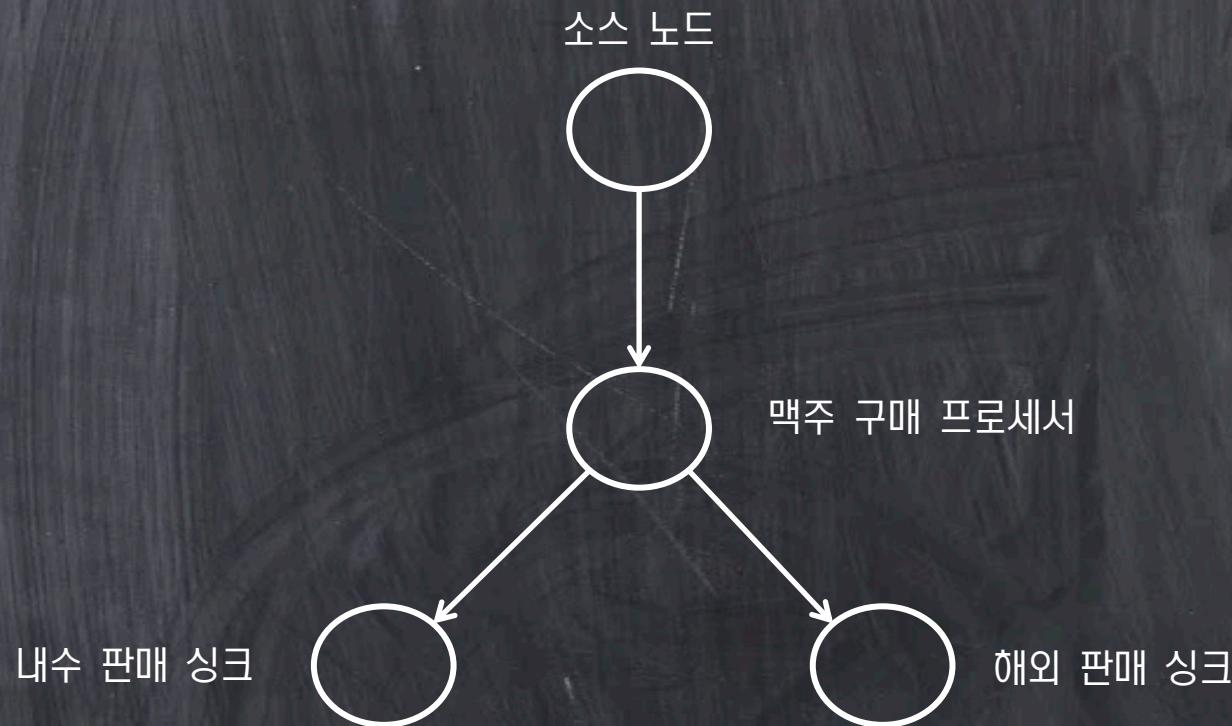
- ✓ 더 높은 수준의 추상화와 더 많은 제어 사이의 트레이드 오프의 적절한 예제는 ORM 프레임워크이다.
- ✓ 훌륭한 ORM 프레임워크는 도메인 객체를 데이터베이스 테이블에 매핑하고 실행 시간에 적합한 SQL 쿼리를 만든다.
- ✓ 단순한 SQL 작업이라면 ORM 프레임워크를 사용하면 많은 시간을 절약할 수 있다.
- ✓ 그러나 ORM 프레임워크에서 지원하지 않는 쿼리가 있을 수 있다. (복잡한 조인, sub-쿼리 등)
- ✓ 이럴 경우 원시 SQL을 사용해야 한다.

Processor API가 할 수 있는 것

- ✓ 일정한 간격으로 액션을 스케줄링
- ✓ 레코드가 다음 스트림에 전송될 때 완벽하게 제어
- ✓ 특정 자식 노드에 레코드 전달
- ✓ Kafka Streams API에 없는 기능 구현

토플로지 구성

여러 지역에 있는 맥주공장 소유자라고 하자.
회사 내에서 주문이 국내인지 해외인지, 영국 파운드화나 유로화를 미국달러로 환산해야 할지를 판단하여 처리해야 한다.



소스 노드 추가

- ✓ 토플로지 구축 시 첫 번째 단계는 소스 노드를 설정하는 것이다.
- ✓ Processor API는 저수준 추상화이기 때문에 소스 노드를 만들 때 키 역직렬화기와 값 역직렬화기를 제공해야 한다.

```
이 소스를 사용하기 위해  
TimeStampExtractor를 명시
```

```
사용할 오프셋 리셋을 지정  
이 노드의 이름을 지정  
값 역직렬화기 지정  
데이터를 소비할 토픽 이름 지정  
키 역직렬화기 지정
```

```
topology.addSource(LATEST,  
                    purchaseSourceNodeName,  
                    new UsePreviousTimeOnInvalidTimestamp(),  
                    stringDeserializer,  
                    beerPurchaseDeserializer,  
                    Topics.POPS_HOPS_PURCHASES.topicName())
```

프로세서 노드 추가

- ✓ 이제, 소스 노드에서 들어오는 레코드로 작업하는 프로세서를 추가할 것이다.
- ✓ Streams API와의 차이점은 반환 유형에 있다.
- ✓ Streams API의 경우 모든 KStream 호출은 새 KStream이나 KTable 인스턴스를 반환한다.
- ✓ Processor API에서는 토플로지 인스턴스를 반환한다.

```
BeerPurchaseProcessor beerProcessor = new BeerPurchaseProcessor(domesticSalesSink, internationalSalesSink);

topology.addSource(LATEST,
    purchaseSourceNodeName,
    new UsePreviousTimeOnInvalidTimestamp(),
    stringDeserializer,
    beerPurchaseDeserializer,
    Topics.POPS_HOPS_PURCHASES.topicName())
    .addProcessor(purchaseProcessor,
        () -> beerProcessor,
        purchaseSourceNodeName);
```

소스노드 이름은 프로세서 노드에서 부모 이름으로 사용된다.
데이터 흐름을 지시하는 부모-자식 관계를 설정한다.

프로세서 노드 이름
위에서 정의한 프로세서를 추가
부모 노드 또는 복수의 부모 노드 이름을 지정

BeerPurchaseProcessor

```

public class BeerPurchaseProcessor extends AbstractProcessor<String, BeerPurchase> {

    private String domesticSalesNode;
    private String internationalSalesNode;

    public BeerPurchaseProcessor(String domesticSalesNode, String internationalSalesNode) {
        this.domesticSalesNode = domesticSalesNode;
        this.internationalSalesNode = internationalSalesNode; ← 레코드가 전달될 각 노드 이름을 설정
    }

    @Override
    public void process(String key, BeerPurchase beerPurchase) { ← 액션이 실행될 process() 메소드
        Currency transactionCurrency = beerPurchase.getCurrency();
        if (transactionCurrency != DOLLARS) {
            BeerPurchase dollarBeerPurchase;
            BeerPurchase.Builder builder = BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(pattern);
            builder.currency(DOLLARS);
            builder.totalSale(Double.parseDouble(
                decimalFormat.format(transactionCurrency.convertToDollars(internationalSaleAmount))
            ));
            dollarBeerPurchase = builder.build();
            context().forward(key, dollarBeerPurchase, internationalSalesNode); ← context() 메소드가 반환하는
        } else {                                                 ProcessorContext를 사용해 레코드를
            context().forward(key, beerPurchase, domesticSalesNode);   international 자식 노드에 전달
        }
    }
}

```

국내 판매 레코드를
domestic 자식 노드로 전송

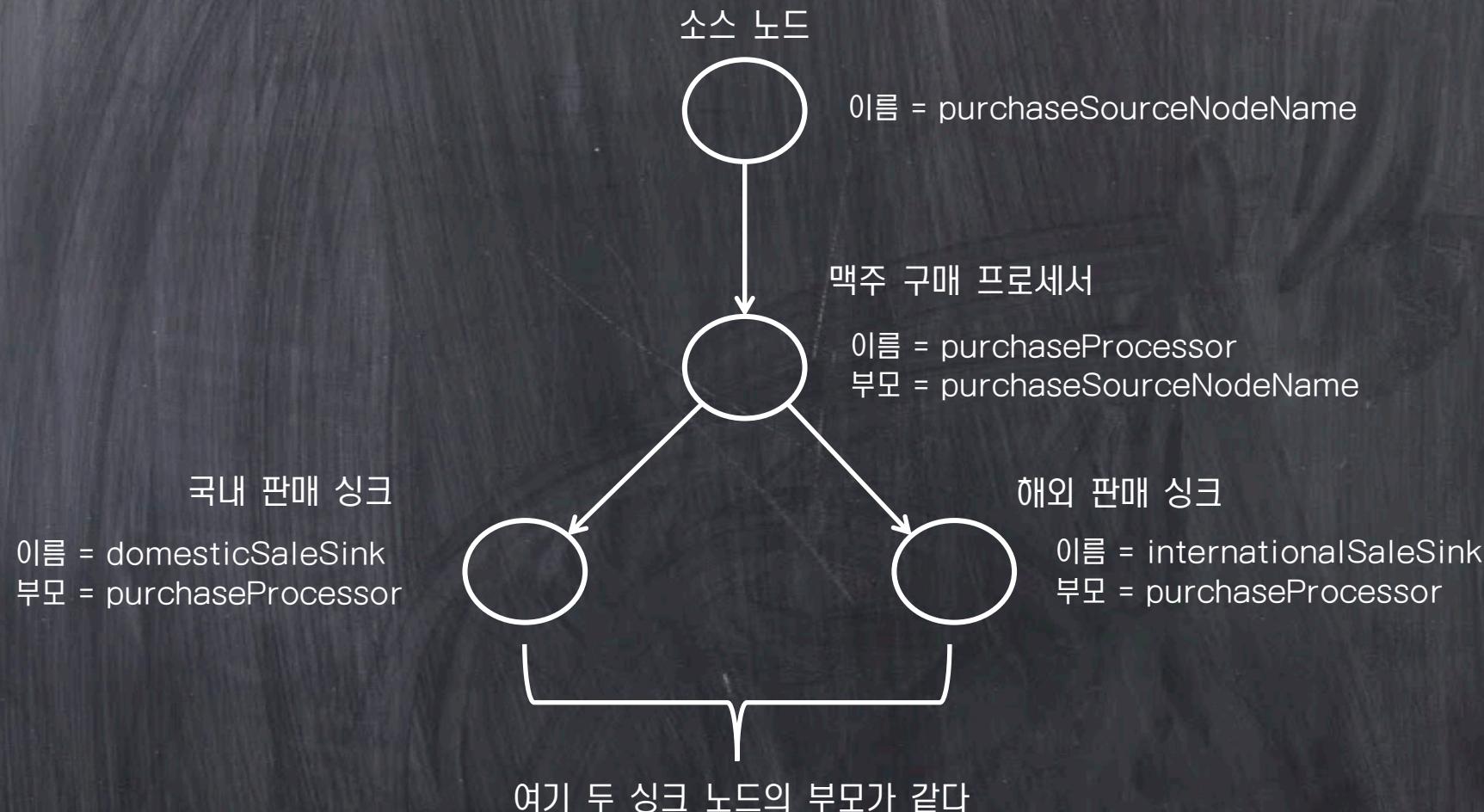
해외 판매액을 달러화로 환산

레코드가 전달될 각 노드 이름을 설정

액션이 실행될 process() 메소드

싱크 노드 추가

- ✓ 소스를 추가하기 위해 `addSource`를 사용했고, 프로세서를 추가하기 위해 `addProcessor`를 사용했다.
- ✓ 싱크노드를 연동하기 위해 `addSink`를 사용



싱크 노드 추가

- ✓ 싱크 노드를 추가해 토플로지를 업데이트 할 수 있다.

```
topology.addSource(LATEST,  
    purchaseSourceNodeName,  
    new UsePreviousTimeOnInvalidTimestamp(),  
    stringDeserializer,  
    beerPurchaseDeserializer,  
    Topics.POPS_HOPS_PURCHASES.topicName())  
.addProcessor(purchaseProcessor,  
    () -> beerProcessor,  
    purchaseSourceNodeName)  
.addSink(internationalSalesSink,  
    "international-sales",  
    stringSerializer,  
    beerPurchaseSerializer,  
    purchaseProcessor)  
.addSink(domesticSalesSink,  
    "domestic-sales",  
    stringSerializer,  
    beerPurchaseSerializer,  
    purchaseProcessor);
```

← 싱크 이름
← 이 싱크가 제공하는 토픽
← 키에 대한 직렬화기
← 값에 대한 직렬화기
← 이 싱크의 부모 노드 이름
← 싱크 이름
← 이 싱크가 제공하는 토픽
← 키에 대한 직렬화기
← 값에 대한 직렬화기
← 이 싱크의 부모 노드 이름

주식 분석 프로세서로 Processor API 자세히 살펴보기

- ✓ 이 주식의 현재 가치 조회
- ✓ 주식 가치가 상승, 하강 추세인지 표시
- ✓ 지금까지 총 주식 거래량과 상승, 하강 추세 여부 포함
- ✓ 추세 변동이 2%인 주식만 레코드를 다운스트림에 전송
- ✓ 계산하기 전에 주식의 최소 샘플 20개를 수집

주식 성과 애플리케이션

```

Topology topology = new Topology();
String stocksStateStore = "stock-performance-store";
double differentialThreshold = 0.02; ← 다운스트림에 전달할 주식 정보에 대한 백분율 임곗값 설정

KeyValueBytesStoreSupplier storeSupplier = Stores.inMemoryKeyValueStore(stocksStateStore); ← 인메모리 키/값 상태 저장소 생성
StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder
    = Stores.keyValueStoreBuilder(storeSupplier, Serdes.String(), stockPerformanceSerde);

topology.addSource("stocks-source",
    stringDeserializer,
    stockTransactionDeserializer,
    "stock-transactions")
.addProcessor("stocks-processor", () -> new StockPerformanceProcessor(
    stocksStateStore, differentialThreshold), "stocks-source") ← 토플리지에 프로세서 추가
.addStateStore(storeBuilder, "stocks-processor") ← stocks 프로세서에 상태 저장소 추가
.addSink("stocks-sink",
    "stock-performance",
    stringSerializer,
    stockPerformanceSerializer,
    "stocks-processor"); ← 결과를 쓰기 위한 싱크 추가

```

init() 메소드의 작업

```

public class StockPerformanceProcessor extends AbstractProcessor<String, StockTransaction> {
    private KeyValueStore<String, StockPerformance> keyValueStore;
    private String stateStoreName;
    private double differentialThreshold;

    public StockPerformanceProcessor(String stateStoreName, double differentialThreshold) {
        this.stateStoreName = stateStoreName;
        this.differentialThreshold = differentialThreshold;
    }

    @Override
    public void init(ProcessorContext processorContext) {
        super.init(processorContext);           ← ProcessorContext 초기화
        keyValueStore = (KeyValueStore) context().getStateStore(stateStoreName);      ← 상태 저장소 조회
        StockPerformancePunctuator punctuator
            = new StockPerformancePunctuator(differentialThreshold, context(), keyValueStore); ← Punctuator 초기화

        context().schedule(10000, PunctuationType.WALL_CLOCK_TIME, punctuator);          ← 10초마다 Punctuator.punctuate()를 호출하도록 스케줄링
    }

    @Override
    public void process(String symbol, StockTransaction transaction) {
        ...
    }
}

```

↑ ProcessorContext 초기화
↑ 상태 저장소 조회
↑ Punctuator 초기화
↑ 10초마다 Punctuator.punctuate()를 호출하도록 스케줄링

process() 구현

```
@Override  
public void process(String symbol, StockTransaction transaction) {  
    if (symbol != null) {  
        StockPerformance stockPerformance = keyValueStore.get(symbol); ← 이전 StockPerformance 통계를 조회,  
        // null 값이다  
  
        if (stockPerformance == null) {  
            stockPerformance = new StockPerformance(); ← 상태 저장소에 없다면 새  
            // StockPerformance 객체를 생성  
        }  
  
        stockPerformance.updatePriceStats(transaction.getSharePrice()); ← 가격 통계를 업데이트  
        stockPerformance.updateVolumeStats(transaction.getShares()); ← 거래량 통계를 업데이트  
        stockPerformance.setLastUpdateSent(Instant.now()); ← 최근 업데이트한 타임스탬프를 설정  
  
        keyValueStore.put(symbol, stockPerformance); ← 업데이트한 StockPerformance  
        // 객체를 상태 저장소에 저장  
    }  
}
```

punctuate 코드

```
@Override  
public void punctuate(long timestamp) {  
    KeyValueIterator<String, StockPerformance> performanceIterator = keyValueStore.all(); ← 상태 저장소에 있는 모든 키와 값을 확인할  
    이터레이터를 조회  
  
    while (performanceIterator.hasNext()) {  
        KeyValue<String, StockPerformance> keyValue = performanceIterator.next();  
        String key = keyValue.key;  
        StockPerformance stockPerformance = keyValue.value;  
  
        if (stockPerformance != null) {  
            if (stockPerformance.priceDifferential() >= differentialThreshold ||  
                stockPerformance.volumeDifferential() >= differentialThreshold) { ← 현재 주식에 대한 임계치 확인  
                context.forward(key, stockPerformance);  
            } ← 임계치에 도달했거나 초과했다면  
        } ← 이 레코드를 전달  
    }  
}
```

Part 3

Kafka Streams 관리

- 7. 모니터링과 성능
- 8. Kafka Streams 애플리케이션 테스트

7. 모니터링과 성능

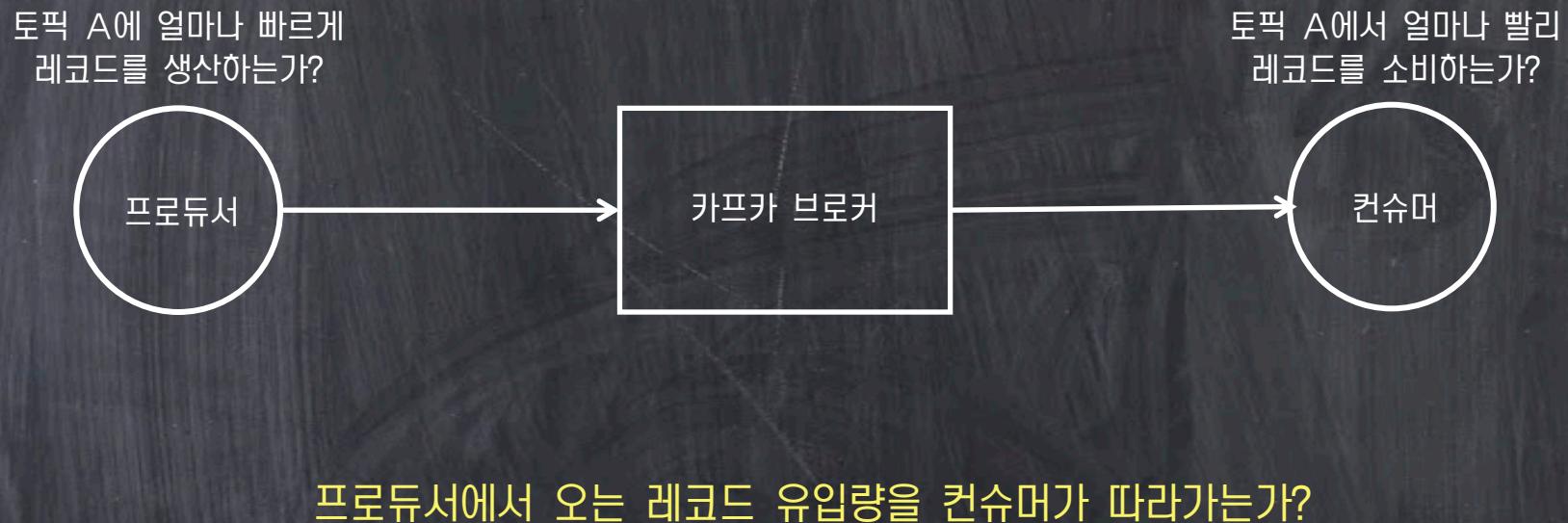
1. 카프카 모니터링

기본적인 카프카 모니터링

- ✓ Kafka Streams API는 카프카의 일부이므로 애플리케이션을 모니터링 할 때 카프카도 일부 모니터링해야 한다.

컨슈머와 프로듀서 성능 측정

- ✓ 프로듀서의 경우 프로듀서가 얼마나 빠르게 브로커로 메시지를 보내느냐가 큰 관심사이며, 처리량이 높으면 높을수록 좋다.
- ✓ 컨슈머의 경우, 브로커에서 얼마나 빠르게 메시지를 읽을 수 있느냐가 성능에 영향을 준다.
- ✓ 컨슈머 성능을 측정하는 또 다른 방법으로 컨슈머 지연이 있다.



컨슈머 지연

- ✓ 프로듀서가 브로커에 기록하는 속도와 컨슈머가 메시지를 읽는 속도 차이를 컨슈머 지연(consumer lag)이라고 부른다.



같은 토픽에 대해 가장 최근 생산된 오프셋과
가장 최근 소비된 오프셋 차이가 바로 컨슈머 지연이다.

컨슈머 지연 확인하기

- ✓ 컨슈머 지연은 명령어를 통해 확인할 수 있다.

컨슈머 그룹목록 조회

```
bash-4.4# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --list
ticket-reception-subscriber
```

컨슈머 그룹 상태

```
bash-4.4# kafka-consumer-groups.sh --bootstrap-server localhost:9092 --group ticket-reception-subscriber --describe
```

Consumer group 'command.mocha.ticket-reception-subscriber' has no active members.

GROUP	TOPIC	PARTITION	CURRENT-OFFSET	LOG-END-OFFSET	LAG	CONSUMER-ID	HOST	CLIENT-ID
ticket-reception-subscriber	ticket-reception	0	5695	5695	0	-	-	-

읽은 메시지 개수

5695

프로듀서가 토픽에
전송한 메시지 개수

컨슈머 지연

프로듀서와 컨슈머 가로채기

- ✓ 인터셉터는 디버깅을 위한 일반적인 도구는 아니지만 카프카 스트리밍 애플리케이션의 동작을 관찰하는 데 유용할 수 있다.

컨슈머 인터셉터

- ✓ 컨슈머 인터셉터는 두 가지 접근점을 제공한다.

Case 1) ConsumerInterceptors.onConsume()

Case 2) ConsumerInterceptor.onCommit()

```
public class StockTransactionConsumerInterceptor implements ConsumerInterceptor<Object, Object> {

    private static final Logger LOG = LoggerFactory.getLogger(StockTransactionConsumerInterceptor.class);

    public StockTransactionConsumerInterceptor() {
        LOG.info("Built StockTransactionConsumerInterceptor");
    }

    @Override
    public ConsumerRecords<Object, Object> onConsume(ConsumerRecords<Object, Object> consumerRecords) {
        LOG.info("Intercepted ConsumerRecords {}", buildMessage(consumerRecords.iterator()));
        return consumerRecords;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> map) {
        LOG.info("Commit information {}", map);
    }
}
```

Case 1)
레코드가 처리되기 전에
컨슈머 레코드와 메타데이터를 로깅

Case 2)
컨슈머가 브로커에 오프셋을 커밋
할 때 커밋 정보를 로깅

프로듀서 인터셉터

- ✓ 프로듀서 인터셉터도 두 가지 접근점을 제공한다.

Case 1) ProducerInterceptor.onSend()

Case 2) ProducerInterceptor.onAcknowledgement()

```
public class ZMartProducerInterceptor implements ProducerInterceptor<Object, Object> {

    private static final Logger LOG = LoggerFactory.getLogger(ZMartProducerInterceptor.class);

    @Override
    public ProducerRecord<Object, Object> onSend(ProducerRecord<Object, Object> record) {
        LOG.info("ProducerRecord being sent out {}", record); ← Case 1)
        return record;
    }

    @Override
    public void onAcknowledgement(RecordMetadata metadata, Exception exception) {
        if (exception != null) {
            LOG.warn("Exception encountered producing record {}", exception);
        } else {
            LOG.info("record has been acknowledged {}", metadata); ← Case 2)
        }
    }
}
```

Case 1)
메시지를 브로커에 전송하기 전에 로깅

Case 2)
브로커 수신 확인 시점에 로깅

참고 자료

- ✓ Apache Kafka – Real-time Stream Processing (udemy)

<https://www.udemy.com/course/kafka-streams-real-time-stream-processing-master-class/>