1 장 카프카 소개

전 세계적으로 데이터 양은 지수적으로 증가하고 있으며 세계 경제 포럼 (WEP, World Economic Forum)에 따르면 이미 세상에 저장된 바이트 수가 관찰 가능한 우주 내 별들의 수를 능가한 상태다.

데이터에 대해 생각할 때 데이터 웨어하우스, 관계형 데이터베이스 또는 분산 파일시스템에 존재하는 바이트 더미를 생각할 수 있다. 이들 시스템은 데이터가 정지 상태에 있다고 생각하도록 우리를 길들여왔다. 다른 말로 데이터는 어딘가 앉아 쉬고 있으며 이를 처리해야 할 때 바이트 더미에 쿼리 또는 잡을 실행시킨다는 것이다.

이러한 관점은 데이터에 대한 보다 전통적인 사고 방식이다. 그러나 데이터가 여러 곳에 쌓일 수 있지만 움직이는 경우가 더욱 많다. 여러분이 알고 있듯이 IoT 센서, 의료 센서, 재무 시스템, 사용자 및 고객 분석 소프트웨어, 애플리케이션 및 서버 로그 등을 포함하여 많은 시스템이 연속적인 데이터 스트림을 생성하고 있다. 최종적으로 쉴 만한 좋은 장소를 찾은 데이터라도 영원한 고향을 찾기 전 어느 시점에는 네트워크를 통해 이동할 가능성이 높다.

데이터가 움직이는 동안 이들을 실시간으로 처리하려 한다면 이들이 어딘 가에 쌓이길 기다리고 그 후 선택한 간격으로 쿼리 또는 잡을 실행할 수는 없다. 이 방법은 어떤 비즈니스 유스케이스를 다룰 수 있지만 많은 중요한 유스케이스는 데이터가 사용 가능할 때마다 증분적으로 데이터를 처리, 보강(enrichment), 변환 및 대응할 것을 요구한다. 따라서 데이터에 대해 매우 다른 관점, 즉 흐르는 상태의 데이터에 액세스할 수 있도록 하는 기술을 갖고 이들 연속적이고 무제한의 데이터 스트림으로 빨리 효율적으로 작업할 수 있도록 하는 무언 가가 필요하다. 이것이 카프카가 등장한 이유이다.

아파치 카프카 (단순히 카프카)는 데이터 스트림을 수집, 저장, 액세스 및 처리하는 스트리밍 플랫폼이다. 전체 플랫폼이 매우 흥미롭지만 이 책에서는 카프카의 가장 매력적인 부분인 스트림 처리 계층에 중점을 둔다. 그러나 카프카 스트림즈와 ksqlDB 를 이해하기 위해서는 플랫폼으로써 카프카의 작동 원리를 이해하는 것이 필요하다 (둘 모두 스트림 처리 계층에서 작동하며 후자는 스트림 수집(ingestion) 계층에서도 작동한다).

따라서 이 장에서는 이 책의 나머지를 위해 필요한 중요한 개념 및 용어를 소개할 것이다. 이미 카프카에 대해 잘 알고 있다면 이 장을 건너뛰어도 좋으며, 그렇지 않다면 계속 읽기 바란다.

이 장에서 답변할 질문 중 일부는 다음을 포함한다:

- 카프카가 시스템 간 통신을 어떻게 단순화하는가?
- 카프카 아키텍처의 주요 구성요소는 무엇인가?
- 어떤 스토리지 추상화가 스트림을 가장 가깝게 모델링하는가?
- 카프카가 데이터를 어떻게 내고장성(fault-tolerant) 및 내구성(durable) 방식으로 저장하는가?
- 데이터 처리 계층에서 고가용성과 내고장성이 어떻게 얻어지는가?

카프카 설치 및 실행 방법을 보여주는 튜토리얼로 이 장을 마무리할 것이다.

우선 카프카 통신 모델을 살펴보자.

통신 모델

시스템 간 가장 일반적인 통신 패턴은 아마도 동기성 클라이언트-서버 모델이다. 이 상황에서 시스템에 대해 이야기할 때 우리는 네트워크를 통해 데이터를 읽고 쓰는 애플리케이션, 마이크로 서비스, 데이터베이스 등을 의미한다. 클라이언트-서버 모델은 처음에는 간단하고 그림 1-1 과 같이 시스템 간 직접 통신을 포함한다.

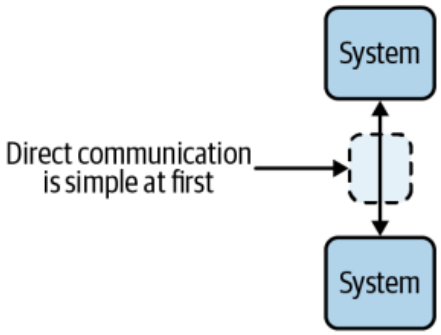

그림 1-1. 점대점(Point-to-Point) 통신은 소규모 시스템의 경우 유지보수 및 추론하기가 단순하다.

예를 들어 어떤 데이터에 대해 동기적으로 데이터베이스에 질의하는 애플리케이션이나 서로 통신하는 마이크로 서비스 모음을 가질 수 있다.

그러나 더욱 많은 시스템들이 통신해야 한다면 점대점 통신은 확장하기 어려워지며, 추론하기 및 유지보수하기 어려울 수 있는 복잡한 통신 경로망으로 이끈다. 그림 1-2 는 비교적 적은 수의 시스템이라도 얼마나 혼동될 수 있는지를 보여준다.

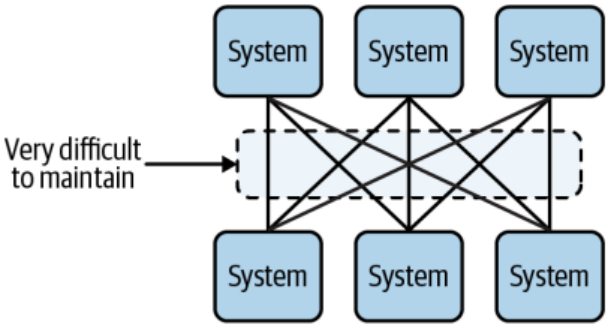

그림 1-2. 시스템을 보다 추가하는 것은 유지보수하기 어려운 복잡한 통신채널망을 만든다.

클라이언트-서버 모델의 단점은 다음을 포함한다:

- 서로 알아야 통신을 할 수 있기 때문에 시스템은 단단히 결합된다. 이는 시스템 유지보수 및 업데이트를 필요 이상으로 어렵게 한다.

- 동기식 통신은 시스템 중 하나가 오프라인이 되는 경우 전달 보증을 하지 않기 때문에 오류의 여지가 많다.
- 시스템이 다양한 통신 프로토콜, 부하 증가에 따른 확장 전략, 고장 처리 전략 등을 사용할 수 있다. 따라서 유지보수해야 할 여러 종 (software speciation)의 시스템이 될 수 있으며 이는 유지관리성을 손상시키고 애플리케이션을 애완 동물 대신 소처럼 다뤄야 한다는 일반적인 지혜를 무시한다 (애완 동물: 메인프레임, 단독 서버, 고가용성 로드 밸런서/방화벽, DB 시스템 등과 같이 다운되서는 안되는 시스템, 소: 자동화 툴을 사용하여 구축된 2 개 이상의 서버로 구성되어 내고장성 등이 보장되는 시스템).
- 수신 시스템이 새로운 요청 또는 데이터가 들어오는 속도를 제어하지 못하기 때문에 쉽게 압도될 수 있다. 요청 버퍼가 없는 경우 요청하는 애플리케이션 작동에 따라 수신 시스템이 작동한다.
- 시스템들 간 무엇이 통신되어야 하는지에 대한 명확한 개념이 없다. 클라이언트-서버 모델 명명법은 요청과 응답을 너무 강조하며 데이터 자체는 충분히 강조하지 않는다. 데이터가 데이터 기반 시스템의 변곡점이어야 한다.
- 통신이 재생될 수 없다. 이는 시스템 상태 재구축을 어렵게 만든다.

카프카는 서로 모르는 시스템들이 데이터를 주고받을 수 있는 중앙 통신 허브 (중추 신경 시스템) 역할을 함으로써 시스템 간 통신을 단순화한다. 구현한 통신 패턴은 발행-구독 패턴 (또는 퍼브/서브, publication/subscription)으로 그림 1-3 에서 보듯이 훨씬 더 단순한 통신 모델이다.

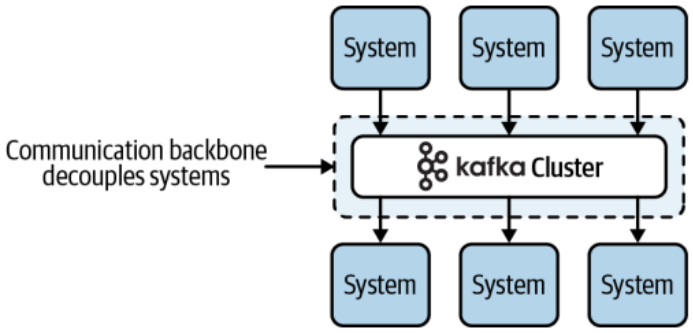

그림 1-3. 카프카는 시스템 간 통신 허브 역할을 함으로써 점대점 통신의 복잡성을 제거한다.

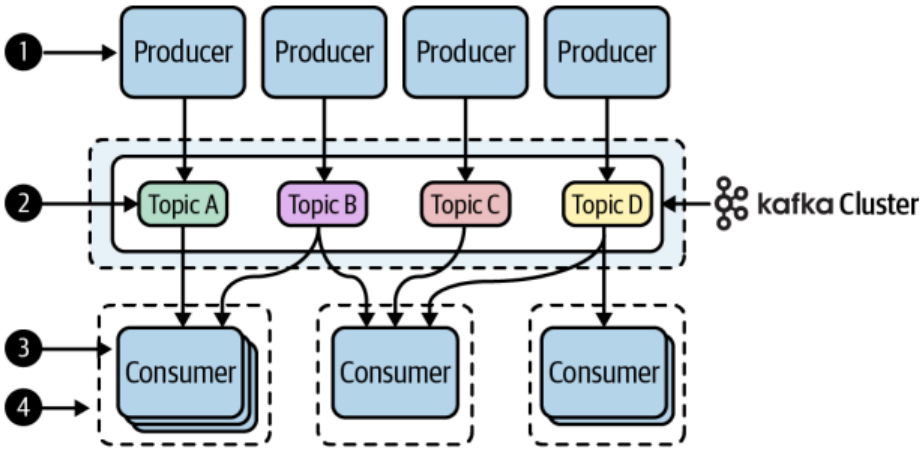

그림 1-4. 카프카 플랫폼의 주요 구성요소를 보여주기 위해 보다 세부적으로 그린 카프카 통신 모델

① 여러 시스템이 서로 직접 통신하게 하는 대신 프로듀서는 데이터를 읽을 대상에 상관없이 하나 이상의 토픽에 단순히 데이터를 발행한다.

② 토픽은 카프카 클러스터에 저장된 연관 데이터의 명명된 스트림 (또는 채널)이다. 이는 데이터베이스 내 테이블과 비슷한 목적을 갖고 있다 (예, 연관 데이터 그룹화). 그러나 특별한 스키마를 부과하지 않으며 작업하기 매우 유연하도록 원시 바이트 데이터를 저장한다[1].

③ 컨슈머는 하나 이상의 토픽에서 데이터를 읽는 (또는 이에 구독한) 프로세스이다. 이는 프로듀서와 직접 통신하지 않으며 스트림에서 발생하는 관심있는 데이터를 수신대기하고 있다.

④ 컨슈머는 여러 프로세스로 작업을 분산시키기 위해 컨슈머 그룹으로 함께 작동할 수 있다.

카프카 통신 모델은 여러 프로세스가 쉽게 읽고 쓸 수 있는 흐르는 데이터 스트림에 더욱 중점을 두며 다음을 포함하여 여러 장점을 갖고 있다:

- 시스템들이 서로 모르는 상태에서 데이터를 생산 및 소비할 수 있기 때문에 시스템이 분리되고 유지보수하기가 더욱 쉬워진다.
- 비동기 통신으로 보다 강력한 전달 보증을 지원한다. 컨슈머가 다운되면 다시 온라인 상태가 될 때 중단된 곳으로부터 단순히 데이터를 가져갈 것이다 (또는 컨슈머 그룹에 여러 컨슈머가 있을 때에는 작업이 다른 멤버 중 하나로 재분배될 것이다).
- 시스템은 통신 프로토콜 (카프카 클러스터와 통신할 때 고성능 바이너리 TCP 프로토콜이 사용된다) 뿐만 아니라 확장 전략과 내고장성 메커니즘 (컨슈머 그룹에 의해 구동)에 대해

[1] 토픽에 저장되는 원시 바이트 배열과 바이트를 JSON 객체/Avro 레코드와 같은 보다 상위 수준의 구조로 역직렬화하는 프로세스에 대해서는 3 장에서 논의한다.

표준화할 수 있다. 이를 통해 대략적으로 일관성이 있고 목적에 맞는 소프트웨어를 작성할 수 있다.

- 컨슈머는 자신들이 처리할 수 있는 속도로 데이터를 처리할 수 있다. 미처리 데이터는 컨슈머가 다시 처리할 준비가 될 때까지 내구성 및 내고장성 방식으로 카프카에 저장되어 있다. 다른 말로 컨슈머가 읽어 들이는 스트림이 갑자기 증가하면 카프카 클러스터가 버퍼 역할을 하여 컨슈머가 압도되는 것을 방지할 것이다.
- 이벤트 형태로 어떤 데이터가 통신되는 지에 대한 더욱 강력한 개념. 이벤트는 "이벤트" 절에서 논의할 특정 구조를 갖는 데이터 조각이다. 요점은 클라이언트-서버 모델과 같이 통신 계층을 푸는데 많은 시간을 소비하는 대신 스트림을 통해 흐르는 데이터에 집중할 수 있다는 것이다.
- 토픽 내 이벤트를 재생함으로써 시스템은 언제라도 상태를 재구축할 수 있다.

발행/구독 모델과 클라이언트-서버 모델 간 한 가지 중요한 차이는 카프카 발행/구독 모델에서 통신이 양방향이 아니라는 것이다. 다른 말로 스트림은 한 방향으로만 흐른다. 시스템이 카프카 토픽에 데이터를 생산하고 데이터를 갖고 무엇을 할 다른 시스템에 의존한다면 (예, 데이터 보강 또는 변환) 보강된 데이터는 다른 토픽에 작성되어야 하고 추후 원래 프로세스에 의해 소비되어야 할 것이다. 이를 중재하는 것은 간단하며 이는 통신에 대해 생각하는 방식을 변경시킨다.

통신 채널이 본질적으로 스트림과 같음을 기억한다면 (즉, 한 방향으로 흐르고 여러 소스와 여러 다운스트림 컨슈머를 가질 수 있다) 단순히 무엇이던 관심있는 흐르는 바이트 스트림을 수신대기하고 하나 이상의 시스템과 데이터를 공유하고자 할 때 데이터를 토픽 (명명된 스트림)에 생산하는 시스템을 설계하는 것은 쉽다. 다음 장들에서 카프카 토픽을 갖고 많은 작업을 수행할 것이며 (각각의 카프카 스트림즈와 ksqlDB 애플리케이션은 카프카 토픽에서 읽고 보통 하나 이상의 카프카 토픽에 작성할 것이다) 따라서 이 책의 끝에 도달할 때까지 토픽은 여러분에게 두번째 자연일 것이다.

지금까지 카프카 통신 모델이 시스템들 간 통신 방식을 단순화하는 방법과 토픽이라는 명명된 스트림이 시스템들 간 통신 매체의 역할을 함을 살펴보았는데, 카프카 스토리지 계층에서 스트림이 하는 역할에 대해 깊게 이해해보자.

스트림 저장 방식

링크드인 엔지니어 팀[2]이 스트림 기반 데이터 플랫폼에서 가능성을 보았을 때 그들은 "무제한의 연속적인 데이터 스트림을 스토리지 계층에서 어떻게 모델링해야 할까?"라는 중요한 질문에 답을 해야 했다.

[2] Jay Kreps, Neha Narkhede와 Jun Rao가 초기에 카프카 개발을 주도했다.

최종적으로 그들이 확인한 스토리지 추상화는 전통적인 데이터베이스, 키-값 저장소, 버전 제어 시스템 등을 포함하여 많은 유형의 데이터 시스템에 이미 존재했었다. 추상화는 단순하지만 강력한 커밋 로그(또는 단순히 로그)이다.

> 이 책에서 로그에 대해 논의할 때 실행 프로세스에 대한 정보를 방출하는 애플리케이션 로그 (예, HTTP 서버 로그)를 언급하는 것이 아니다. 대신 다음 단락에서 기술되는 특정 데이터 구조를 언급하는 것이다.

로그는 순서있는 이벤트 시퀀스를 포착하는 추가 전용(append-only) 데이터 구조이다. 기울임꼴 속성을 보다 자세히 살펴보고 커맨드 라인에서 간단한 로그를 생성함으로써 로그 관련 직관을 구축해보자. 예를 들어 user_purchases 라는 로그를 생성하고 다음 명령을 사용하여 더미 데이터로 채워보자.

```
# create the logfile
touch users.log
# generate four dummy records in our log
echo "timestamp=1597373669,user_id=1,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=2,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=3,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=4,purchases=1" >> users.log
```

이제 생성한 로그를 살펴본다면 이는 하나를 구매한 4 명의 사용자를 포함하고 있다:

```
# print the contents of the log
cat users.log
# output
timestamp=1597373669,user_id=1,purchases=1
timestamp=1597373669,user_id=2,purchases=1
timestamp=1597373669,user_id=3,purchases=1
timestamp=1597373669,user_id=4,purchases=1
```

로그의 첫번째 속성은 추가 전용 방식으로 작성된다는 것이다. 이는 users_id=1 이 오고 두번째 구매를 하더라도 각 레코드가 로그에서 불변이기 때문에 첫번째 레코드를 업데이트 하지 않는다는 것을 의미한다. 대신 새로운 레코드를 끝에 단지 추가한다:

```
# append a new record to the log
```

```
echo "timestamp=1597374265,user_id=1,purchases=2" >> users.log

# print the contents of the log

cat users.log

# output

timestamp=1597373669,user_id=1,purchases=1 ①

timestamp=1597373669,user_id=2,purchases=1

timestamp=1597373669,user_id=3,purchases=1

timestamp=1597373669,user_id=4,purchases=1

timestamp=1597374265,user_id=1,purchases=2 ②
```

① 레코드가 로그에 작성되면 이는 불변으로 고려된다. 따라서 업데이트를 해도 (예, 사용자에 대한 구매 카운트를 변경해도) 원래 레코드는 변경되지 않고 유지된다.

② 업데이트를 모델링하기 위해 로그 끝에 새로운 값을 단순히 추가한다. 로그는 둘 모두 불변인 이전 레코드와 새로운 레코드를 포함할 것이다.

각 사용자에 대한 구매 카운트를 조사하려는 시스템은 단순히 로그 내 각 레코드를 읽을 수 있으며 user_id=1 인 마지막 레코드가 업데이트된 구매량을 포함할 것이다. 이는 로그의 두번째 속성으로 순서가 있다는 것이다.

이전 로그는 타임스탬프 순서대로 발생하지만 (첫번째 칼럼 참조) 우리가 의미하는 순서가 있다는 것은 아니다. 사실 카프카는 로그에 각 레코드의 타임스탬프를 저장하지만 레코드가 타임스탬프 순서일 필요는 없다. 로그가 순서가 있다라고 할 때 이는 로그 내 레코드 위치가 고정되어 절대로 변경되지 않는다는 것을 의미한다. 로그를 라인 숫자와 함께 다시 출력하면 첫번째 칼럼의 위치를 볼 수 있다.

```
# print the contents of the log, with line numbers

cat -n users.log

# output

1 timestamp=1597373669,user_id=1,purchases=1

2 timestamp=1597373669,user_id=2,purchases=1

3 timestamp=1597373669,user_id=3,purchases=1

4 timestamp=1597373669,user_id=4,purchases=1

5 timestamp=1597374265,user_id=1,purchases=2
```

이제 순서가 보장되지 않을 수 있는 시나리오를 상상해보자. 여러 프로세스가 다른 순서로 user_id=1 의 업데이트를 읽을 수 있고 이는 사용자에 대한 실제 구매 카운트에 대해 불일치를 야기한다. 로그가 순서가 있음을 보장함으로써 데이터가 여러 프로세스[3]에 의해 결정론적으로[4] 처리될 수 있다.

또한 이전 예에서 각 로그 엔트리의 위치로 라인 숫자를 사용했지만 카프카는 분산 로그에서 각 엔트리 위치를 오프셋으로 간주한다. 오프셋은 0 에서 시작하며 중요한 동작을 가능케한다: 여러 컨슈머 그룹이 동일 로그에서 각각 읽으며, 읽어 들이는 로그/스트림 내에서 자신의 위치를 유지할 수 있도록 한다. 그림 1-5 는 이를 보여준다.

커맨드 라인에서 자체 로그를 생성하여 카프카 로그 기반 스토리지 계층과 관련하여 직관을 얻었기 때문에 이러한 아이디어를 카프카 통신 모델에서 식별했던 보다 상위 수준의 구성에 연결해보자. 주제에 대한 논의를 계속하고 파티션이라는 것에 대해 배울 것이다.

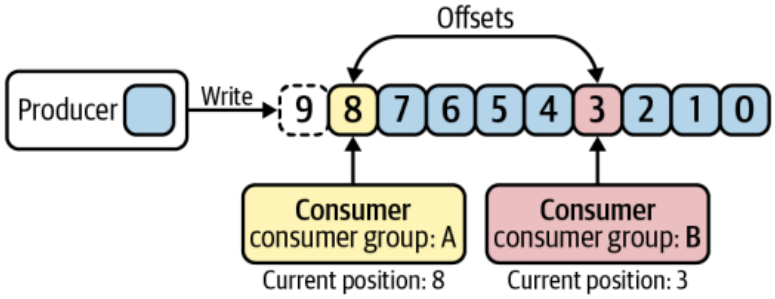

그림 1-5. 여러 컨슈머 그룹이 각각 읽었던/처리했던 오프셋에 기반하여 위치를 유지하면서 동일 로그에서 읽을 수 있다.

토픽과 파티션

카프카 통신 모델 논의에서 카프카가 토픽이라는 명명된 스트림의 개념을 갖고 있음을 배웠다. 또한 카프카 토픽은 내부에 저장하는 것에 매우 유연하다. 예를 들어 단지 한 가지 유형의 데이터만 갖는 동종 토픽 또는 여러 유형의 데이터를 갖는 이종 토픽을 가질 수 있다[5]. 그림 1-6 은 이러한 다른 전략의 설명을 보여준다.

---

[3] 이것이 전통적인 데이터베이스가 복제에 로그를 사용하는 이유이다. 로그는 리더 데이터베이스에 대한 각각의 쓰기 작업을 포착하여 다른 머신에 동일 데이터셋을 결정론적으로 재생성하기 위해 복제 데이터베이스에 순서대로 동일 쓰기를 처리하는데 사용된다.

[4] 결정론적이라는 것은 동일 입력이 동일 출력을 산출할 것임을 의미한다.

[5] Martin Kleppmann이 이 주제에 대해 흥미로운 기사를 작성하였다. 이는 https://oreil.ly/tDZMm에서 찾을 수 있다. 그는 다양한 트레이드 오프와 여러 전략 중 하나를 선택하는 이유에 대해서 논의하였다. 또한 Robert Yokota의 후속 기사에서는 스키마 관리/진화를 위해 컨플루언트 스키마 레지스트리를 사용할 때 여러 이벤트 유형을 지원하는 방법을 보다 심도있게 다루었다.

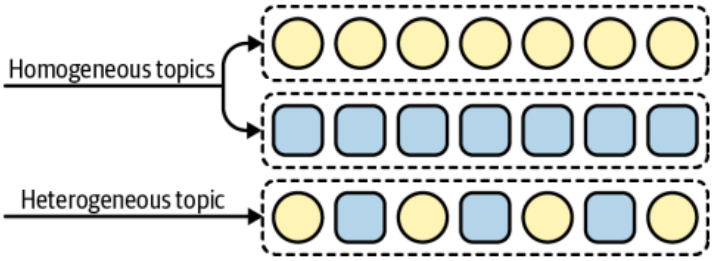

그림 1-6. 토픽에 이벤트 저장을 위한 다른 전략이 존재한다; 동종 토픽은 일반적으로 한 가지 이벤트 유형 (예, 클릭)을 포함하지만 이종 토픽은 여러 이벤트 유형 (예, 클릭과 페이지 뷰)을 포함한다.

또한 카프카 스토리지 계층에서 추가 전용 커밋 로그가 스트림을 모델링하기 위해 사용된다는 것도 배웠다. 그렇다면 각 토픽이 로그 파일과 상호 연관됨을 의미하는가? 정확히 그렇지는 않다. 여러분이 알고 있듯이 카프카는 분산 로그로 단지 어떤 것 중 하나만을 분산하는 것은 어렵다. 따라서 로그 분산 및 처리 방식으로 어떤 수준의 병렬화를 얻으려면 이들을 많이 생성해야 한다. 이것이 카프카 토픽이 파티션이라는 보다 작은 단위로 나뉜 이유이다.

파티션은 데이터가 생산되고 소비되는 개별 로그이다 (즉, 이전 절에서 논의했던 데이터 구조). 커밋 로그 추상화가 파티션 수준에서 구현되어 있기 때문에 이는 각 파티션이 일련의 자체 오프셋을 갖고 순서가 보장되는 수준이다. 토픽 수준에서 글로벌 순서는 지원되지 않으며 이는 프로듀서가 종종 연관 레코드를 동일 파티션으로 전달하는 이유이다[6].

이상적으로 데이터는 토픽 내 모든 파티션에 비교적 균등하게 분산될 것이다. 그러나 다른 크기의 파티션이 될 수도 있다. 그림 1-7 은 3 개의 다른 파티션을 갖는 토픽 예를 보여준다.

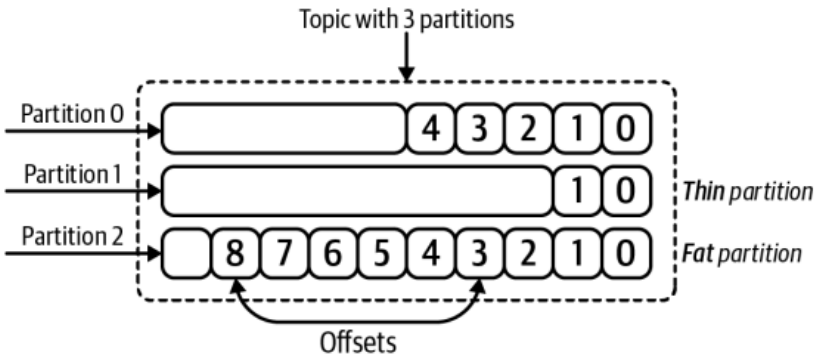

그림 1-7. 3 개의 파티션으로 구성된 카프카 토픽

주어진 토픽에 대해 파티션 수는 구성 가능하며 너무 많은 파티션을 갖는 것에 대해 트레이드 오프가 존재하지만 토픽에 보다 많은 파티션을 만드는 것은 일반적으로 보다 높은 병렬화와 처리량으로

---

[6] 파티셔닝 전략은 구성 가능하며 카프카 스트림즈와 ksqlDB에 구현되어 있는 전략을 포함하여 인기 있는 전략은 (레코드 페이로드에서 추출될 수 있거나 또는 명시적으로 설정된) 레코드 키에 기반해 파티션을 설정하는 것이다. 이를 다음 몇 장에서 보다 세부적으로 논의할 것이다.

변환된다[7]. 이를 책을 통해 논의할 것이며 중요한 요점은 컨슈머 그룹 당 하나의 컨슈머만이 파티션에서 소비할 수 있다는 것이다 (그림 1-5 에서 보듯이 다른 컨슈머 그룹에 대한 개별 멤버는 동일 파티션에서 소비할 수 있다).

따라서, 단일 컨슈머 그룹에서 N 개의 컨슈머에 처리 부하를 분산하려면 N 개의 파티션이 필요하다. 컨슈머 그룹에 소스 토픽의 파티션 수보다 적은 멤버가 있다면 각 컨슈머가 여러 파티션을 처리할 수 있기 때문에 이것은 괜찮다. 컨슈머 그룹에 소스 토픽의 파티션 수보다 많은 멤버가 있다면 일부 컨슈머는 유휴 상태일 것이다.

이를 염두에 두고 토픽이 무엇인지에 대한 정의를 개선할 수 있다. 토픽은 여러 파티션으로 구성된 명명된 스트림으로 각 파티션은 완전히 정렬된 추가 전용 시퀀스에 데이터를 저장하는 커밋 로그로 모델링된다. 그렇다면 토픽 파티션에 정확히 무엇이 저장되는가? 다음 절에서 이를 논의할 것이다.

이벤트

이 책에서는 토픽 내 데이터 처리에 대한 논의에 많은 시간을 할애한다. 그러나 카프카 토픽 (그리고 보다 구체적으로 토픽 파티션)에 어떤 종류의 데이터가 저장되는지에 대해서는 아직까지 완전히 이해하지 못했다.

공식 문서를 포함하여 카프카에 대한 기존 문헌 중 많은 것들이 메시지, 레코드와 이벤트를 포함하여 토픽 내 데이터를 기술하기 위해 다양한 용어를 사용한다. 이들 용어는 종종 상호 교환되어 사용되지만 이 책에서 선호하는 용어는 이벤트이다 (다른 용어를 우연히 사용하고 있지만). 이벤트는 발생한 무언가를 기록한 타임스탬프를 갖는 키-값 쌍이다. 그림 1-8 은 토픽 파티션에 저장되는 각 이벤트의 기본 구조를 보여준다.

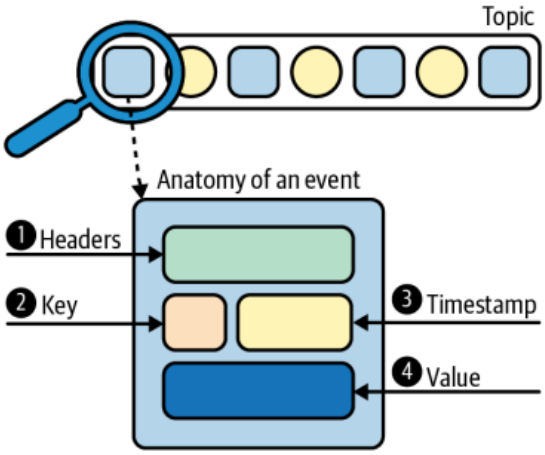

그림 1-8. 토픽 파티션에 저장되는 이벤트 구조

---

[7] 트레이드 오프는 특정 실패 시나리오로 보다 긴 복구 기간, 리소스 이용률 증가 (파일 설명자, 메모리)와 양단 간 지연 증대를 포함한다.

① 애플리케이션 수준 헤더는 이벤트에 대한 선택적인 메타 데이터를 포함한다. 이 책에서는 헤더를 통해 거의 작업을 하지 않는다.

② 키 또한 선택사항이지만 데이터가 파티션에 분산되는 방식에서 중요한 역할을 한다. 이를 다음 몇 장에 걸쳐 살펴볼 것이며 일반적으로 이는 연관 레코드를 식별하는데 사용된다.

③ 각 이벤트는 타임스탬프와 연관된다. 5 장에서 타임스탬프에 대해 자세히 배울 것이다.

④ 값은 바이트 배열로 인코딩된 실제 메시지 컨텐츠를 포함한다. 원시 바이트를 보다 의미있는 구조(예, JSON 객체 또는 Avro 레코드)로 역직렬화하는 것은 클라이언트의 몫이다. "직렬화/역직렬화"에서 바이트 배열 역직렬화를 자세히 논의할 것이다.

이제 토픽에 어떤 데이터가 저장되는 지를 이해했기 때문에 카프카 클러스터 배치 모델을 보다 깊게 살펴보자. 이는 카프카에 데이터가 물리적으로 저장되는 방식에 대해 보다 많은 정보를 제공할 것이다.

카프카 클러스터와 브로커

중앙화된 통신 접점을 갖는다는 것은 신뢰성과 내고장성이 매우 중요하다는 것을 의미한다. 이는 또한 통신 백본이 확장가능해야 함을 의미한다. 즉 부하 증가량을 처리할 수 있어야 한다. 이것이 카프카가 클러스터로 동작하는 이유로 브로커라고 하는 여러 머신이 데이터 스토리지와 검색에 관여되어 있다.

카프카 클러스터는 매우 클 수 있으며 심지어 여러 데이터 센터와 지리적인 지역으로 확장될 수 있다. 그러나 이 책에서는 보통 단일 노드 카프카 클러스터로 작업할 것이다. 왜냐하면 그것이 카프카 스트림즈와 ksqlDB 로 작업을 시작하는데 필요한 전부이기 때문이다. 운영 환경에서는 최소한 3 개의 브로커를 원할 가능성이 높고 데이터가 여러 브로커에 복제되도록 토픽의 복제를 설정하길 원할 것이다 (이 장 튜토리얼 후반부에 이를 살펴볼 것이다). 이를 통해 하나의 머신이 다운되는 경우에도 고가용성을 얻고 데이터 손실을 방지할 수 있다.

이제 브로커에 저장 및 복제되는 데이터에 대해 논의할 때 실제로는 토픽의 개별 파티션에 대해 논의하고 있는 것이다. 예를 들어 그림 1-9 에서 보듯이 토픽은 3 개의 브로커에 대해 분산되어 있는 3 개의 파티션을 가질 수 있다.

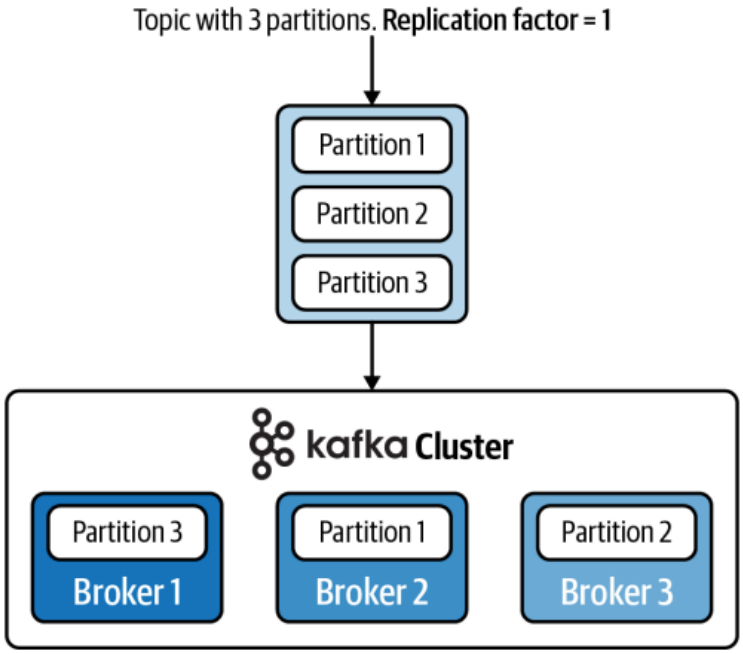

그림 1-9. 파티션은 사용가능한 브로커에 분산되어 있으며 이는 토픽이 카프카 클러스터 내 여러 머신에 걸쳐 있을 수 있음을 의미한다.

위에서 볼 수 있듯이 이를 통해 단일 머신의 용량을 초과하여 커질 수 있기 때문에 토픽은 매우 클 수 있다. 내고장성과 고가용성을 얻기 위해 토픽 구성 시 복제 수(replication factor)를 설정할 수 있다. 예를 들어 복제 수 2 를 통해 2 개의 다른 브로커에 파티션이 저장될 수 있다. 그림 1-10 은 이를 보여준다.

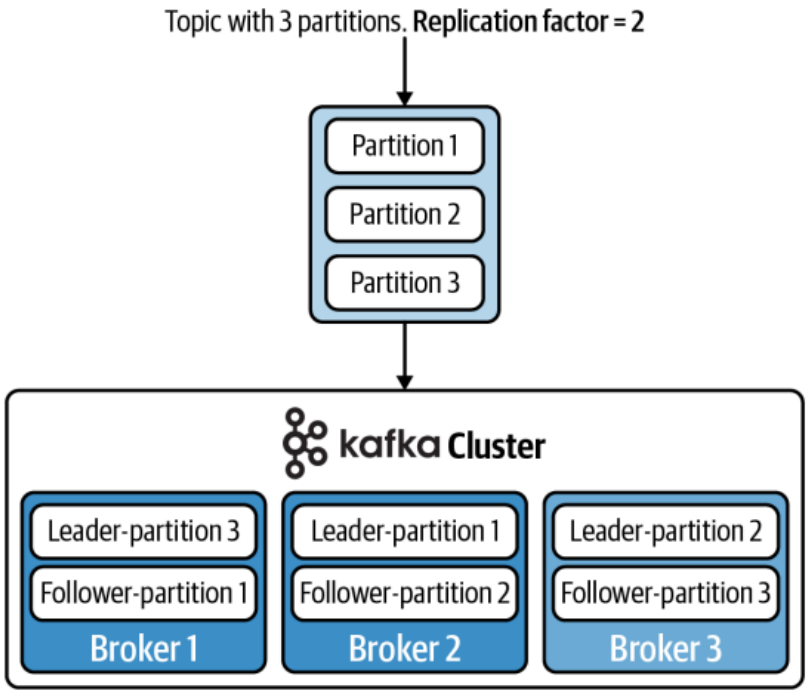

그림 1-10. 복제 수를 2 로 늘림으로써 2 개의 다른 브로커에 파티션이 저장될 것이다.

파티션이 여러 브로커로 복제될 때 한 브로커가 리더로 지정될 것인데, 이는 주어진 파티션에 대해 프로듀서/컨슈머로부터의 모든 읽기/쓰기 요청을 리더가 처리할 것임을 의미한다. 복제 파티션을 갖는 다른 브로커는 팔로워로 불리는데 이들은 리더로부터 데이터를 단순히 복제한다. 리더가 고장나는 경우 팔로워 중 하나가 새로운 리더로 선출될 것이다.

또한 시간이 지남에 따라 클러스터 부하가 증가한다면 여러 브로커를 추가하고 파티션 재할당을 통해 클러스터를 확장할 수 있다. 이를 통해 오래된 머신으로부터 새로운 머신으로 데이터를 이관할 수도 있다.

마지막으로 브로커는 컨슈머 그룹의 멤버십 유지에 중요한 역할을 한다. 다음 절에서 이를 논의할 것이다.

컨슈머 그룹

카프카는 높은 처리량과 낮은 지연에 최적화되어 있다. 컨슈머 쪽에서 이를 이용하기 위해서는 여러 프로세스로 작업을 병렬화할 수 있어야 한다. 이는 컨슈머 그룹을 통해 가능하다.

컨슈머 그룹은 협업하는 여러 컨슈머로 구성되며 이들 그룹의 멤버십은 시간에 따라 변할 수 있다. 예를 들어 처리 부하를 확장하기 위해 새로운 컨슈머가 들어올 수 있고 계획 유지보수 또는 예기치 못한 고장으로 인해 컨슈머가 오프라인이 될 수 있다. 따라서 카프카는 각 그룹의 멤버십을 관리하고 필요 시 작업을 재분배하는 방식이 필요하다.

이를 용이하게 하기 위해 모든 컨슈머 그룹은 컨슈머로부터 하트비트를 수신하고 컨슈머가 죽었다고 표시될 때마다 작업 재분배를 트리거링하는 역할을 하는 그룹 중재자라는 특수 브로커에 배정된다. 그림 1-11 은 그룹 중재자에 하트비트를 전송하는 컨슈머에 대한 설명을 보여준다.

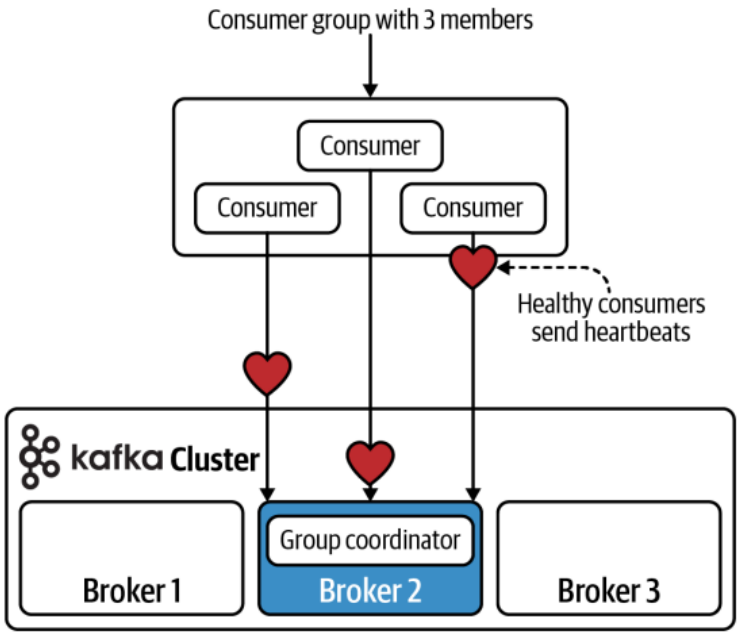

그림 1-11. 그룹 중재자에 하트비트를 전송하는 그룹 내 3 개의 컨슈머

컨슈머 그룹의 모든 활성 멤버는 파티션 배정을 수신할 수 있다. 예를 들어 정상 컨슈머에 대한 작업 분배는 그림 1-12 와 같이 보일 수 있다.

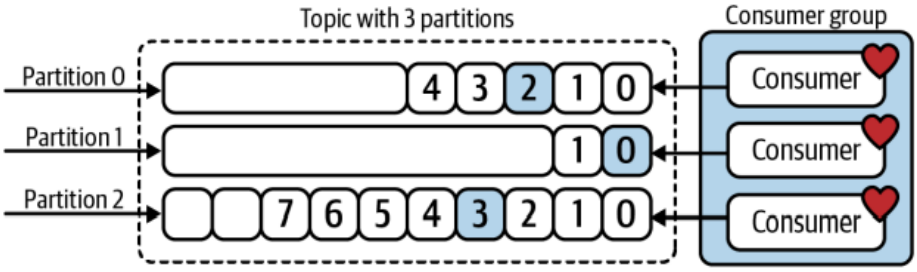

그림 1-12. 3 개 파티션을 갖는 토픽의 읽기/처리 워크로드를 분리하는 3 개의 정상 컨슈머

그러나 컨슈머 인스턴스가 비정상이 되고 클러스터에 하트비트를 전송할 수 없는 경우 작업은 정상 컨슈머에 자동으로 재할당될 것이다. 예를 들어 그림 1-13 에서 중간의 컨슈머에 이전에 비정상 컨슈머가 처리하던 파티션이 할당되었다.

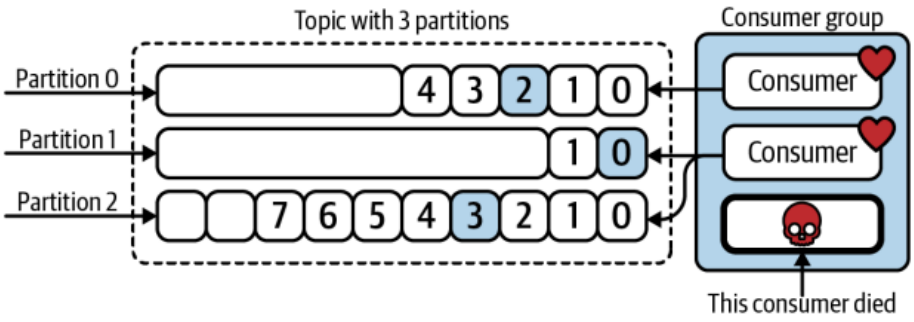

그림 1-13. 컨슈머 프로세스가 실패일 때 작업이 재분배된다.

위에서 볼 수 있듯이 컨슈머 그룹은 데이터 처리 계층에서 고가용성과 내고장성을 얻는데 있어 매우 중요하다. 이것으로 카프카 설치 방법을 배워 튜토리얼을 시작해보자.

캬프카 설치

공식 문서에 수동으로 카프카 설치를 위한 세부 지침이 존재한다. 그러나 가능한 단순하게 유지하기 위해 이 책 대부분의 튜토리얼은 컨테이너 환경 내부에 카프카와 스트림 처리 애플리케이션을 배치할 수 있도록 하는 도커를 활용한다.

따라서 도커 컴포즈를 사용하여 카프카를 설치할 것이며 컨플루언트[8]가 게시한 도커 이미지를 사용할 것이다. 첫번째 단계는 도커 설치 페이지로부터 도커를 다운로드받아 설치하는 것이다.

---

[8] 카프카 실행을 위해 선택할 수 있는 많은 도커 이미지가 있다. 그러나 컨플루언트가 이 책에서 사용할 ksqlDB와 컨플루언트 스키마 레지스트리를 포함하여 다른 기술들에 대한 도커 이미지도 제공하기 때문에 컨플루언트 이미지가 편리한 선택이다.

다음은 docker-compose.yml 파일에 다음 구성을 저장한다:

```
---
version: '2'
services:
  zookeeper: ①
    image: confluentinc/cp-zookeeper:6.0.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
  kafka: ②
    image: confluentinc/cp-enterprise-kafka:6.0.0
    hostname: kafka
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

① zookeeper 라는 이름의 컨테이너는 주키퍼 설치를 포함할 것이다. 이 책 작성 시점에 주키퍼가 카프카로부터 제거되는 중이기 때문에 이 소개에서 주키퍼에 대해서는 논의하지 않았다. 그러나 이는 토픽 구성과 같은 메타 데이터 저장을 위한 중앙 서비스이다. 이는 조만간 카프카에 더 이상 포함되지 않을 것이지만 주키퍼가 완전히 제거되기 전에 이 책이 발표되었기 때문에 이를 포함하였다.

② kafka 라는 이름의 컨테이너는 카프카 설치를 포함할 것이다. 이는 브로커 (단일 노드 클러스터를 구성)가 실행되고 클러스터와 상호작용을 위해 카프카 컨솔 스크립트를 실행할 장소이다.

마지막으로 로컬 카프카 클러스터를 시작하기 위해 다음 명령을 실행한다:

```
docker-compose up
```

카프카 클러스터가 실행 중으로 튜토리얼을 진행할 준비를 마쳤다.

헬로우 카프카

이 간단한 튜토리얼에서는 카프카 토픽 생성, 프로듀서를 사용한 토픽에 데이터 작성, 마지막으로 컨슈머를 사용한 토픽에서 데이터 읽기를 어떻게 하는지 보여줄 것이다. 해야 할 첫번째 일은 카프카가 설치된 컨테이너에 로그인하는 것이다. 다음 명령을 실행하여 로그인할 수 있다"

```
docker-compose exec kafka bash
```

이제 users 라는 토픽을 생성해보자. 카프카에 포함되어 있는 컨솔 스크립트 중 하나인 (kafka-topics)를 사용할 것이다. 다음 명령은 토픽 생성 방법을 보여준다:

```
kafka-topics ₩ ①
        --bootstrap-server localhost:9092 ₩ ②
        --create ₩ ③
        --topic users ₩ ④
        --partitions 4 ₩ ⑤
        --replication-factor 1 ⑥
# output
Created topic users.
```

① kafka-topics 는 카프카에 포함되어 있는 컨솔 스크립트이다.

② 부트스트랩 서버는 하나 이상의 브로커에 대한 호스트/포트 쌍이다.

③ --list, --describe 와 --delete 를 포함하여 카프카 토픽과 상호작용하기 위한 많은 플래그가 존재한다. 여기서는 새로운 토픽을 생성하기 때문에 --create 플래그를 사용한다.

④ 토픽명은 users 이다.

⑤ 토픽을 4 개의 파티션으로 나눈다.

⑥ 단일 노드 클러스터를 실행하고 있기 때문에 복제 수를 1 로 설정할 것이다. 운영 환경에서는 고가용성을 보장하기 위해 (3 과 같은) 큰 값으로 설정하길 원할 것이다.

> 이 절에서 사용하는 컨솔 스크립트는 카프카 소스 배포판에 포함되어 있다. 바닐라 카프카 설치의 경우 이들 스크립트의 확장자는 .sh 이다 (예, kafka-topics.sh, kafka-consoleproducer.sh 등). 그러나 컨플루언트 플랫폼에서는 확장자가 제거되어 있다 (이전 코드 스니펫에서 kafka-topics.sh 이 아닌 kafka-topics 를 실행한 이유이다).

토픽이 생성되었다면 다음 명령을 사용하여 구성을 포함한 토픽의 설명을 출력할 수 있다.

```
kafka-topics ₩
```

```
        --bootstrap-server localhost:9092 ₩
        --describe ₩ ①
        --topic users
# output
Topic: users PartitionCount: 4 ReplicationFactor: 1 Configs:
        Topic: users Partition: 0 Leader: 1 Replicas: 1 Isr: 1
        Topic: users Partition: 1 Leader: 1 Replicas: 1 Isr: 1
         Topic: users Partition: 2 Leader: 1 Replicas: 1 Isr: 1
        Topic: users Partition: 3 Leader: 1 Replicas: 1 Isr: 1
```

① --describe 플래그를 통해 토픽에 대한 구성 정보를 볼 수 있다.

이제 내장 kafka-console-producer 스크립트를 사용하여 데이터를 생산해보자.

```
kafka-console-producer ₩ ①
        --bootstrap-server localhost:9092 ₩
        --property key.separator=, ₩ ②
        --property parse.key=true ₩
        --topic users
```

① kafka-console-producer 스크립트는 토픽에 데이터를 생산하기 위해 사용될 수 있다. 그러나 카프카 스트림즈와 ksqlDB 로 작업한다면 프로듀서 프로세스는 기반 Java 라이브러리에 내장될 것이다. 따라서 테스팅과 개발 목적 외에 이 스크립트를 사용할 필요는 없다.

② users 토픽에 일련의 키-값 쌍을 생산할 것이다. 이 속성은 키와 값이 , 문자로 구분될 것임을 나타낸다.

이전 명령을 통해 상호대화식 프롬프트로 갈 것이다. 여기서 users 토픽에 생산할 몇 개의 키-값 쌍을 입력할 수 있다. 완료한 후 프롬프트에서 나오기 위해 키보드의 Control-C 를 누르기 바란다:

```
>1,mitch
>2,elyse
>3,isabelle
>4,Sammy
```

토픽에 데이터를 생산한 후 데이터를 읽기 위해 kafka-console-consumer 스크립트를 사용할 수 있다. 이를 위해 다음 명령을 실행한다:

```
kafka-console-consumer ₩ ①
        --bootstrap-server localhost:9092 ₩
        --topic users ₩
        --from-beginning ②
# output
mitch
elyse
isabelle
Sammy
```

① kafka-console-consumer 또한 카프카 배포판에 포함되어 있다. Kafka-console-producer 스크립트에 대해 언급한 바와 비슷하게 이 책 대부분의 튜토리얼은 (테스팅 목적에 유용한) 이 독립형 콘솔 스크립트 대신 카프카 스트림즈와 ksqlDB 내에 구축된 컨슈머 프로세스를 활용한다.

② --from-beginning 플래그는 토픽의 처음부터 소비함을 나타낸다.

기본적으로 kafka-console-consumer 는 메시지 값만 출력할 것이다. 그러나 초반에 배웠듯이 이벤트는 키, 타임스탬프, 헤더를 포함하여 실제 보다 많은 정보를 포함하고 있다. 타임스탬프와 키 값 또한 볼 수 있도록 컨솔 컨슈머에 추가적인 속성을 전달해보자[9]:

```
kafka-console-consumer ₩
        --bootstrap-server localhost:9092 ₩
        --topic users ₩
        --property print.timestamp=true ₩
        --property print.key=true ₩
        --property print.value=true ₩
        --from-beginning
# output
```

[9] 2.7 버전에서 메시지 헤더 출력을 위해 --property print.header=true 플래그를 사용할 수 있다.

```
CreateTime:1598226962606 1 mitch

CreateTime:1598226964342 2 elyse

CreateTime:1598226966732 3 isabelle

CreateTime:1598226968731 4 sammy
```

이게 다다. 이제 카프카 클러스터와 기본적인 상호작용을 수행하는 방법을 배웠다. 마지막 단계는 다음 명령을 사용하여 로컬 클러스터를 없애는 것이다.

```
docker-compose down
```

요약

카프카 통신 모델은 여러 시스템이 통신하는 것을 쉽게 만들며, 빠르고 내구성있는 추가 전용 스토리지 계층은 빠르게 이동 중인 데이터 스트림과의 작업을 쉽게 할 수 있도록 한다. 클러스터 배치를 사용하여 카프카는 브로커라는 여러 머신에 데이터를 복제함으로써 스토리지 계층에서 고가용성 및 내고장성을 얻을 수 있다. 또한 컨슈머 프로세스에서 하트비트를 수신하고 컨슈머 그룹의 멤버십을 업데이트할 수 있는 클러스터 능력은 스트림 처리 및 소비 계층에서 고가용성, 내고장성 및 워크로드 확장성을 허용한다. 이러한 기능 모두는 카프카를 존재하는 가장 유명한 스트림 처리 플랫폼 중 하나로 만든다.

이제 카프카 스트림즈와 ksqlDB 를 시작하기 위해 카프카에 대한 충분한 배경 지식을 얻었다. 다음에는 카프카 스트림즈가 카프카 생태계에서 어디에 적합한 지 살펴 보고 스트림 처리 계층의 데이터로 작업하기 위해 이 라이브러리를 사용하는 방법을 배움으로써 여행을 시작할 것이다.