2 장 카프카 스트림즈 시작하기

카프카 스트림즈는 실시간 데이터 스트림의 보강, 변환 및 처리를 위한 경량이지만 강력한 Java 라이브러리이다. 이 장에서는 고수준에서 카프카 스트림즈를 소개할 것인데, 첫번째 데이트로 생각해 보자. 카프카 스트림즈에 대한 배경을 배우고 기능을 처음 살펴볼 것이다.

데이트 후, 즉 이 장 끝에서 다음을 이해할 수 있을 것이다:

- 카프카 생태계에서 카프카 스트림즈의 적합한 위치
- 카프카 스트림즈의 최초 개발 이유
- 카프카 스트림즈 라이브러리의 기능 및 작동 특성
- 카프카 스트림즈가 적합한 대상
- 카프카 스트림즈와 다른 스트림 처리 솔루션과의 비교
- 기본적인 카프카 스트림즈 애플리케이션 작성 및 실행 방법

따라서 더 이상 고민하지 않고 카프카 생태계에서 카프카 스트림즈의 위치에 대한 간단한 질문으로 은유적 데이트를 시작한다.

카프카 생태계

카프카 스트림즈는 집합적으로 카프카 생태계로 간주되는 기술 그룹 중에 속한다. 1 장에서 우리는 아파치 카프카가 메시지를 생산하고 메시지를 읽어 올 수 있는 분산된 추가 전용 로그임을 배웠다. 또한 핵심 카프카 코드는 (토픽이라는 메시지 카테고리로 분리되는) 이 로그와 상호작용하기 위한 일부 중요한 API를 포함하고 있다. 표 2-1에 요약된 카프카 생태계 내 3 가지 API는 카프카로/카프카로부터의 데이터 이동과 관련이 있다.

표 2-1. 카프카로/카프카로부터의 데이터 이동을 위한 API

API	토픽 상호작용	예
프로듀서 API	카프카 토픽으로 메시지 작성	filebeat rsyslog 맞춤형 프로듀서
컨슈머 API	카프카 토픽에서 메시지 읽기	logstash kafkacat 맞춤형 컨슈머
커넥트 API	외부 데이터 저장소, API 및 파일 시스템과 카프카 토픽을 연결. 토픽에서 읽는 싱크 커넥터와 토픽에 작성하는 소스 커넥터 포함	JDBC 소스 커넥터 Elasticsearch 싱크 커넥터 맞춤형 커넥터

그러나 카프카를 통해 데이터를 움직이는 것이 데이터 파이프라인 생성에 분명히 중요하지만 어떤 비즈니스 문제는 카프카에서 사용가능 할 때 데이터를 처리하고 이에 대응하는 것을 필요로 한다. 이는

스트림 처리로 알려져 있는데 카프카를 사용해 스트림 처리 애플리케이션을 구축하기 위한 여러 방법이 존재한다. 따라서 카프카 스트림즈 도입 이전에 스트림 처리 애플리케이션이 어떻게 구현되었는 지와 전용 스트림 처리 라이브러리가 카프카 생태계 내 다른 API와 함께 어떻게 존재하게 되었는 지를 살펴보자.

카프카 스트림즈 이전

카프카 스트림즈가 등장하기 전 카프카 생태계에는 빈틈이 있었다.[1] 여러분을 활기차게 하고 밝게 만드는 오전 명상 중에 만날 수 있는 유형의 빈틈이 아니라 스트림 처리 애플리케이션 구축을 필요이상으로 어렵게 만드는 빈틈이었다. 여기서는 카프카 토픽에서 데이터를 처리하기 위한 라이브러리 지원의 부족을 이야기하고 있다.

초기 카프카 생태계에는 카프카 기반 스트림 처리 애플리케이션을 구축하기 위한 2 가지 주요 옵션이 있었다.

- 컨슈머 및 프로듀서 API 직접 사용
- 다른 스트림 처리 프레임워크 사용 (예, 아파치 Spark Streaming, 아파치 Flink)

컨슈머 및 프로듀서 API를 통해 Python, Java, Go, C/C++, Node.js 등 다양한 프로그래밍 언어를 사용하여 직접 이벤트 스트림 읽기 및 쓰기를 할 수 있고 처음부터 많은 코드를 작성하고자 한다면 원하는 어떤 유형의 데이터 처리 로직도 수행할 수 있다. 이 API는 매우 기본적인 것으로 다음을 포함하여 스트림 처리 API로 인정받을 만한 많은 프리미티브 (primitive)가 부족하다.

- 로컬 및 내고장성 상태[2]
- 데이터 스트림 변환을 위한 풍부한 연산자들
- 스트림에 대한 보다 우수한 표현[3]
- 복잡한 시간 처리[4]

따라서 레코드 집계, 별개의 데이터 스트림 조인, 타임 윈도우 버킷 내 이벤트 그룹화 또는 스트림에

---

[1] 여기서는 아파치 카프카 프로젝트 하에 유지되고 있는 모든 컴포넌트들을 포함하는 공식 생태계를 의미한다.

[2] 아파치 카프카의 원 개발자 중 한 명인 Jay Kreps는 2014년 게시된 O'Reilly 블로그에서 이를 세부적으로 논의하였다.

[3] 이는 추후 논의할 예정인 집계된 스트림/테이블을 포함한다.

[4] 이를 중점으로 다루는 한 장을 포함하였으며 2019년 카프카 서밋의 Matthias J. Sax의 발표를 참조하기 바란다.

대한 임시 쿼리 수행과 같이 사소하지 않은 무언가를 하길 원한다면 매우 빨리 복잡성의 벽에 부딪힐 것이다. 프로듀서 및 컨슈머 API는 이러한 유형의 유스케이스에 대해 도움이 될 만한 어떤 추상화도 포함하고 있지 않으며 따라서 이벤트 스트림에 대해 보다 고급의 무언가를 해야 할 때마다 스스로 무언가를 해야 할 것이다.

아파치 Spark 또는 아파치 Flink와 같이 본격적인 스트리밍 플랫폼 채택을 포함하는 두번째 옵션은 불필요한 많은 복잡성을 끌어들인다. "다른 시스템과의 비교"에서 이 접근방법의 단점에 대해 논의하는데 간략히 말해 간결함과 능력에 대해 최적화하고자 한다면 처리 클러스터의 간접비 없이 스트림 처리 프리미티브를 제공하는 솔루션을 필요로 한다는 것이다. 또한 소스 및 싱크 토픽 외부에서 데이터의 중간 표현을 사용하고자 할 때 카프카와의 더 나은 통합을 필요로 한다.

운좋게도 카프카 커뮤니티는 카프카 생태계에서 스트림 처리 API에 대한 니즈를 인식하였고 이를 구축하기로 결정하였다[5].

카프카 스트림즈 들어가기

2016년 카프카 스트림즈의 첫번째 버전 (스트림즈 API)이 발표되었을 때 카프카 생태계는 영원히 바뀌었다. 시작과 함께 수작업으로 구현했던 기능에 크게 의존했던 스트림 처리 애플리케이션들은 실시간 이벤트 스트림 변환 및 처리를 위해 커뮤니티에서 개발된 패턴과 추상화를 활용했던 보다 첨단 애플리케이션에 길을 양보하였다.

프로듀서, 컨슈머 및 커넥트 APIs와 달리 카프카 스트림즈는 데이터를 단지 카프카로 그리고 카프카로부터 옮기는 것이 아니라[6] 실시간 데이터 스트림 처리에 전념한다. 따라서 실시간 이벤트 스트림이 데이터 파이프라인을 통해 이동할 때 이들을 소비하고, 풍부한 스트림 처리 연산자와 프리미티브 집합을 사용하는 데이터 변환 로직을 적용하며 선택적으로 새로운 데이터 표현을 카프카에 다시 적재하는 것을 쉽게 한다 (예, 변환된 또는 보강된 이벤트를 다운스트림 시스템에서 사용가능 하길 원하는 경우).

그림 2-1은 카프카 생태계에서 이전에 논의된 API를 나타내는데 카프카 스트림즈가 스트림 처리 계층에서 작동한다.

---

[5] 카프카 스트림즈 개발에 주요 역할을 했던 Guozhang Wang은 추후 카프카 스트림즈가 된 원래 KIP를 제출한 것에 대해 인정을 받을 만하다. https://oreil.ly/l2wbc참조.

[6] 카프카 커넥트는 단일 메시지 변환(single message transform)을 지원함으로써 이벤트 처리 영역으로 약간 방향을 바꾸었지만 이는 카프카 스트림즈가 할 수 있는 것에 비해 극히 제한적이다.

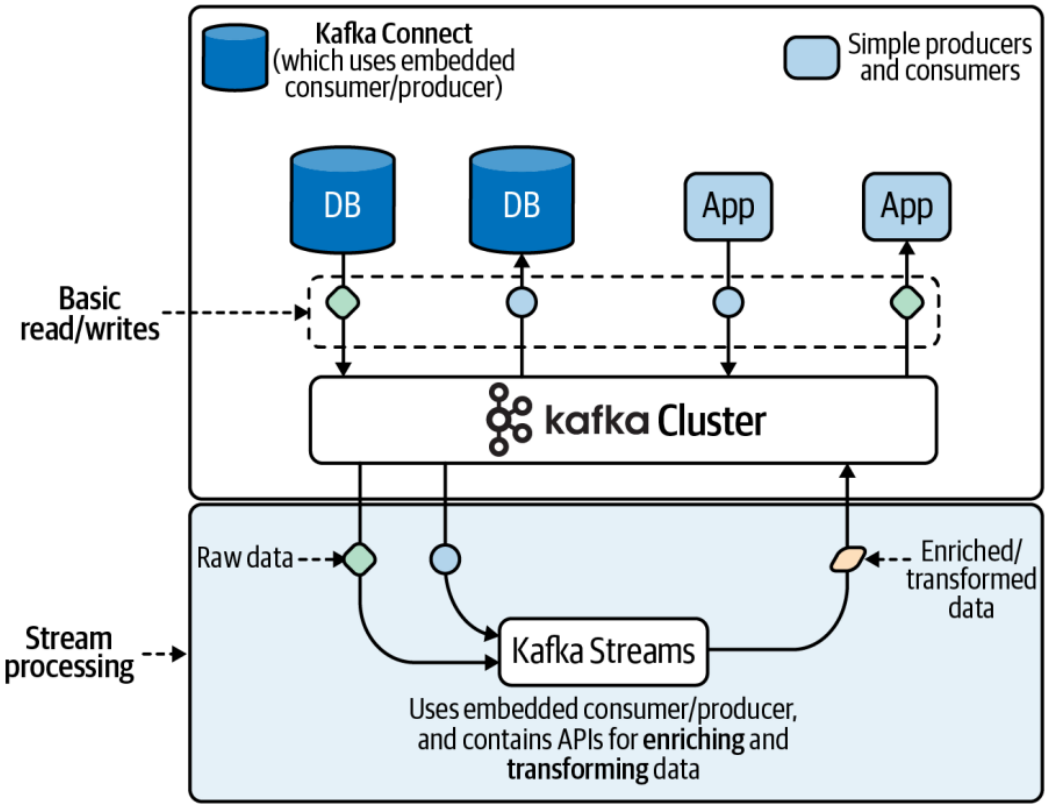


그림 2-1. 카프카 스트림즈는 카프카 생태계의 두뇌로 이벤트 스트림으로부터 레코드를 소비하고, 데이터를 처리하며 선택적으로 보강된 또는 변환된 레코드를 다시 카프카로 적재한다.

그림 2-1에서 볼 수 있듯이 카프카 스트림즈는 카프카 생태계의 흥미로운 계층, 즉 많은 소스들로부터 오는 데이터가 모이는 장소에서 작동한다:. 이는 복잡한 데이터 보강, 변환 및 처리가 발생할 수 있는 계층이다. 이 계층은 카프카 이전 세상에서 컨슈머/프로듀서 API를 사용하여 스스로 장황하게 스트림 처리 추상화를 작성했거나 다른 프레임워크를 사용하여 복잡성을 흡수했던 동일한 장소이다. 이제, 이 계층에서 재미있고 효율적으로 작동할 수 있도록 하는 카프카 스트림즈의 특징을 먼저 살펴보자.

기능 개요

카프카 스트림즈는 현대 스트림 처리 애플리케이션에 대해 탁월한 선택이 되도록 하는 많은 기능을 제공한다. 다음 장에서 이를 세부적으로 살펴볼 것인데, 다음은 기대할 수 있는 일부 기능이다:

- 자바 스트리밍 API와 같은 모습과 느낌이 나는 고수준 DSL (Domain Specific Language). DSL은 데이터 스트림 처리에 대해 배우기 쉽고 사용하기 쉬운 능숙하고 기능적인 접근방법을 제공한다.
- 개발자가 필요한 경우 미세하게 제어할 수 있도록 하는 저수준 프로세서 API.
- 데이터를 스트림 또는 테이블로 모델링하기 위한 편리한 추상화.
- 데이터 변환과 보강에 유용한 스트림과 테이블의 조인

- 스테이트리스(Stateless, 무상태) 및 스테이트풀(Stateful, 상태유지) 스트림 처리 애플리케이션 구축을 위한 연산자와 유틸리티
- 윈도우 및 주기적 기능을 포함하여 시간 기반 작업 지원
- 손쉬운 설치. 카프카 스트림즈는 단순히 라이브러리로 어떠한 자바 애플리케이션에도 카프카 스트림즈를 추가할 수 있다.[7]
- 확장성, 신뢰성, 유지보수성

이 책에서 이러한 기능들을 탐구함에 따라 여러분은 이 라이브러리가 왜 널리 사용되고 사랑받고 있는지를 빠르게 알아챌 것이다. 고수준 DSL과 저수준 프로세서 API 모두 배우기 쉬울 뿐만 아니라 기능적으로 매우 강력하다. (라이브로 이동 중인 데이터 스트림 조인과 같이) 첨단 스트림 처리 태스크를 약간의 코드를 통해 수행할 수 있는데, 이는 개발하는 것이 매우 재미있다고 느끼도록 한다.

이제 마지막 부문은 스트림 처리 애플리케이션의 장기간 안정성과 관련이 있다. 결국 초기에 많은 기술들을 배우는 것은 흥미롭지만 실제로 중요한 것은 이 라이브러리를 실세계에서 장기간 사용함으로써 관계가 점점 복잡해짐에 따라 카프카 스트림즈가 계속해서 좋은 선택으로 남을지 여부이다. 따라서 더 깊숙이 들어가기 전에 카프카 스트림즈의 장기 생존 능력을 평가하는 것은 의미가 있다. 그렇다면 이를 어떻게 하면 될까? 카프카 스트림즈의 운영 특성을 살펴보는 것으로 시작해보자.

운영 특성

Martin Kleppmann의 훌륭한 책인 Designing Data-Intensive Applications (O'Reilly)에서 저자는 데이터 시스템에 대해 3 가지 중요한 목표를 강조한다:

- 확장성
- 신뢰성
- 유지보수성

확장성

부하가 증가하더라도 시스템이 대처할 수 있고 효율적으로 유지될 수 있을 때 시스템은 확장가능한 것으로 고려된다. 1 장에서 카프카 토픽을 확장하는 것은 더 많은 파티션을 추가하고 필요 시 더 많은 카프카 브로커를 추가하는 것으로 배웠다 (후자의 경우 토픽이 카프카 클러스터의 기존 용량을 넘어서는 경우에만 필요하다).

---

[7] 카프카 스트림즈는 Scala 및 Kotlin을 포함하여 다른 JVM-기반 언어와도 잘 작동할 것이다. 이 책에서는 Java만을 사용한다.

비슷하게 카프카 스트림즈에서 작업 단위는 단일 토픽-파티션으로 카프카는 컨슈머 그룹[8]이라는 협업하는 컨슈머들의 그룹에 작업을 자동으로 분배한다. 이는 2 가지 중요한 의미를 갖는다.

- 카프카 스트림즈에서 작업 단위가 단일 토픽-파티션이고 더 많은 파티션을 추가함으로써 토픽이 확장될 수 있기 때문에 카프카 스트림즈 애플리케이션이 수행할 수 있는 작업의 양은 소스 토픽에 대해 파티션 수를 증가시킴으로써 확장될 수 있다[9].
- 컨슈머 그룹을 활용함으로써 카프카 스트림즈 애플리케이션이 다루는 총 작업양은 복수의 협업하는 애플리케이션 인스턴스들에 분배될 수 있다.

두 번째 요점에 대한 간략한 주석: 카프카 스트림즈 애플리케이션을 배치할 때 여러분은 거의 항상 각각 작업의 일부를 담당할 복수의 애플리케이션 인스턴스들을 배치할 것이다 (예, 소스 토픽이 32 개의 파티션을 갖고 있다면 32 개의 작업 단위가 있고 이들이 모든 협업 컨슈머에 분배될 수 있다). 예를 들어 각자 8 개의 파티션 (4*8=32)을 다루는 4 개의 애플리케이션 인스턴스를 배치하거나 각자 2 개의 파티션 (16*2=32)을 다루는 16 개의 애플리케이션 인스턴스를 배치할 수 있다.

그러나 얼마나 많은 애플리케이션 인스턴스가 배치되는 지와 상관없이 더 많은 파티션 추가 (작업 단위) 및 더 많은 애플리케이션 추가 (워커)를 통해 증가되는 부하에 대처할 수 있는 카프카 스트림즈의 능력은 카프카 스트림즈를 확장가능성 있도록 만든다.

비슷한 주석으로 카프카 스트림즈는 또한 탄성적이라고 하는데 이는 토폴로지를 위해 생성된 태스크의 수가 스케일아웃 경로에 대한 제한이지만 애플리케이션 인스턴스의 수를 (수동이지만) 유연하게 늘리거나 줄일 수 있도록 한다. "태스크 및 스트림 쓰레드"에서 태스크를 보다 상세하게 논의할 것이다.

신뢰성

신뢰성은 엔지니어링 측면(시스템 내 어떤 고장으로 인해 새벽 3시에 일어나고 싶지 않음) 뿐만 아니라 고객 측면 (우리는 어떤 눈에 띄는 방식으로든 시스템이 오프라인이 되는 것을 원치 않으며 데이터 분실 또는 손상을 견딜 수 없음)에서 데이터 시스템의 중요한 특징이다. 카프카 스트림즈는 몇몇 내고장성 특징[10]이 있으며 가장 명백한 것은 컨슈머 그룹에서 이미 다룬 것이다: 컨슈머 그룹을 통한 자동 장애조치와 파티션 리밸런싱.

여러분이 복수의 카프카 스트림즈 애플리케이션 인스턴스를 배치하였고 하나가 시스템 내 특정 고장 (예, 하드웨어 고장)으로 인해 오프라인이 된다면 카프카는 부하를 다른 정상 인스턴스들에 자동적으

---

[8] 여러 컨슈머 그룹이 단일 토픽으로부터 소비할 수 있는데 각각의 컨슈머 그룹은 다른 컨슈머 그룹에 상관없이 메시지를 처리한다.

[9] 파티션이 기존 토픽에 추가될 수 있지만 권고되는 패턴은 원하는 파티션 수를 갖는 새로운 소스 토픽을 생성하고 기존의 모든 워크로드를 새로운 토픽으로 이동시는 것이다.

[10] 4 장에서 논의할 스테이트풀 애플리케이션에 특정적인 일부 특징을 포함

로 재분배할 것이다. 문제가 해결되거나 (또는 쿠버네티스와 같이 오케스트레이션 시스템을 활용하는 보다 최신 아키텍처에서 애플리케이션이 정상 모드로 돌아갈 때) 카프카는 작업을 다시 재조정할 것이다. 이렇게 고장을 우아하게 다루는 능력은 카프카 스트림즈를 신뢰성 있도록 한다.

유지보수성

소프트웨어의 대부분 비용은 초기 개발에 소요되는 것이 아니라 버그 수정, 시스템 운영 유지, 고장 조사 등의 지속적인 유지보수에 소요된다는 것은 잘 알려진 사실이다. – Martin Kleppmann

카프카 스트림즈는 Java 라이브러리로 독립 실행형 애플리케이션으로 작업하기 때문에 버그 문제 해결 및 수정은 상대적으로 수월하며, Java 애플리케이션 문제해결 및 모니터링 모두에 대한 패턴은 잘 확립되어 있고 이미 여러분의 조직 내에서 사용 중일 수 있다 (애플리케이션 로그 수집 및 분석, 애플리케이션 및 JVM 지표 (Metrics) 수집, 프로파일링 및 트레이싱 등).

더구나 카프카 스트림즈 API는 명료하고 직관적이기 때문에 코드 수준의 유지보수는 보다 복잡한 라이브러리에 대해 예상되는 것보다 시간이 덜 소요되고 초보자와 전문가 모두 이해하기 쉽다. 여러분이 카프카 스트림즈 애플리케이션을 구축한 후 몇 달간 손을 대지 않은 경우라도 일상적인 프로젝트 기억상실로 고생할 것 같지 않으며 작성했던 이전 코드를 이해하기 위해 많은 시간이 필요할 것 같지 않다. 동일한 이유로 새로운 프로젝트 유지보수자는 카프카 스트림즈 애플리케이션에 대해 꽤 빨리 따라잡을 수 있으며 더욱더 유지보수성을 향상시킨다.

다른 시스템과의 비교

이제부터 여러분은 카프카 스트림즈를 장기 관점에서 사용하기로 시작한 것에 대해 편안함을 느끼기 시작해야 한다. 그러나 상황이 너무 심각해지기 전에 시장에 다른 시스템이 있는지 살펴보자.

실제로 기술이 경쟁 기술에 비해 어떻게 구성되어 있는지 모르면서 얼마나 좋은지를 평가하는 것은 어렵다. 따라서 스트림 처리 분야[11]에서 카프카 스트림즈가 다른 인기있는 기술과 어떻게 비교되는지를 살펴보자.

배치 모델

카프카 스트림즈는 스트림 처리에 대해 아파치 Flink 및 아파치 Spark Streaming과 같은 기술과는 다른 접근방법을 취한다. 후자의 시스템들은 스트림 처리 프로그램을 제출 및 작동시키기 위해 전용 처리 클러스터를 설정해야 한다. 이는 많은 복잡도와 간접비를 도입할 수 있다. 잘 확립된 회사의 경험이 풍부한 엔지니어도 처리 클러스터의 간접비가 사소하지 않다고 인정하였다. 넷플릭스의 Nitin Sharma와의 인터뷰에서 아파치 Flink의 뉘앙스에 적응하여 매우 신뢰성있는 운영 환경의 아파치 Flink 애플리

[11] 스트림 처리 분야가 끊임없이 성장하는 특성으로 인해 모든 솔루션을 카프카 스트림즈와 비교하는 것은 어려우며 따라서 이 책 작성 시점에 사용가능한 가장 인기있고 성숙한 스트림 처리 솔루션에 중점을 두었다.

케이션 및 클러스터를 구축하는데 대략 6 개월이 걸렸음을 알았다.

반면 카프카 스트림즈는 Java 라이브러리로 구현되어 있으며 따라서 클러스터 관리자에 대해 걱정할 필요가 없기 때문에 시작하기가 매우 수월하다; 단순히 Java 애플리케이션에 카프카 스트림즈 의존성을 추가하면 된다. 스트림 처리 애플리케이션을 독립실행형 프로그램으로 구축할 수 있다는 것은 여러분이 코드를 모니터링, 패키징 및 배포하는 방법 측면에서 많은 자유를 가질 수 있음을 의미한다. 예를 들어 Mailchimp에서 카프카 스트림즈 애플리케이션은 다른 내부 Java 애플리케이션에 대해 사용한 동일한 패턴 및 tooling을 사용하여 배포된다. 이렇게 여러분의 회사 시스템애 즉각적으로 통합될 수 있는 능력은 카프카 스트림즈의 큰 장점이다.

다음에는 카프카 스트림즈의 처리 모델이 이 분야의 다른 시스템과 어떻게 비교되는지를 살펴보자.

처리 모델

카프카 스트림즈와 아파치 Flink 또는 Trident와 같은 시스템들 간 다른 주요 차이는 카프카 스트림즈가 한번에 한 이벤트 (event-at-a-time) 처리를 구현하고 있다는 것으로, 따라서 이벤트는 들어오는 경우 한번에 하나씩 즉각적으로 처리된다. 이는 실제 스트리밍으로 간주되며 마이크로 배칭이라는 다른 접근방법보다 낮은 지연을 제공한다. 마이크로 배칭은 레코드를 작은 그룹 (또는 배치)으로 그룹화하는 것을 포함하며 이는 메모리에 버퍼링되어 추후 특정 간격 (예, 매 500 밀리초)으로 방출된다. 그림 2-2는 한 번에 한 이벤트와 마이크로 배치 처리 간 차이를 보여준다.

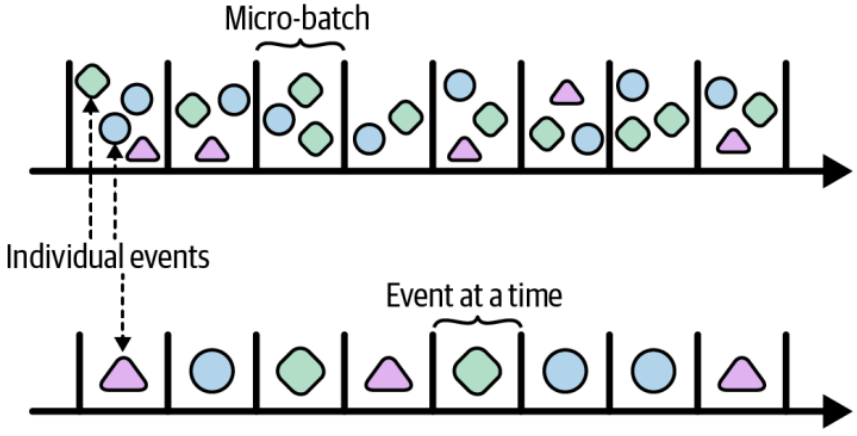


그림 2-2. 레코드의 소규모 배치로의 그룹화 및 특정 간격으로 다운스트림 처리기에 배출하는 마이크로 배칭; 한 번에 한 이벤트 처리는 배치가 만들어지길 기다리는 대신 각 이벤트가 들어오자마자 처리될 수 있도록 한다.

> 마이크로 배칭을 사용하는 프레임워크는 보다 긴 지연을 희생하면서 보다 많은 처리를 하도록 최적화되어 있다. 여러분은 카프카 스트림즈로 데이터를 많은 파티션으로 분할함으로써 높은 처리량을 유지하면서 매우 낮은 지연을 얻을 수 있다.

마지막으로 카프카 스트림즈의 데이터 처리 아키텍처를 살펴보고 스트리밍에 대한 초점이 다른 시스템과 어떻게 다른 지를 알아보자.

카파(Kappa) 아키텍처

카프카 스트림즈를 다른 솔루션과 비교할 때 다른 중요한 고려사항은 여러분의 유스케이스가 배치와 스트림 처리 모두를 지원하는지 여부이다. 이 책 작성 시점에서 카프카 스트림즈는 스트리밍 유스케이스[12] (카파 아키텍처라고 불린다)에만 전적으로 중점을 두고 있으며 반면 아파치 Flink와 아파치 Spark와 같은 프레임워크는 배치와 스트림 처리 모두를 지원한다 (이는 람다 아키텍처로 불린다). 그러나 배치와 스트리밍 유스케이스 모두를 지원하는 아키텍처가 단점이 없는 것은 아니다. Jay Kreps는 카프카 스트림즈가 카프카 생태계에 도입되기 2년전에 하이브리드 시스템의 일부 단점을 논의하였다:

> 2 가지 시스템을 실행 및 디버깅하는 운영 부담은 매우 높아지고 있다. 그리고 어떠한 새로운 추상화도 이러한 2 가지 시스템의 교집합에 의해 지원하는 특징만을 제공한다.

이러한 문제들이 배치와 스트림 처리에 대해 통합된 프로그래밍 모델을 정의한 아파치 Beam과 같은 프로젝트가 최근 인기를 얻는 것을 막지는 못했다. 그러나 아파치 Beam은 아파치 Flink와 동일한 방식으로 카프카 스트림즈와 비교할 수 없다. 예를 들어 아파치 Flink와 아파치 Spark 모두 아파치 Beam의 실행 엔진 (종종 러너로 불림)으로 사용될 수 있다. 따라서 카프카 스트림즈를 아파치 Beam과 비교할 때 Beam API 자체 외에 사용할 계획인 실행 엔진을 고려해야 한다.

더구나 아파치 Beam 기반 파이프라인은 카프카 스트림즈에서 제공되는 일부 중요한 특징이 부족하다. 실험적인 카프카 스트림즈 Beam Runner를 만들었고 카프카 생태계의 몇몇 혁신적인 프로젝트를 유지하고 있는[13] Robert Yokota는 다른 스트리밍 프레임워크의 비교에서 다음과 같이 말했다:

두 시스템 간 차이를 말하는 한 가지 방법은 다음과 같다:

- 카프카 스트림즈는 스트림-관계형 처리 플랫폼이다.
- 아파치 Beam은 스트림-only 처리 플랫폼이다.

스트림-관계형 처리 플랫폼은 일반적으로 스트림-only 처리 플랫폼에 부족한 다음과 같은 능력을 갖고 있다:

- 관계 (또는 테이블)이 일류 시민이다. 즉 각각이 독립된 identity를 갖고 있다.
- 관계는 다른 관계로 변환될 수 있다.
- 관계는 임시적으로 쿼리될 수 있다.

우리는 다음의 여러 장에서 기술된 특징 각각을 보여줄 것이며, 지금은 카프카 스트림즈의 가장 강력

---

[12] 오래되었지만 카프카 스트림즈에서 배치 처리를 지원하기 위한 공개된 제안이 있었다.

[13] KarelDB라는 카프카 지원 관계형 DB, 카프카 스트림즈 기반 그래프 분석 라이브러리 등. https://yokota.blog 참조.

한 많은 특징들을 아파치 Beam 또는 다른 보다 일반화된 프레임워크[14]에서는 사용할 수 없다고 말하는 것으로 충분한 듯하다. 더구나 카파 아키텍처는 데이터 스트림으로 작업할 때 보다 단순하고 보다 전문화된 접근방법을 제공하며 이는 개발 경험을 개선하고 소프트웨어의 운영 및 유지보수를 단순화할 수 있다. 따라서 유스케이스에 배치 처리가 필요치 않은 경우 하이브리드 시스템은 불필요한 복잡도를 도입할 것이다.

이제 경쟁 기술에 대해 살펴보았으미 카프카 스트림즈의 유스케이스를 살펴보자.

유스케이스

카프카 스트림즈는 무제한의 데이터셋을 빠르게 효율적으로 처리하기 위해 최적화되어 있으며 따라서 저지연이라는 시간이 중요한 분야의 문제를 해결하기 위한 훌륭한 솔루션이다. 일부 유스케이스는 다음을 포함한다:

- 재무 데이터 처리 (Flipkart), 구매 모니터링, 사기 탐지
- 알고리즘 트레이딩
- 주식 시장/암호화폐 거래 모니터링
- 실시간 재고 추적 및 채움 (Walmart)
- 이벤트 예약, 좌석 선택 (Ticketmaster)
- 이메일 전달 추적 및 모니터링 (Mailchimp)
- 비디오 게임 원격 처리 (Activision, Call of Duty 발행자)
- 검색 인덱싱 (Yelp)
- 지리공간 추적/계산 (예, 거리 비교, 도착 예상)
- 스마트 홈/IoT 센서 처리 (때때로 AIOT 또는 Artificial Intelligence of Things로 불림)
- 변경 데이터 획득 (Change Data Capture) (Redhat)
- 스포츠 브로드캐스팅/실시간 위젯 (Gracenote)

---

[14] 이 책 작성 시점에서 아파치 Flink는 API 자체가 덜 성숙한 상태이고 공식적인 Flink 문서에 다음과 같은 경고가 있음에도 불구하고 최근 쿼리가능한 상태의 베타 버전을 출시하였다: "쿼리가능한 상태에 대한 클라이언트 API는 현재 진화 중이며 제공된 인터페이스의 안정성에 대해 보장을 하지 못한다. 추후 Flink 버전에서는 클라이언트 측면에서 급격한 API 변경이 있을 수도 있다". 따라서 아파치 Flink 팀이 이 간극을 좁히기 위해 작업하고 있지만 카프카 스트림즈는 상태 쿼리에 대해 더욱 성숙하고 운영 환경 수준의 API를 갖고 있다.

- 실시간 광고 플랫폼 (Pinterest)
- 예측 헬스케어, 활력 모니터링 (Children's Healthcare of Atlanta)
- 챗 인프라 (Slack), 챗봇, 가상 비서
- 머신러닝 파이프라인 (Twitter) 및 플랫폼 (Kafka Graphs)

유스케이스는 계속 늘어나고 있는데, 이 모든 사례의 공통적인 특성은 실시간 의사결정 또는 데이터 처리를 필요로 하거나 최소한 이로부터 혜택을 얻고 있다는 것이다. 이러한 유스케이스의 스펙트럼과 여러분이 실생활에서 만날 다른 사례는 실제 꽤 매력적이다. 스펙트럼의 한 끝에서 여러분은 스마트 홈 디바이스로부터의 센서 출력을 분석함으로써 취미 수준에서 스트림을 처리할 수 있다. 그러나 Childrent's Healthcare of Atlanta에서 했듯이 트라우마 희생자의 상태를 모니터링하고 변경에 대응하기 위해 헬스케어 환경에서 카프라 스트림을 사용할 수도 있다.

카프카 스트림즈는 실시간 이벤트 스트림 기반으로 마이크로 서비스를 구축하는데도 좋은 선택이다. 카프카 스트림즈는 일반적인 스트림 처리 작업 (데이터 필터링, 조인, 윈도우 및 변환)을 단순화할 뿐만 아니라 "상호대화형 쿼리"에서 볼 수 있듯이 상호대화형 쿼리라는 기능을 사용하여 스트림 상태를 노출시킬 수도 있다. 스트림 상태는 일종의 집계 (예, 스트리밍 플랫폼에서 각 비디오의 총 시청 수) 또는 이벤트 스트림에서 빠르게 변하는 엔티티의 최신 표현 (예, 주어진 주식 심볼의 최신 주가)일 수 있다.

카프가 스트림을 누가 사용하고 있고 어떤 유스케이스에 적합한지를 알았는데, 코드 작성을 시작하기 전에 카프카 스트림즈의 아키텍처를 빠르게 살펴보자.

프로세서 토폴로지

카프카 스트림즈는 프로그림을 일련의 입력, 출력 및 처리 단계로 표현하는 데이터 중심 방법인 데이터 플로우 프로그래밍 (Data Flow Programming, DFP)이라는 프로그래밍 패러다임을 활용한다. 이는 스트림 처리 프로그램을 작성하는데 있어 매우 자연스럽고 직관적인 방식으로 이끌며 초보자가 카프카 스트림즈를 선택하는 많은 이유 중 하나이다.

> 이 절에서는 카프카 스트림즈 아키텍처를 약간 깊게 들어갈 것이다. 여러분이 카프카 스트림즈를 경험하고 추후 이 절을 다시 보고 싶다면 "튜토리얼 소개: 헬로우, 스트림" 부분으로 넘어가도 좋다.

프로그램을 일련의 단계로 구축하는 대신 카프카 스트림즈 애플리케이션에서 스트림 처리 로직은 방향성 비순환 그래프 (Directed Acyclic Graph, DAG)로 구조화된다. 그림 2-3은 데이터가 일련의 스트림 프로세서를 통해 어떻게 흐르는지를 나타내는 DAG 예를 보여준다. 노드 (그림의 사각형)는 처리 단계 또는 프로세서를 나타내며 엣지(그림에서 노드를 연결하는 선)는 (데이터가 한 프로세서에서 다른 프로세서 흐르는) 입력 및 출력 스트림을 나타낸다.

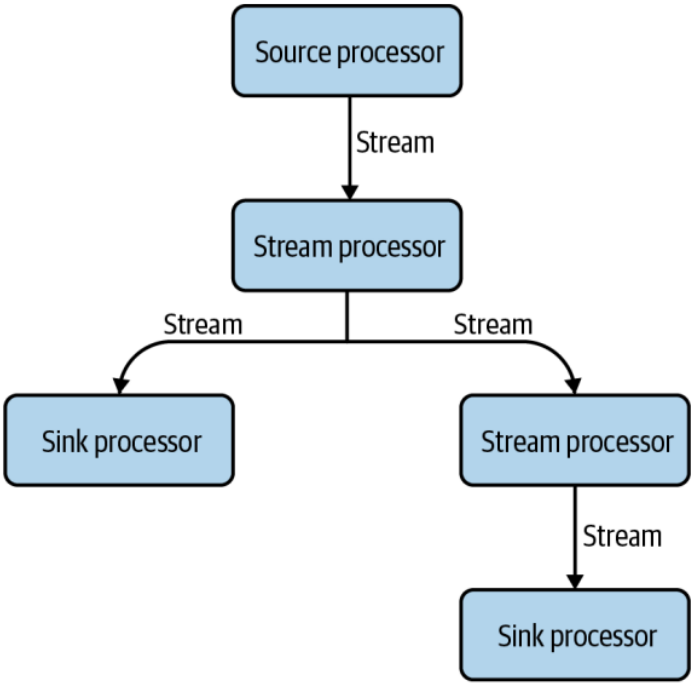


그림 2-3. 카프카 스트림즈는 데이터 플로우 프로그래밍에서 일부 설계를 차용했으며 스트림 처리 프로그램을 데이터가 흐르는 프로세서 그래프로 구조화한다.

카프카 스트림즈에는 3 가지 프로세서가 존재한다:

소스 프로세서

소스는 카프카 스트림즈 애플리케이션 내로 정보가 흘러들어가는 곳이다. 데이터는 카프카 토픽으로부터 읽혀지고 하나 이상의 스트림 프로세서로 보내진다.

스트림 프로세서

이 프로세서는 입력 스트림에 데이터 처리/변환 로직 적용을 담당한다. 고수준 DSL에서 이 프로세서는 카프카 스트림즈 라이브러리에 의해 노출되는 일련의 내장 연산자를 사용하여 정의되는데, 다음 장에서 세부적으로 논의할 것이다. 일부 연산자 예는 filter, map, flatMap과 join이다.

싱크 프로세서

싱크는 보강, 변환, 필터링 또는 다른 처리된 레코드가 카프카로 다시 쓰여지는 곳으로 다른 스트림 처리 애플리케이션에 의해 처리되거나 카프카 커넥트와 같은 것을 통해 다운스트림 데이터 저장소로 보내진다. 소스 프로세서와 같이 싱크 프로세서도 카프카 토픽에 연결된다.

프로세서 모음은 프로세서 토폴로지를 구성하는데 이 책과 카프카 스트림즈 커뮤니티에서 토폴로지로 간주된다. 이 책에서 각각의 튜토리얼을 진행할 때 우선 소스, 스트림 및 싱크 프로세서를 연결하는 DAG를 생성함으로써 토폴로지를 설계할 것이다. 그 후 Java 코드를 작성함으로써 토폴로지를 구현할 것이다. 이를 보여주기 위해 일부 프로젝트 요구사항을 (DAG로 표현되는) 프로세서 토폴로지로 변환

하는 예제를 진행해보자. 이는 카프카 스트림즈 개발자와 같이 생각하는 법을 배울 수 있도록 한다.

시나리오

카프카 스트림즈로 챗봇을 만들고 있고 봇인 @StreamsBot을 언급한 모든 Slack 메시지를 포함하는 slack-mentions이 토픽의 이름이라고 해보자. 각각의 멘션 다음에 @StreamsBot restart myservice와 같은 명령어가 나오도록 봇을 설계할 것이다.

우리는 이러한 Slack 메시지를 전처리/확인하는 기본적인 프로세서 토폴로지를 구현하려고 한다. 우선적으로 소스 토픽에서 각 메시지를 소비하고 명령어가 유효한지를 확인하며, 유효한 경우 Slack 메시지를 valid-mentions이라는 토픽에 쓸 필요가 있다. 명령어가 유효하지 않은 경우 (예, @StreamsBot restart serverrr와 같이 봇을 언급할 때 스펠링 에러) invalid-mentions이라는 토픽에 쓸 것이다.

이 경우 그림 2-4에 보여지는 토폴로지로 이러한 요구사항을 변환할 수 있다.

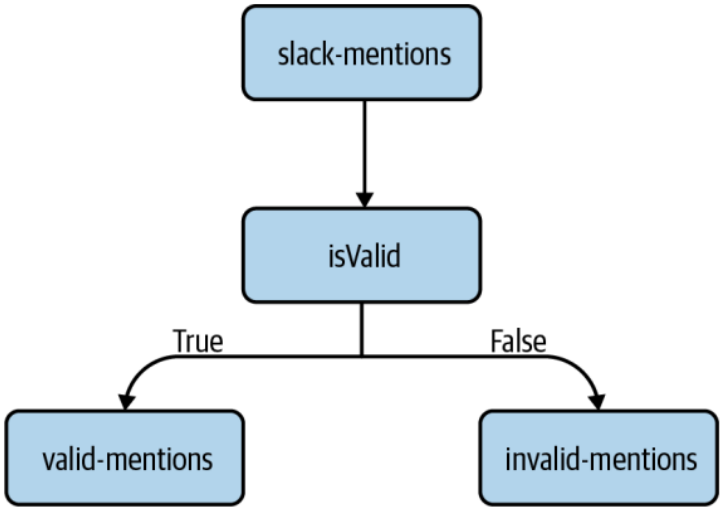


그림 2-4. 카프카 (slack-mentions)로부터 Slack 메시지를 읽기 위한 단일 소스 프로세서, 각 메시지의 유효성을 검사하는 단일 스트림 프로세서 (isValid)와 이전 검사에 기반하여 메시지를 2 가지 출력 토픽 중 하나로 전송하는 2 개의 싱크 프로세서 (valid-mentions, invalid-mentions)를 포함하는 예제 프로세서 토폴로지

다음 장에서 튜토리얼을 시작할 때 카프카 스트림즈 API를 사용하여 설계한 토폴로지를 구현하기 시작할 것이다. 그러나 우선 관련 개념을 살펴보자: 서브-토폴로지

서브-토폴로지

카프카 스트림즈는 서브-토폴로지의 개념 또한 갖고 있다. 이전 예에서 단일 소스 토픽 (slack-mentions)으로부터 이벤트를 소비하고 원시 챗 메시지 스트림에 대해 전처리를 수행하는 프로세서 토폴로지를 설계했다. 그러나 우리 애플리케이션이 복수의 소스 토픽으로부터 소비하는 경우 카프카 스

트림즈는 (대다수 상황에서[15]) 작업을 더욱 병렬화하기 위해 토폴로지를 보다 작은 서브-토폴로지로 나눌 것이다. 이는 하나의 입력 스트림에 대한 연산이 다른 입력 스트림에 대한 연산과 독립적으로 수행될 수 있기 때문에 가능하다.

예를 들어 2 개의 새로운 스트림 프로세서를 추가하여 챗봇을 계속 만든다고 해보자: valid-mentions로부터 소비하여 StreamsBot에 발행된 모든 명령(예, restart server)을 수행하는 프로세서와 invalid-mentions으로부터 소비하여 에러 응답을 Slack으로 전송하는 다른 프로세서[16].

그림 2-5에서 볼 수 있듯이 토폴로지는 slack-mentions, valid-mentions과 invalid-mentions의 3 개 토픽으로부터 데이터를 소비한다. 우리가 새로운 소스 토픽으로부터 읽어 들일때마다 카프카 스트림즈는 토폴로지를 독립적으로 실행할 수 있는 보다 작은 부분으로 나눈다. 이 예에서 챗봇 애플리케이션을 위해 3 개의 서브-토폴로지가 존재하며 각각을 별 모양으로 표시하였다.

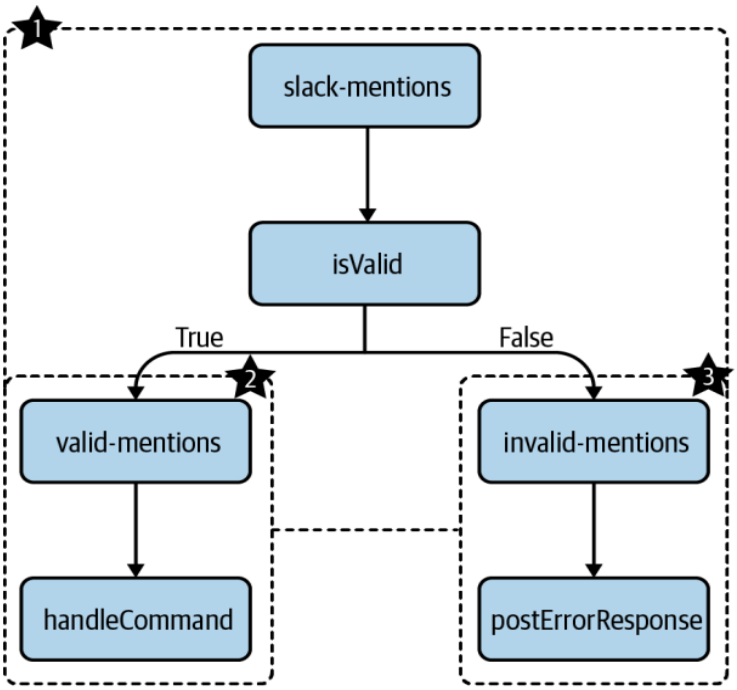


그림 2-5. (점선으로 구분된) 서브 토폴로지로 나누어진 프로세서 토폴로지

Valid-mentions과 invalid-mentions 토픽 모두 첫번째 서브-토폴로지에서는 싱크 프로세서 역할을 하

---

[15] 예외는 토픽이 조인될 때로, 이 경우 서브-토폴로지로 나누지 않고 단일 토폴로지가 조인에 포함된 각각의 소스 토픽으로부터 읽어 들일 것이다. 이는 조인이 작동하기 위해 필요하다. 보다 자세한 정보는 "Co-Partitioning"을 참조하기 바란다.

[16] 이 예에서 2 개의 중간 토픽 (valid-mentions과 invalid-mentions)에 쓴 후 각 토픽에서 데이터를 즉시 소비한다. 이와 같이 중간 토픽을 사용하는 것은 특정 작업 (예를 들어 데이터 리파티셔닝)에 대해서만 필요하다. 여기서는 논의 목적으로만 사용한다.

지만 두번째 및 세번째 서브 토폴로지에서는 소스 프로세서 역할을 함을 주목하기 바란다. 이런 경우 서브-토폴로지 간 직접적인 데이터 교환은 없다. 레코드는 싱크 프로세서에서 카프카로 생산되고 소스 프로세서에 의해 카프카로 다시 읽혀진다.

이제 스트림 처리 프로그램을 프로세서 토폴로지로 표현하는 방법을 이해했는데 카프카 스트림즈 애플리케이션에서 데이터가 실제 상호 연결된 프로세서를 통해 어떻게 흐르는지를 살펴보자.

깊이 우선 (Depth-First) 처리

카프카 스트림즈는 데이터를 처리할 때 깊이 우선 전략을 사용한다. 새로운 레코드가 수신될 때 이는 다른 레코드가 처리되기 전에 토폴로지 내 각각의 스트림 프로세서를 통해 전달된다. 카프카 스트림즈를 통한 데이터 흐름은 그림 2-6에 보여지는 것과 같다.

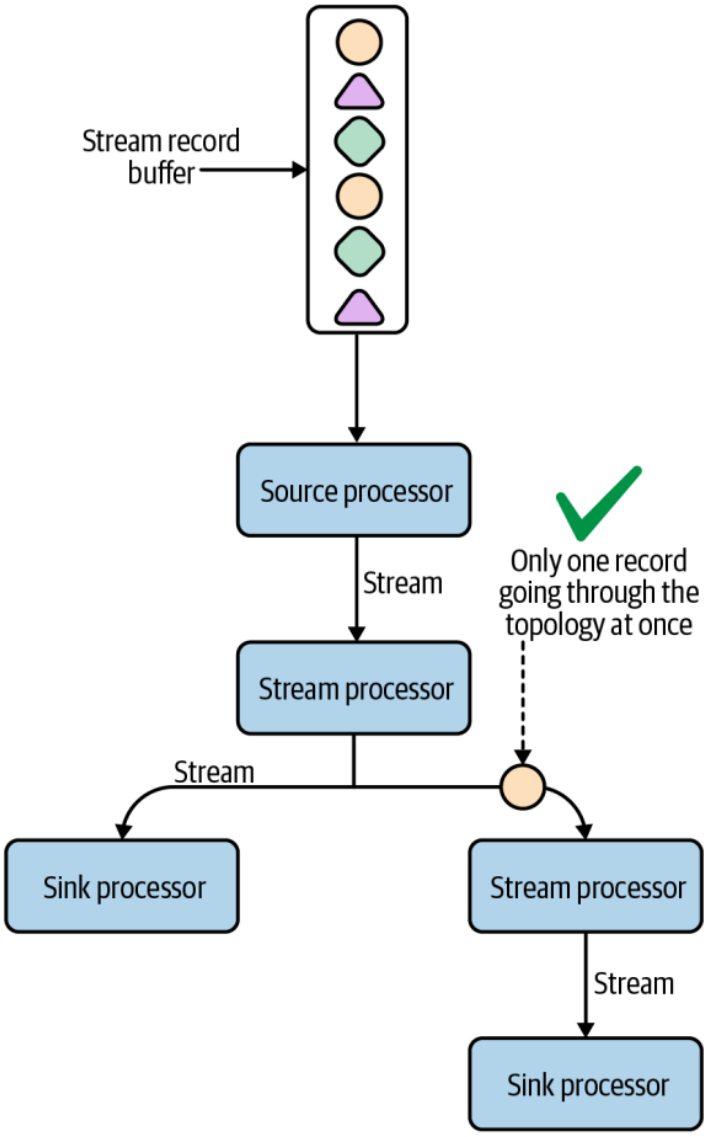


그림 2-6. 깊이 우선 처리에서 다른 레코드가 처리되기 전에 하나의 레코드가 전체 토폴로지를 통해 이동한다.

이 깊이 우선 전략은 데이터 흐름을 보다 쉽게 추론할 수 있게 하며 느린 스트림 처리 작업이 동일 쓰레드에서 다른 레코드가 처리되는 것을 막을 수 있음을 의미한다. 그림 2-7은 카프카 스트림즈에서 절대 일어날 수 없는 것을 보여준다: 한 번에 토폴로지를 통과하는 복수의 레코드.

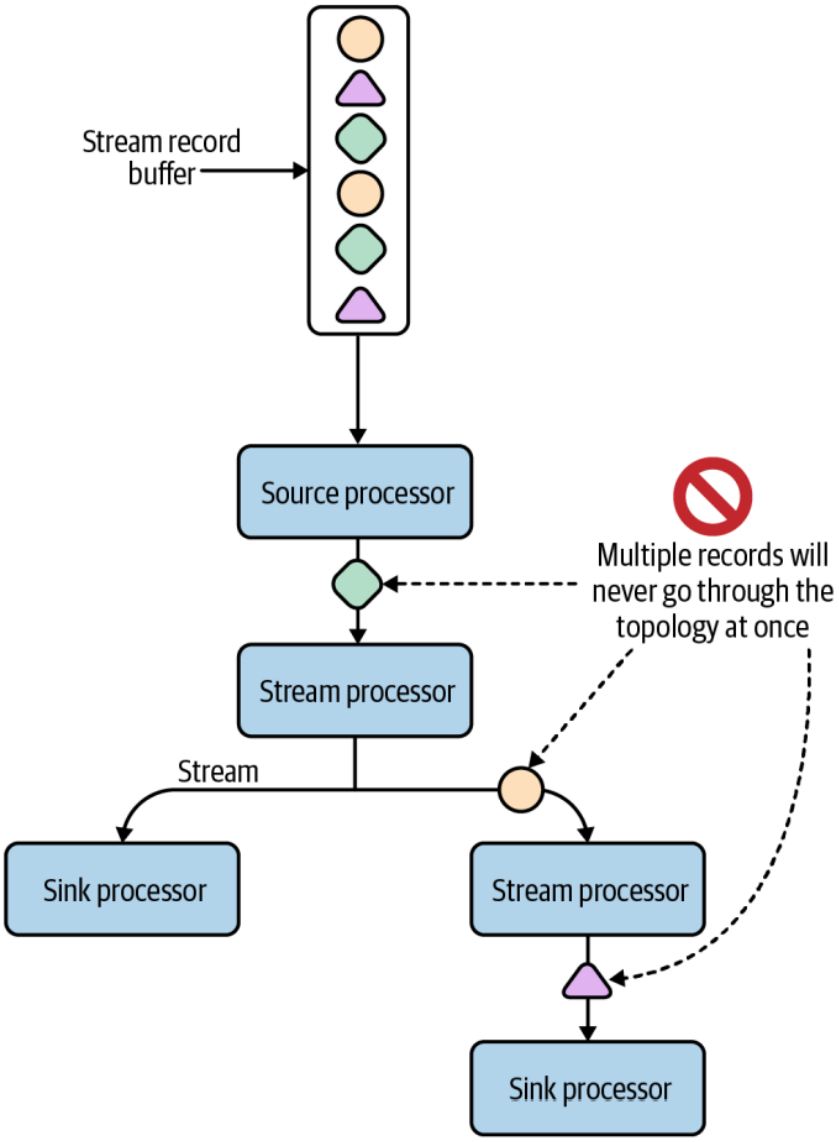


그림 2-7. 복수의 레코드가 동시에 토폴로지를 절대 통과하지 못할 것이다.

> 복수의 서브 토폴로지가 존재할 때 단일 이벤트 규칙은 전체 토폴로지가 아닌 각각의 서브-토폴로지에 적용된다.

이제 프로세서 토폴로지를 어떻게 설계하고 데이터가 토폴로지를 어떻게 통과하는 지를 알았는데 스트림 처리 애플리케이션 구축에 대한 데이터 중심 접근 방법의 장점을 살펴보자.

데이터 플로우 프로그래밍의 장점

스트림 처리 애플리케이션 구축을 위해 카프카 스트림즈와 데이터 플로우 프로그래밍 모델을 사용하는 것은 여러 장점을 갖는다. 우선적으로 프로그램을 방향성 그래프로 표현하는 것은 추론을 수월하게

한다. 데이터가 애플리케이션을 통해 어떻게 흐르는 지를 이해하기 위해 많은 조건부와 제어 로직을 따를 필요가 없다. 데이터가 프로그램 어디로 들어가고 나오는 지를 결정하기 위해 단순히 소스 및 싱크 프로세서를 찾으면 되며 데이터가 도중에 어떻게 처리되고, 변환되며 보강되는지를 이해하기 위해 이들 사이의 스트림 프로세서를 보면 된다.

또한 스트림 처리 프로그램을 방향성 그래프로 표현함으로써 스트리밍 솔루션 구축 방식을 표준화할 수 있다. 내가 작성한 카프카 스트림즈 애플리케이션은 이전에 자신의 프로젝트에서 카프카 스트림즈를 사용했던 누구라도 라이브러리 자체에서 사용 가능한 재사용성 추상화로 인해 그리고 공통적인 문제 해결 방식 덕분에 어느 정도 익숙할 것이다.

방향성 그래프는 비기술적 이해관계자에 대해서도 데이터 흐름을 시각화 할 수 있는 직관적인 방식이다. 종종 엔지니어링 팀과 비엔지니어링 팀 간에 프로그램이 어떻게 작동하는 지에 대해 단절이 존재한다. 때때로 이로 인해 비기술적 팀이 소프트웨어를 폐쇄 박스로 간주하도록 한다. 이는 데이터 개인정보 보호 법과 GDPR (General Data Protection Regulation) 준수 시대에 위험한 부작용을 야기할 수 있으며 엔지니어, 법률 팀 및 다른 이해관계자 간 밀접한 조정이 필요하다. 따라서 애플리케이션 내에서 데이터가 어떻게 처리되는지를 간단히 의사소통 할 수 있다는 것은 비즈니스 문제의 다른 측면에 중점을 두고 있는 사람들에게 애플리케이션의 설계를 이해하거나 나아가서 이에 기여할 수 있도록 한다.

마지막으로 소스, 싱크 및 스트림 프로세서를 포함하는 프로세서 토폴로지는 복수의 쓰레드 및 애플리케이션 인스턴스를 통해 쉽게 인스턴스화 및 병렬화 될 수 있는 템플릿의 역할을 한다. 따라서 이러한 방식으로 데이터 흐름을 정의하는 것은 데이터 양 증가 시 스트림 처리 프로그램을 쉽게 복제할 수 있기 때문에 성능과 확장성 측면에서 혜택을 얻을 수 있도록 한다.

이제 이러한 토폴로지 복제 프로세스가 어떻게 동작하는 지를 이해하기 위해 태스크, 스트림 쓰레드 및 파티션 간 관계를 이해할 필요가 있다.

태스크 및 스트림 쓰레드

카프카 스트림즈에서 토폴로지를 정의할 때 우리는 실제 프로그램을 실행시키는 것은 아니다. 대신 데이터가 애플리케이션을 통해 어떻게 흘러야 하는 지에 대한 템플릿을 작성하는 것이다. 이 템플릿 (토폴로지)는 단일 애플리케이션 인스턴스로 여러 번 인스턴스화 될 수 있고 많은 태스크 및 스트림 쓰레드를 통해 병렬화 될 수 있다 (앞으로 단순히 쓰레드라고 할 것이다[17]). 태스크/쓰레드의 수와 스트림 처리 애플리케이션이 다룰 수 있는 작업량 간에는 밀접한 관계가 있으며 따라서 카프카 스트림즈를 통해 좋은 성능을 얻기 위해서는 이 절의 내용을 이해하는 것이 특히 중요하다.

---

[17] Java 애플리케이션은 많은 다양한 유형의 쓰레드를 실행시킬 수 있다. 현재는 프로세서 토폴로지 실행을 위해 카프카 스트림즈에 의해 생성 및 관리되는 스트림 쓰레드에 중점을 둔다.

태스크를 살펴보면서 시작하자:

> 태스크는 카프카 스트림즈 애플리케이션에서 병렬로 수행될 수 있는 가장 작은 작업 단위이다.
>
> 간단히 말해서 애플리케이션이 동작할 수 있는 최대 병렬 처리는 최대 스트림 태스크 수로 제한을 받으며 이는 애플리케이션이 읽어 들이는 입력 토픽의 최대 파티션 수에 의해 결정된다.
>
> - Andy Bryant

위 인용문을 공식으로 변환시킨다면 주어진 카프카 스트림즈 서브-토폴로지[18]에 대해 생성될 수 있는 태스크 수는 다음 공식으로 계산될 수 있다:

```
max(source_topic_1_partitions, ..., source_topic_n_partitions)
```

예를 들어 토폴로지가 16 개 파티션을 포함하는 하나의 소스 토픽으로부터 읽는 다면 카프카 스트림즈는 16 개 태스크를 생성할 것이고 각각은 자신의 프로세서 토폴로지 복제본을 인스턴스화 할 것이다. 카프카 스트림즈가 모든 태스크를 생성한다면 각 태스크로부터 읽어 들일 소스 파티션이 할당될 것이다.

여러분이 볼 수 있듯이 태스크는 프로세서 토폴로지를 인스턴스화 및 구동하는데 사용되는 논리 단위이다. 반면 쓰레드는 실제 태스크를 실행하는 실체이다. 카프카 스트림즈에서 스트림 쓰레드는 격리되고 쓰레드로부터 안전하도록 설계되어 있다[19]. 더구나 태스크와 달리 애플리케이션이 얼마나 많은 쓰레드를 실행해야 하는지를 알기 위해 카프카 스트림즈가 적용할 어떤 공식도 없다. 대신 구성 특성인 num.stream.threads를 사용하여 쓰레드 수를 지정해야 한다. 사용할 수 있는 쓰레드 수의 상한은 태스크 수이고 실행시켜야 할 스트림 쓰레드의 수를 결정하기 위해서는 다양한 전략이 존재한다[20].

이제 다른 수의 쓰레드를 갖는 2 가지 별도의 구성을 사용하여 태스크와 쓰레드가 어떻게 생성되는 지를 시각화함으로써 이러한 개념의 이해도를 높여보자. 각 예에서 카프카 스트림즈 애플리케이션은 4 개의 파티션을 포함하는 소스 토픽을 읽어 들인다 (그림 2-8에서 p1 – p4).

---

[18] 카프카 스트림즈 토폴로지가 복수의 서브-토폴로지로 구성될 수 있음을 기억하자. 따라서 전체 프로그램에 대한 태스크 수를 얻기 위해서는 모든 서브-토폴로지에 대한 태스크 수를 합해야 한다.

[19] 제대로 구현되지 않은 스트림 프로세서가 동시성 문제의 영향을 받지 않음을 의미하지는 않는다. 그러나 기본적으로 스트림 쓰레드는 어떤 상태도 공유하지 않는다.

[20] 예를 들어 애플리케이션이 액세스하는 코어 수는 실행시켜야 할 쓰레드 수에 대한 정보를 제공할 수 있다. 애플리케이션 인스턴스가 4 코어 머신에서 동작하고 토폴로지가 16 개의 태스크를 지원한다면 쓰레드 수를 4로 설정할 수 있으며 이는 각 코어에 대해 쓰레드를 제공할 것이다. 반면 16 개의 태스크 애플리케이션이 48 코어 머신에서 동작하고 있다면 16 쓰레드를 실행시킬 수 있다 (상한이 태스크 수로 16이기 때문에 48 개의 쓰레드를 실행시킬 수는 없다).

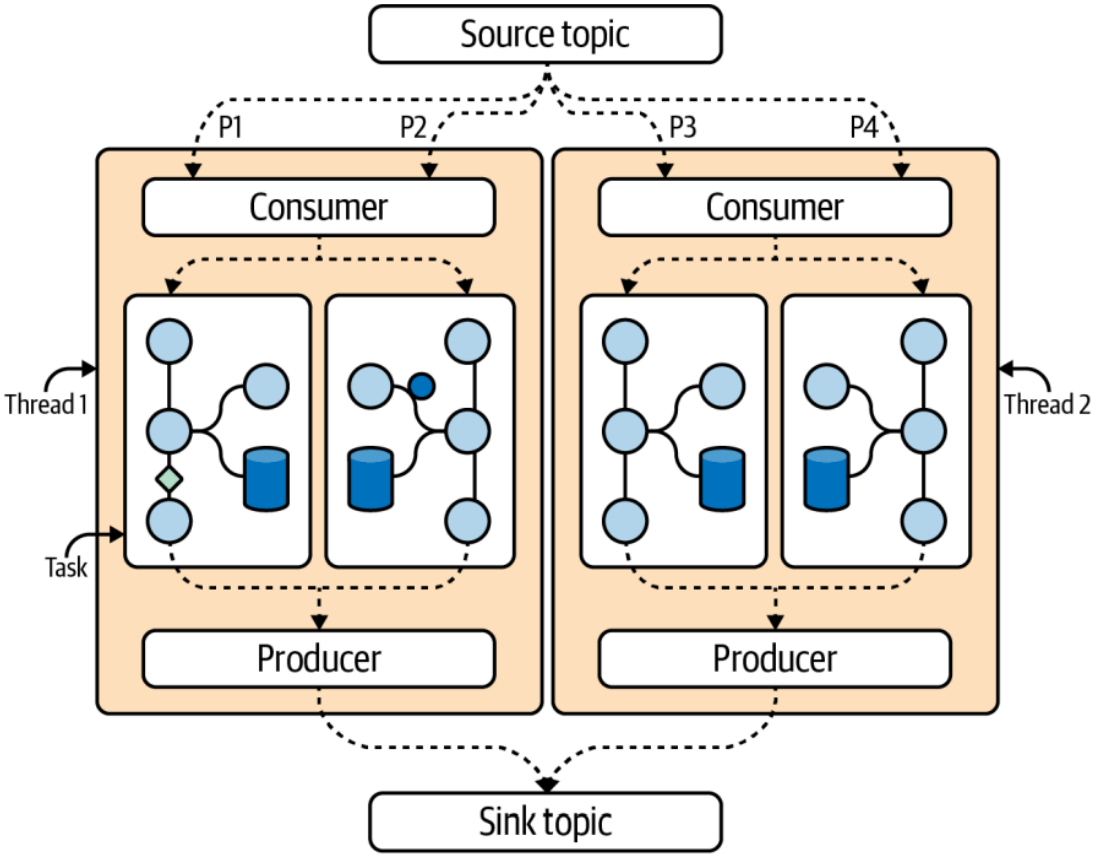


그림 2-8. 2 개의 쓰레드에서 실행되는 4 개의 카프카 스트림즈 태스크

우선적으로 2 개의 쓰레드 (num.stream.threads = 2)로 실행되는 애플리케이션을 구성해보자. 소트 토픽이 4 개의 파티션을 갖고 있기 때문에 4 개의 태스크가 생성될 것이고 각 쓰레드로 분배될 것이다. 최종적으로 그림 2-8과 같이 태스크/쓰레드가 배치된다.

쓰레드 당 하나 이상의 태스크를 실행하는 것은 완벽하게 좋지만 때때로 사용가능한 CPU 자원을 완전히 이용하기 위해 보다 많은 수의 쓰레드를 실행하는 것이 바람직하다. 쓰레드 수를 증가시킨다고 해서 태스크 수가 변경되지는 않으며 쓰레드 간 태스크의 분배를 변경시킨다. 예를 들어 동일한 카프카 스트림즈 애플리케이션을 4 개의 쓰레드 (num.stream.threads = 4)로 실행시키도록 재구성한다면 그림 2-9와 같이 태스크/쓰레드가 배치된다.

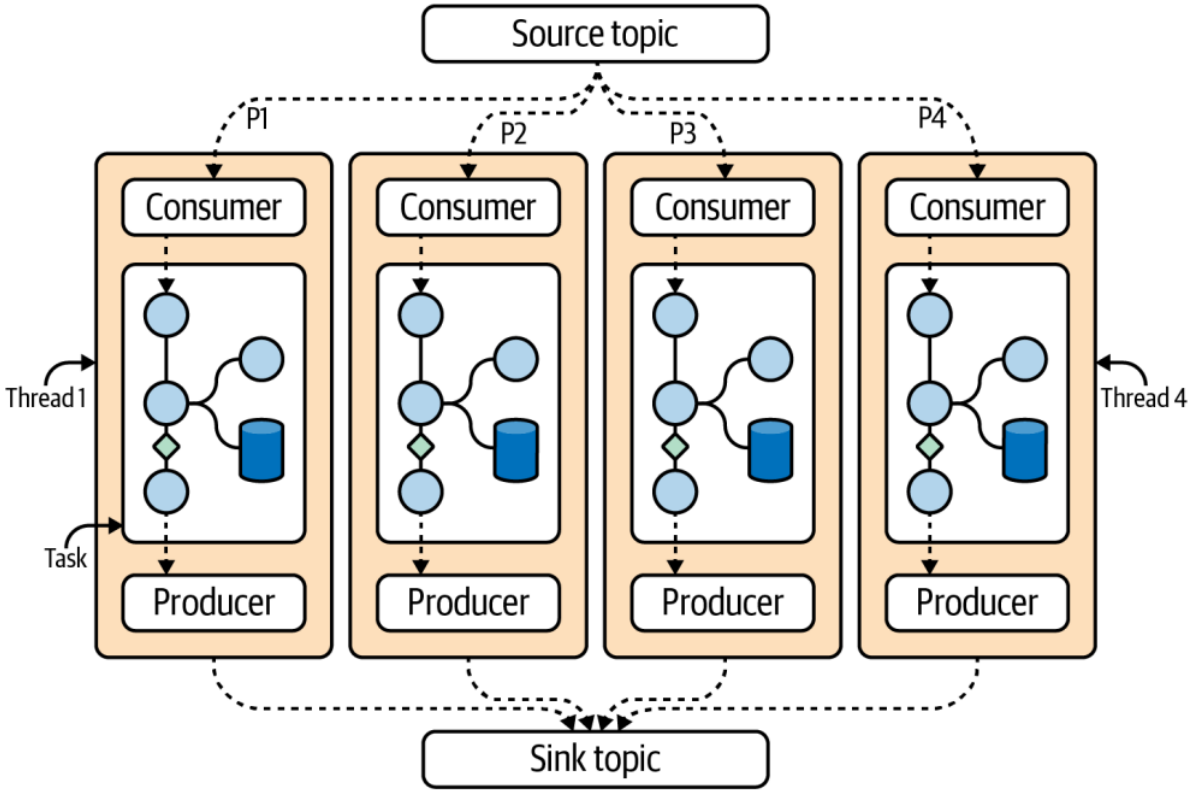


그림 2-9. 4 개의 쓰레드에서 실행되는 4 개의 카프카 스트림즈 태스크

이제 카프카 스트림즈 아키텍처에 대해 배웠는데 스트림 처리 애플리케이션 작성을 위해 카프카 스트림즈가 노출하는 API를 살펴보자.

고수준 DSL 대 저수준 프로세서 API

> 다른 솔루션은 다른 추상화 계층에서 나타난다.
>
> - James Clear[21]

소프트웨어 엔지니어링 분야의 공통적인 개념은 추상화에는 항상 비용이 따른다는 것이다: 더욱 더 추상화할 수록 세부사항이 더욱 사라지며 소프트웨어가 더욱 더 마술처럼 보일수록 더욱 더 제어할 수 없다는 것이다. 카프카 스트림즈로 시작할 때 저수준 컨슈머/프로듀서 API를 직접 사용하여 솔루션을 설계하는 대신 고수준 라이브러리를 사용하여 스트림 처리 애플리케이션을 구현함으로써 어떤 유형의 제어를 포기하는 것인지 의아스럽게 생각할 수 있다.

운좋게도 카프카 스트림즈는 프로젝트와 개발자의 경험 및 선호도에 따라 그들에게 가장 잘 맞는 추상화 수준을 선택할 수 있도록 한다.

여러분이 선택할 수 있는 2 가지 API:

---

[21] First Principles: Elon Musk on the Power of □inking for Yourself

- 고수준 DSL
- 저수준 프로세서 API

고수준 DSL과 저수준 프로세서 API에 대한 상대적인 추상화 수준을 그림 2-10에 나타냈다.

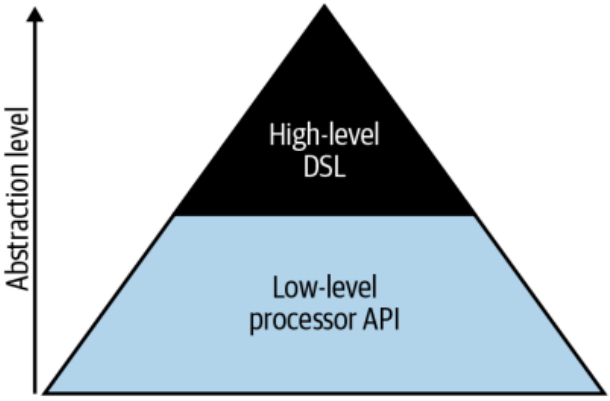


그림 2-10. 카프카 스트림즈 API의 추상화 수준

고수준 DSL은 프로세서 API 기반으로 생성되며 각각이 노출하는 인터페이스가 약간 다르다. 여러분이 기능적 프로그래밍 스타일을 사용하여 스트림 처리 애플리케이션을 구축하고자 하거나 데이터 (스트림 및 테이블)를 사용하여 작업하기 위해 보다 고수준의 추상화를 활용하고자 한다면 DSL이 좋은 선택이다.

반면 데이터에 대한 저수준 액세스(예, 메타데이터를 읽기 위한 액세스), 주기적 기능 스케줄링, 애플리케이션 상태에 대한 보다 세분화된 액세스 또는 특정 연산 타이밍에 대한 보다 미세한 제어가 필요하다면 프로세서 API가 보다 좋은 선택이다. 다음 튜토리얼에서 여러분은 DSL과 프로세서 API 둘 모두에 대한 예제를 볼 것이다. 추후 보다 세부적으로 둘 모두를 탐구할 것이다.

이제 이러한 2 가지 추상화 수준 간 차이를 보기 위한 가장 좋은 방법은 코드 예제를 사용하는 것이다. 첫번째 카프카 스트림즈 튜토리얼로 들어가 보자: 헬로우 스트림.

튜토리얼 소개: 헬로우 스트림

이 절에서는 카프카 스트림즈에 대한 첫번째 실습을 진행할 것이다. 이는 새로운 프로그래밍 언어와 라이브러리를 배울 때 표준이 된 "헬로우 월드" 튜토리얼을 변형한 것이다. 이 튜토리얼에서는 고수준 DSL과 저수준 프로세서 API를 사용하는 2 가지 예를 다룬다. 둘 모두 기능적으로 동일하며 카프카 users 토픽으로 메시지를 받자마자 간단한 인사를 출력할 것이다 (예, Mitch 메시지를 받자마자 각 애플리케이션은 Hello, Mitch를 출력할 것이다).

시작하기 위해 프로젝트 환경 설정 방법을 살펴보자.

프로젝트 설정

이 책의 모든 튜토리얼은 작동 중인 카프카 클러스터를 필요로 하며 각 장의 소스 코드는 도커를 사용

하여 개발 클러스터를 작동시킬 수 있도록 docker-compose.yml 파일을 포함할 것이다. 카프카 스트림즈 애플리케이션은 카프카 클러스터 외부에서 작동하기 때문에 (예, 브로커와 다른 머신 상) 카프카 클러스터를 필요하지만 카프카 스트림즈 애플리케이션과는 별개의 인프라로 보는 것이 최선이다.

카프카 클러스터를 작동시키기 위해 저장소를 복제한 후 이 장의 튜토리얼을 포함한 디렉토리로 이동한다. 다음 명령어를 사용할 수 있다:

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

$ cd mastering-kafka-streams-and-ksqldb/chapter-02/hello-streams
```

다음에 docker-compose up 명령어를 실행하여 카프카 클러스터를 실행시킨다.

브로커는 29092 포트[22]에서 수신대기하고 있을 것이다. 또한 이전 명령은 이 튜토리얼에 필요한 users 토픽을 미리 생성할 컨테이너를 시작할 것이다. 이제 작동 중인 카프카 클러스터를 이용하여 카프카 스트림즈 애플리케이션 생성을 시작할 수 있다.

새로운 프로젝트 생성

이 책에서는 카프카 스트림즈 애플리케이션을 컴파일 및 실행시키기 위해 Gradle[23]이라는 빌드 툴을 사용할 것이다. Maven과 같은 다른 빌드 툴도 지원하지만 빌드 파일들의 보다 나은 가독성으로 인해 Gradle을 선택하였다.

코드 컴파일 및 실행 외에 Gradle은 이 책 예제가 아닌 여러분이 구축한 카프카 스트림즈 애플리케이션을 빠르게 부트스트랩하기 위해 사용될 수 있다. 이는 프로젝트가 놓이는 디렉토리를 생성하고 그 디렉토리 안에서 gradle init 명령을 실행시킴으로써 이루어진다. 예는 다음과 같다:

```
$ mkdir my-project && cd my-project

$ gradle init ₩
        --type java-application ₩
        --dsl groovy ₩
        --test-framework junit-jupiter ₩
        --project-name my-project ₩
        --package com.example
```

이 책의 소스 코드는 각 튜토리얼에 대한 초기화된 프로젝트 구조를 이미 포함하고 있으며 따라서 여

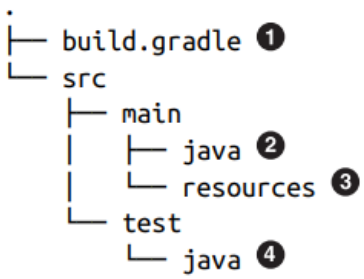
---

[22] 확인하고 싶고 telnet이 설치되어 있다면 echo 'exit' | telnet localhost 29092를 실행시킬 수 있다. 포트가 열려 있다면 "Connected to localhost"라는 출력을 볼 수 있을 것이다.

[23] Gradle 설치 방법은 https://gradle.org에 찾을 수 있다. 이 책에서는 6.6.1 버전을 사용하였다.

러분 스스로 새로운 프로젝트를 시작하는 경우가 아니라면 gradle init 명령을 실행시킬 필요는 없다. 여기서는 여러분이 어떤 시점에 스스로 카프카 스트림즈 애플리케이션을 작성하고 다음 프로젝트를 부트스트랩하기 위한 빠른 방법을 원할 것이라는 가정 하에 이를 언급하고 있다.

다음은 카프카 스트림즈 애플리케이션의 기본 프로젝트 구조이다:



① 이는 프로젝트 빌드 파일로 애플리케이션이 필요로 하는 모든 의존도를 지정할 것이다 (카프카 스트림즈 라이브러리 포함).

② src/main/java 디렉토리에 소스 코드와 토폴로지 정의를 저장할 것이다.

③ src/main/resources 디렉토리는 일반적으로 설정 파일 저장에 사용된다.

④ "카프카 스트림즈 테스팅"에서 논의할 단위 및 토폴로지 테스트는 src/test/java 디렉토리에 놓일 것이다.

새로운 카프카 스트림즈 프로젝트를 어떻게 부트스트랩 하는 지를 배웠고 프로젝트 구조를 살펴보았는데 프로젝트에 카프카 스트림즈를 추가하는 방법을 살펴보자.

카프카 스트림즈 의존도 추가

카프카 스트림즈를 사용하기 위해 빌드 파일에 단순히 카프카 스트림즈 라이브러리를 추가한다 (Gradle 프로젝트에서 빌드 파일은 build.gradle이다). 빌드 파일 예는 다음과 같다:

```
plugins {
    id 'java'
    id 'application'
} repositories {
    jcenter()
} dependencies {
    implementation 'org.apache.kafka:kafka-streams:2.7.0' ①
}
```

```
task runDSL(type: JavaExec) { ②
    main = 'com.example.DslExample'
    classpath sourceSets.main.runtimeClasspath
}
task runProcessorAPI(type: JavaExec) { ③
    main = 'com.example.ProcessorApiExample'
    classpath sourceSets.main.runtimeClasspath
}
```

① 카프카 스트림즈 의존도를 프로젝트에 추가한다

② 여기서는 2 가지 다른 버전의 토폴로지를 생성할 것이기 때문에 이 책의 튜토리얼과 비교해서 이 튜토리얼은 고유하다. 이 라인은 애플리케이션의 DSL 버전을 실행시키기 위해 Gradle 태스크를 추가한다.

③ 비슷하게 이 라인은 애플리케이션의 프로세서 API 버전을 실행시키기 위해 Gradle 태스크를 추가한다.

(외부 저장소로부터 프로젝트 내로 의존도를 실제 가져올) 프로젝트를 빌드하기 위해 다음 명령을 실행시킬 수 있다:

```
./gradlew build
```

이게 전부로 카프카 스트림즈가 설치되어 사용할 준비가 되어 있다. 계속 튜토리얼을 진행해보자.

DSL

DSL 예제는 예외적으로 간단하다. 우선 프로세서 토폴로지 생성을 위해 StreamsBuilder 카프카 스트림즈 클래스를 사용해야 한다:

```
StreamsBuilder builder = new StreamsBuilder();
```

다음에 프로세서 토폴로지에서 배웠듯이 카프카 토픽에서 데이터를 읽기 위해 소스 프로세서를 추가한다 (이 예에서는 users 토픽). 여기서 데이터를 어떻게 모델링하는 지에 따라 사용할 수 있는 몇몇 방법이 있지만 지금은 데이터를 스트림으로 모델링한다. 다음 라인은 소스 프로세서를 추가한다:

```
KStream<Void, String> stream = builder.stream("users"); ①
```

① 다음 장에서 세부적으로 논의할 것이지만 KStream<Void, String>에서 제네릭은 키와 값 타입을 의미한다. 이 경우 키는 없고 (void) 값은 String 타입이다.

이제 스트림 프로세서를 추가할 시점이다. 각 메시지에 대해 간단한 인사를 출력하기 때문에 다음과

같이 단순한 람다 표현식을 사용하는 foreach 연산을 사용할 수 있다:

```
Streamforeach(
  (key, value) -> {
      System.out.println("(DSL) Hello, " + value);
  });
```

마지막으로 토폴로지를 생성하고 스트림 처리 애플리케이션을 실행시킨다:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);
streams.start();
```

일부 상용구를 포함하여 프로그램을 실행시키기 위해 필요한 전체 코드는 예제 2-1과 같다.

예제 2-1. 헬로우, 월드 – DSL 예

```
class DslExample {
    public static void main(String[] args) {
        StreamsBuilder builder = new StreamsBuilder(); ①
        KStream stream = builder.stream("users"); ②
        stream.foreach( ③
            (key, value) -> { System.out.println("(DSL) Hello, " + value); });
    // omitted for brevity
    Properties config = ...; ④
    KafkaStreams streams = new KafkaStreams(builder.build(), config); ⑤
    streams.start();
    // close Kafka Streams when the JVM shuts down (e.g., SIGTERM)
    Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); } } ⑥
```

① 토폴로지를 생성하기 위해 builder를 사용한다.

② users 토픽에서 데이터를 읽기 위해 소스 프로세서를 추가한다.

③ 메시지를 출력하기 위해 DSL의 foreach 연산자를 사용한다. DSL은 추후 장에서 논의할 많은 연산자를 포함하고 있다.

④ 간결함을 위해 카프카 스트림즈 설정을 생략했는데 추후 장에서 논의할 것이다. 다른 것보다 이

설정을 통해 애플리케이션이 어떤 카프카 클러스터에서 데이터를 읽는 지와 이 애플리케이션이 어떤 컨슈머 그룹에 속하는 지를 지정할 수 있다.

⑤ 토폴로지를 생성하고 스트리밍을 시작한다.

⑥ JVM이 종료될 때 카프카 스트림즈를 닫는다.

애플리케이션을 실행시키기 위해 단순히 다음 명령을 실행한다:

```
./gradlew runDSL --info
```

이제 카프카 스트림즈 애플리케이션이 작동하여 들어오는 데이터를 수신대기하고 있을 것이다. "Hello, Kafka"에서 상기할 수 있듯이 kafka-console-producer 컨솔 스크립트를 사용하여 카프카 클러스터 내에 일부 데이터를 넣을 수 있다. 이를 위해 다음 명령을 실행한다:

```
doker-compose exec kafka bash ①
kaka-console-producer ₩ ②
  --bootstrap-server localhost:9092 ₩
  --topic users
```

① 컨솔 스크립트는 개발 클러스터 내 브로커가 작동하고 있는 kafka 컨테이너에서 사용할 수 있다. 또한 공식 Kafka 배포판의 일부로 이들 스크립트를 다운로드할 수도 있다.

② users 토픽에 데이터를 쓰기 할 로컬 프로듀서를 시작한다.

프로듀서 프롬프트에서 user 이름을 타이핑하고 엔터 키를 입력하여 하나 이상의 레코드를 생성한다. 데이터 입력 종료 시에는 프롬프트에서 나가기 위해 Control-C를 누른다:

```
>angie
>guy
>kate
>mark
```

카프카 스트림즈 애플리케이션은 다음 인사말을 산출해야 한다:

```
(DSL) Hello, angie
(DSL) Hello, guy
(DSL) Hello, kate
(DSL) Hello, mark
```

애플리케이션이 예상한 바와 같이 작동하는 것을 확인하였으며 다음 여러 장에 걸쳐 보다 흥미로운 유

스케이스를 다룰 것이다. 그러나 토폴로지 정의 및 애플리케이션 구동 프로세스는 구축 토대이다. 다음에는 저수준 프로세서 API를 사용하여 동일한 카프카 스트림즈 토폴로지를 생성하는 법을 살펴보자.

프로세서 API

프로세서 API는 고수준 DSL에서 사용가능한 추상화 일부가 부족하며 구문이 Topology.addSource, Topology.addProcessor와 Topology.addSink (마지막은 이 예에서 사용하지 않는다)를 사용함으로써 프로세서 토폴로지를 생성하고 있음을 더욱 직접적으로 알려준다. 프로세서 토폴로지를 사용하는데 있어서 첫번째 단계는 다음과 같이 새로운 Topology 인스턴스를 인스턴스화하는 것이다:

```
Topology topology = new Topology();
```

다음에 users 토픽에서 데이터를 읽어 들이기 위해 소스 프로세서를 생성하고 인사말을 출력하기 위해 스트림 프로세서를 생성할 것이다. 스트림 프로세서는 곧 구현할 SayHelloProcessor 클래스를 참조한다:

```
topology.addSource("UserSource", "users"); ①

topology.addProcessor("SayHello", SayHelloProcessor::new, "UserSource"); ②
```

① addSource 메소드의 첫번째 인자는 스트림 프로세서를 위한 임의의 이름으로 이 예에서 프로세서를 UserSource로 부른다. 다음 라인에서 이 이름을 참조하여 자식 프로세서와 연결하며 이는 토폴로지를 통해 데이터가 어떻게 흘러야 하는 지를 정의한다. 두번째 인자는 소스 프로세서가 읽어 들이는 토픽명으로 users이다.

② 이 라인은 처리 로직인 SayHelloProcessor 클래스 (다음 절에서 이를 생성할 것이다)에 정의되어 있는 SayHello 다운스트림 프로세서를 생성한다. 프로세서 API에서는 부모 프로세서의 이름을 지정함으로써 한 프로세서와 다른 프로세서를 연결할 수 있다. 이 경우 UserSource 프로세서를 SayHello 프로세서의 부모로 지정하며 이는 데이터가 UserSource에서 SayHello로 흐를 것임을 의미한다.

이전에 DSL 튜토리얼에서 보았듯이 이제 토폴로지를 생성하고 이를 실행하기 위해 streams.start()를 호출해야 한다:

```
KafkaStreams streams = new KafkaStreams(topology, config);

streams.start();
```

코드를 실행시키기 전에 SayHelloProcessor 클래스를 구현해야 한다. 프로세서 API를 사용하여 맞춤형 스트림 프로세서를 생성할 때마다 여러분은 Processor 인터페이스를 구현해야 한다. 인터페이스는 스트림 프로세서 초기화 (init), 단일 레코드에 스트림 처리 로직 적용 (process)과 정리 함수 (close)를 위한 메소드를 지정한다. 초기화와 정리 함수는 이 예에서는 필요치 않다.

다음은 이 예에서 사용할 SayHelloProcessr의 단순한 구현체이다. 7 장에서 보다 복잡한 예제와 프로세서 인터페이스의 모든 인터페이스 메소드 (init, process 및 close)를 세부적으로 다룰 것이다.

```
public class SayHelloProcessor implements Processor<Void, String, Void, Void> { ①

  @Override

  public void init(ProcessorContext<Void, Void> context) {} ②

  @Override

  public void process(Record<Void, String> record) {③

    System.out.println("(Processor API) Hello, " + record.value()); }

  @Override

  public void close() {} ④
```

① Processor 인터페이스의 최초 2 개의 제네릭 (Processor<Void, String, ..., ...>)은 입력 키와 값 타입을 의미한다. 키가 null이고 값이 usernames (텍스트 문자열)이기 때문에 Void와 String은 적절한 선택이다. 다음 2 개의 제네릭 (Processor<..., ..., Void, Void>)은 출력 키와 값 타입으로 이 예에서 SayHelloProcessor이 단순히 인사말을 출력하며 출력 키와 값을 다운스트림으로 전달하지 않기 때문에 Void는 2 개의 제네릭에 대해 적절한 타입이다[24].

② 이 예에서는 특별한 초기화가 필요치 않으며 따라서 메소드 바디는 비어 있다. ProcessorContext 인터페이스의 제네릭 (ProcessorContext<Void, Void>)은 출력 키와 값 타입을 의미한다 (이 예에서는 어떤 메시지도 다운스트림으로 전달하지 않기 때문에 둘 모두 Void이다).

③ 처리 로직은 Processor 인터페이스의 process 메소드에 담겨있다. 여기서는 단순히 인사말을 출력하는 것이다. Record 인터페이스의 제네릭은 입력 레코드의 키와 값 타입을 의미한다.

④ 이 예에서는 특별한 정리가 필요치 않다.

DSL 예에서와 같이 동일한 명령을 사용하여 코드를 실행시킬 수 있다:

```
./gradlew runProcessorAPI --info
```

카프카 스트림즈 애플리케이션이 예상한 바와 같이 작동함을 나타내기 위해 다음과 같은 출력이 나와야 한다:

```
(Processor API) Hello, angie

(Processor API) Hello, guy
```

---

[24] 이 버전의 Processor 인터페이스는 카프카 스트림즈 버전 2.7에 도입되었고 2.6 이전 버전에서 사용가능했던 인터페이스 초기 버전은 더 이상 사용되지 않는다. Processor 인터페이스 초기 버전에서는 단지 입력 타입만 지정되었다. 이는 타입 안정성 검사 시 몇 가지 문제를 야기했으며 따라서 새로운 형태의 Processor 인터페이스 사용이 권고된다.

(Processor API) Hello, kate

(Processor API) Hello, mark

7 장에서 볼 것이지만 프로세서 API의 능력에도 불구하고 DSL을 사용하는 것이 보다 선호된다. 다른 장점 중에서도 스트림과 테이블이라는 2 가지 매우 강력한 추상화를 포함하기 때문이다. 다음 절에서 이 추상화를 살펴볼 것이다.

스트림과 테이블

예제 2-1을 자세히 보면 카프카 토픽을 스트림 내로 읽어 들이기 위해 stream이라는 DSL 연산자를 사용했음을 알 수 있을 것이다. 관련 코드는 다음과 같다:

```
KStream<Void, String> stream = builder.stream("users");
```

그러나 카프카 스트림즈는 데이터를 테이블로 보기 위한 추가적인 방법 또한 지원한다. 이 절에서는 두 옵션 모두를 논의하며 언제 스트림과 테이블을 사용하는 지를 배울 것이다.

"프로세서 토폴로지"에서 논의했듯이 프로세서 토폴로지 설계 시에는 애플리케이션이 읽어 들이고 쓰기 하는 토픽에 해당하는 일련의 소스 및 싱크 프로세스를 지정한다. 그러나 카프카 토픽을 직접 사용하는 대신 카프카 스트림즈 DSL은 토픽의 다른 표현을 사용할 수 있도록 하며 각각은 다른 유스케이스에 적합하다. 카프카 토픽에서 데이터를 모델링하는 2 가지 방식이 존재한다: 스트림 (레코드 스트림) 또는 테이블 (체인지로그 스트림). 이 두 모델을 비교하기 위한 가장 쉬운 방법은 예제를 보는 것이다.

가령 표 2-2와 같이 사용자 ID가 키인 레코드로 구성된 ssh 로그를 포함하는 토픽을 갖고 있다고 해보자.

표 2-2. 단일 토픽-파티션 내 키 값이 있는 레코드

키	값	오프셋
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

이 데이터를 소비하기 전에 사용할 추상화를 결정할 필요가 있다: 스트림 또는 테이블. 이 결정을 할 때 해당 키의 최신 상태/표현만을 추적할 지 아니면 메시지의 전체 이력을 추적할 지를 고려해야 한다. 두 옵션을 하나씩 비교해보자:

스트림

이는 데이터베이스 용어로 insert로 생각할 수 있다. 각각의 고유 레코드가 로그 뷰에 존재한다. 토픽의 스트림 표현은 표 2-3과 같이 나타낼 수 있다.

표 2-3. ssh 로그의 스트림 뷰

키	값	오프셋
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

테이블

테이블은 데이터베이스의 update로 생각할 수 있다. 이 뷰에서는 각 키의 현재 상태 (해당 키의 가장 최신 레코드 또는 일종의 집계)만 포함된다. 테이블은 보통 압축된 토픽으로부터 생성된다 (즉, cleanup.policy = compact로 구성된 토픽으로 이는 카프카에게 각 키의 최신 표현만을 유지하라는 설정이다). 토픽의 테이블 표현은 표 2-4와 같이 나타낼 수 있다.

표 2-4. ssh 로그의 테이블 뷰

키	값	오프셋
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

테이블은 본질적으로 상태를 저장하며 카프카 스트림즈에서 종종 집계 수행 시 사용된다[25]. 표 2-4에서 실제 수학적 집계를 수행하지는 않았고 단지 각 사용자 ID에 대한 최신 ssh 이벤트만을 유지했다. 그러나 테이블은 또한 수학적 집계도 지원한다. 예를 들어 각 키에 대해 최신 레코드를 추적하는 대신 쉽게 롤링 카운트 계산을 할 수 있다. 이 경우 count 집계 결과를 포함하는 약간 다른 테이블을 가질 것이다. 표 2-5에서 카운트 집계된 테이블을 볼 수 있다.

표 2-5. ssh 로그의 집계된 테이블 뷰

키	값	오프셋
mitch	2	1
elyse	1	2
isabelle	1	3

세심한 독자라면 카프카 저장 계층 (분산된 추가 전용 로그)과 테이블 설계 간 불일치를 알아챘을 것이다. 카프카로 쓰여지는 레코드는 불변인데 카프카 토픽의 테이블 표현을 사용하여 update로 데이터를 모델링하는 것이 가능한가?

답은 간단하다: 테이블은 키-값 저장소를 사용하여 카프카 스트림즈 측에서 구체화되는데 기본적으로

---

[25] 사실 테이블은 집계된 스트림으로 간주된다. 이 주제를 보다 다루는 Michael Noll의 Of Streams and Tables in Kafka and Stream Processing, Part 1을 참조하기 바란다.

이는 RocksDB[26]를 사용하여 구현되어 있다. 순서가 있는 이벤트 스트림을 소비하여 클라이언트 측 키-값 저장소 (카스카 스트림 용어로 보통 상태 저장소)에 각 키에 대한 최신 레코드만을 저장함으로써 데이터에 대한 테이블 또는 맵 형태의 표현을 얻을 수 있다. 다른 말로 테이블은 카프카에서 소비하는 어떤 것도 아니며 클라이언트 측에서 생성되는 어떤 것이다.

여러분은 이 기본 아이디어를 구현하기 위해 몇 줄의 Java 코드를 실제 작성할 수 있다. 다음 코드 스니펫에서 List는 순서가 있는 레코드 모음[27]을 포함하기 때문에 스트림을 나타내며 리스트를 통해 반복하고(stream.forEach) Map을 사용하여 해당 키의 최신 레코드만을 포함함으로써 테이블이 생성된다. 다음 Java 코드는 이를 보여준다:

```
import java.util.Map.Entry;

var stream = List.of(

    Map.entry("a", 1),

    Map.entry("b", 1),

    Map.entry("a", 2));

var table = new HashMap<>();

stream.forEach((record) -> table.put(record.getKey(), record.getValue()));
```

이 코드 실행 후 스트림과 테이블을 출력한다면 다음 출력을 볼 수 있을 것이다:

```
stream ==> [a=1, b=1, a=2]

table ==> {a=2, b=1}
```

물론 이 카프카 스트림즈 구현은 보다 복잡하며 인메모리 Map과 달리 내고장성 데이터 구조를 활용할 수 있다. 그러나 무제한 스트림의 테이블 표현을 생성할 수 있는 능력은 추후 더 논의할 것인 스트림과 테이블의 보다 복잡한 관계 중 한 측면일 뿐이다.

스트림/테이블 이중성

스트림과 테이블의 이중성은 테이블이 스트림으로 표현될 수 있고 스트림이 테이블을 재구축하는데 사용될 수 있다는 사실에 기인한다. 이전 절에서 우리는 카프카 추가 전용 불변 로그와 데이터에 대해 업데이트를 수용하는 가변 테이블 구조의 개념 간 불일치를 논의할 때 스트림에서 테이블로의 후자 변

---

[26] RocksDB는 원래 페이스북에서 개발된 고속의 내장 키-값 저장소이다. 4-6 장에 RocksDB와 키-값 저장소에 대해 보다 논의를 할 것이다.

[27] 비유로 더 깊이 들어가기 위해 리스트 내 각 항목의 인덱스 위치는 카프카 토픽 내 레코드의 오프셋을 나타낸다.

환을 보았다.

스트림으로부터 테이블을 재구축하는 것은 카프카 스트림즈에 고유한 것은 아니며 사실 다양한 유형의 스토리지에서는 일반적인 것이다. 예를 들어 MySQL의 복제 프로세스는 다운스트림 복제본에 소스 테이블을 재구축하기 위해 이벤트 스트림 (예, 행 변경)을 취하는 동일한 개념에 의존한다. 비슷하게 Redis는 인메모리 키-값 저장소에 쓰여지는 모든 명령을 수집하는 추가 전용 파일 (append-only file, AOF) 개념을 갖고 있다. Redis 서버가 오프라인이 되면 AOF의 명령 스트림은 데이터셋을 재구축하기 위해 재생될 수 있다.

동전의 다른 면 (테이블을 스트림으로 표현하는 것)은 어떤가? 테이블을 볼 때는 스트림에 대한 단일 시점 표현을 보는 것이다. 이전에 보았듯이 테이블은 새로운 레코드가 도착 시 업데이트될 수 있다. 테이블에서 스트림으로의 뷰를 변경함으로써 단순히 update를 insert로 간단히 처리할 수 있으며 키 업데이트 대신 로그 말미에 새로운 레코드를 추가할 수 있다. 이에 숨어있는 직관은 다음의 몇 줄 안 되는 Java 코드를 사용하여 볼 수 있다:

```
var stream = table.entrySet().stream().collect(Collectors.toList());

stream.add(Map.entry("a", 3));
```

이제 스트림의 내용을 출력한다면 업데이트 시맨틱을 더 이상 사용하지 않으며 대신 insert 시맨틱을 사용함을 볼 것이다.

```
stream ==> [a=2, b=1, a=3]
```

지금까지 스트림과 테이블 관련 직관을 얻기 위해 Java 표준 라이브러리로 작업해왔다. 그러나 카프카 스트림즈에서 스트림과 테이블을 사용해 작업할 때는 일련의 보다 전문화된 추상화를 사용할 것이다. 다음에 이들 추상화를 살펴볼 것이다.

KStream, KTable, GlobalKTable

카프가 스트림에서 고수준 DSL이 저수준 프로세서 API에 비해 갖는 장점 중 하나는 전자가 스트림과 테이블을 사용한 작업을 매우 쉽게 만드는 일련의 추상화를 포함한다는 것이다.

다음은 각각에 대한 고수준 개요를 포함한다:

KStream

- 파티션된 레코드 스트림의 추상화로 insert 시맨틱을 사용하여 데이터가 표현된다 (즉, 각 이벤트가 다른 이벤트와 독립적으로 고려된다).

KTable

- 파티션된 테이블의 추상화 (예, 체인지로그 스트림)로 데이터가 update 시맨틱을 사용하여 표현된다 (키의 가장 최신 표현이 애플리케이션에 의해 추적된다). KTables이 파티션되어 있기

때문에 각 카프카 스트림즈 태스크는 전체 테이블 중 일부 만을 포함한다.[28]

GlobalKTable

- 각각의 GlobalKTable이 데이터에 대해 (파티션되지 않은) 완전한 복제본을 포함한다는 것을 제외하고는 KTable과 유사하다. 4 장에서 KTable과 GlobalKTable을 언제 사용할지를 배울 것이다.

카프카 스트림즈 애플리케이션은 복수의 스트림/테이블 추상화 또는 단지 하나의 추상화만을 사용할 수 있다. 이는 유스케이스에 전적으로 달려 있으며 다음의 몇 장을 진행하면서 각각을 언제 사용할지를 배울 것이다. 이제 스트림과 테이블에 대한 초기 논의를 완료하였고 다음 장으로 넘어가 카프카 스트림즈를 좀 더 자세하게 논의해보자.

요약

축하한다. 여러분은 카프카 스트림즈와의 첫번째 데이트를 마쳤으며 다음은 여러분이 배운 것들이다:

- 카프카 스트림즈는 카프카 생태계의 스트림 처리 계층에 존재한다. 이는 복잡한 데이터 처리, 변환 및 보강이 일어나는 곳이다.
- 카프카 스트림즈는 간단한 기능적 API와 프로젝트에 대해 재사용할 수 있는 일련의 스트림 처리 프리미티브를 통해 스트림 처리 애플리케이션의 개발을 단순화하기 위해 만들어졌다. 보다 많은 제어가 필요한 경우 토폴로지를 정의하는데 저수준의 프로세서 API가 사용될 수도 있다.
- 카프카 스트림즈는 아파치 Flink와 아파치 Spart Streaming와 같은 클러스터 기반 솔루션보다 친화적인 학습 곡선과 보다 쉬운 배치 모델을 갖고 있다. 이는 또한 실제 스트링으로 간주되는 한번에 한 이벤트 처리 또한 지원한다.
- 카프카 스트림즈는 실시간 의사결정과 데이터 처리를 필요로 하거나 이로부터 혜택을 얻기 위한 문제 해결에 우수하다. 더구나 이는 신뢰성있고 유지보수가능하며 확장가능하고 탄력적이다.
- 카프카 스트림즈 설치 및 실행은 간단하며 이장의 코드 https://github.com/mitch-seymour/mastering-ka□astreams-and-ksqldb에서 찾을 수 있다.

다음 장에서는 카프카 스트림즈의 스테이트리스 처리에 대해 배울 것이다. 또한 보다 고급의 강력한 스트림 처리 애플리케이션 구축에 도움이 될 몇몇 새로운 DSL 연산자를 실제 경험할 것이다.

---

[28] 소스 토픽이 하나 이상의 파티션을 갖는다고 가정