4 장 스테이트풀 처리

이전 장에서는 카프카 스트림즈에서 사용가능한 KStream 추상화와 풍부한 일련의 스테이트리스 연산자를 사용하여 레코드 스트림에 대해 스테이트리스 변환을 수행하는 법을 배웠다. 스테이트리스 변환은 이전에 보았던 이벤트에 대해 어떤 기억도 필요치 않기 때문에 추론 및 사용하기가 쉽다. 모든 이벤트가 불변의 사실로 처리되고 다른 이벤트와 상관없이 처리된다.

그러나 카프카 스트림즈는 소비하는 이벤트에 대한 정보를 포착 및 기억할 수 있는 능력 또한 제공한다. 포착된 정보 또는 상태는 데이터 조인 및 집계를 포함하여 보다 고급 스트림 처리 연산을 처리할 수 있도록 한다. 이 장에서는 스테이트풀 스트림 처리를 세부적으로 논의할 것이다. 다룰 주제는 다음을 포함할 것이다:

- 스테이트풀 처리의 장점
- 사실과 행동 간 차이
- 카프카 스트림즈에서 사용가능한 스테이트풀 연산자 종류
- 카프카 스트림즈에서 포착 및 쿼리되는 상태
- 로컬 파티션된 상태를 표현하기 위해 사용될 수 있는 KTable 추상화
- 글로벌, 복제된 상태를 표현하기 위해 사용될 수 있는 GlobalKTable 추상화
- 데이터 조인 및 집계를 포함하여 스테이트풀 처리 수행 방법
- 상태를 노출하기 위해 상호대화형 쿼리를 사용하는 방법

이전 장과 같이 튜토리얼 기반 접근방법을 사용하여 이들 개념을 탐구할 것이다. 이 장의 튜토리얼은 비디오 게임 산업에 고무된 것으로 많은 카프카 스트림즈의 스테이트풀 연산 사용을 요구할 실시간 리더보드를 구축할 것이다. 또한 조인이 스테이트풀 애플리케이션에서 가장 일반적인 데이터 보강 형태 중 하나이기 때문에 조인을 논의하는데 많은 시간을 할애할 것이다. 튜토리얼을 시작하기 전에 스테이트풀 처리의 장점 중 일부를 살펴보면서 시작해보자.

스테이트풀 처리의 장점

스테이트풀 처리를 통해 보다 고급 처리 유스케이스를 위한 이벤트 간 관계를 이해하고 이 관계를 활용할 수 있다. 이벤트가 다른 이벤트와 어떻게 관련되는지를 이해할 수 있을 때 다음을 할 수 있다:

- 이벤트 스트림에서 패턴 및 행동 인지
- 집계 수행
- 조인을 사용한 보다 복잡한 방식으로 데이터 보강

스테이트풀 처리의 다른 장점은 데이터 표현을 위해 추가적인 추상화를 제공한다는 것이다. 한 번에 한 이벤트씩 이벤트 스트림을 재생하고 내장된 키-값 저장소에 각 키의 최신 상태를 저장함으로써 연속적이고 무제한의 레코드 스트림에 대한 한 시점 표현을 구축할 수 있다. 이러한 한 시점의 표현 또는 스냅샷은 테이블로 간주되며 카프카 스트림즈는 이 장에서 배울 다양한 유형의 테이블 추상화를 포함하고 있다.

테이블은 스테이트풀 스트림 처리의 핵심일 뿐만 아니라 구체화될 때 쿼리도 할 수 있다. 빠르게 이동하는 이벤트 스트림의 실시간 스냅샷에 쿼리할 수 있는 능력은 카프카 스트림즈를 스트림-관계형 처리 플랫폼[1]으로 만드는 것으로 스트림 처리 애플리케이션 구축 뿐만 아니라 저지연의 이벤트 기반 마이크로 서비스도 구축할 수 있도록 한다.

마지막으로 스테이트풀 처리는 보다 복잡한 정신 모델을 사용하여 데이터를 이해할 수 있도록 한다. 한 가지 특히 흥미로운 관점은 이벤트 우선 사고에 대한 논의해서 사실과 행동 간 차이를 논의한 Neil Avery 의 견해이다:

> 이벤트는 발생했던 어떤 사실을 표현한다; 이는 불변이다.

이전 장에서 논의했던 스테이트리스 애플리케이션은 사실 기반이다. 각 이벤트는 독립적이고 원자적인 사실로 처리되며 이는 불변 시맨틱 (절대 끝나지 않는 스트림에서 insert 를 생각해보자)을 사용하여 처리되고 그 후 잊혀 질 수 있다.

그러나 사실을 필터링, 분기, 병합 및 변환하기 위해 스테이트리스 연산자를 활용하는 것 외에 스테이트풀 연산자를 사용하여 행동을 모델링하는 방법을 배운다면 데이터에 대해 보다 고급 질문을 할 수 있다. 그렇다면 작동은 무엇인가? Neil 에 따르면:

> 사실의 축적이 행동을 포착한다.

여러분은 실제 세계에서 이벤트 (또는 사실)가 거의 별개로 발생하지 않음을 알고 있다. 모든 것은 상호 연결되어 있으며 사실을 포착 및 기억함으로써 의미를 이해할 수 있다. 이는 이벤트를 보다 큰 이력 컨텍스트에서 이해하고 애플리케이션에 의해 포착 및 저장된 다른 관련 이벤트를 살펴봄으로써 가능하다.

유명한 예로는 복수의 사실로 구성된 행동인 장바구니 포기이다: 사용자가 하나 이상의 항목을 장바구니에 추가하고 그 후 세션이 수동으로 (사용자 로그 오프) 또는 자동으로 (장기간의 비활성으로 인해) 종료된다. 각각의 사실을 독립적으로 처리한다면 사용자가 결제 프로세스에 있는지에 대해 거의 말하지 못한다. 그러나 각각의 사실을 수집, 기억 및 분석 (스테이트풀 처리가 가능하도록 하는

---

[1] 스트림-관계형 처리 플랫폼에 대해 보다 자세한 정보는 Robert Yokota의 2018년 블로그를 참조하기 바란다.

것)한다면 행동을 인지하고 이에 대응할 수 있으며 세상을 일련의 관련없는 이벤트로 보는 것보다 많은 비즈니스 가치를 제공할 수 있다.

이제 스테이트풀 스트림 처리의 장점과 그리고 사실과 행동 간 차이를 이해했기 때문에 카프카 스트림즈의 스테이트풀 처리 연산자를 미리 살펴보자.

스테이트풀 연산자 미리보기

카프카 스트림즈는 프로세서 토폴로지에서 사용할 수 있는 여러 스테이트풀 연산자를 포함하고 있다. 표 4-1 은 이 책에서 작업할 여러 연산자에 대한 개요를 보여준다.

표 4-1. 스테이트풀 연산자 및 목적

유스케이스	목적	연산자
데이터 조인	별도 스트림 또는 테이블에 포착된 추가적인 정보 또는 컨텍스트를 이용해 데이터 보강	• join (inner join) • leftJoin
데이터 집계	연속적으로 업데이트되고 있는 관련 이벤트의 수학적 또는 조합 변환 계산	• aggregate • count • reduce
데이터 윈도우	시간적으로 근접한 이벤트 그룹화	• windowedBy

더구나 이벤트 간 보다 복잡한 관계/행동을 이해하기 위해 카프카 스트림즈의 스테이트풀 연산자를 결합할 수 있다. 예를 들어 윈도우 조인을 수행함으로써 특정 기간 동안 개별 이벤트 스트림이 어떻게 관련되어 있는지를 이해할 수 있다. 다음 장에서 볼 것인데 윈도우 집계는 스테이트풀 연산자를 결합하는 다른 유용한 방법이다.

이제 이전 장에서 다뤘던 스테이트리스 연산자와 비교하여 스테이트풀 연산자는 내부적으로 더욱 복잡하고 추가적인 계산 및 저장 요건[2]을 갖는다. 이러한 이유로 표 4-1 의 스테이트풀 연산자를 사용하여 시작하기 전에 카프카 스트림즈의 스테이트풀 처리의 내부 작동 원리에 대해 배우는데 시간을 할애할 것이다.

아마도 가장 중요한 시작점은 카프카 스트림즈에서 상태가 어떻게 저장 및 쿼리되는 지를 살펴보는 것이다.

상태 저장소

스테이트풀 연산은 애플리케이션으로 하여금 이전에 보았던 이벤트를 일부 기억할 것을 요구함을 이미 확립하였다. 예를 들어 에러 로그의 수를 세는 애플리케이션은 각 키에 대해 하나의 숫자를 추적해야 한다: 새로운 에러 로그가 소비될 때마다 업데이트되는 롤링 카운트. 이 카운트는 레코드에 대한 이력 컨텍스트를 표현하며 레코드 키와 함께 애플리케이션 상태의 일부가 된다.

---

[2] 메모리, 디스크 또는 이들의 조합

스테이트풀 연산을 지원하기 위해서는 애플리케이션의 각 스테이트풀 연산자가 필요로 하는 기억된 데이터 또는 상태를 저장 및 검색하는 방법이 필요하다 (예, aggregate, join 등). 카프카 스트림즈에서 이러한 니즈를 다루는 저장 추상화는 상태 저장소로 불리며 하나의 카프카 스트림즈 애플리케이션이 많은 스테이트풀 연산자를 활용할 수 있기 때문에 단일 애플리케이션이 여러 상태 저장소를 포함할 수도 있다.

> 이 절에서는 카프카 스트림즈에서 상태가 포착 및 저장되는 방법에 대한 일부 저수준의 정보를 제공한다. 튜토리얼을 시작하고 싶은 경우 "튜토리얼 소개: 비디오 게임 리더보드"로 건너뛰고 추후 이 절을 다시 보기 바란다.

카프카 스트림즈에는 사용가능한 많은 상태 저장소 구현이 존재하고 다양하게 구성할 수 있으며 각각은 특정 장점, 트레이드 오프 및 유스케이스를 갖고 있다. 카프카 스트림즈 애플리케이션에서 스테이트풀 연산자를 사용할 때마다 연산자에 대해 필요한 상태 저장소 유형과 최적화 기준 (예, 높은 처리량, 운영 단순성, 고장 발생 시 빠른 복구 시간 등)에 기반하여 상태 저장소를 구성하는 방법을 고려하는 것이 도움이 된다. 대부분의 경우 상태 저장소 유형을 명시적으로 지정하지 않거나 상태 저장소의 구성 특성을 재정의하지 않는다면 카프카 스트림즈는 합리적인 기본 값을 선택할 것이다.

상태 저장소 유형과 구성의 변형으로 인해 매우 깊은 주제이기 때문에 처음에는 모든 기본 상태 저장소 구현의 공통적인 특성에 대한 논의에 중점을 둘 것이고 그 후 상태 저장소의 2 가지 넓은 범주를 살펴볼 것이다: 영구 및 인메모리 저장소. 상태 저장소에 대한 보다 깊은 논의는 6 장에서 다룰 것이며 튜토리얼에서 특정 주제를 접할 것이다.

공통 특성

카프카 스트림즈에 포함되어 있는 기본 상태 저장소 구현은 일부 공통적인 특성을 공유한다. 상태 저장소 작동 방법을 더 잘 이해하기 위해 이 절에서는 이러한 공통점을 논의할 것이다.

내장형

카프카 스트림즈에 포함되어 있는 기본 상태 저장소 구현은 태스크 수준에서 애플리케이션 내에 내장되어 있다 ("태스크 및 스트림 쓰레드"에서 태스크를 처음 논의하였다). 외부 저장 엔진 사용과 반대로 내장형 상태 저장소의 장점은 후자의 경우 상태에 액세스할 필요가 있을 때마다 네트워크 호출을 필요로 하며 따라서 불필요한 지연과 처리 병목을 야기한다는 것이다. 더구나 태스크 수준에서 상태 저장소가 내장되기 때문에 공유 상태 액세스에 대한 전체적인 동시성 문제가 제거된다.

이외에 상태 저장소가 원격에 있다면 카프카 스트림즈 애플리케이션으로부터 멀리 떨어져 있는 원격 시스템의 가용성에 대해 걱정해야 한다. 카프카 스트림즈가 로컬 상태 저장소를 관리할 수 있도록 함으로써 항상 가용할 것임을 보장하고 에러들을 상당히 줄인다. 중앙화된 원격 저장소는 모든 애플리케이션 인스턴스에 대해 단일 실패 지점이 될 것이기 때문에 더욱 더 좋지 않을 것이다. 따라서 애플리케이션 자체와 함께 애플리케이션의 상태를 배치한다는 카프카 스트림즈의 전략은 성능 (이전 단락에서 논의했던) 뿐만 아니라 가용성도 향상시킨다.

모든 기본 상태 저장소는 내부적으로 RocksDB 를 활용한다. RocksDB 는 원래 페이스북에서 개발한 고속의 내장형 키-값 저장소로 키-값 쌍 저장을 위해 임의의 바이트 스트림을 지원하기 때문에 직렬화와 저장을 분리하는 카프카와도 잘 작동한다. 더구나 분기된 LevelDB 코드[3]의 풍부한 일련의 최적화 덕분에 읽기와 쓰기 모두 매우 빠르다.

다중 액세스 모드

상태 저장소는 다중 액세스 모드와 쿼리 패턴을 지원한다. 프로세서 토폴로지는 상태 저장소에 대한 읽기 및 쓰기 액세스를 필요로 한다. 그러나 뒤에 논의할 카프카 스트림즈의 상호대화형 쿼리 기능을 사용하여 마이크로 서비스를 구축할 때 클라이언트는 단지 기본 상태에 대한 읽기 액세스만 필요하다. 이는 프로세서 토폴로지 외부에서 상태가 절대로 변하지 않음을 보장하며 카프카 스트림즈 애플리케이션의 상태를 안전하게 쿼리하기 위해 클라이언트가 사용할 수 있는 읽기 전용 랩퍼를 통해 가능하다.

내고장성

기본적으로 상태 저장소는 카프카에서 체인지로그 토픽에 백업된다[4]. 고장 시 애플리케이션을 재구축하기 위해 체인지로그 토픽에서 개별 이벤트를 재생함으로써 상태 저장소는 복구될 수 있다. 더구나 카프카 스트림즈는 애플리케이션을 재구축하는데 걸리는 시간을 줄이기 위해 대기 복제본을 사용할 수 있도록 한다. 이 대기 복제본 (종종 새도우 토픽이라 불림)은 상태 저장소를 이중화하며, 이는 고가용성 시스템의 중요한 특성이다. 이외에 상태가 쿼리될 수 있도록 하는 애플리케이션은 다른 애플리케이션 인스턴스가 오프라인이 될 때 쿼리 트래픽을 처리하기 위해 대기 복제본에 의존할 수 있으며 이 또한 고가용성에 기여한다.

키 기반

상태 저장소를 활용하는 연산은 키 기반이다. 레코드 키는 현재 이벤트와 다른 이벤트 간 관계를 정의한다. 데이터 구조는 사용하는 상태 저장소 유형[5]에 따라 다를 것이지만 각 구현은 키가 단순하거나 일부 경우 보다 복잡한 다차원일 수 있는 일종의 키-값 저장소로 개념화될 수 있다[6].

---

[3] LevelDB는 구글에서 개발하였지만 페이스북 엔지니어가 이를 사용하기 시작했을 때 내장형 워크플로우에 대해 너무 느린 것을 발견하였다. LevelDB의 단일 쓰레드 압축을 다중 쓰레드 압축 프로세스로 변경하고 읽기에 대해 블룸 필터를 활용함으로써 읽기 및 쓰기 성능 모두 극적으로 개선되었다.

[4] 상태 저장소가 구성가능하고 체인지 로깅 작동을 비활성화함으로써 내고장성을 끌 수 있음을 언급하였다.

[5] 예를 들어, inMemoryKeyValueStore가 red-black 트리 기반 Java TreeMap을 사용하는 반면 모든 영구 키-값 저장소는 RockDB를 사용한다.

[6] 예를 들어 윈도우 저장소는 키-값 저장소인데 키로 레코드 키 외에 윈도우 타임 또한 포함한다.

조금 복잡하게 카프카 스트림즈는 모든 기본 상태 저장소가 키 기반이지만 특정 유형의 상태 저장소를 명시적으로 키-값 저장소로 간주한다. 이 장과 이 책의 다른 곳에서 키-값 저장소를 언급할 때 비윈도우 상태 저장소를 언급하는 것이다 (윈도우 상태 저장소는 다음 장에 논의될 것이다).

이제 카프카 스트림즈의 기본 상태 저장소 간 공통점을 이해했기 때문에 특정 구현 간 차이를 이해하기 위해 2 가지 상태 저장소 범주를 살펴보자.

영구 대 인메모리 저장소

다양한 상태 저장소 구현 간 가장 중요한 차이 중 하나는 상태 저장소가 영구적인지 아니면 인메모리(RAM)에 단순히 기억된 정보를 저장하는지 여부이다. 영구 상태 저장소는 비동기적으로 상태를 디스크에 플러시하며 이는 2 가지 주요 장점을 갖는다:

- 상태가 사용가능 메모리 크기를 초과할 수 있다.
- 고장 발생 시 영구 저장소는 인메모리 저장소보다 빨리 복구될 수 있다.

첫번째 요점을 명확히 한다면 영구 저장소는 상태 일부를 인메모리에 유지할 수 있으며 상태 크기가 너무 커질 때 (이를 spilling to disk 라 부름) 또는 쓰기 버퍼가 설정된 값을 초과할 때 디스크에 쓴다. 두번째 애플리케이션 상태가 디스크에 저장되어 있기 때문에 상태 손실 시 (예, 시스템 고장, 인스턴스 이관 등으로 인해) 카프카 스트림즈는 상태 저장소를 재구축하기 위해 전체 토픽을 재생할 필요가 없다. 애플리케이션이 다운되었던 시간과 복구된 시간 사이에 누락된 데이터만 재생하면 된다.

영구 저장소에 사용되는 상태 저장소 디렉토리를 StreamsConfig.STATE_DIR_CONFIG 특성을 사용하여 설정할 수 있다. 기본 위치는 /tmp/kafka-streams 지만 /tmp 외부 디렉토리로 이를 재정의할 것이 권고된다.

단점은 영구 저장소가 운영 측면에서 보다 복잡하고 항상 RAM 으로부터 데이터를 가져오는 순수 인메모리 저장소보다 느릴 수 있다는 것이다. 추가적인 운영 복잡성은 이차 저장 요건 (예, 디스크 기반 저장)과 상태 저장소를 조정하는 경우 RocksDB 와 이의 구성을 이해해야 한다는 것에 기인한다 (후자는 대다수 애플리케이션에 대해 문제가 아닐 수도 있다).

인메모리 상태 저장소의 성능 향상과 관련하여 이는 인메모리 상태 저장소의 사용을 보장할 만큼 극적이지 않을 수 있다 (고장 복구에 오랜 시간이 걸리기 때문). 애플리케이션에서 보다 성능을 향상시켜야 한다면 작업을 병렬화하기 위해 더욱 많은 파티션을 추가하는 것이 항상 옵션이다. 따라서 영구 저장소로 시작하고 눈에 띌 만한 성능 향상이 필요한 경우에만 인메모리 저장소로 변경하고 빠른 복구가 중요한 경우 (예, 애플리케이션 상태 손실 시) 복구 시간을 줄이기 위해 대기 복제본을 사용하는 것이 권고된다.

이제 상태 저장소가 무엇이고 그들이 스테이트풀/동작 중심 처리를 가능케 하는 방법을 어느 정도 이해했기 때문에 이 장의 튜토리얼을 살펴보고 아이디어 중 일부를 실제 확인해보자.

튜토리얼 소개: 비디오 게임 리더보드

이 장에서는 카프카 스트림즈로 비디오 게임 리더보드를 구현함으로써 스테이트풀 처리에 대해 배울 것이다. 비디오 게임 산업은 게이머와 게임 시스템 모두 저지연 처리와 즉각적인 피드백을 필요로 하기 때문에 스트림 처리가 뛰어난 대표적인 분야이다. 이는 Activision(Call of Duty 와 Crash Ban-dicoot and Spyro 리마스터와 같은 게임을 제작)과 같은 회사가 비디오 게임 원격측정을 위해 카프카 스트림즈를 사용하는 이유 중 하나이다[7].

구축할 리더보드는 이제껏 탐구하지 않았던 방식으로 데이터를 모델링할 것을 요구한다. 구체적으로 데이터를 일련의 업데이트로 모델링하기 위해 카프카 스트림즈의 테이블 추상화를 사용하는 방법을 살펴볼 것이다. 그 후 다중 이벤트 간 관계를 이해 또는 계산할 때마다 유용한 데이터 조인과 집계를 자세히 살펴볼 것이다. 이를 앎으로써 카프카 스트림즈를 통해 보다 복잡한 비즈니스 문제를 해결하는데 도움이 될 것이다.

일련의 새로운 스테이트풀 연산자를 사용하여 실시간 리더보드를 만들었다면 상호대화형 쿼리를 사용하여 최신 리더보드 정보를 카프카 스트림즈에 쿼리하는 방법을 보여줄 것이다. 이 기능에 대한 논의를 통해 카프카 스트림즈를 사용한 이벤트 기반 마이크로 서비스를 구축하는 방법을 배울 것이며, 이는 스트림 처리 애플리케이션에서 데이터를 공유할 수 있는 클라이언트 유형을 넓힐 것이다[8].

더 이상 고민하지 말고 비디오 게임 리더보드의 아키텍처를 살펴보자. 그림 4-1 은 이 장에서 구현할 토폴로지 설계를 보여준다. 각 단계에 대한 추가 정보는 그림 다음에 포함되어 있다.

---

[7] Tim Berglund와 Yaroslav Tkachenko가 Streaming Audio podcast에서 Activision 유스케이스에 대해 논의한다.

[8] 처리된/보강된 데이터를 다운스트림 애플리케이션에 푸쉬할 수 있도록 하는 카프카 스트림즈가 직접 출력 토픽에 쓸 수 있는 방법을 이미 살펴보았다. 그러나 상호대화형 쿼리는 카프카 스트림즈 애플리케이션에 대해 임시 쿼리를 수행하기 원하는 클라이언트가 사용할 수 있다.

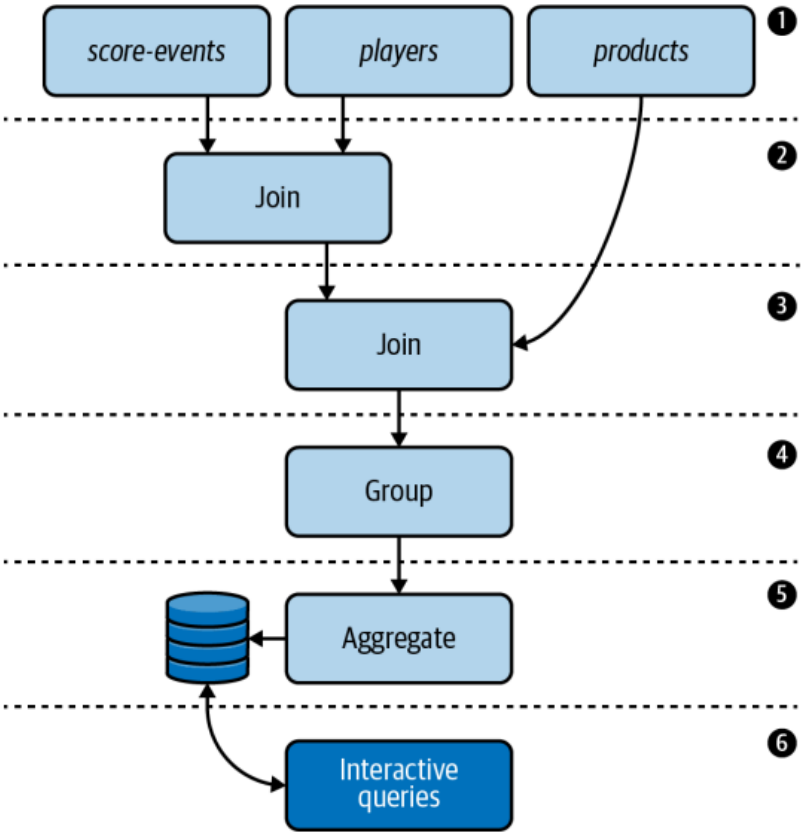

그림 4-1. 스테이트풀 비디오 게임 리더보드 애플리케이션에 구현할 토폴로지

① 카프카 클러스터는 3 개의 토픽을 포함한다:

Score-events 토픽은 게임 점수를 포함한다. 레코드는 키가 없으며 따라서 토픽 파티션에 라운드-로빈 방식으로 분배된다.

Players 토픽은 플레이어의 프로파일을 포함한다. 각 레코드의 키는 플레이어 ID 이다.

Products 토픽은 다양한 비디오 게임에 대한 제품 정보를 포함한다. 각 레코드의 키는 제품 ID 이다.

② score-events 데이터를 세부 플레이어 정보로 보강해야 한다. 조인을 사용해 이를 수행할 수 있다.

③ 플레이어 데이터로 score-events 데이터를 보강했다면 해당 스트림에 세부 제품 정보를 추가해야 한다. 조인을 사용하여 이 또한 수행할 수 있다.

④ 데이터 그룹화가 집계의 선결조건이기 때문에 보강된 스트림을 그룹화해야 한다.

⑤ 각 게임에 대해 상위 3 개의 고득점을 계산해야 한다. 이 목적으로 카프카 스트림즈의 집계 연산자를 사용할 수 있다.

⑥ 마지막으로 외부에 각 게임에 대한 고득점을 노출해야 한다. 카프카 스트림즈의 상호대화형 쿼리 기능을 사용하여 RESTful 마이크로 서비스를 구축함으로써 이를 수행할 것이다.

토폴로지 설계를 마쳤고 이제 프로젝트 환경설정으로 이동할 수 있다.

프로젝트 환경설정

이 장의 코드는 https://github.com/mitch-seymour/masteringkafka-streams-and-ksqldb.git 에 위치한다.

각 토폴로지 단계를 통해 작업할 때 코드를 참조하고 싶다면 저장소를 복제하고 이 장 튜토리얼을 포함한 디렉토리로 이동한다. 다음 명령을 사용할 수 있다:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git

$ cd mastering-kafka-streams-and-ksqldb/chapter-04/video-game-leaderboard
```

다음 명령을 실행하여 언제라도 프로젝트를 빌드할 수 있다:

```
$ ./gradlew build –info
```

프로젝트 환경 설정이 끝났고 비디오 게임 리더보드 구현을 시작해보자.

데이터 모델

평소와 같이 데이터 모델 정의로 시작할 것이다. 소스 토픽이 JSON 데이터를 포함하고 있기 때문에 POJO 데이터 클래스를 사용하여 데이터 모델을 정의할 것이며 추후 선택한 JSON 직렬화 라이브러리를 사용하여 직렬화 및 역직렬화할 것이다 (이 책을 통해 Gson 을 사용하지만 Jackson 또는 다른 라이브러리도 쉽게 사용할 수 있다)[9].

데이터 모델을 프로젝트 내 전용 패키지, 예를 들어 com.magicalpipelines.model 에 그룹화할 수 있다. 이 튜토리얼의 데이터 클래스가 위치한 곳의 파일시스템 뷰는 다음과 같다:

```
src/
└── main
    └── java
        └── com
            └── magicalpipelines
                └── model
                    ├── ScoreEvent.java ❶
                    ├── Player.java ❷
                    └── Product.java ❸
```

① ScoreEvent.java 데이터 클래스는 score-events 토픽의 레코드를 표현하기 위해 사용될 것이다.

② Player.java 데이터 클래스는 players 토픽의 레코드를 표현하기 위해 사용될 것이다.

③ Product.java 데이터 클래스는 products 토픽의 레코드를 표현하기 위해 사용될 것이다.

---

[9] 3 장에 언급했듯이 토픽이 Avro 데이터를 포함한다면 Avro 스키마 파일에 데이터 모델을 정의할 수 있다.

이제 어떤 데이터 클래스를 구현해야 할 지 알았기 때문에 각 토픽에 대해 데이터 클래스를 생성해보자. 표 4-2 는 이 튜토리얼에서 구현했던 POJO 를 보여준다.

표 4-2. 각 토픽에 대한 예제 레코드와 데이터 클래스

카프카 토픽	예제 레코드	데이터 클래스

score-events	{
	"score": 422,
	"product_id": 6,
	"player_id": 1
	}
		public class ScoreEvent {
		private Long playerId;
		private Long productId;
		private Double score;
		}

players	{
	"id": 2,
	"name": "Mitch"
	}
		public class Player {
		private Long id;
		private String name;
		}

products	{
	"id": 1,
	"name": "Super Smash Bros"
	}
		public class Product {
		private Long id;
		private String name;
		}

이미 직렬화와 역직렬화를 논의하였고, 스스로 맞춤형 직렬화기, 역직렬화기 및 Serdes 를 구현하였다. 따라서 여기에 많은 시간을 할애하지 않을 것이며 표 4-2 에 보이는 데이터 클래스 각각에 대 Serdes 를 어떻게 구현했는지를 보기 위해서는 이 튜토리얼의 코드를 확인할 수 있다.

소스 프로세서 추가하기

데이터 클래스를 정의했다면 소스 프로세서를 설정할 수 있다. 이 토폴로지에서는 3 개의 소스 토픽으로부터 읽어 들일 것이기 때문에 3 개가 필요하다. 소스 프로세서를 추가할 때 해야 할 첫번째는 토픽에서 데이터를 표현하기 위해 어떤 카프카 스트림즈 추상화를 사용할지를 결정하는 것이다.

지금까지는 스테이트리스 레코드 스트림을 표현하기 위해 사용했던 KStream 추상화만으로 작업을 수행하였다. 그러나 토폴로지는 조회를 위해 products 와 players 토픽 모두를 사용해야 하며 따라서 이는 이들 토픽에 대해 테이블과 같은 추상화가 적절할 수 있음을 잘 나타낸다[10]. 토픽과 카프카 스트림즈 추상화 간 매핑을 시작하기 전에 카프카 토픽에 대한 KStream, KTable 과 GlobalKTable 간 차이를 우선 검토해보자. 각 추상화를 검토함에 따라 표 4-3 의 각 토픽에 대해 적절한 추상화를 채울 것이다.

표 4-3. 다음 절에서 업데이트할 토픽-추상화 매핑

카프카 토픽	추상화

[10] 조회/조인 연산을 위해 KStream을 사용할 수 있지만 이는 항상 윈도우 연산으로 다음 장까지 이 주제에 대한 논의를 유보한다.

Score-events	???
Players	???
products	???

KStream

사용할 추상화를 결정할 때 토픽 특성, 토픽 구성 및 소스 토픽 내 레코드의 키 공간을 결정하는 것이 도움이 된다. 스테이트풀 카프카 스트림즈 애플리케이션이 하나 이상의 테이블 추상화를 사용하지만 하나 이상의 데이터 소스에 대해 가변 테이블 시맨틱이 필요치 않은 경우 KTable 또는 GlobalKTable 과 함께 스테이트리스 KStreams 를 사용하는 것이 매우 일반적이다.

이 튜토리얼에서 score-events 는 비압축 토픽에 키가 없는 (따라서 라운드-로빈 방식으로 분배되는) 원시 점수 이벤트를 포함하고 있다. 테이블은 키 기반이기 때문에 이는 키가 없는 score-events 토픽에 대해 KStream 을 사용해야 함을 잘 나타낸다. 키생성(keying) 전략을 업스트림에서 (소스 토픽에 생산하고 있는 애플리케이션이 무엇이든) 변경할 수 있지만 항상 가능한 것은 아니다. 더구나 애플리케이션은 각 플레이어에 대해 최신 점수가 아닌 가장 높은 점수를 관리하며 따라서 (주어진 키에 대해 가장 최신 레코드만 유지하는) 테이블 시맨틱은 키가 있더라도 score-events 토픽을 사용하려는 의도에 대해 잘 해석되지 않는다.

따라서 score-events 토픽에 대해 KStream 을 사용할 것이며 따라서 이 결정을 반영하여 표 4-3 을 다음과 같이 업데이트해보자:

카프카 토픽	추상화
Score-events	KStream
Players	???
products	???

나머지 두 토픽인 players 과 products 는 키가 존재하며 토픽에서 각각의 고유 키에 대해 최신 레코드만 관리한다. 따라서 이들 토픽에 대해 KStream 추상화는 적합하지 않다. 이제 더 나아가서 KTable 추상화가 이들 토픽 각각에 적절한지를 알아보자.

KTable

Players 토픽은 플레이어 프로파일을 포함하고 각 레코드의 키가 플레이어 ID 인 압축 토픽이다. 플레이어에 대해 최신 상태만 관리하기 때문에 테이블 기반 추상화 (KTable 또는 GlobalKTable)를 사용하여 이 토픽을 표현하는 것은 의미가 있다.

KTable 또는 GlobalKTable 을 언제 사용할 지 결정할 때 고려해야 할 한 가지 중요한 것은 키 공간이다. 키 공간이 매우 크거나 매우 큰 키 공간으로 성장할 것으로 예상된다면 전체 상태의 부분들을 실행 중인 모든 애플리케이션 인스턴스에 분배할 수 있도록 KTable 을 사용하는 것이 더욱 의미가 있다. 이런 방식으로 상태를 파티셔닝함으로써 각 개별적인 카프카 스트림즈 인스턴스에 대해 로컬 저장 오버헤드를 낮출 수 있다.

KTable 또는 GlobalKTable 간 선택 시 아마도 보다 중요한 고려사항은 시간 동기화된 처리의 필요성 여부이다. KTable 은 시간 동기화되며 따라서 카프카 스트림즈가 다중 소스로부터 읽을 때 (예, 조인의 경우) 다음에 처리할 레코드를 결정하기 위해 타임스탬프를 볼 것이다. 이는 특정 시점에 결합된 레코드가 존재했음을 의미하며 이로 인해 조인 작동은 더욱 예측가능하다. 반면 GlobalKTable 은 시간 동기화되지 않고 어떤 처리이던 종료되기 전에 완전히 채워진다[11]. 따라서 조인은 항상 GlobalKTable 의 가장 최신 버전에 대해 수행되며 이는 프로그램의 시맨틱을 변경시킨다.

여기서는 다음 장에서 타임과 카프카 스트림즈에서 이의 역할에 대해 논의할 것이기 때문에 두 번째 고려사항에 대해서는 많이 논의하지 않는다. 키 공간 관련 player 는 각각의 고유 플레이어에 대한 레코드를 포함한다. 이는 회사 또는 제품 라이프사이클에서 어디에 존재하는지에 따라 작을 수도 있지만 시간이 지남에 따라 상당히 커질 것으로 예상하는 수이며 따라서 이 토픽에 대해 KTable 추상화를 사용할 것이다.

그림 4-2 는 KTable 을 사용하여 실행 중인 다중 애플리케이션 인스턴스에 상태가 분배되는 방법을 보여준다.

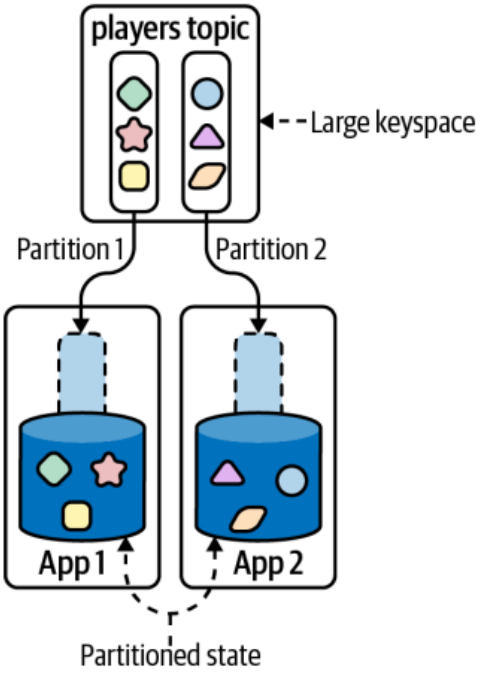

그림 4-2. 상태를 다중 애플리케이션에 파티셔닝하고 시간 동기화된 처리가 필요할 때 KTable 이 사용되어야 한다.

업데이트된 추상화 테이블은 이제 다음과 같다:

카프카 토픽	추상화

[11] Florian Trossbach와 Matthias J. Sax는 "Crossing the Streams: Joins in Apache Kafka" 기사에서 이 주제를 깊이 다루었다.

Score-events	KStream
Players	KTable
products	???

이제 products 토픽만 남아있다. 이 토픽은 비교 작으며 따라서 모든 애플리케이션 인스턴스에 전체 상태를 복제할 수 있어야 한다. 이를 할 수 있도록 하는 추상화 GlobalKTable 을 살펴보자.

GlobalKTable

Products 토픽은 구성 (압축되어 있음)과 제한적인 키 공간 (각각 고유 제품 ID 에 대해 최신 레코드만 유지하며 추적할 제품의 수도 고정되어 있음) 측면에서 player 토픽과 유사하다. 그러나 products 토픽은 player 토픽보다 더 적은 cardinality (예, 더 적은 고유 키)를 가지며 리더보드가 수백개의 게임에서 높은 점수를 추적하더라도 이는 메모리에 완전히 들어갈 만큼 작은 공간으로 변환된다.

더 적다는 것 외에 products 토픽의 데이터는 비교적 정적이다. 비디오 게임 제작에는 오랜 시간이 걸리며 따라서 products 토픽에 많은 업데이트가 있을 것으로 예상하지 않는다.

이러한 2 가지 특성 (작고 정적인 데이터)은 GlobalKTable 이 설계된 이유이다. 따라서 products 토픽에 대해 GlobalKTable 을 사용할 것이다. 따라서 카프카 스트림즈 인스턴스 각각은 제품 정보에 대해 전체 복제본을 저장할 것이며 이는 뒤에서 볼 것이지만 조인 수행을 더욱 쉽게 만들 것이다.

그림 4-3 은 각각의 카프카 스트림즈 인스턴스가 products 테이블의 전체 복제본을 어떻게 유지하는지를 보여준다.

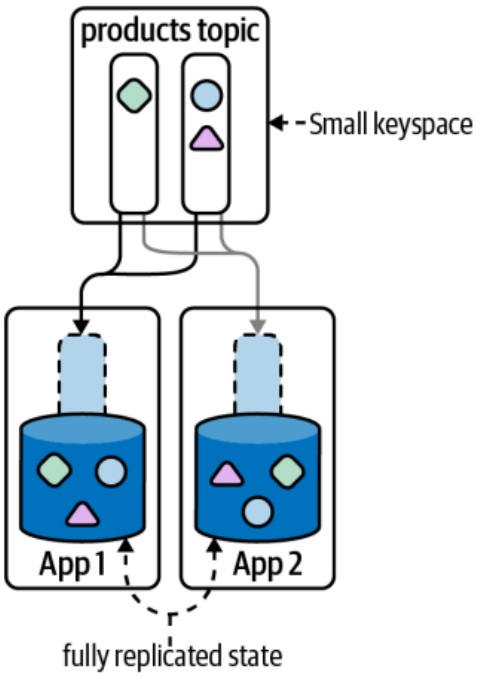

그림 4-3. GlobleKTable 은 키 공간이 작고, 조인의 co-partition 요건을 피하고 싶으며 시간 동기화가 필요치 않을 때 사용되어야 한다.

이제 토픽-추상화 매핑에 최종 업데이트를 할 수 있다:

카프카 토픽	추상화
Score-events	KStream
Players	KTable
products	GlobalKTable

이제 소스 토픽 각각에 대해 사용할 추상화를 결정했기 때문에 스트림과 테이블을 등록할 수 있다.

스트림 및 테이블 등록하기

스트림과 테이블 등록은 간단하다. 다음 코드 블록은 적절한 빌더 메소드를 사용하여 KStream, KTable 과 GlobalKTable 을 생성하기 위해 고수준 DSL 을 사용하는 방법을 보여준다:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], ScoreEvent> scoreEvents =
        builder.stream(
                "score-events",
                Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent())); ①
KTable<String, Player> players =
        builder.table(
                "players",
                Consumed.with(Serdes.String(), JsonSerdes.Player())); ②
GlobalKTable<String, Product> products =
        builder.globalTable(
                "products",
                Consumed.with(Serdes.String(), JsonSerdes.Product())); ③
```

① score-events 토픽에서 현재 키가 없는 데이터를 표현하기 위해 KStream 을 사용한다.

② KTable 추상화를 사용하여 players 토픽에 대해 파티셔닝된 (또는 샤딩된) 테이블을 생성한다.

③ products 토픽에 대해 각 애플리케이션 인스턴스로 전체가 복제될 GlobalKTable 을 생성한다.

소스 토픽을 등록함으로써 (그림 4-1 의) 리더보드 토폴로지의 1 단계를 구현하였다. 다음 단계로 넘어가 보자: 스트림과 테이블 조인

조인

관계형 세계에서 데이터셋을 결합하는 가장 일반적인 방법은 조인을 통한 것이다[12]. 관계형 시스템에서 데이터는 종종 매우 고차원으로 많은 다른 테이블에 흩어져 있다. 카프카에서도 이러한 동일 패턴을 보는 것은 일반적이다. 왜냐하면 이벤트가 다중 위치로부터 들어오고, 개발자가 관계형 모델이

[12] UNION 쿼리는 데이터셋 결합을 위한 다른 방법으로 카프카 스트림즈의 merge 연산자의 작동이 UNION 쿼리가 작동 방식과 더 밀접하게 관련되어 있다.

편리하거나 또는 이에 익숙하기 때문에 또는 특정 카프카 통합 (예, JDBC 카프카 커넥터, Debezium, Maxwell 등)이 소스 시스템에서 원시 데이터와 데이터 모델 모두를 가져오기 때문이다.

데이터가 카프카에 어떻게 흩어져 있는지에 상관없이 관계에 기반하여 별도 스트림과 테이블의 데이터를 결합할 수 있다는 것은 카프카 스트림즈에서 보다 고급 데이터 보강 기회를 제공한다. 더구나 데이터셋 결합을 위한 조인 방법과 그림 3-6 에서 보듯이 단순히 스트림을 병합하는 것은 매우 다르다. 카프카 스트림즈에서 merge 연산자를 사용할 때 merge 양측의 레코드는 무조건 단일 스트림으로 결합된다. 간단한 merge 연산은 병합되는 이벤트에 대해 추가적인 컨텍스트를 필요로 하지 않기 때문에 스테이트리스다.

그러나 조인은 이벤트 간 관계에 대해 주의하고 레코드가 출력 스트림에 그대로 복제되는 것이 아니라 결합되는 특수한 유형의 조건부 병합으로 간주될 수 있다. 더구나 이러한 관계는 조인을 용이하게 하기 위해 병합 시점에 포착, 저장 및 참조되어야 하며 이는 조인을 스테이트풀 연산으로 만든다. 그림 4-4 는 한 가지 유형의 조인의 작동 원리를 간단하게 설명한 것이다 (표 4-5 에 볼 것이지만 여러 유형의 조인이 존재한다).

관계형 시스템과 같이 카프카 스트림즈는 여러 유형의 조인을 지원한다. 따라서 score-events 스트림과 players 테이블을 조인하는 방법을 배우기 전에 특정 유스케이스에 대해 최상을 선택할 수 있도록 사용가능한 다양한 조인 연산자에 익숙해지자.

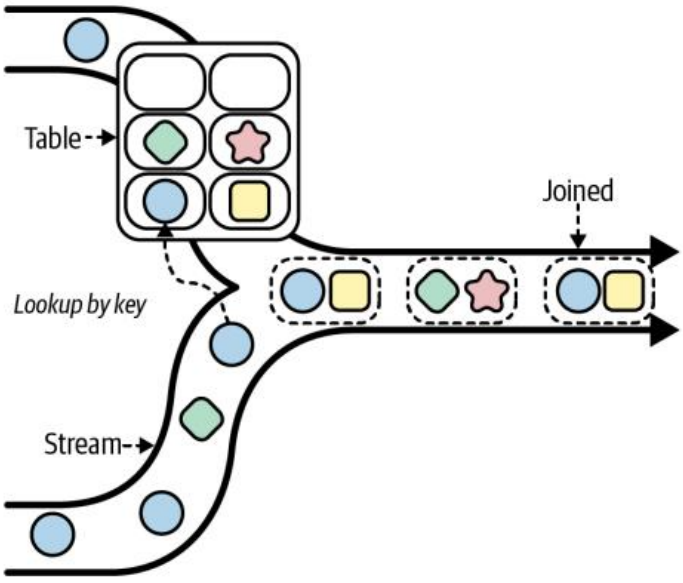

그림 4-4. 메시지 조인

조인 연산자

카프카 스트림즈는 스트림과 테이블 조인을 위해 3 가지 다른 조인 연산자를 포함하고 있다. 표 4-4 에 각 연산자를 자세히 설명하였다.

표 4-4. 조인 연산자

연산자	설명
join	이너 조인. 조인 양쪽의 입력 레코드가 동일 키를 공유할 때 조인이 트리거된다.
leftJoin	레프트 조인. 조인 의미가 조인 유형에 따라 다르다. - 스트림-테이블 조인의 경우: 조인 왼쪽의 레코드가 수신될 때 조인이 트리거된다. 조인 오른쪽에 동일 키를 갖는 레코드가 없다면 오른쪽 값은 null 로 설정된다. - 스트림-스트림과 테이블-테이블 조인의 경우: 조인 오른쪽의 입력이 조회 또한 트리거할 수 있다는 것을 제외하고는 스트림-스트림 레프트 조인과 동일한 의미. 오른쪽이 조인을 트리거하고 왼쪽에 일치하는 키가 없다면 조인은 결과를 산출하지 않을 것이다.
outerJoin	아우터 조인. 조인 양쪽 중 한쪽에 레코드가 수신될 때 조인이 트리거된다. 조인 반대쪽에 동일 키를 갖는 일치하는 레코드가 없는 경우 해당 값은 null 로 설정된다.

조인 연산자 간 차이를 논의할 때 조인의 다른 쪽을 언급했다. 조인의 오른쪽이 관련 조인 연산자에 파라미터로 전달됨을 기억하기 바란다. 예를 들어:

```
KStream scoreEvents = ...;
KTable players = ...;
scoreEvents.join(players, ...); ①
```

① scoreEvents 는 조인의 왼쪽이고 players 는 조인의 오른쪽이다.

이제 이들 연산자로 생성할 수 있는 조인 유형을 살펴보자.

조인 유형

카프카 스트림즈는 표 4-5 에서 보듯이 많은 다양한 유형의 조인을 지원한다. Co-partitioning 칼럼은 뒤에 논의할 것과 관련이 있다. 지금은 co-partitioning 이 실제 조인을 수행하기 위해 필요한 여분의 조건임을 이해하는 것으로 충분하다.

표 4-5 조인 유형

유형	윈도우	연산자	Co-partitioning 필요
KStream-KStream	예*	• join • leftJoin • outerJoin	예
KTable-KTable	아니오	• join • leftJoin • outerJoin	예
KStream-KTable	아니오	• join • leftJoin	예
KStream-GlobalKTable	아니오	• join • leftJoin	아니오

*주목해야 할 한 가지 사실은 KStream-KStream 조인이 윈도우라는 것이다. 다음 장에서 이를 세부적으로 논의할 것이다.

이 장에서 수행해야 하는 2 가지 유형의 조인은 다음과 같다:

- Score-events KStream 과 players KTable 을 조인하기 위한 KStream-KTable
- 이전 조인의 출력과 products GlobalKTable 을 조인하기 위한 KStream-GlobalKTable

여기서는 양쪽이 일치하는 경우에만 조인이 트리거되는 것을 원하기 때문에 조인 각각에 대해 join 연산자를 사용하는 이너 조인을 사용할 것이다. 그러나 이를 하기 전에 생성할 첫번째 조인 (KStream-KTable)이 co-partitioning 이 필요함을 볼 수 있다. 코드 작성 전에 이것의 의미를 살펴보자.

Co-Partitioning

숲에 나무가 쓰러졌고 주위에 듣는 이가 아무도 없다면 소리가 나는 것인가?

-경구

이 유명한 사고 실험은 이벤트 발생 시 (여기서는 숲에서 나는 소리) 관찰자의 역할에 대해 의문을 제기한다. 비슷하게 카프카 스트림즈에서 우리는 관찰자가 이벤트 처리에 미치는 영향을 항상 알아야 한다.

"태스트 및 스트림 쓰레드"에서 각 파티션이 단일 카프카 스트림즈 태스크에 할당된다는 것을 배웠는데, 이들 태스크가 이벤트를 실제 소비 및 처리하는 역할을 하기 때문에 비유에서 관찰자 역할을 할 것이다. 다른 파티션의 이벤트가 동일한 카프카 스트림즈 태스크에 의해 처리될 것임을 보장하지 못하기 때문에 잠재적인 관찰 가능성 문제를 갖고 있다. 그림 4-5 는 기본적인 관찰 가능성 문제를 보여준다. 연관된 이벤트가 다른 태스크에 의해 처리되는 경우 2 개의 별도 관찰자를 갖기 때문에 이벤트 간 관계를 정확히 결정할 수 없다. 데이터 조인의 목적이 연관 데이터를 결합하는 것이기 때문에 관찰 가능성 문제는 조인이 성공해야 할 때 실패하게 만들 것이다.

(조인을 통해) 이벤트 간 관계를 이해하거나 이벤트 시퀀스에 대해 집계를 계산하기 위해 연관된 이벤트가 동일 파티션으로 전달되고 따라서 동일 태스크에 의해 처리됨을 보장해야 한다.

연관된 이벤트가 동일 파티션으로 전달됨을 보장하기 위해 다음의 co-partitioning 요건이 충족됨을 보장해야 한다:

- 양쪽의 레코드가 동일 필드의 키를 가져야 하며 동일한 파티셔닝 전략을 사용하여 그 키로 파티셔닝되어야 한다.
- 조인 양쪽의 입력 토픽은 동일한 수의 파티션을 가져야 한다 (이는 기동 시 확인되는 한 가지 요건이다. 이 요건이 충족되지 못하면 TopologyBuilderException 예외가 발생할 것이다).

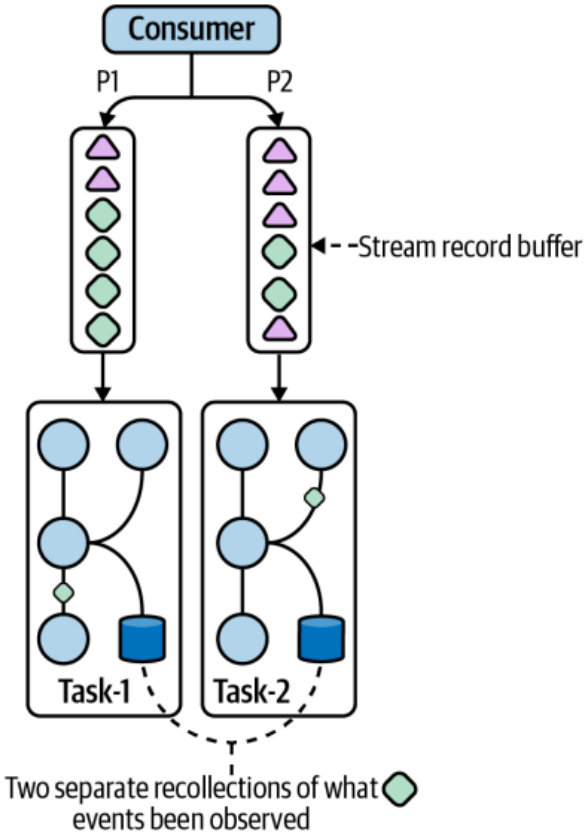

그림 4-5. 연관된 레코드를 조인하려고 하지만 이들 레코드가 항상 동일 태스크에 의해 처리되지 않는다면 관찰 가능성 문제를 갖는다.

이 튜토리얼에서 첫번째를 제외하고는 KStream-KTable 조인 수행 요건을 모두 충족한다. Score-events 토픽의 레코드는 키가 없지만 플레이어 ID 가 키인 players KTable 과 조인할 것임을 상기하기 바란다. 따라서 조인 수행에 앞서 score-events 의 데이터를 플레이어 ID 로 키를 재생성(rekeying)을 해야 한다. 이는 예제 4-1 에서 보듯이 selectKey 연산자를 사용하여 달성될 수 있다.

예제 4-1. selectKey 연산자를 레코드의 키를 재생성할 수 있도록 한다; 이는 특정 유형의 조인을 수행하기 위한 co-partitioning 요건을 충족시키기 위해 종종 필요하다.

```
KStream<String, ScoreEvent> scoreEvents =
        builder
                .stream(
                        "score-events",
                        Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
                .selectKey((k, v) -> v.getPlayerId().toString()); ①
```

① selectKey 는 레코드의 키를 재생성하는데 사용된다. 이 경우 조인 양쪽의 레코드가 동일 필드의 키를 가짐을 보장하는 첫번째 co-partitioning 요건을 충족시키는데 도움을 준다.

레코드의 키가 어떻게 재생성되는지를 그림 4-6 에 시각화하여 나타냈다.

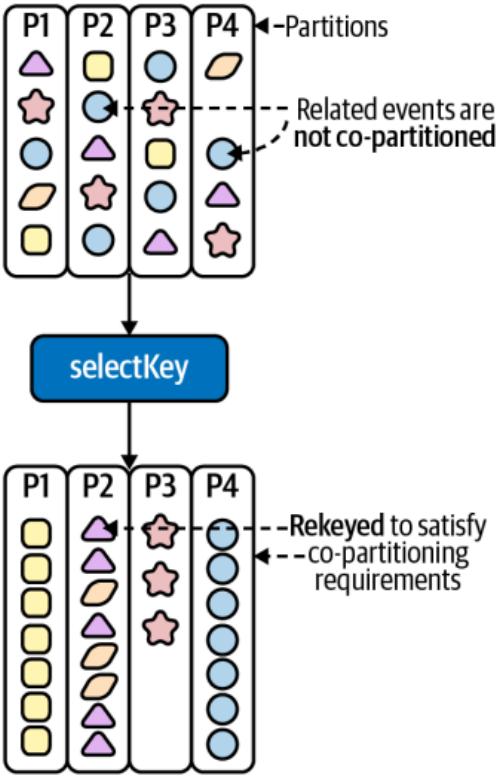

그림 4-6. 메시지 키재생성은 연관된 레코드가 동일 파티션에 나타남을 보장한다.

토폴로지에 키 변경 연산자를 추가할 때 데이터는 리파티셔닝을 위해 표시될 것이다. 이는 새로운 키를 읽는 다운스트림 연산자를 추가하자 마자 카프카 스트림즈가 다음을 할 것임을 의미한다:

- 키가 재생성된 데이터를 내부 리파티션 토픽에 전송한다.
- 새롭게 키가 재생성된 데이터를 다시 카프카 스트림즈로 읽어 들인다.

이 프로세스는 연관된 데이터 (예, 동일 키를 공유하는 레코드)가 추후 토폴로지 단계에서 동일 태스크에 의해 처리될 것임을 보장한다. 그러나 특수한 리파티션 토픽으로 데이터 재전달에 필요한 네트워크 이동은 키재생성 연산이 과도할 수 있음을 의미한다.

Products 토픽 조인을 위한 KStream-GlobaleKTable 은 어떤가? 표 4-5 에서 보듯이 상태가 카프카 스트림즈 앱의 각 인스턴스로 전부 복제되기 때문에 GlobalKTable 조인에는 co-partitioning 이 필요치 않다. 따라서 GlobalKTable 조인의 경우 이러한 유형의 관찰 가능성 문제가 전혀 일어나지 않을 것이다.

스트림과 테이블을 조인할 준비가 거의 되었다. 그러나 우선 조인 연산 동안 레코드가 실제 결합되는 방식을 살펴보자.

밸류 조이너

전통적인 SQL 을 사용하여 조인을 수행할 때는 결합되는 조인 레코드의 형태 (또는 투영)를 지정하기 위해 조인 연산자를 SELECT 문과 함께 사용하면 된다. 예를 들어:

```
SELECT a.customer_name, b.purchase_amount ①
FROM customers a
LEFT JOIN purchases b
ON a.customer_id = b.customer_id
```

① 결합되는 조인 레코드의 투영은 2 개의 칼럼을 포함한다.

그러나 카프카 스트림즈에서는 다른 레코드를 어떻게 결합할지를 지정하는 ValueJoiner 를 사용해야 한다. ValueJoiner 는 단순히 조인에 참여하는 각 레코드를 취해서 새로운 결합된 레코드를 산출한다. Score-events KStream 과 players KTable 을 조인해야 하는 첫번째 조인을 살펴보면 밸류 조이너의 작동은 다음 의사코드를 사용하여 표현될 수 있다:

```
(scoreEvent, player) -> combine(scoreEvent, player);
```

그러나 이것보다 더 좋게 할 수 있다. 다음 중 하나를 하는 전용 데이터 클래스를 갖는 것이 더욱 일반적이다:

- 조인에 포함된 값 각각을 래핑한다.
- 조인 양쪽에서 관련 필드를 추출하고 추출된 값을 클래스 특성으로 저장한다.

다음에 이들 모두를 탐구할 것이다. 우선 score-events->players 조인을 위한 간단한 래퍼 클래스로 시작해보자. 예제 4-2 는 조인 양쪽의 레코드를 래핑하는 데이터 클래스의 간단한 구현을 보여준다.

예제 4-2. 조인된 socre-events->players 레코드를 구축하기 위해 사용할 데이터 클래스

```
public class ScoreWithPlayer {
  private ScoreEvent scoreEvent;
  private Player player;
  public ScoreWithPlayer(ScoreEvent scoreEvent, Player player) { ①
        this.scoreEvent = scoreEvent; ②
        this.player = player;
  }
  // accessors omitted from brevity
}
```

① 생성자는 조인 양쪽에 대한 파라미터를 포함한다. 조인 왼쪽은 ScoreEvent 를 오른쪽은 Player 를 포함한다.

② 조인에 포함된 각 레코드에 대한 참조를 래퍼 클래스 내부에 저장한다.

새로운 래퍼 클래스를 ValueJoiner 의 반환 타입으로 사용할 수 있다. 예제 4-3 은 (score-events KStream 으로부터) ScoreEvent 와 (Player KTable 로부터) Player 를 ScoreWIthPlayer 인스턴스로 결합하는 ValueJoiner 구현을 보여준다.

예제 4-3. Score-events 와 player 를 결합하기 위한 ValueJoiner

```
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
        (score, player) -> new ScoreWithPlayer(score, player); ①
```

① ScoreWithPlayer::new 와 같이 정적 메소드 참조를 사용할 수 있다.

두번째 조인으로 넘어가보자. 이 조인은 (첫번째 조인의 출력으로부터의) ScoreWithPlayer 과 (products GlobalKTable 로부터의) Product 를 결합하기 위해 필요하다. 래퍼 패턴을 재사용할 수 있으며 조인 양쪽으로부터 필요한 특성을 단순히 추출하고 나머지는 버릴 것이다.

다음 코드 블록은 두 번째 패턴을 따르는 데이터 클래스의 구현을 보여준다. 단순히 원하는 값을 추출하고 이들을 적절한 클래스 속성으로 저장한다:

```
public class Enriched {
    private Long playerId;
    private Long productId;
    private String playerName;
    private String gameName;
    private Double score;
    public Enriched(ScoreWithPlayer scoreEventWithPlayer, Product product) {
        this.playerId = scoreEventWithPlayer.getPlayer().getId();
        this.productId = product.getId();
        this.playerName = scoreEventWithPlayer.getPlayer().getName();
        this.gameName = product.getName();
        this.score = scoreEventWithPlayer.getScoreEvent().getScore();
    }
    // accessors omitted from brevity
}
```

새로운 데이터 클래스 준비를 마쳤고 예제 4-4 의 코드를 사용하여 KStream-GlobalKTable 에 대한 ValueJoiner 를 구축할 수 있다.

예제 4-4. 조인에 사용할 ValueJoiner, 람다식으로 표현

```
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
        (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
```

이제 조인 레코드를 결합하는 법을 카프카 스트림즈에 알려줬기 때문에 실제 조인을 수행할 수 있다.

KStream to KTable Join (players Join)

Score-events KStream 과 players KTable 을 조인할 시간이다. ScoreEvent 레코드가 (레코드 키를 사용하여) Player 레코드와 일치될 수 있을 때에만 조인을 트리거하길 원하기 때문에 다음과 같이 join 연산자를 사용하여 이너 조인을 수행할 것이다.

```
Joined<String, ScoreEvent, Player> playerJoinParams =
  Joined.with( ①
```

```
            Serdes.String(),
            JsonSerdes.ScoreEvent(),
            JsonSerdes.Player()
    );
KStream<String, ScoreWithPlayer> withPlayers =
  scoreEvents.join( ②
            players,
            scorePlayerJoiner, ③
            playerJoinParams
  );
```

① 조인 파라미터는 조인 레코드의 키와 값이 어떻게 직렬화되는 지를 정의한다.

② join 연산자가 이너 조인을 수행한다.

③ 이는 예제 4-3 에서 생성했던 ValueJoiner 이다. 새로운 ScoreWithPlayer 값이 두 조인 레코드로부터 생성된다. 조인 값의 왼쪽 및 오른쪽이 생성자로 어떻게 전달되는지를 확인하기 위해 예제 4-2 의 ScoreWithPlayer 데이터 클래스를 확인하기 바란다.

이는 간단하다. 더구나 이 시점에서 코드를 실행시키고 카프카 클러스터에서 사용가능한 모든 토픽을 열거한다면 카프카 스트림즈가 2 개의 새로운 내부 토픽을 생성했음을 확인할 수 있다.

이들 토픽은 다음과 같다:

- 예제 4-1 에서 수행했던 키재생성 연산을 처리하기 위한 리파티션 토픽
- join 연산자에 의해 사용되는 상태 저장소를 백업하기 위한 체인지로그 토픽. 이는 초기에 논의했던 내고장성 작동의 일부이다.

Kafka-topics 컨솔 스크립트를 사용하여 확인할 수 있다[13].

```
$ kafka-topics --bootstrap-server kafka:9092 --list

            players
            products
            score-events
            dev-KSTREAM-KEY-SELECT-0000000001-repartition ①
            dev-players-STATE-STORE-0000000002-changelog ②
```

① 카프카 스트림즈가 생성한 내부 리파티션 토픽. 카프카 스트림즈 애플리케이션의 ID (dev)가 앞에 붙어 있다.

[13] 컨플루언트 플랫폼을 사용하고 있지 않다면 스크립트는 kafka-topics.sh이다.

② 카프카 스트림즈가 생성한 내부 체인지로그 토픽. 리파티션 토픽과 마찬가지로 카프카 스트림즈 애플리케이션의 ID (dev)가 앞에 붙어 있다.

자 이제 두번째 조인으로 넘어갈 준비가 되었다.

KStream to GlobalKTable Join (products Join)

Co-partitioning 요건에서 논의했듯이 GlobalKTable 조인 양쪽의 레코드는 동일 키를 공유할 필요가 없다. 로컬 태스크가 테이블의 전체 복제본을 갖고 있기 때문에 조인의 스트림 쪽[14]에 레코드 값 자체의 속성을 사용하여 조인을 실제 수행할 수 있으며 이는 연관된 레코드가 동일 태스크에 의해 처리됨을 보장하기 위해 리파티션 토픽을 통해 레코드의 키를 재생성하는 것보다 효율적이다.

KStream-GloablKTable 조인을 수행하기 위해서는 KStream 레코드를 GlobalKTable 레코드로 매핑하는 법을 지정하는 것이 목적인 KeyValueMapper 을 생성해야 한다. 이 튜토리얼의 경우 다음과 같이 이들 레코드를 Product 로 매핑하기 위해 ScoreWithPlayer 값으로부터 제품 ID 를 추출할 수 있다.

```
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper =
        (leftKey, scoreWithPlayer) -> {
                return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());
        };
```

KeyValueMapper 가 준비되었고 예제 4-4 에 생성했던 ValueJoiner 를 사용하여 조인을 수행할 수 있다:

```
KStream<String, Enriched> withProducts =
        withPlayers.join(products, keyMapper, productJoiner);
```

이로써 리더보드 토폴로지의 2 및 3 단계를 완료하였다. 해결해야 할 다음 단계는 집계를 수행할 수 있도록 보강된 레코드를 그룹화하는 것이다.

레코드 그룹화

카프카 스트림즈에서 스트림 또는 테이블 집계를 수행하기 전에 집계 대상인 KStream 또는 KTable 을 우선 그룹화해야 한다. 그룹화의 목적은 조인에 앞서 레코드의 키를 재생성하는 것과 동일하다: 연관된 레코드가 동일한 관찰자 또는 카프카 스트림즈 태스크에 의해 처리됨을 보장.

스트림과 테이블 그룹화에는 약간의 차이가 있으며 따라서 이들 각각을 살펴볼 것이다.

스트림 그룹화

KStream 을 그룹화하는데 사용될 수 있는 2 개의 연산자가 있다.

- groupBy

[14] 조인의 GlobalKTable 쪽은 조회를 위해 레코드 키를 여전히 사용할 것이다.

- groupByKey

groupBy 를 사용하는 것은 selectKey 를 사용하여 스트림의 키를 재생성하는 프로세스와 유사하다. 왜냐하면 이 연산자가 키 변경 연산자로 카프카 스트림즈로 하여금 리파티셔닝을 위해 스트림에 표시하게 하기 때문이다. 새로운 키를 읽는 다운스트림 연산자가 추가된다면 카프카 스트림즈는 자동적으로 리파티션 토픽을 생성하고 키재생성 프로세스를 완료하기 위해 데이터를 다시 카프카로 전달할 것이다.

예제 4-5 는 KStream 을 그룹화하기 위해 groupBy 연산자를 사용하는 방법을 보여준다.

예제 4.5. 동시에 KStream 키재생성 및 그룹화하기 위해 groupBy 연산자를 사용한다.

```
KGroupedStream<String, Enriched> grouped =
    withProducts.groupBy(
            (key, value) -> value.getProductId().toString(), ①
            Grouped.with(Serdes.String(), JsonSerdes.Enriched())); ②
```

① groupBy 연산자가 KeyValueMapper 를 예상하고 함수 인터페이스가 되기 때문에 새로운 키를 선택하기 위해 람다 표현식을 사용할 수 있다.

② Grouped 는 레코드를 직렬화할 때 사용하는 키 및 값 Serdes 를 포함하여 그룹화를 위해 추가적인 옵션을 전달할 수 있도록 한다.

그러나 레코드의 키재생성이 필요치 않다면 대신 groupByKey 연산자를 사용하는 것이 선호된다. groupByKey 는 리파티셔닝을 위해 스트림에 표시를 하지 않으며 따라서 리파티셔닝을 위해 데이터를 다시 카프카로 전송하기 위한 추가적인 네트워크 호출을 피하기 때문에 더욱 성능이 우수할 것이다. groupByKey 구현은 다음과 같다:

```
KGroupedStream<String, Enriched> grouped =
    withProducts.groupByKey(
            Grouped.with(Serdes.String(),
            JsonSerdes.Enriched()));
```

각 제품 ID 에 대해 높은 점수를 계산하길 원하고 보강된 데이터의 키가 현재 플레이어 ID 이기 때문에 리더보드 토폴로지에서는 예제 4-5 의 groupBy 변형을 사용할 것이다.

스트림을 그룹화하는데 어떤 연산자를 사용하는지에 상관없이 카프카 스트림즈는 이전에 논의하지 않았던 새로운 타입을 반환할 것이다: KGroupedStream. KGroupedStream 은 집계를 수행할 수 있도록 하는 스트림의 중간 표현이다. 짧게 집계를 살펴볼 것이지만 우선 KTable 을 그룹화하는 방법을 살펴보자.

테이블 그룹화

스트림 그룹화와 달리 테이블 그룹화에 사용가능한 연산자는 단지 하나이다: gorupBy. 또한 KGroupedStream 을 반환하는 대신 KTable 에 대해 groupBy 를 호출하는 것은 다른 중간 표현을

반환한다: KGroupedTable. 그렇지 않은 경우 KTable 그룹화 프로세스는 KStream 그룹화 프로세스와 동일하다. 예를 들어 추후 일부 집계를 수행할 수 있도록 (예, 플레이어 수 세기) players KTable 을 그룹화하길 원한다면 다음 코드를 사용할 것이다.

```
KGroupedTable<String, Player> groupedPlayers =
  players.groupBy(
      (key, value) -> KeyValue.pair(key, value),
      Grouped.with(Serdes.String(), JsonSerdes.Player()));
```

이전 코드 블록은 players 테이블을 그룹화할 필요가 없기 때문에 이 튜톨리얼에는 필요치 않은데 여기서는 개념을 설명하기 위해 나타냈다. 이제 스트림과 테이블 그룹화 방법을 알았고 프로세서 토폴로지의 4 단계를 완료했다. 다음으로 카프카 스트림즈에서 집계를 수행하는 방법을 배울 것이다.

집계

리더보드 토폴로지에 필요한 마지막 단계 중 하나는 각 게임에 대해 높은 점수를 계산하는 것이다. 카프카 스트림즈는 이러한 유형의 집계를 매우 쉽게 수행하도록 하는 일련의 연산자를 제공한다:

- Aggregate
- Reduce
- Count

고수준에서 집계는 다중 입력 값을 단일 출력 값으로 결합하는 방법이다. 집계를 수학 연산으로 생각하는 경향이 있지만 그럴 필요는 없다. Count 는 키 당 이벤트 수를 계산하는 수학 연산이지만 aggregate 와 reduce 연산 모두는 보다 일반적이고 지정한 계산 로직을 사용하여 값을 결합할 수 있다.

> Reduce 는 aggregate 와 매우 유사하다. 차이는 반환 타입에 있다. Reduce 연산자는 집계의 출력이 입력과 동일한 타입이어야 하지만 aggregate 연산자는 출력 레코드에 다른 타입을 지정할 수 있다.

더구나 집계는 스트림과 테이블 모두에 적용될 수 있다. 스트림은 불변이고 테이블은 가변이기 때문에 시맨틱은 약간 다르다. 이는 aggregate 와 reduce 연산자에 대해 약간 다른 버전으로 변환되는데 스트림 버전은 2 개의 파라미터 initializer 과 adder 을 받아들이고 테이블 버전은 3 개의 파라미터인 initializer, adder 과 substractor 을 받아들인다[15].

높은 점수 집계를 생성함으로써 스트림을 집계하는 방법을 살펴보자.

스트림 집계하기

---

[15] 스트림은 추가 전용이며 따라서 substractor을 필요로 하지 않는다.

이 절에서 레코드 스트림에 집계를 적용하는 방법을 배울 것인데 이는 새로운 집계 값을 초기화하기 위한 함수 (initializer)와 주어진 키에 대해 새로운 레코드가 들어올 때 추후 집계를 수행하기 위한 함수 (adder) 생성을 포함한다. 우선 initializer 에 대해 배울 것이다.

Initializer

카프카 스트림즈 토폴로지에서 새로운 키를 볼 때 집계를 초기화하는 방법이 필요하다. 이를 돕는 인터페이스는 Initializer 로 카프카 스트림즈 API 의 많은 클래스와 같이 함수 인터페이스 (예, 단일 메소드를 포함)로 람다 표현식으로 정의될 수 있다.

예를 들어 count 집계의 내부를 살펴본다면 집계의 초기 값을 0 으로 설정하는 initializer 을 볼 수 있다:

```
Initializer countInitializer = () -> 0L; ①
```

① Initializer 은 함수 인터페이스이기 때문에 람다 표현식으로 정의된다

보다 복잡한 집계의 경우 스스로 맞춤형 initializer 을 제공할 수 있다. 예를 들어 비디오 게임 리더보드를 구현하기 위해서는 주어진 게임에 대해 상위 3 개의 고득점을 계산하는 방법이 필요하다. 이를 위해 상위 3 개의 점수를 추적하는 로직을 포함하고 집계가 초기화될 필요가 있을 때마다 이 클래스의 새로운 인터페이스를 제공할 별도의 클래스를 생성할 수 있다.

이 튜토리얼에서는 집계 클래스 역할을 할 HighScores 맞춤형 클래스를 생성할 것이다. 이 클래스는 주어진 비디오 게임에 대해 상위 3 개의 점수를 유지하기 위해 데이터 구조를 필요로 할 것이다. 한 가지 방법은 Java 표준 라이브러리에 포함된 순서 집합인 TreeSet 을 사용하는 것인데 고득점을 유지하기에 매우 편리하다 (본질적으로 순서가 있다).

고득점 집계에 사용할 데이터 클래스의 최초 구현은 다음과 같다:

```
public class HighScores {
  private final TreeSet<Enriched> highScores = new TreeSet<>();
}
```

이제 새로운 데이터 클래스를 초기화하는 방법을 카프카 스트림즈에 알려야 한다. 클래스 초기화는 간단하며 단지 이를 인스턴스화하면 된다:

```
Initializer highScoresInitializer = HighScores::new;
```

집계를 위한 initializer 을 가졌다면 실제 집계를 수행할 로직을 구현하면 된다 (이 경우 각 비디오 게임에 대해 상위 3 개의 고득점을 추적)

Adder

스트림 집계기를 구축하기 위해 해야 할 다음 일은 2 가지 집계를 결합하기 위한 로직을 정의하는 것이다. 이는 Initializer 와 같이 람다 표현식을 사용하여 구현될 수 있는 함수 인터페이스인 Aggregator 인터페이스를 사용하여 달성된다. 함수 구현은 3 개의 파라미터를 받아들여야 한다.

- 레코드 키
- 레코드 값
- 현재 집계 값

다음 코드를 통해 고득점 집계기를 생성할 수 있다.

```
Aggregator<String, Enriched, HighScores> highScoresAdder =
        (key, value, aggregate) -> aggregate.add(value);
```

Aggregate 이 HighScores 인스턴스임을 주목하기 바라며 집계기가 HighScores.add 메소드를 호출하기 때문에 HighScores 클래스에 이를 구현하면 된다. 다음 코드 블록에서 볼 수 있듯이 코드는 매우 간단한데 add 메소드는 새로운 고득점을 내부 TreeSet 에 단순히 추가하고 3 개의 고득점보다 많은 경우 가장 낮은 점수를 제거한다:

```
public class HighScores {
   private final TreeSet<Enriched> highScores = new TreeSet<>();
   public HighScores add(final Enriched enriched) {
        highScores.add(enriched); ①
        if (highScores.size() > 3) { ②
                highScores.remove(highScores.last());
        }
        return this;
   }
}
```

① 카프카 스트림즈가 adder 메소드 (HighScores.add)를 호출할 때마다 각 엔트리를 자동으로 정렬할 TreeSet 에 새로운 레코드를 추가한다.

② TreeSet 에 3 개 이상의 고득점이 존재하는 경우 가장 낮은 점수를 제거한다.

TreeSet 이 Enriched 객체 정렬 방법을 알 수 있도록 (따라서 고득점 집계가 3 개 이상의 값을 가질 때 제거할 가장 낮은 점수를 갖는 Enriched 레코드를 식별할 수 있도록) 예제 4-6 에 보듯이 Comparable 인터페이스를 구현할 것이다.

예제 4-6. Comparable 인터페이스를 구현하는 업데이트된 Enriched 클래스

```
public class Enriched implements Comparable<Enriched> { ①
    @Override
    public int compareTo(Enriched o) { ②
        return Double.compare(o.score, score);
    }
    // omitted for brevity
}
```

① Comparable 을 구현하도록 Enriched 클래스를 업데이트할 것이다. 왜냐하면 상위 3 개의 고득점을 결정하기 위해 하나의 Enriched 객체를 다른 것과 비교해야 하기 때문이다.

② compareTo 메소드 구현은 2 개의 다른 Enriched 객체 비교 메소드로 score 특성을 사용한다.

이제 initializer 와 adder 함수 모두를 준비했기 때문에 예제 4-7 의 코드를 사용하여 집계를 수행할 수 있다.

예제 4-7. 고득점 집계를 수행하기 위해 카프카 스트림즈의 aggregate 연산자를 사용한다

```
KTable highScores =
        grouped.aggregate(highScoresInitializer, highScoresAdder);
```

테이블 집계하기

테이블 집계 프로세스는 스트림 집계 프로세스와 매우 비슷하다. Initializer 와 add 함수가 필요하며 테이블이 가변이기 때문에 키가 삭제될 때마다 집계 값을 업데이트할 수 있어야 한다[16]. 또한 subtractor 함수라는 세번째 파라미터 또한 필요하다.

Subtractor

리더보드 예의 경우 필요하지 않지만 players KTable 의 플레이어 수를 카운트하길 원한다고 가정해보자. Count 연산자를 사용할 수 있지만 subtractor 함수를 구축하는 방법을 보여주기 위해 count 연산자와 본질적으로 동일한 집계 함수를 구축할 것이다. (KStream 과 KTable 집계에 필요한 initializer 와 adder 뿐만 아니라) subtractor 함수를 사용하는 집계의 기본 구현은 다음과 같다.

```
KGroupedTable<String, Player> groupedPlayers =
  players.groupBy(
      (key, value) -> KeyValue.pair(key, value),
      Grouped.with(Serdes.String(), JsonSerdes.Player()));
    groupedPlayers.aggregate(
      () -> 0L, ①
      (key, value, aggregate) -> aggregate + 1L, ②
      (key, value, aggregate) -> aggregate - 1L); ③
```

① initializer 함수는 집계를 0 으로 초기화한다.

② add 함수는 새로운 키를 볼 때 현재 카운트를 증분한다.

③ subtractor 함수는 키가 제거될 때 현재 카운트를 감소시킨다.

---

[16] 키 삭제에 대해서는 논의하지 않았는데 상태 저장소 정리를 논의하는 6 장에서 이 주제를 다룰 것이다.

이제 (그림 4-1 의) 리더보드 토폴로지의 5 단계를 마쳤다. 적당한 양의 코드를 작성했는데 따라서 다음 절에서는 개별 코드 부분이 어떻게 맞는지 확인해보자

함께 모으기

이제 리더보드 토폴로지의 개별 처리 단계들을 구축했기 때문에 함께 모아보자. 예제 4-8 은 지금까지 생성했던 토폴로지 단계가 함께 수행되는 방법을 보여준다.

예제 4-8. 비디오 게임 리더 보드 애플리케이션의 프로세서 토폴로지

```
// the builder is used to construct the topology
StreamsBuilder builder = new StreamsBuilder();
// register the score events stream
KStream<String, ScoreEvent> scoreEvents = ①
    builder
        .stream(
            "score-events",
            Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
        .selectKey((k, v) -> v.getPlayerId().toString()); ②
// create the partitioned players table
KTable<String, Player> players = ③
  builder.table("players", Consumed.with(Serdes.String(), JsonSerdes.Player()));
// create the global product table
GlobalKTable<String, Product> products = ④
  builder.globalTable(
        "products",
        Consumed.with(Serdes.String(), JsonSerdes.Product()));
// join params for scoreEvents - players join
Joined<String, ScoreEvent, Player> playerJoinParams =
  Joined.with(Serdes.String(), JsonSerdes.ScoreEvent(), JsonSerdes.Player());
// join scoreEvents - players
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
  (score, player) -> new ScoreWithPlayer(score, player);
KStream<String, ScoreWithPlayer> withPlayers =
  scoreEvents.join(players, scorePlayerJoiner, playerJoinParams); ⑤
// map score-with-player records to products
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper =
  (leftKey, scoreWithPlayer) -> {
        return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());
  };
// join the withPlayers stream to the product global ktable
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
```

```
    (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
KStream<String, Enriched> withProducts =
    withPlayers.join(products, keyMapper, productJoiner); ⑥
// Group the enriched product stream
KGroupedStream<String, Enriched> grouped =
    withProducts.groupBy( ⑦
            (key, value) -> value.getProductId().toString(),
            Grouped.with(Serdes.String(), JsonSerdes.Enriched()));
// The initial value of our aggregation will be a new HighScores instance
Initializer<HighScores> highScoresInitializer = HighScores::new;
// The logic for aggregating high scores is implemented in the HighScores.add method
Aggregator<String, Enriched, HighScores> highScoresAdder =
    (key, value, aggregate) -> aggregate.add(value);
// Perform the aggregation, and materialize the underlying state store for querying
KTable<String, HighScores> highScores =
    grouped.aggregate( ⑧
            highScoresInitializer,
            highScoresAdder);
```

① score-events 를 KStream 내로 읽어들인다.

② 조인에 필요한 co-partitioning 요건을 충족시키기 위해 메시지의 키를 재생성한다.

③ 키 공간이 크고 (다중 애플리케이션 인스턴스에 대해 상태를 샤딩할 수 있도록) score-events -> players 조인에 대해 시간 동기화된 처리를 원하기 때문에 players 토픽을 KTable 내로 읽어들인다.

④ 키 공간이 작고 시간 동기화된 처리가 필요가 없기 때문에 products 토픽을 GlobalKTable 내로 읽어들인다.

⑤ score-events 스트림과 players 테이블을 조인한다.

⑥ 보강된 score-events 를 products 테이블과 조인한다.

⑦ 보강된 스트림을 그룹화한다. 이는 집계를 위한 선결 조건이다.

⑧ 그룹화된 스트림을 집계한다. 집계 로직은 HighScores 클래스 내에 존재한다.

애플리케이션에 필요한 구성을 추가하고 스트리밍을 시작해보자:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

이 시점에서 애플리케이션은 리더보드에 대해 레코드를 수신하고 고득점 계산을 시작할 준비를 마쳤다. 그러나 리더보드 결과를 외부 클라이언트에 노출하기 위해 해결해야 할 한 가지 최종 단계가 존재한다. 프로세서 토폴로지의 최종 단계로 이동해서 상호대화형 쿼리를 사용하여 카프카 스트림즈 애플리케이션의 상태를 노출하는 방법을 배워보자.

상호대화형 쿼리

카프카 스트림즈를 정의하는 기능 중 하나는 로컬과 외부 모두에 애플리케이션 상태를 노출할 수 있다는 것이다. 후자는 매우 낮은 지연을 갖는 이벤트 기반 마이크로 서비스 구축을 쉽게 한다. 이 튜토리얼에서는 고득점 집계를 노출하기 위해 상호대화형 쿼리를 사용할 수 있다.

이를 위해 상태 저장소를 구체화해야 한다. 다음 절에서 이를 하는 방법을 배울 것이다.

구체화된 (materialized) 저장소

이미 aggregate, count, reduce 등의 스테이트풀 연산자가 내부 상태를 관리하기 위해 상태 저장소를 활용한다는 것을 알고 있다. 그러나 예제 4-7 의 고득점을 집계하기 위한 메소드를 자세히 살펴보면 상태 저장소에 대한 어떤 언급도 찾지 못할 것이다.

Aggregate 메소드의 이 변형은 프로세서 토폴로지만 액세스할 수 있는 내부 상태 저장소를 사용한다.

임시 쿼리를 위해 상태 저장소에 대해 읽기 전용 액세스를 활성화하려면 상태 저장소의 구체화를 로컬에서 강제하기 위해 오버로딩된 메소드 중 하나를 사용할 수 있다. 구체화된 저장소는 명시적인 이름을 가지며 프로세서 토폴로지 외부에서 쿼리가 가능하다는 점에서 내부 상태 저장소와 다르다. 이는 Materialized 클래스가 유용한 곳으로 예제 4-9 는 상호대화형 쿼리를 사용할 수 있도록 하는 Materialized 클래스를 사용하여 영구 키-값 저장소를 구체화하는 방법을 보여준다.

예제 4-9. 최소 구성을 갖는 구체화된 저장소

```
KTable<String, HighScores> highScores =
    grouped.aggregate(
            highScoresInitializer,
            highScoresAdder,
            Materialized.<String, HighScores, KeyValueStore<Bytes, byte[]>> ①
                    as("leader-boards")②
                    .withKeySerde(Serdes.String()) ③
                    .withValueSerde(JsonSerdes.HighScores()));
```

① Materialized.as 메소드의 변형은 3 개의 제네릭을 포함한다.

- 저장소의 키 타입 (이 경우 String)
- 저장소의 값 타입 (이 경우 HighScores)

- 상태 저장소 타입 (이 경우 KeyValueStore<Bytes, byte[]>로 표현되는 간단한 키-값 저장소를 사용할 것이다).

② 프로세서 토폴로지 외부에서 쿼리할 수 있도록 저장소에 명시적인 이름을 제공한다.

③ 키와 값 Serde 를 포함한 다양한 파라미터와 6 장에서 논의할 다른 옵션을 사용하여 구체화된 상태 저장소를 맞춤화할 수 있다.

일단 leader-boards 상태 저장소를 구체화했다면 임시 쿼리를 통해 이 데이터를 노출할 준비가 거의 된 것이다. 그러나 해야 할 첫번째 일은 카프카 스트림즈로부터 저장소를 검색하는 것이다.

읽기 전용 상태 저장소 액세스하기

읽기 전용 모드로 상태 저장소에 액세스해야 할 때 2 가지 정보가 필요하다:

- 상태 저장소 이름
- 상태 저장소 타입

예제 4-9 에서 보듯이 상태 저장소 이름은 leader-boards 로 QueryableStoreTypes 팩토리 클래스를 사용하여 상태 저장소에 대한 적절한 읽기 전용 래퍼를 검색해야 한다. 다음을 포함하여 다중 상태 저장소가 지원된다:

- QueryableStoreTypes.keyValueStore()
- QueryableStoreTypes.timestampedKeyValueStore()
- QueryableStoreTypes.windowStore()
- QueryableStoreTypes.timestampedWindowStore()
- QueryableStoreTypes.sessionStore()

이 예에서는 간단한 키-값 저장소를 사용하며 따라서 QueryableStoreType.KeyValueStore() 메소드가 필요하다. 상태 저장소 이름과 상태 저장소 타입 모두를 이용하여 예제 4-10 에 보이는 바와 같이 kafkaStreams.store() 메소드를 사용함으로써 상호대화형 쿼리에 사용되는 쿼리가능 상태 저장소의 인스턴스를 인스턴스화할 수 있다.

예제 4-10. 상호대화형 쿼리를 수행하는데 사용할 수 있는 키-값 저장소를 인스턴스화한다.

```
ReadOnlyKeyValueStore<String, HighScores> stateStore =
      streams.store(
            StoreQueryParameters.fromNameAndType(
                  "leader-boards",
                  QueryableStoreTypes.keyValueStore()));
```

상태 저장소 인스턴스가 존재할 때 쿼리할 수 있다. 다음 절에서는 키-값 저장소에서 사용가능한 다양한 쿼리 타입을 논의한다.

비윈도우 키-값 저장소에 쿼리하기

각각의 상태 저장소 타입은 다른 유형의 쿼리를 지원한다. 예를 들어 윈도우 저장소(예, ReadOnlyWindowStore)는 시간 범위를 사용한 키 조회를 지원하며 간단한 키-값 저장소(ReadOnlyKeyValueStore)는 포인트 조회, 범위 스캔과 카운트 쿼리를 지원한다.

다음 절에서 윈도우 상태 저장소를 논의할 것이며 지금은 leader-boards 저장소에 할 수 있는 쿼리 유형을 살펴보자.

상태 저장소 타입에 대해 어떤 쿼리 유형이 사용가능한지를 결정하는 가장 쉬운 방법은 기본 인터페이스를 확인하는 것이다. 다음 스니펫의 인터페이스 정의로부터 볼 수 있듯이 간단한 키-값 저장소는 몇 개의 다른 유형의 쿼리를 지원한다:

```
public interface ReadOnlyKeyValueStore<K, V> {
        V get(K key);
        KeyValueIterator<K, V> range(K from, K to);
        KeyValueIterator<K, V> all();
        long approximateNumEntries();
}
```

첫번째 포인트 조회(get())로 시작하여 이들 쿼리 유형 각각을 살펴보자.

포인트 조회

아마도 가장 일반적인 쿼리 유형인 포인트 조회는 개별 키에 대해 상태 저장소에 쿼리하는 것이다. 이 유형의 쿼리를 수행하기 위해 주어진 키에 대한 값을 검색하는 get 메소드를 사용할 수 있다. 예를 들어:

```
HighScores highScores = stateStore.get(key);
```

포인트 조회는 값의 역직렬화된 인스턴스 (이 경우 상태 저장소에 저장하고 있는 HighScores 객체) 또는 키가 존재하지 않는 경우 null 을 반환함을 주목하기 바란다.

범위 스캔

간단한 키-값 저장소는 또한 범위 스캔 쿼리도 지원한다. 범위 스캔은 키의 포함 범위에 대해 반복자를 반환한다. 메모리 누수를 막기 위해 일단 쿼리가 종료되면 반복자를 종료하는 것이 매우 중요하다.

다음 코드 블록은 범위 쿼리를 수행하고, 각 결과에 대해 반복하며 반복자를 종료하는 방법을 보여준다:

```
KeyValueIterator<String, HighScores> range = stateStore.range(1, 7); ①
while (range.hasNext()) {
        KeyValue<String, HighScores> next = range.next(); ②
```

```
String key = next.key;
HighScores highScores = next.value; ③
// do something with high scores object
range.close();
```

① 선택 범위에서 각 키를 통해 반복하기 위해 사용될 수 있는 반복자를 반환한다.

② 반복에서 다음 요소를 가져온다.

③ next.value 특성에서 HighScores 값이 사용가능하다.

④ 메모리 누수를 막기 위해 반복자를 종료하는 것이 매우 중요하다. 다른 종료 방법은 반복자를 얻을 때 try-with-resources 문을 사용하는 것이다.

모든 엔트리

범위 스캔과 비슷하게 all() 쿼리는 키-값 쌍에 대한 반복자를 반환하며 필터링 조건이 없는 SELECT * 쿼리와 유사하다. 그러나 이 쿼리 유형은 특정 키 범위 내의 엔트리 대신 상태 저장소 내 모든 엔트리에 대한 반복자를 반환할 것이다. 범위 쿼리와 같이 메모리 누수를 막기 위해 쿼리가 종료되면 반복자를 종료하는 것이 중요하다. 다음 코드는 all() 쿼리를 실행하는 방법을 보여준다. 결과에 대해 반복하고 반복자를 종료하는 것은 범위 스캔 쿼리와 동일하며 따라서 편의상 이 로직은 생략하였다:

```
KeyValueIterator range = stateStore.all();
```

엔트리 수

마지막으로 최종 쿼리 유형은 COUNT(*) 쿼리와 유사하며 기본 상태 저장소 내 엔트리의 대략적인 수를 반환한다.

> RocksDB 영구 저장소를 사용할 때 반환 값은 정확한 카운트를 계산하는 것이 과도할 수 있기 때문에 근사 값으로 RocksDB 지원 저장소의 경우 도전적인 문제이다. RocksDB FAQ 로부터 발췌:
>
> > RocksDB 와 같은 LSM 데이터베이스에서 키의 정확한 수를 얻는 것은 DB 가 정확한 키의 수를 얻기 위해 전체 압축을 필요로 하는 중복 키와 삭제 엔트리 (예, 툼스톤)를 갖고 있기 때문에 도전적인 문제이다. 이외에 RocksDB 데이터베이스가 merge 연산자를 포함하고 있다면 키에 대해 평가된 수는 덜 정확할 것이다.
>
> 반면 인메모리 저장소를 사용한다면 카운트는 정확할 것이다.

간단한 키-값 저장소에 대해 이러한 유형의 쿼리를 수행하기 위해 다음 코드를 실행시킬 수 있다:

```
long approxNumEntries = stateStore.approximateNumEntries();
```

이제 키-값 저장소에 쿼리하는 방법을 알고 있기 때문에 이들 쿼리를 실행할 수 있는 위치를 살펴보자.

로컬 쿼리

카프카 스트림즈 애플리케이션의 각 인스턴스는 자신의 로컬 저장소에 쿼리할 수 있다. 그러나 GlobalKTable 을 구체화하지 않거나 카프카 스트림즈 앱[17]의 단일 인스턴스를 실행시키지 않는다면 로컬 상태는 단지 전체 애플리케이션 상태의 부분적 뷰만 표현할 것임을 기억하는 것은 중요하다 (이는 KTable 의 특성이다).

운좋게도 카프카 스트림즈는 분산 상태 저장소에 접속하여 애플리케이션의 전체 상태를 쿼리할 수 있도록 하는 원격 쿼리를 실행하는 것을 쉽게 하는 추가적인 메소드를 제공한다. 다음에 원격 쿼리에 대해 배울 것이다.

원격 쿼리

애플리케이션의 전체 상태를 쿼리하기 위해 다음이 필요하다:

- 애플리케이션 상태의 다양한 부분들을 포함하는 인스턴스 발견
- 로컬 상태를 다른 실행 중인 애플리케이션 인스턴스[18]로 노출하기 위해 RPC (Remote Procedure Call) 또는 REST 서비스 추가
- 실행 중인 애플리케이션 인스턴스로부터 원격 상태 저장소에 쿼리하기 위한 RPC 또는 REST 클라이언트 추가

마지막 두 요점과 관련하여 원격 통신에 사용할 서버 및 클라이언트 컴포넌트를 선택할 때 많은 유연성이 존재한다. 이 튜토리얼에서는 REST 서비스를 구현하기 위해 간단한 API 를 갖는 Javalin 을 사용할 것이다. 또한 REST 클라이언트에 대해서는 손쉬운 사용으로 인해 Square 에 의해 개발한 OkHttp 를 사용할 것이다. 다음과 같이 build.gradle 파일을 업데이트하여 애플리케이션에 이들의 의존성을 추가하자:

```
dependencies {
        // required for interactive queries (server)
implementation 'io.javalin:javalin:3.12.0'
        // required for interactive queries (client)
 implementation 'com.squareup.okhttp3:okhttp:4.9.0'
        // other dependencies
}
```

---

[17] 후자는 바람직하지 않다. 단일 카프카 스트림즈 애플리케이션을 실행하는 것은 전체 애플리케이션 상태를 단일 인스턴스로 통합하지만 카프카 스트림즈는 성능 최대화와 내고장성을 위해 분산 방식으로 실행된다.

[18] 원하는 경우 다른 클라이언트, 예, 사람

이제 인스턴스 발견이라는 문제를 해결해보자. 특정 시점에 실행 중인 인스턴스와 이 인스턴스의 위치를 브로드캐스팅할 방법이 필요하다. 후자는 다음과 같이 카프카 스트림즈의 호스트와 포트 쌍을 지정하는 APPLICATION_SERVER_CONFIG 파라미터를 사용하여 달성될 수 있다:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_SERVER_CONFIG, "myapp:8080"); ①
// other Kafka Streams properties omitted for brevity
KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

① 엔드 포인트를 구성한다. 이는 카프카의 컨슈머 그룹 프로토콜을 통해 다른 실행 중인 인스턴스로 전달될 것이다. 애플리케이션과 통신하기 위해 다른 인스턴스가 사용할 수 있는 IP 와 포트를 사용하는 것이 중요하다 (예, 인스턴스에 따라 다른 IP 가 될 것이기 때문에 localhost 는 작동하지 않을 것이다).

APPLICATION_SERVER_CONFIG 파라미터 구성을 설정하는 것이 카프카 스트림즈에게 구성한 포트에서 대기요청을 시작하라는 것은 아님을 주목하기 바란다. 사실 카프카 스트림즈는 내장 RPC 서비스를 포함하고 있지 않다. 그러나 이 호스트/포트 정보는 다른 실행 중인 인스턴스로 전송되며 뒤에 논의할 전용 API 메소드를 통해 사용가능해진다. 그러나 우선 적절한 포트 (예, 8080)에서 대기요청을 시작하기 위해 REST 서비스를 설정하자.

코드 유지보수성 측면에서 토폴로지 정의와 별도로 전용 파일에 리더보드 REST 서비스를 정의하는 것이 의미가 있다. 다음 코드 블록은 리더보드 서비스의 간단한 구현을 보여준다.

```
class LeaderboardService {
    private final HostInfo hostInfo; ①
    private final KafkaStreams streams; ②
    LeaderboardService(HostInfo hostInfo, KafkaStreams streams) {
        this.hostInfo = hostInfo;
        this.streams = streams;
    }
    ReadOnlyKeyValueStore<String, HighScores> getStore() { ③
        return streams.store(
                StoreQueryParameters.fromNameAndType(
                        "leader-boards",
                        QueryableStoreTypes.keyValueStore()));
    }
    void start() {
        Javalin app = Javalin.create().start(hostInfo.port()); ④
        app.get("/leaderboard/:key", this::getKey); ⑤
    }
}
```

① HostInfo 는 호스트네임과 포트를 포함하는 카프카 스트림즈의 래퍼 클래스이다. 이를 인스턴스화하는 법을 곧 볼 것이다.

② 로컬 카프카 스트림즈 인스턴스를 추적해야 하는데 다음 코드 불록에서 이 인스턴스에 대한 API 메소드를 사용할 것이다.

③ 리더보드 집계를 포함한 상태 저장소를 검색하기 위해 전용 메소드를 추가한다. 이는 예제 4-10 에서 보았던 읽기 전용 상태 저장소 래퍼를 검색하기 위한 것과 동일한 메소드를 따른다.

④ 구성된 포트에서 Javalin 기반 웹 서비스를 시작한다.

⑤ Javalin 으로 엔드 포인트를 추가하는 것은 쉽다. URL 경로를 곧 구현할 메소드에 매핑한다. 선행 콜론 (예, :key)으로 지정된 경로 파라미터를 통해 동적인 엔트포인트를 생성할 수 있다. 이는 포인트 조회 쿼리에 이상적이다.

이제 주어진 키 (이 경우 제품 ID)에 대해 고득점을 보여줄 /leaderboard/:key 엔드포인트를 구현해보자. 최근에 배운 것과 같이 상태 저장소로부터 단일 값을 검색하기 위해 포인트 조회를 사용할 수 있다. 다음은 전문적이지 못한 구현이다.

```
void getKey(Context ctx) {
  String productId = ctx.pathParam("key");
  HighScores highScores = getStore().get(productId); ①
  ctx.json(highScores.toList()); ②
}
```

① 로컬 상태 저장소로부터 값을 검색하기 위해 포인트 조회를 사용한다.

② 비고: toList() 메소드가 소스 코드에서 사용가능하다.

불행히도 이는 충분치 않다. 카프카 스트림즈 애플리케이션이 실행 중인 2 개의 인스턴스가 있는 예를 고려해보자. 어떤 인스턴스에 언제 쿼리할 수 있는지에 따라 요청된 값을 검색하지 못할 수도 있다. 그림 4-7 은 이러한 수수께끼를 보여준다.

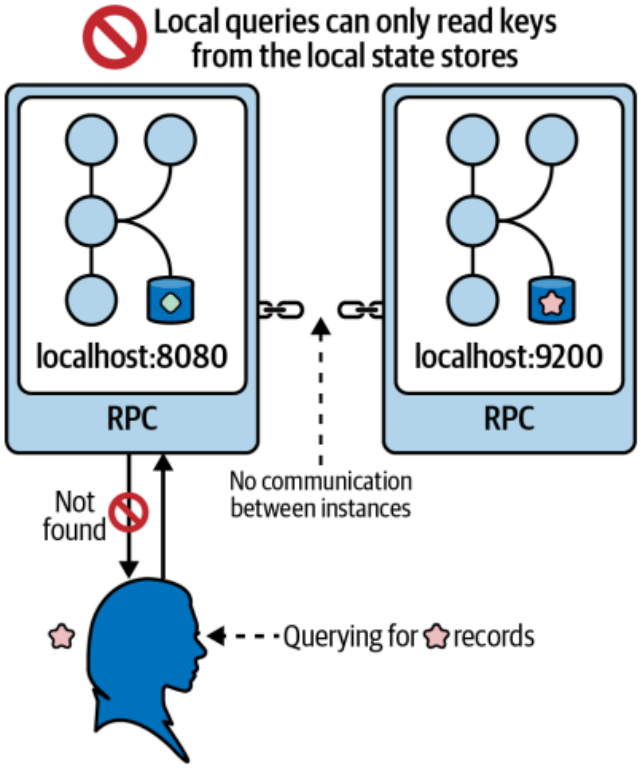

그림 4-7. 상태가 다중 애플리케이션 인스턴스에 대해 파티션되어 있을 때 로컬 쿼리는 충분치 않다.

운좋게도 카프카 스트림즈는 지정된 키가 존재하는 애플리케이션 인스턴스 (로컬 또는 원격)를 발견할 수 있도록 하는 queryMetadataForKey [19]메소드를 제공한다. getKey 메소드의 개선된 구현을 예제 4-11 에 나타냈다.

예제 4-11. 다른 애플리케이션 인스턴스로부터 데이터를 가져오기 위해 원격 쿼리를 활용하는 getKey 메소드의 업데이트된 구현

```
void getKey(Context ctx) {
 String productId = ctx.pathParam("key");
 KeyQueryMetadata metadata =
            streams.queryMetadataForKey(
                        "leader-boards", productId, Serdes.String().serializer()); ①
 if (hostInfo.equals(metadata.activeHost())) {
HighScores highScores = getStore().get(productId); ②
 if (highScores == null) { ③
             // game was not found
             ctx.status(404);
             return;
```

[19] 2.5 미만 버전에서 널리 사용되었던 metadataForKey 메소드를 대신. 그러나 공식적으로 유지보수가 지원되지 않음

```
}
// game was found, so return the high scores
ctx.json(highScores.toList()); ④
return;
}
// a remote instance has the key
String remoteHost = metadata.activeHost().host();
int remotePort = metadata.activeHost().port();
String url =
        String.format(
                "http://%s:%d/leaderboard/%s",
                remoteHost, remotePort, productId); ⑤
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder().url(url).build();
try (Response response = client.newCall(request).execute()) { ⑥
        ctx.result(response.body().string());
} catch (Exception e) {
        ctx.status(500);
}
}
```

① queryMetadataForKey 를 통해 특정 키가 존재하는 호스트를 찾을 수 있다.

② 로컬 인스턴스가 키를 갖고 있다면 단지 로컬 상태 저장소에 쿼리한다.

③ queryMetadataForKey 메소드는 키가 존재하는지 실제 확인하지 않는다. 키가 존재하는 경우 어떤 키가 존재할 것인지를 결정하기 위해 기본 스트림 파티셔너를 사용한다. 따라서 (키가 발견되지 않는 경우 반환되는) null 을 확인할 수 있으며 존재하지 않는 경우 404 응답을 반환할 것이다.

④ 고득점을 포함하는 포맷된 응답을 반환한다.

⑤ 여기까지 왔다면 키가 존재하는 경우 키는 원격 호스트에 존재한다. 따라서 지정된 키를 포함할 카프카 스트림즈 인스턴스의 호스트와 포트를 포함하는 메타데이터를 사용하여 URL 을 생성한다.

⑥ 요청을 호출하고 성공인 경우 결과를 반환한다.

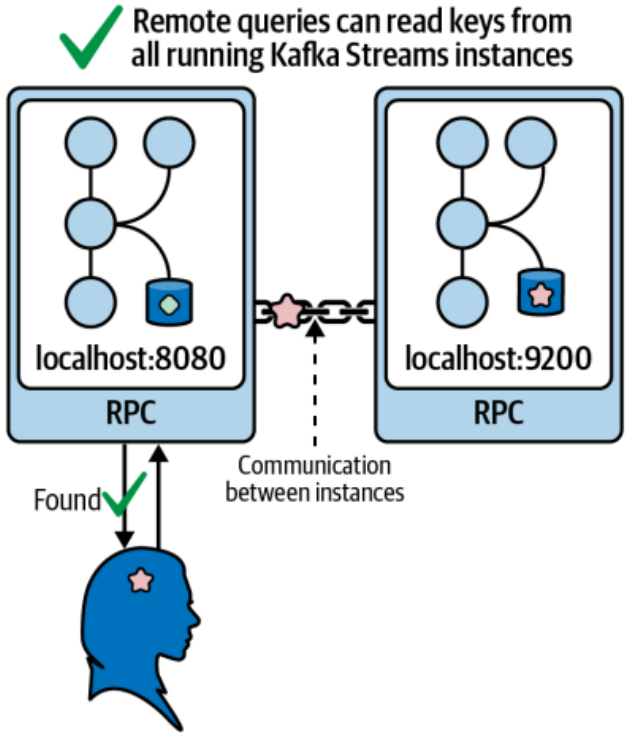

그림 4-8. 원격 쿼리를 통해 실행 중인 다른 애플리케이션 인스턴스의 상태를 쿼리할 수 있다.

그러나 단일 키에 대해 작동하지 않는 쿼리를 실행시킬 필요가 있다면? 예를 들어 분산 상태 저장소 모두에 대한 엔트리 수를 셀 필요가 있다면? queryMetadataForKey 는 단일 키를 지정해야 하기 때문에 이 경우 잘 작동하지 않을 것이다. 대신 동일한 애플리케이션 ID 를 공유하고 제공된 저장소 이름에 대해 최소한 하나의 활성 파티션을 갖고 있는 모든 실행 중인 카프카 스트림즈 애플리케이션에 대한 엔드 포인트를 반환하는 allMetadataForStore 메소드를 활용할 것이다.

실행 중인 모든 애플리케이션 인스턴스에 대해 고득점 레코드 수를 노출하는 리더보드 서비스에 새로운 엔드 포인트를 추가해보자:

```
app.get("/leaderboard/count", this::getCount);
```

이제 이전 코드에서 참조된 getCount 메소를 구현할 것인데, 각각 원격 상태 저장소 내 레코드의 총 수를 얻기 위해 allMetadataForStore 메소드를 활용한다.

```
void getCount(Context ctx) {
 long count = getStore().approximateNumEntries(); ①
                                                  ②
 for (StreamsMetadata metadata : streams.allMetadataForStore("leader-boards")) {
 if (!hostInfo.equals(metadata.hostInfo())) {
         continue; ③
 }
 count += fetchCountFromRemoteInstance( ④
         metadata.hostInfo().host(),
```

```
            metadata.hostInfo().port());
    }
    ctx.json(count);
  }
```

① 카운트를 로컬 상태 저장소 내 엔트리 수로 초기화한다.

② 다음 라인에서 쿼리하고자 하는 상태의 부분을 포함하고 있는 각 카프카 스트림즈 인스턴스의 호스트/포트 쌍을 검색하는 allMetadataForStore 메소드를 사용한다.

③ 메타데이터가 현재 호스트에 대한 것이라면 이미 로컬 상태 저장소에서 엔트리 카운트를 가져왔기 때문에 루프를 통해 계속 진행한다.

④메타데이터가 로컬 인스턴스와 관련이 없다면 원격 인스턴스로부터 카운트를 검색한다. REST 클라이언트를 인스턴스화했고 원격 애플리케이션 인스턴스에 대해 요청을 발행했던 예제 4-11 과 비슷하기 때문에 이 텍스트에서는 fetchCountFromRemoteInstance 의 구현 세부 사항을 생략하였다. 구현 세부 사항이 관심이 있다면 이 장의 소스 코드를 확인하기 바란다.

소스 토픽 각각에 대한 더미 데이터는 예제 4-12 에 나타냈다.

키가 있는 토픽의 경우 (players 와 products) 레코드 키의 포맷은 <key>|<value>이다. Score-events 토픽의 경우 더미 레코드 포맷은 <value>이다.

예제 4-12. 소스 토픽에 생산할 더미 레코드

```
# players
1|{"id": 1, "name": "Elyse"}
2|{"id": 2, "name": "Mitch"}
3|{"id": 3, "name": "Isabelle"}
4|{"id": 4, "name": "Sammy"}
# products
1|{"id": 1, "name": "Super Smash Bros"}
6|{"id": 6, "name": "Mario Kart"}
# score-events
{"score": 1000, "product_id": 1, "player_id": 1}
{"score": 2000, "product_id": 1, "player_id": 2}
{"score": 4000, "product_id": 1, "player_id": 3}
{"score": 500, "product_id": 1, "player_id": 4}
{"score": 800, "product_id": 6, "player_id": 1}
{"score": 2500, "product_id": 6, "player_id": 2}
{"score": 9000.0, "product_id": 6, "player_id": 3}
{"score": 1200.0, "product_id": 6, "player_id": 4}
```

이 더미 레코드를 적절한 토픽에 생산한 후 리더보드 서비스에 쿼리하면 카프카 스트림즈 애플리케이션이 고득점을 처리할 뿐만 아니라 스테이트풀 연산 결과를 노출하고 있음을 확인할 것이다. 상호대화형 쿼리에 대한 응답 예는 다음 코드 블록과 같다:

```
$ curl -s localhost:7000/leaderboard/1 | jq '.'
[
 {
 "playerId": 3,
 "productId": 1,
 "playerName": "Isabelle",
 "gameName": "Super Smash Bros",
 "score": 4000
 },
 {
 "playerId": 2,
 "productId": 1,
 "playerName": "Mitch",
 "gameName": "Super Smash Bros",
 "score": 2000
 },
 {
 "playerId": 1,
 "productId": 1,
 "playerName": "Elyse",
 "gameName": "Super Smash Bros",
 "score": 1000
 }
]
```

요약

이 장에서는 카프카 스트림즈가 소비하는 이벤트에 대한 정보를 수집하는 방법과 다음을 포함하여 보다 고급 스트림 처리 태스크를 수행하기 위해 기억된 정보 (상태)를 활용하는 방법을 배웠다:

- KStream-KTable 조인 수행
- 특정 조인 유형에 대해 co-partitioning 요건을 충족시키기 위한 메시지 키재생성
- KStream-GlobalKTable 조인 수행
- 집계를 위한 데이터 준비를 위해 레코드를 중간 표현으로 그룹화 (KGroupedStream, KGroupedTable)
- 스트림과 테이블 집계

- 로컬과 원격 쿼리 모두를 사용하여 애플리케이션의 상태를 노출하기 위해 상호대화형 쿼리 사용

다음 장에서는 애플리케이션이 본 이벤트 뿐만 아니라 언제 발생했는지와 관련된 스테이트풀 프로그래밍의 다른 측면을 논의할 것이다. 시간은 스테이트풀 처리에서 중요한 역할을 하는데 따라서 시간의 다른 개념과 카프카 스트림즈 라이브러리의 일부 시간 기반 추상화를 이해함으로써 스테이트풀 처리에 대한 지식을 보다 확장할 수 있다.