5장 윈도우와 시간

우리가 시간의 흐름을 통해 삶을 측정할 만큼 시간은 중요한 개념이다. 매년 여섯 명이 내 주위에 서서 생일 축하 노래를 부르며 마지막 음이 사라짐에 따라 케이크가 이 신비로운 기운 말미에 제공된다. 나는 케이크가 나를 위한 것이라고 생각하고 싶지만 시간을 위한 것이다.

시간은 실세계에 너무 복잡하게 얽혀 있을 뿐만 아니라 우리의 이벤트 스트림을 퍼뜨린다. 카프카 스트림즈의 완전한 능력을 밝히기 위해서는 이벤트와 시간 사이의 관계를 이해해야 한다. 이 장에서는 이 관계를 세부적으로 다루며 윈도우를 통해 직접 실습을 진행할 것이다. 윈도우는 이벤트를 명시적인 시간 버킷으로 그룹화할 수 있도록 하고 (이전 장에서 우선 다루었던) 보다 고급 조인 및 집계를 생성하는데 사용될 수 있다.

이 장 말미에서 여러분은 다음을 이해할 것이다:

- 이벤트 시간, 인입 시간과 처리 시간 간 차이
- 이벤트를 특정 타임스탬프 및 타임 시맨틱과 연관시키기 위해 맞춤형 타임 추출기를 구축하는 방법
- 시간이 카프카 스트림즈를 통해 데이터 흐름을 제어하는 방법
- 윈도우를 갖는 조인 수행 방법
- 윈도우를 갖는 집계 수행 방법
- 지연 및 비정상 이벤트 처리에 사용가능한 전략
- 윈도우 최종 결과를 처리하기 위해 suppress 연산자를 사용하는 방법
- 윈도우를 갖는 키-값 저장소에 쿼리하는 방법

이전 장과 같이 튜토리얼을 통해 이러한 개념을 소개할 것이다. 따라서 더 이상 고민하지 말고 이 장에 구축할 애플리케이션을 살펴보자.

튜토리얼 소개: 환자 모니터링 애플리케이션

시간 중심 스트림 처리에 대한 가장 중요한 유스케이스 중 상당 부분은 의료 분야에 있다. 환자 모니터링 시스템은 초당 수백 개의 계측치를 생산할 수 있으며 이 데이터를 빨리 처리/응답하는 것이 특정 유형의 의료 상태를 처리하는데 중요하다. 이는 Children's Healthcare of Atlanta가 머리 외상을 가진 아이들이 가까운 시기에 외과 개입이 필요한지 여부에 대해 실시간 예측을 하기 위해 카프카 스트림즈와 ksqlDB를 사용한 이유이다[1]. 이 유스케이스에 고무되어 우리는 환자 바이탈을 모니터링하는 애

[1] 이는 두개내 압력을 측정하고 측정치를 집계하여 예측 모델에 집계를 전송함으로써 달성된다. 압력이 다음 30분 내에 위험한 수준에 도달할 것임을 모델이 예측할 때 의료 전문가에 통지되어 적절한

플리케이션을 구축함으로써 몇몇 시간 중심 스트리밍 개념을 보여줄 것이다. 머리 외상 모니터링 대신 전신 염증 반응 증후군 (Systemic Inflammatory Response Syndrome, SIRS)이라는 의학적 상태의 존재를 탐지하려고 할 것이다. 남캘리포니아 의과 대학의 의사 조수인 Bridgette Kadri에 따르면 SIRS의 지표로 사용될 수 있는 체온, 혈압 및 심박수를 포함한 몇몇 생체 신호가 존재한다. 이 튜토리얼에서는 이들 중 체온과 심박수 측정치를 살펴볼 것이다. 이 바이탈 모두 사전 정의된 임계치 (심박수 >= 100 beats per minute, 체온 >= 100.4 °F)에 도달할 때 적절한 의료 인력에 알려주기 위해 alerts 토픽에 레코드를 보낼 것이다[2].

환자 모니터링 시스템의 아키텍처를 살펴보자. 그림 5-1은 이 장에서 구현할 토폴로지 설계를 보여준다. 각 단계에 대한 추가적인 정보는 그림 이후에 기술된다.

조치를 취할 수 있다. 이 모두가 시간 인식 스트림 처리를 통해 가능하다. 이 유스케이스를 깊게 살펴보기 위해 Tim Berglund의 Ramesh Sringeri와의 인터뷰를 참조하기 바란다.

[2] 카프카 스트림즈의 시간 중심 특징을 보여줄 필요가 없기 때문에 모니터링 애플리케이션을 실제 구현하지는 않을 것이다.

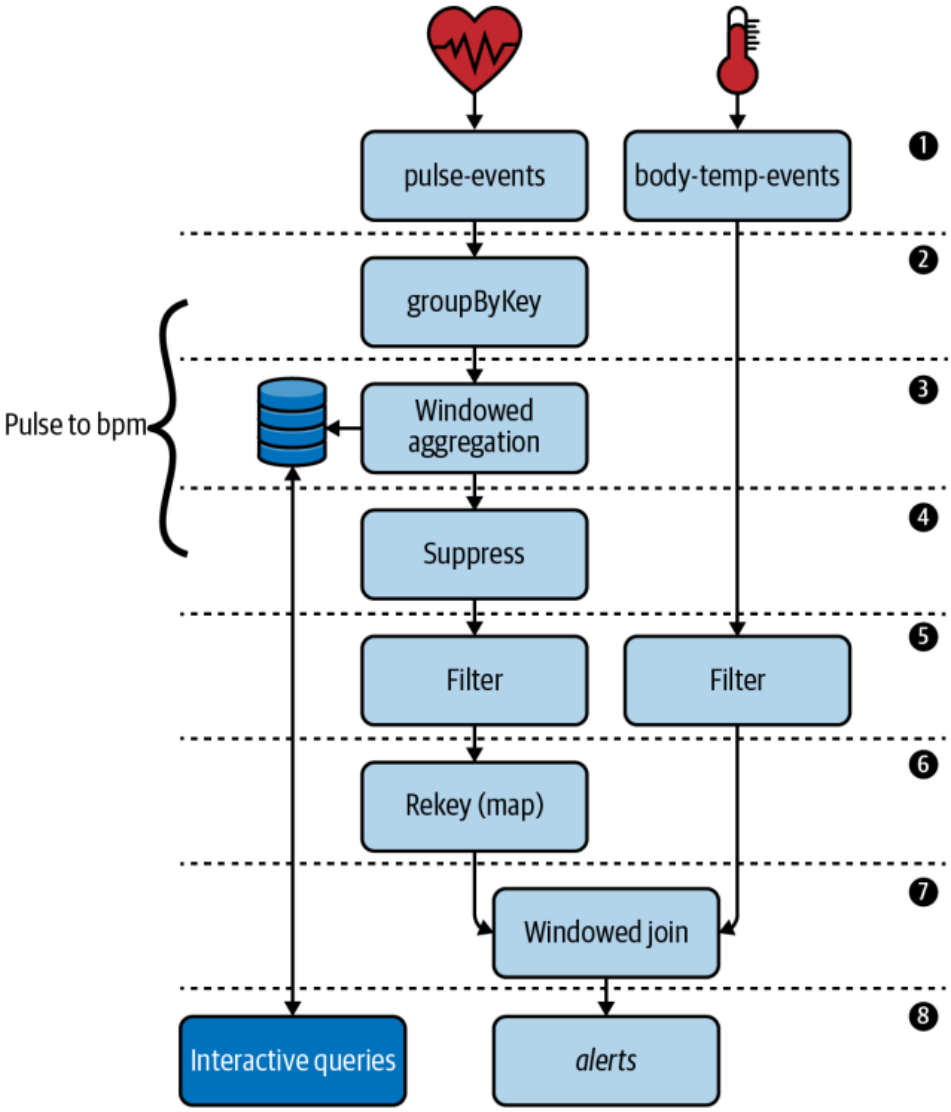

그림 5-1. 환자 모니터링 애플리케이션 구현을 위한 토폴로지

①카프카 클러스터는 환자의 바이탈 측정치를 수집하는 2 개의 토픽을 포함한다:

- Pulse-events 토픽은 심박수 센서에 의해 채워진다. 센서가 환자의 심박수를 수집하자 마자 이 토픽에 레코드를 추가한다. 레코드의 키는 환자 ID이다.
- Body-temp-events 토픽은 무선 체온 센서에 의해 채워진다. 환자의 주요 체온이 수집되자 마자 이 토픽에 레코드를 추가한다. 이 레코드의 키 또한 환자 ID이다.

② 심박수 증가를 탐지하기 위해 원시 펄스 이벤트를 심박수 (beats per minute, bpm을 사용해 측정)로 변환해야 한다. 이전 장에서 배웠듯이 집계 수행을 위한 카프카 스트림즈의 선결 조건을 충족시키기 위해 레코드를 우선 그룹화해야 한다.

③ 펄스 이벤트를 심박수로 변환하기 위해 윈도우를 갖는 집계를 사용할 것이다. 측정 단위가 bpm 이기 때문에 윈도우 크기는 60 초일 것이다.

④ Bpm 윈도우의 최종 계산 결과를 방출하기 위해 suppress 연산자를 사용할 것이다. 추후 장에서 논의한다면 이 연산자가 필요한 이유를 알 것이다.

⑤ 감염을 탐지하기 위해 사전 정의된 일련의 임계치들을 넘어서는 모든 바이탈 측정치를 필터링할 것이다 (심박수 >= 100 beats per minute, 체온 >= 100.4 °F).

⑥ 곧 살펴보겠지만 윈도우를 갖는 집계는 레코드 키를 변경시킨다. 따라서 레코드 조인을 위한 co-partitioning 요건을 충족시키기 위해 심박수 레코드를 환자 ID로 키를 설정해야 한다.

⑦ 2 개의 바이탈 스트림을 결합하기 위해 윈도우를 갖는 조인을 수행할 것이다. 증가된 bpm과 체온 측정치를 필터링 한 후 조인을 수행하기 때문에 각각 조인된 레코드는 SIRS에 대한 경고 조건을 나타낼 것이다.

⑧ 마지막으로 상호대화형 쿼리를 통해 심박수 집계 결과를 보일 것이다. 또한 조인된 스트림의 결과를 alerts 토픽에 쓸 것이다.

이 튜토리얼을 따라갈 수 있도록 프로젝트 환경설정을 통해 빠르게 실행시켜 보자.

프로젝트 환경 설정

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-05/patient-monitoring
```

다음 명령을 실행하여 언제라도 프로젝트를 빌드할 수 있다:

```
$ ./gradlew build –info
```

프로젝트 환경 설정이 끝났고 환자 모니터링 애플리케이션 구현을 시작해보자.

데이터 모델

평소와 같이 데이터 모델 정의로 시작할 것이다. 각 바이탈 측정치는 타임스탬프와 연관되어 있기 때문에 각 데이터 클래스가 구현할 간단한 인터페이스를 작성할 것이다. 이 인터페이스를 통해 일관된 방식으로 해당 레코드로부터 타임스탬프를 추출할 수 있으며 추후 타임스탬프 추출기를 구현할 때 이 인터페이스가 도움이 될 것이다. 다음 코드 블록은 각 데이터 클래스가 구현할 인터페이스를 보여준다:

```
public interface Vital {
    public String getTimestamp();
}
```

다음은 우리가 사용한 데이터 클래스이다. 비고: 액세스 메소드 (getTimestamp 포함)는 편의를 위해 생략되었다.

카프카 토픽	레코드 예	데이터 클래스

Pulse-events	{
	"timestamp": "2020-11-05T09:02:00.000Z"
	}
		public class Pulse implements Vital {
		private String timestamp;
		}

Body-temp-events	{
	"timestamp": "2020-11-04T09:02:06.500Z",
	"temperature": 101.2,
	"unit": "F"
	}
		public class BodyTemp implements Vital {
		private String timestamp;
		private Double temperature;
		private String unit;
		}

소스 데이터의 구조를 이해했는데 입력 스트림을 등록할 준비가 된 상태이다. 그러나 이 애플리케이션은 시간에 대해 특별한 주의가 필요한데 지금까지 이 책에서는 레코드를 타임스탬프와 연관시키는 방법에 대해서는 생각을 많이 하지 않았다. 따라서 입력 스트림 등록 전에 카프카 스트림즈에서의 다양한 시간 시맨틱을 살펴보자.

시간 시맨틱

카프카 스트림즈에는 몇 가지 다른 시간 개념이 존재하며 윈도우 기반 조인 및 집계를 포함하여 시간 기반 연산을 수행할 때 정확한 시맨틱을 선택하는 것이 중요하다. 이 절에서는 일부 정의로 시작하여 카프카 스트림즈에서의 다른 시간 개념을 이해할 것이다:

이벤트 타임

소스에서 이벤트가 생성된 시간. 이 타임스탬프는 이벤트 페이로드에 삽입되거나 또는 0.10.0 버전의 카프카 프로듀서 클라이언트를 사용하여 직접 설정될 수 있다.

인입 타임

이벤트가 카프카 브로커 토픽에 추가된 시간. 이는 항상 이벤트 타임 이후에 발생한다.

처리 타임

이벤트가 카프카 스트림즈 애플리케이션에 의해 처리된 시간. 이는 항상 이벤트 및 인입 타임 이후에 발생한다. 이는 이벤트 타임보다 덜 정적으로 (버그 수정 등을 위한) 동일 데이터 재처리는 새로운 처리 타임을 야기할 것이며 따라서 비결정적인 윈도우 작동을 야기할 것이다.

이러한 시간 개념을 이벤트 스트림에서 물리적으로 나타나는 것을 보기 위해 그림 5-2를 참조하기 바란다.

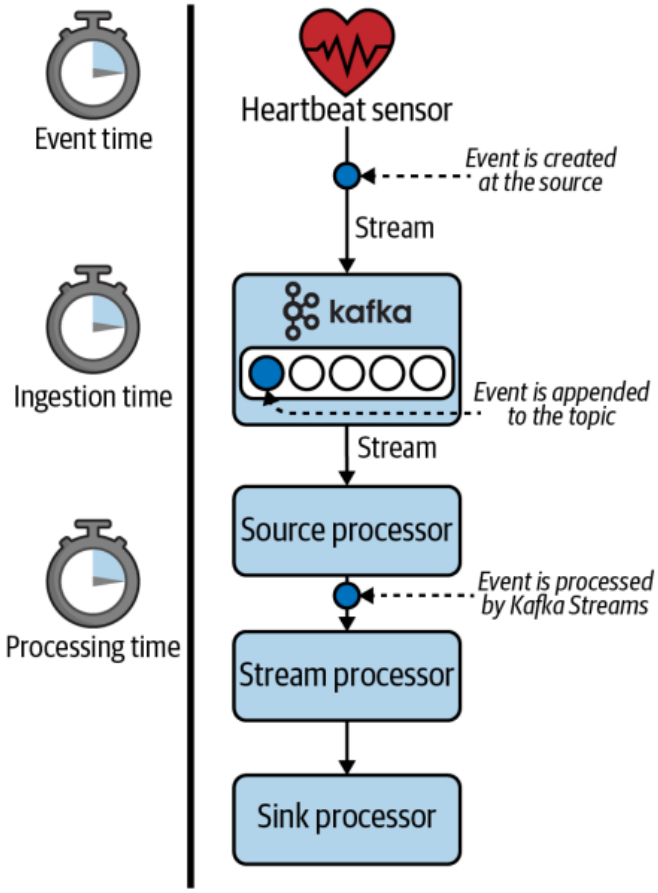

그림 5-2. 심박수 센서를 통해 보는 바와 같이 카프카 스트림즈에서 다른 타임 시맨틱

이벤트 타임은 실제 이벤트가 발생한 시간을 나타내기 때문에 아마도 가장 직관적인 타임 개념이다. 예를 들어 심박수 센서가 오전 9시 2분에 레코드를 읽는 경우 이벤트 타임은 오전 9시 2분이다.

이벤트 타임은 다음 코드 예에서 보듯이 일반적으로 페이로드에 삽입된다:

```
{
        "timestamp": "2020-11-12T09:02:00.000Z", ①
        "sensor": "smart-pulse"
}
```

① 추출해야 하는 삽입된 이벤트 타임스탬프

대안으로 카프카 프로듀서는 각 레코드에 대한 설정을 재정의하는 기본 타임스탬프를 허용하며 이는 이벤트-타임 시맨틱을 얻기 위해 사용될 수 있다. 그러나 이벤트와 타임스탬프를 연관시키는데 이 방법을 사용하는 시스템에 대해 실수로 인입-타임 시맨틱이 사용되지 않음을 보장하기 위해 2 가지 카프카 설정 (하나는 브로커 수준이고 다른 하나는 토픽 수준)을 아는 것이 중요하다. 관련 설정은 다음과 같다:

- log.message.timestamp.type (브로커 수준)
- message.timestamp.type (토픽 수준)

이 설정에는 2 가지 가능한 값이 존재한다: CreateTime 또는 LogAppendTime. 또한 토픽 수준 설정은 브로커 수준 설정을 우선한다. 토픽이 LogAppendTime 타임스탬프 타입으로 설정된 경우[3] 프로듀서가 메시지에 추가하는 타임스탬프는 레코드가 토픽에 추가될 때마다 브로커의 로컬 시스템으르 재정의될 것이다 (따라서 의도하지 않았지만 인입-타임 시맨틱으로 작업을 할 것이다). 이벤트-타임 시맨틱을 얻고 싶은데 프로듀서 타임스탬프에 의존하고 있다면 메시지 타임스탬프 타입으로 CreateTime을 사용하고 있음을 확인하기 바란다.

이벤트-타임 시맨틱을 사용하는 장점은 이 타임스탬프가 이벤트 자체에 대해 보다 의미있고 따라서 사용자에 대해 보다 직관적이라는 것이다. 이벤트 타임은 또한 시간 종속적인 연산을 결정론적으로 만들 수 있다 (예, 데이터 재처리 시). 처리 타임을 사용하는 경우 이는 해당되지 않는다. 처리 타임은 보통 시간 기반 연산을 활용하지 않거나 이벤트가 처리되는 시간이 이벤트가 최초 발생한 시간보다 애플리케이션 시맨틱에 보다 의미있거나 또는 어떤 이유로 이벤트와 타임스탬프를 연관시키지 않는 경우에 사용된다. 충분히 흥미로운데 이벤트 타임이 레코드와 연관될 수 없는 후자의 경우는 인입 타임을 사용함으로써 종종 다루어진다. 이벤트 생성 시간과 이벤트가 추후 토픽에 추가되는 시간 사이에 지연이 많이 없는 시스템에서 인입 타임이 이벤트 타임을 근사하기 위해 사용될 수 있으며 따라서 이벤트 타임을 사용할 수 없는 경우 대안으로 사용가능하다[4].

카프카 스트림즈에서 다른 시간 개념을 배웠는데 선택한 타임 시맨틱을 실제 어떻게 활용하는지? 다음 절에서 이를 배울 것이다.

타임스탬프 추출기

카프카 스트림즈에서 타임스탬프 추출기는 해당 레코드와 타임스탬프를 연관시키는 역할을 하는데 이 타임스탬프는 윈도우 조인 및 집계와 같은 시간 종속적인 연산에 사용된다. 타임스탬프 추출기 구현은 다음 인터페이스를 준수해야 한다:

```
public interface TimestampExtractor {
    long extract(
        ConsumerRecord<Object, Object> record, ①
        long partitionTime ②
    );
}
```

① 처리 중인 현재 컨슈머 레코드

---

[3] 토픽 수준 설정을 통해 직접 설정하거나 또는 브로커 수준 설정을 사용하여 간접적으로 설정. 토픽 수준 설정을 재정의하지 않음

[4] Matthias J. Sax는 이 부분과 이 장에서 다루는 다른 주제에 대해 논의한 훌륭한 발표 자료를 갖고 있다.

② 카프카 스트림즈는 소비하고 있는 각 파티션에 대해 본 가장 최근 타임스탬프를 추적하며 이 타임스템프를 partitionTime 파라미터를 사용하여 extract 메소드에 전달한다.

두번째 파라미터 partitionTime는 타임스탬프가 추출될 수 없는 경우 fallback으로 사용될 수 있기 때문에 흥미롭다. 곧 자세히 알아볼 것이지만 우선 카프카 스트림즈에 포함된 타임스탬프 추출기를 살펴보자.

포함된 타임스탬프 추출기

카프카 스트림즈의 기본 타임스탬프 추출기인 FailOnInvalidTimestamp는 컨슈머 레코드로부터 타임스탬프를 추출하는데 이벤트 타임(message.timestamp.type이 CreateTime인 경우) 또는 인입 타임(message.timestamp.type이 LogAppendTime인 경우) 둘 중 하나이다. 이 추출기는 타임스탬프가 유효하지 않은 경우 StreamsException 예외처리를 할 것이다. 타임스탬프는 음의 값 (0.10.0 이전의 메시지 포맷을 사용하여 레코드가 생산된 경우) 인 경우 유효하지 않은 것으로 간주된다. 이 책 작성 시점에서 0.10.0 버전 배포 이후 4년 이상이 지났고 따라서 음의 값/유효하지 않은 타임스탬프는 이 시점에서 점점 더 코너 케이스가 되고 있다 (여러 가지 변수와 환경의 복합적인 상호작용으로 발생하는 문제).

LogAndSkipOnInvalidTimestamp 추출기 또한 이벤트 타임 시맨틱을 얻기 위해 사용될 수 있으며 FailOnInvalidTimestamp 추출기와 달리 유효하지 않은 타임스탬프를 만날 때 단순히 경고를 기록한다. 이는 레코드를 건너 뛰어 카프카 스트림즈가 유효하지 않은 타임스탬프를 만날 때 계속 처리를 할 수 있도록 한다.

처리 타임 시맨틱을 원하는 경우 사용할 수 있는 다른 내장 추출기도 존재한다. 다음 코드에서 볼 수 있듯이 WallclockTImestampExtractor는 스트림 처리 애플리케이션의 로컬 시스템을 반환한다:

```
public class WallclockTimestampExtractor implements TimestampExtractor {
    @Override
    public long extract(
      final ConsumerRecord record,
      final long partitionTime
       ) {
        return System.currentTimeMillis(); ①
      }
}
```

① WallclockTimestampExtractor은 카프카 스트림즈의 내장 타임스탬프 추출기 중 하나로 단순히 현재 시스템 타임스탬프를 반환한다.

어떤 타임스탬프 추출기를 사용하더라도 예제 5-1에서 보듯이 DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG 특성을 설정함으로써 기본 타임스탬프 추출기를 재정의할 수 있다.

예제 5-1. 카프카 스트림즈의 기본 타임스탬프 추출기 재정의 방법 예

```
Properties props = new Properties();
props.put( ①
   StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG, WallclockTimestampExtractor.class );
// ... other configs
KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

① 기본 타임스탬프 추출기 재정의

카프카 스트림즈 애플리케이션이 환자 모니터링 애플리케이션과 같이 윈도우를 활용할 때 처리 타임 시맨틱을 사용하는 것은 의도치 않은 부작용을 야기할 수 있다. 예를 들어 1분 윈도우 내에서 심박수를 수집하길 원한다고 해보자. 윈도우 집계에 처리 타임 시맨틱 (예, WallclockTimestampExtractor을 통해)을 사용한다면 윈도우 경계는 전혀 펄스 타임을 나타내지 않으며 대신 카프카 스트림즈 애플리케이션이 펄스 이벤트를 관측한 시간을 나타낼 것이다. 애플리케이션이 몇 초의 지연을 경험하더라도 이벤트는 의도한 윈도우 밖에 놓이며 따라서 특정 방식으로 우리의 예상에 영향을 미칠 수 있다 (예, 증가된 심박수 탐지 능력).

타임스탬프가 추출되어 추후 레코드와 연관될 때 레코드가 찍혔다고 한다.

내장 타임스탬프 추출기를 검토한 후 우리의 경우 각 바이탈 데이터 (펄스와 체온 이벤트)의 페이로드에 이벤트 타임이 내장되기 때문에 맞춤형 타임스탬프 추출기가 필요함이 명확하다. 다음 절에서 맞춤형 타임스탬프 추출기 구축 방법을 논의할 것이다.

맞춤형 타임스탬프 추출기

이벤트 타임 시맨틱이 필요하고 이벤트 타임스탬프가 레코드 페이로드에 내장될 때 맞춤형 타임스탬프 추출기를 구축하는 것은 일반적이다. 다음 코드 블록은 환자 바이탈 측정치의 타임스탬프를 추출하기 위해 사용할 맞춤형 타임스탬프 추출기를 보여준다. 이전에 언급했듯이 맞춤형 타임스탬프 추출기는 카프카 스트림즈에 포함된 TimestampExtractor을 구현한다.

```
public class VitalTimestampExtractor implements TimestampExtractor {
  @Override
  public long extract(ConsumerRecord record, long partitionTime) {
    Vital measurement = (Vital) record.value(); ①
    if (measurement != null && measurement.getTimestamp() != null) { ②
      String timestamp = measurement.getTimestamp(); ③
      return Instant.parse(timestamp).toEpochMilli(); ④
    }
    return partitionTime; ⑤
  }
}
```

① 레코드 값을 Vital 객체로 캐스팅한다. 이는 Pulse와 BodyTemp 레코드로부터 일관된 방식으로 타

임스탬프를 추출하도록 하기 때문에 인터페이스가 유용한 경우이다.

② 레코드를 파싱하기 전에 타임스탬프를 포함하고 있는지를 확인한다.

③ 레코드로부터 타임스탬프를 추출한다.

④ TimestampExtractor.extract 메소드는 밀리초 단위의 레코드 타임스탬프를 반환한다. 따라서 여기서 타임스탬프의 밀리초 변환을 수행한다.

타임스탬프 추출기를 사용할 때 고려해야 할 한 가지는 유효한 타임스탬프가 없는 레코드를 어떻게 처리해야 하는 지이다. 3 가지 가장 일반적인 옵션은 다음과 같다:

- 예외처리 후 처리를 중지한다 (개발자에게 버그 해결 기회 제공)
- 파티션 타임으로 fallback
- 음의 타임스탬프를 반환하여 카프카 스트림즈가 레코드를 건너 뛰어 계속 처리를 할 수 있도록 한다.

VitalTimestampExtractor 구현에서는 파티션 타임으로 fallback 하기로 결정하였으며 이는 현재 파티션에 대해 이미 관측되었던 가장 최근 타임스탬프를 사용할 것이다.

타임스탬프 추출기를 생성하였고 입력 스트림을 등록해보자.

타임스탬프 추출기를 사용해 스트림 등록하기

일련의 입력 스트림 등록은 이제 익숙할 것이지만 현재는 타임스탬프 추출기를 맞춤형 타임스탬프 출기 구현 (VitalTimestampExtractor)을 명시적으로 설정할 Consumed 파라미터로 전달할 것이다. 예제 5-2는 맞춤형 타임스탬프 추출기를 사용하여 (그림 5-1의) 프로세서 토폴로지의 첫번째 단계인 2 개의 소스 스트림 등록 방법을 보여준다.

예제 5-2. 소스 스트림에 대한 타임스탬프 추출기 재정의 방법 예

```
StreamsBuilder builder = new StreamsBuilder(); ①
Consumed<String, Pulse> pulseConsumerOptions =
  Consumed.with(Serdes.String(), JsonSerdes.Pulse())
          .withTimestampExtractor(new VitalTimestampExtractor()); ②
KStream<String, Pulse> pulseEvents =
  builder.stream("pulse-events", pulseConsumerOptions); ③
Consumed<String, BodyTemp> bodyTempConsumerOptions =
  Consumed.with(Serdes.String(), JsonSerdes.BodyTemp())
          .withTimestampExtractor(new VitalTimestampExtractor()); ④
KStream<String, BodyTemp> tempEvents =
  builder.stream("body-temp-events", bodyTempConsumerOptions); ⑤
```

① 항상 DSL에서와 같이 프로세서 토폴로지 생성을 위한 StreamsBuilder을 사용

② 카프카 스트림즈에 바이탈 타임스탬프 추출을 위해 맞춤형 타임스탬프 추출기 (VitalTimestampExtractor)를 사용하라고 지시하기 위해 Consumed.withTimestampExtractor 사용

③ 펄스 이벤트 수집을 위한 스트림 등록

④ 체온의 경우에도 맞춤형 타임스탬프 추출기 사용

⑤ 체온 이벤트 수집을 위한 스트림 등록

대안으로 예제 5-1의 메소드를 사용하여 기본 타임스탬프 추출기를 재정의 할 수도 있다. 각각의 방법 모두 좋지만 지금은 각 입력 스트림의 추출기를 직접 설정하는 것을 택할 것이다. 소스 스트림을 등록하였고 프로세서 토폴로지 (그림 5-1)의 두번째 및 세번째 단계를 진행해보자: pulse-events 스트림 그룹화 및 윈도잉

스트림 윈도잉

환자의 심박이 기록될 때마다 pulse-events 토픽은 데이터를 수신한다. 그러나 모니터링 목적을 위해서는 심박수에 관심이 있으며 이는 bpm (beats per minute)에 의해 측정된다. 심박수를 세기 위해 count 연산자가 사용될 수 있음을 알고 있지만 매 60 초 윈도우 내의 레코드만 세는 방법이 필요하다. 이에는 윈도우 집계가 사용된다. 윈도잉은 집계 및 조인을 위한 목적으로 다른 시간 기반 서브 그룹으로 레코드를 그룹화하는 방법이다. 카프카 스트림즈는 몇몇 다른 유형의 윈도우를 지원하며 따라서 환자 모니터링 시스템에 어떤 구현이 필요한지를 결정하기 위해 각 유형을 살펴보자.

윈도우 타입

윈도우는 시간적으로 근접한 레코드를 그룹화하는데 사용된다. 시간적 근접성은 어떤 타임 시맨틱을 사용하는 지에 따라 다른 것을 의미할 수 있다. 예를 들어 이벤트 타임 시맨틱의 경우 이는 동일 시간대에 발생한 레코드를 의미하며 반면 처리 타임 시맨틱의 경우 동일 시간대에 처리된 레코드를 의미한다. 대부분의 경우 동일 시간대는 윈도우 크기 (예, 5분, 1시간 등)에 의해 정의되며 (간략하게 논의할)세션 윈도우의 경우 활동 기간을 활용한다..

카프카 스트림즈에는 4 가지 다른 유형의 윈도우가 존재한다. 다음에 각 윈도우 유형의 특성을 논의하며 이 특성을 사용하여 튜토리얼에서 어떤 윈도우를 사용할지를 결정하기 위해 의사결정 트리를 만들 것이다 (여러분의 애플리케이션에서 의사결정 트리를 사용할 수도 있다).

텀블링 윈도우 (Tumbling Window)

텀블링 윈도우는 전혀 중복되지 않는 고정 크기의 윈도이다. 이는 단일 특성인 밀리초 단위의 윈도우 크기에 의해 정의되며 이포크와 정렬되기 때문에 예측가능한 시간 범위를 갖는다[5]. 다음 코드는 카프

[5] Java docs에서 발췌: 이포크에 정렬한다는 것은 첫번째 윈도우가 타임스탬프 0으로 시작함을 의미한다. 다른 말로 윈도우 크기 5,000의 경우 0-5,000, 5,000-10,000 등의 경계를 가질 것이다. 시작 시

카 스트림즈에서 텀블링 윈도우를 생성하는 방법을 보여준다:

```
TimeWindows tumblingWindow =
 TimeWindows.of(Duration.ofSeconds(5)); ①
```

① 윈도우 크기는 5초이다.

그림 5-3에서 보듯이 텀블링 윈도우는 시각적으로 추론하기 쉽다. 윈도우 경계 중복 또는 한 타임 버킷 이상에 나타나는 레코드에 대해 걱정할 필요가 없다.

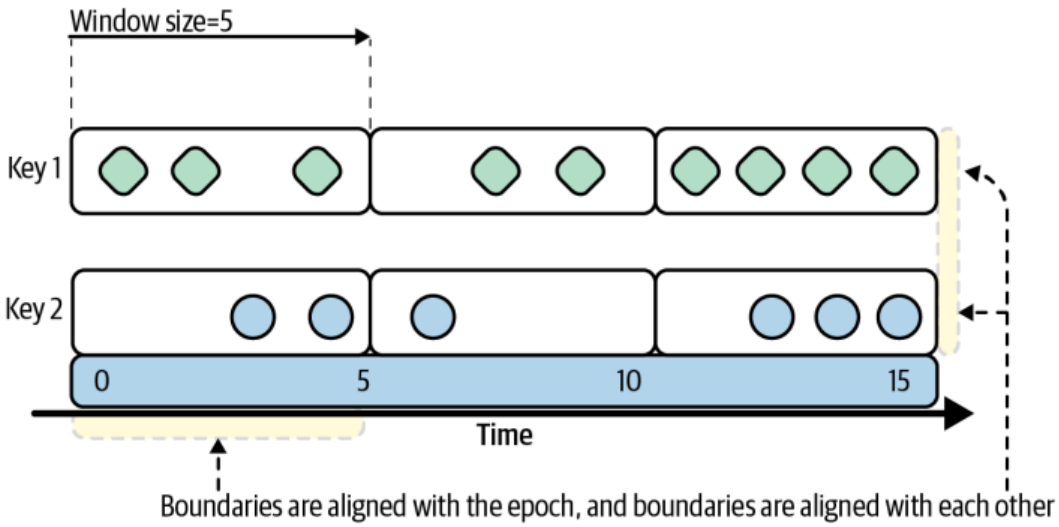

그림 5-3. 텀블링 윈도우

호핑 윈도우 (Hopping Window)

호핑 윈도우는 중복될 수 있는 고정 크기의 윈도우이다[6]. 호핑 윈도우를 설정할 때는 윈도우 크기와 전진 간격 (윈도우가 앞으로 얼마나 이동할지) 모두를 지정해야 한다. 그림 5-4와 같이 전진 간격이 윈도우 크기보다 작을 때 윈도우는 중복되고 일부 레코드는 복수 윈도우에서 나타날 수 있다. 더구나 호핑 윈도우는 이포크와 정렬되기 때문에 예측가능한 시간 범위를 가지며 시작 시간은 포함되고 끝 시간은 포함되지 않는다. 다음 코드는 카프카 스트림즈에서 간단한 호핑 윈도우 생성 방법을 나타낸다:

```
TimeWindows hoppingWindow = TimeWindows
  .of(Duration.ofSeconds(5)) ①
  .advanceBy(Duration.ofSeconds(4)); ②
```

① 윈도우 크기는 5초이다.

간은 포함되지만 끝 시간은 포함되지 않음을 주목하기 바란다.

[6] 일부 시스템에서는 호핑 윈도우를 언급하기 위해 슬라이딩 윈도우를 사용한다. 그러나 카프카 스트림즈에서 호핑 윈도우는 슬라이딩 윈도우와 다르다.

② 윈도우는 4 초 전진 간격 (홉)을 갖는다.

그림 5-4에 호핑 윈도우를 시각화해서 나타냈다.

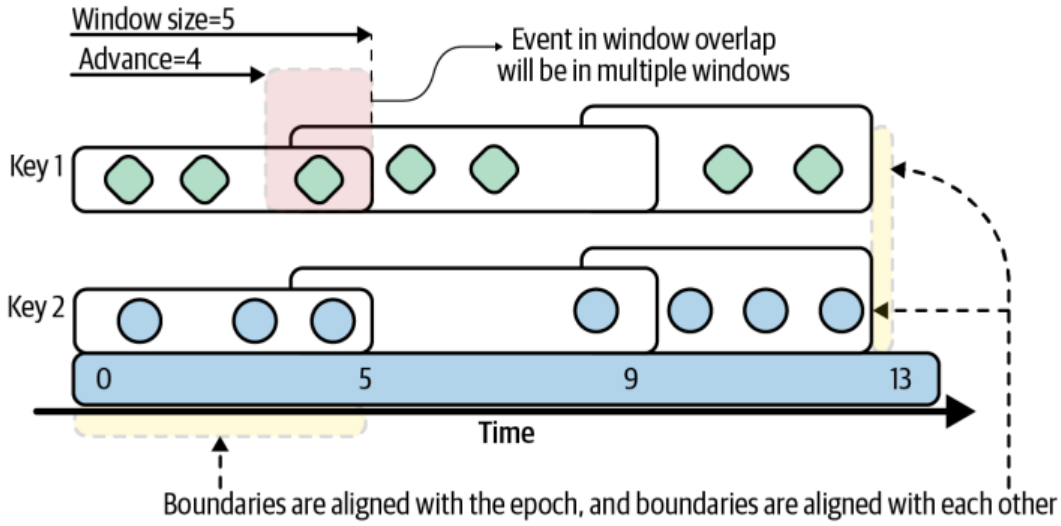

그림 5-4 호핑 윈도우

세션 윈도우 (Session Window)

세션 윈도우는 활동 기간과 비활동 갭에 의해 결정되는 가변 크기 윈도우이다. 세션 윈도우를 정의하기 위해 비활동 갭이라는 단일 파라미터가 사용된다. 비활동 갭이 5초라면 동일 키를 갖는 이전 레코드에 대해 5 초 내의 타임스탬프를 갖는 각 레코드는 동일 윈도우로 합쳐질 것이다. 그렇지 않고 새로운 레코드의 타임스탬프가 비활동 갭보다 크다면 (이 경우 5초) 새로운 윈도우가 생성될 것이다. 텀블링 및 호핑 윈도우와 달리 하한 및 상한 경계 모두 포함된다. 다음 코드 스니펫은 세션 윈도우 정의 방법을 보여준다:

```
SessionWindows sessionWindow = SessionWindows
  .with(Duration.ofSeconds(5)); ①
```

① 비활동 갭 5초의 세션 윈도우

그림 5-5에서 보듯이 세션 윈도우는 정렬되지 않고 (범위가 각 키에 특정적) 크기가 변한다. 범위는 레코드 타임스탬프에 전적으로 의존하며 활동이 많은 키는 긴 윈도우 범위 그리고 덜 활동적인 키는 보다 짧은 윈도우 범위를 야기한다.

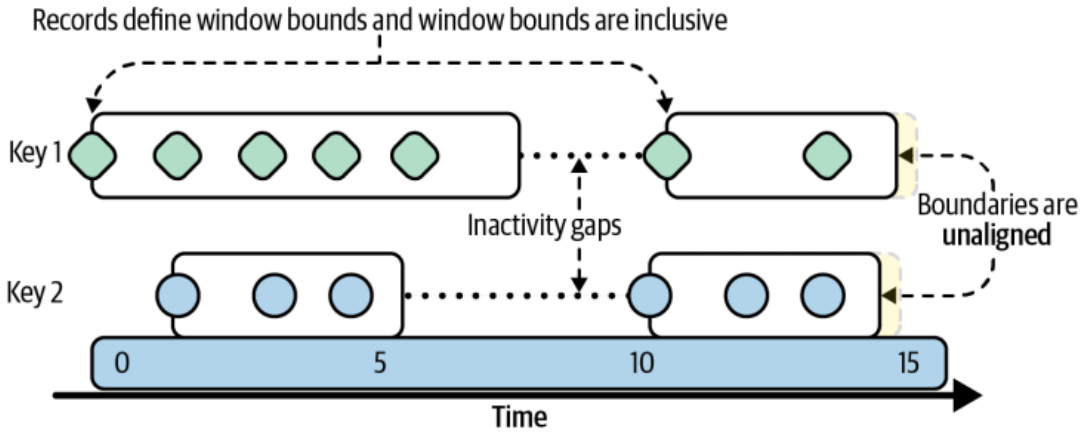

그림 5-5. 세션 윈도우

슬라이딩 조인 윈도우 (Sliding Join Window)

슬라이딩 조인 윈도우는 조인에 사용되는 고정 크기 윈도우로 JoinWiIndows 클래스를 사용해 생성된다. 2 개 레코드의 경우 그들 간 타임스탬프가 윈도우 크기 이하면 동일 윈도우 내에 들어간다. 따라서 세션 윈도우와 비슷하게 상한 및 하한 경계 모두 포함된다. 다음은 5 초 간격의 조인 윈도우를 생성하는 법의 예이다:

```
JoinWindows joinWindow = JoinWindows
  .of(Duration.ofSeconds(5)); ①
```

① 동일 윈도우 내에 들어가기 위해 타임스탬프는 5초 이하로 떨어져 있어야 한다.

조인 윈도우는 복수의 입력 스트림을 포함할 수 있도록 조인에 사용되기 때문에 시각적으로 조인 윈도우는 다른 유형의 윈도우와는 다르게 보인다. 그림 5-6은 이를 보여주는데 어떤 레코드가 윈도우 조인에 결합되는지를 결정하기 위해 5초 윈도우가 어떻게 사용되는지를 보여준다.

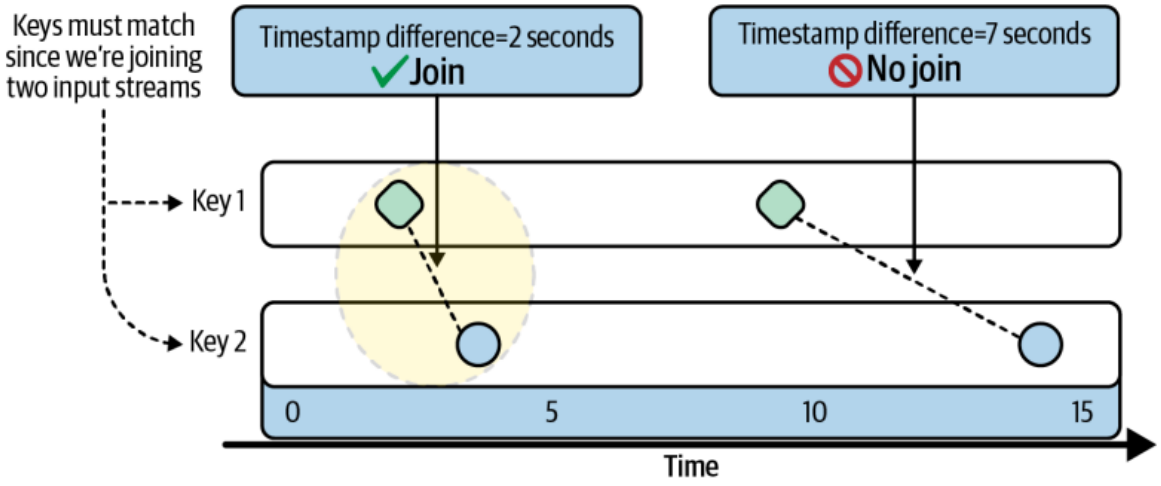

그림 5-6. 조인 윈도우

슬라이딩 집계 윈도우 (Sliding Aggregation Window)

이전 절에서 조인에 사용되는 특수한 유형의 슬라이딩 윈도우를 논의하였다. 카프카 스트림즈 2.7.0 이후 슬라이딩 윈도우는 집계에도 사용될 수 있다. 슬라이딩 조인 윈도우와 같이 슬라이딩 집계 윈도우에서 윈도우 경계는 (이포크가 아닌) 레코드 타임스탬프에 정렬되어 있고 상한 및 하한 경계 모두 포함된다. 이외에 레코드 간 타임스탬프가 지정된 윈도우 크기 내에 존재한다면 동일 윈도우 내로 들어갈 것이다. 다음은 5초 간격과 0 초의 유예 기간을 갖는 슬라이딩 윈도우 생성 방법의 예이다:[7]

```
SlidingWindows slidingWindow = SlidingWindows
  .withTimeDifferenceAndGrace(
    Duration.ofSeconds(5),
    Duration.ofSeconds(0));
```

윈도우 선택하기

이제 카프카 스트림즈에서 지원되는 윈도우 타입을 배웠는데 윈도우 집계를 사용해 원시 펄스 이벤트를 심박수로 변환하는데 어떤 유형이 적합한지 결정해야 한다.

세션 윈도우는 스트림에 활동이 있다면 윈도우 크기가 무한정 확장될 수 있기 때문에 심박수 측정에 적합하지 않다. 이는 60초의 고정 크기 윈도우를 가져야 한다는 요건을 충족시키지 못한다. 또한 슬라이딩 조인 윈도우는 조인을 위해서만 사용되며 따라서 이 또한 배제할 수 있다 (이 튜토리얼에서 추후 슬라이딩 조인 윈도우를 사용할 것이지만).

집계에 어떤 윈도우 유형도 사용할 수 있지만 문제를 단순화하기 위해 윈도우 경계를 이포크에 정렬시키고 (슬라이딩 집계 윈도우 배제) 중복 윈도우 (호핑 윈도우를 필요로 하지 않음을 의미)를 피해보자. 이제 심박수 집계를 위해서는 텀블링 윈도우만 남았다. 윈도우 유형이 선택되었으며 윈도우 집계를 수행할 준비가 된 것이다.

윈도우 집계

심박수 집계에 텀블링 윈도우를 사용하기로 결정했기 때문에 윈도우 집계를 구축할 수 있다. 우선 TimeWindows.of 메소드를 사용하여 윈도우를 만들고 다음에 집계 수행 전 windowedBy 연산자를 사용하여 스트림을 윈도잉할 것이다. 다음 코드는 이 둘 모두를 달성할 방법을 보여준다:

```
TimeWindows tumblingWindow =
  TimeWindows.of(Duration.ofSeconds(60));
KTable<Windowed<String>, Long> pulseCounts =
 pulseEvents
  .groupByKey() ①
  .windowedBy(tumblingWindow)  ②
  .count(Materialized.as("pulse-counts")); ③
```

---

[7] 슬라이딩 집계 윈도우는 명시적으로 유예 기간 설정이 필요한 유일한 윈도우이다. 이 장 후반부에 유예 기간을 논의할 것이다.

```
    pulseCounts
      .toStream() ④
      .print(Printed.<Windowed<String>, Long>toSysOut().withLabel("pulse-counts"));
```

① 레코드 그룹화는 집계 수행의 선결 조건이다. 그러나 pulse-events 토픽 내 레코드가 원하는 스킴(예, 환자 ID)을 사용하여 이미 키가 부여되어 있으며 따라서 불필요한 데이터 재분배를 피하기 위해 groupBy 대신 groupByKey를 사용할 수 있다.

② 60초 텀블링 윈도우를 사용하여 스트림을 윈도잉한다. 이를 통해 원시 펄스 이벤트를 bpm으로 측정되는 심박수로 변환할 수 있다.

③ 상호대화형 쿼리를 사용하여 심박수를 구체화한다 (이는 프로세서 토폴로지의 8 단계에서 필요, 그림 5-1)

④ 디버깅 목적을 위해 KTable을 스트림으로 변환하며 따라서 내용을 컨솔로 출력할 수 있다.

⑤ 윈도우 스트림의 내용을 컨솔로 출력한다. Print 문은 로컬 개발에 유용하지만 애플리케이션을 제품 단계로 배치하기 전에 제거되어야 한다.

이 코드에서 강조할 한 가지 흥미로운 사실은 KTable의 키가 String에서 Windowed<String>으로 변경되었단 것이다. 이는 windowedBy 연산자가 KTables을 윈도우 KTables로 변환하였기 때문으로 원래 레코드 키 뿐만 아니라 윈도우의 시간 범위를 포함하는 다차원 키도 갖는다. 이는 키를 서브 그룹 (윈도우)으로 그룹화하는 방법이 필요하고 단순히 원래 키를 사용한다면 모든 펄스 이벤트가 동일 서브 그룹에 포함되기 때문에 의미가 있다. 우리는 몇몇 레코드를 pulse-events 토픽으로 생산하고 윈도우 스트림의 출력된 결과를 검토함으로써 이러한 다차원 키 변환이 동작함을 볼 수 있다. 우리가 생산할 레코드는 다음과 같다 (레코드 키와 값은 | 문자에 의해 구분된다):

```
1|{"timestamp": "2020-11-12T09:02:00.000Z"}
1|{"timestamp": "2020-11-12T09:02:00.500Z"}
1|{"timestamp": "2020-11-12T09:02:01.000Z"}
```

이 레코드의 생산은 예제 5-3의 출력을 야기한다. 레코드 각각에 대한 이전 키 1이 다음 포맷으로 변경되었음을 주목하기 바란다:

```
[<oldkey>@<window_start_ms>/<window_end_ms>]
```

예제 5-3. Print 연산자의 출력. 윈도우 pulseCounts 테이블의 다차원 키를 보여준다.

```
[pulse-counts]: [1@1605171720000/1605171780000], 1  ①
[pulse-counts]: [1@1605171720000/1605171780000], 2
[pulse-counts]: [1@1605171720000/1605171780000], 3
```

① 다차원 레코드 키, 원래 키 뿐만아니라 윈도우 경계 또한 포함한다.

키 변환 로직과 별개로 이전 출력은 카프카 스트림즈의 특이한 동작을 강조한다. 윈도우 집계에서 계산한 심박수 카운트가 새로운 심박이 기록될 때마다 업데이트된다. 예제 5-3에서 이를 볼 수 있는데,

첫번째 심박수는 1이고 다음은 2, 3 등이다. 따라서 다운스트림 연산자는 윈도우의 최종 결과 (bpm) 뿐만 아니라 윈도우의 중간 결과 (지금까지 60초 윈도우 내 심박수)를 볼 것이다. 저지연에 최적화된 애플리케이션에서 이러한 중간 또는 불완전환 윈도우 결과는 유용하다. 그러나 이 경우 윈도우 끝에 도달할 때까지 환자의 삼박수를 실제 모르며 따라서 오해의 소지가 있다. 카프카 스트림즈가 왜 각 중간 결과를 방출하는지를 살펴보고 이 동작을 조정할 수 있는지를 보자.

윈도우 결과 방출

윈도우 계산을 언제 방출할지에 대한 결정은 스트림 처리 시스템의 경우 매우 복잡하다. 복잡성은 2가지 사실에 기인한다:

- 무제한의 이벤트 스트림은 이벤트 타임 시맨틱을 사용할 지라도 항상 타임스탬프의 순서가 맞지 않을 수도 있다[8]

카프카는 파티션 수준에서 이벤트가 항상 오프셋 순서대로 있음을 보장한다. 이는 모든 컨슈머가 토픽에 추가된 동일 순서대로 이벤트를 읽을 것임을 의미한다 (오프셋 값 오름차순).

- 이벤트는 종종 지연된다.

타임스탬프 순서가 없다는 것은 우리가 특정 타임스탬프를 포함하는 레코드를 봤다 하더라도 그 타임스탬프 이전에 도착해야 하는 모든 레코드를 보았고 따라서 우리가 생각하기에 최종 윈도우 결과를 방출할 수 있다고 가정할 수 없음을 의미한다. 더구나 지연되거나 순서가 뒤바뀐 데이터는 선택이 필요하다: 모든 데이터가 도착할 때까지 특정 시간을 기다릴지 또는 업데이트될 때마다 윈도우 결과를 출력할지 (예제 5-3에서 보았듯이)? 이는 완전성과 지연 간 트레이드 오프이다. 데이터를 기다리는 것이 보다 완벽한 결과를 산출할 것 같기 때문에 이 방법은 완전성을 최적화한다. 반면 (완전하지 않지만) 업데이트를 즉시 다운스트림으로 업데이트하는 것은 지연을 줄인다. 무엇을 최적화할 지는 당신과 아마도 서비스 수준 협의 (Service Level Agreement, SLA)에 달려있다.

그림 5-7은 이러한 문제 모두가 어떻게 발생하는지를 보여준다. 환자 #1에는 간헐적으로 네트워킹 문제가 있는 바이탈 모니터링 머신이 부착되어 있다. 이는 프로듀서 재시도를 야기하며 일부 바이탈 측정치가 카프카 클러스터에 지연되어 도착하도록 한다. 더구나 pulse-events 토픽에 복수의 프로듀서가 쓰고 있기 때문에 이는 또한 이벤트 타임스탬프 측면에서 일부 이벤트의 순서가 없음을 야기한다.

---

[8] 물론 message.timestamp.type을 LogAppendTime으로 설정하여 인입 타임 시맨틱을 사용한다면 레코드 타임스탬프는 항상 순서가 맞을 것이다. 이는 타임스탬프가 토픽 추가 시간을 덮어쓰기 때문이다. 그러나 인입 타임은 이벤트 자체에 대해 항상 임의적이고 최상 시나리오의 경우 이벤트 타임을 단지 가깝게 근사한다 (이벤트가 생성 후 바로 쓰여진다고 가정할 때).

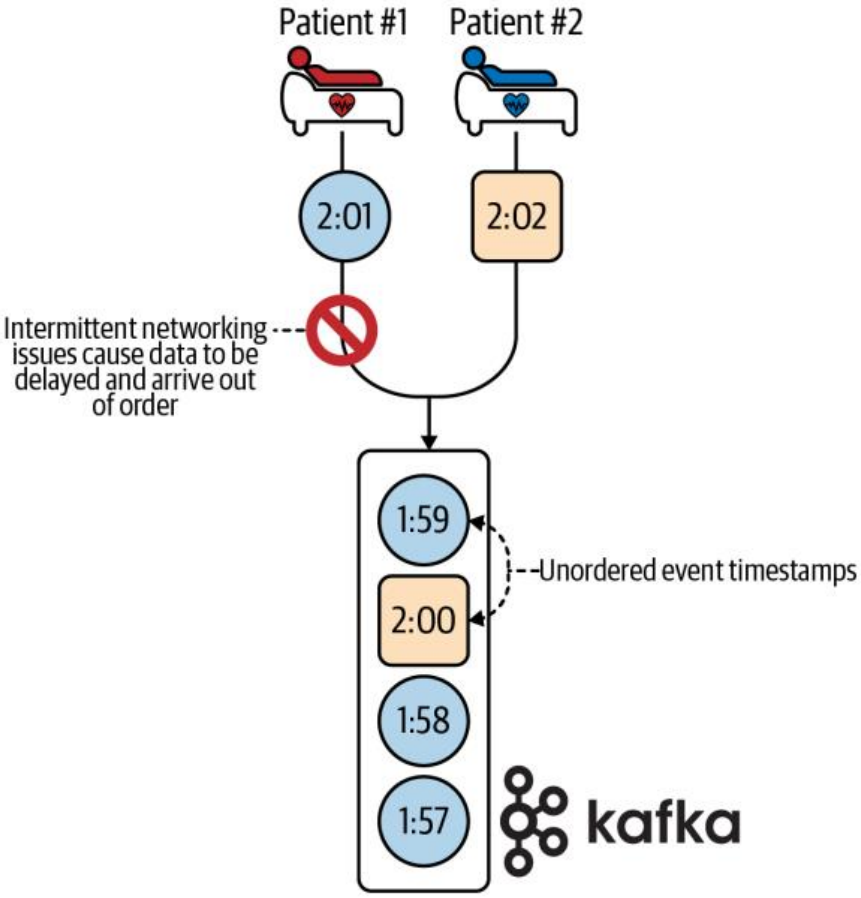

그림 5-7. 많은 이유로 인해 타임스탬프 순서가 맞지 않게 데이터가 도착할 수 있다; 한 가지 공통적인 예는 경쟁 조건을 야기하는 복수의 프로듀서를 갖는 토픽이다.

주목해야 할 한 가지 중요한 점은 순서가 잘못된 데이터를 야기하기 위해 일부 실패 시나리오가 필요치 않다는 것이다. 이는 복수의 프로듀서가 동일 토픽에 쓰려고 할 때와 같이 정상적인 운영 상황에서도 발생할 수 있다.

이전에 언급했듯이 이러한 문제를 해결하기 위해 우리는 지연 또는 완전성에 대해 최적화해야 한다. 기본적으로 카프카 스트림즈는 지속적인 개선이라는 방법을 사용하여 지연에 대해 최적화한다. 지속적인 개선은 새로운 이벤트가 윈도우에 추가될 때마다 카프카 스트림즈가 즉시 새로운 계산을 방출할 것임을 의미한다. 이것이 환자 모니터링 애플리케이션에 일부 레코드를 생산했을 때 각각의 중간 윈도우 결과를 본 이유이다 (예제 5-3). 그러나 지속적인 개선을 사용하더라도 각 결과는 잠정적으로 불완전하게 보여 지며 방출된 이벤트가 최종적으로 윈도우에 들어갈 모든 레코드를 처리했음을 의미하는 것은 아니다. 더구나 지연된 데이터는 예기치 못한 시간에 이벤트가 계속 방출되도록 할 수 있다.

다음 두 절에서는 환자 모니터링 애플리케이션에 이러한 문제 각각을 어떻게 다룰지를 논의한다. 우선 카프카 스트림즈에서 지연된 데이터를 다루는 전략을 살펴보자. 그 후 카프카 스트림즈의 suppress 연산을 사용하여 중간 윈도우 계산을 막는 방법을 배울 것이다.

유예 기간

스트림 처리 시스템이 직면한 가장 큰 문제 중 하나는 지연 데이터 처리 방법이다. 영향력있는 데이터 플로우 모델 (예, 아파치 Flink)을 고수하는 프레임워크를 포함하여 많은 프레임워크는 워터마크를 활용한다. 워터 마크는 주어진 윈도우에 대해 모든 데이터가 언제 도착해야 하는 지를 평가하기 위해 사용된다 (보통 이벤트에 대한 윈도우 크기와 허용된 지연을 설정). 그 후 사용자는 (워터마크에 의해 결정된) 늦은 이벤트가 어떻게 처리되어야 하는지를 지정해야 하며 보통 (Dataflow, Flink 등에서) 기본은 늦은 이벤트를 버리는 것이다.

워터마크 접근방법과 비슷하게 카프카 스트림즈는 유예 기간을 사용하여 이벤트의 허용 지연을 설정할 수 있다. 윈도우에 대해 지연된/순서없는 이벤트를 허용하기 위해 유예 기간 설정은 특정 시간 동안 윈도우를 계속 열려 있는 상태로 유지할 것이다. 예를 들어 다음 코드를 사용하여 텀블링 윈도우를 초기에 설정했다:

```
TimeWindows tumblingWindow =
  TimeWindows.of(Duration.ofSeconds(60));
```

그러나 (심박수를 계산하기 위해 사용할) 펄스 이벤트에 대해 5 초의 지연을 견딜 수 있길 원하는 경우 유예 기간을 갖는 텀블링 윈도우를 정의할 수 있다:

```
TimeWindows tumblingWindow =
  TimeWindows
  .of(Duration.ofSeconds(60))
  .grace(Duration.ofSeconds(5));
```

유예 기간을 더 늘릴 수 있지만 트레이드 오프를 기억하기 바란다: 큰 유예 기간은 윈도우를 더 길게 열려 있는 상태로 유지하기 때문에 (데이터의 지연을 허용) 유예 기간이 클수록 완전성에 대해 최적화하지만 보다 큰 지연을 희생해야 한다 (유예 기간이 끝나야 윈도우가 닫힐 것이다).

이제 윈도우 심박수 집계에서 방출되는 중간 결과 문제를 해결하는 방법을 살펴보자.

억제

이전 절에서 배웠듯이 새로운 데이터가 도착할 때마다 윈도우의 결과 방출을 포함하는 카프카 스트림즈의 지속적인 개선 전략은 저지연에 대해 최적화할 때 이상적이며 윈도우로부터 방출되는 불완전한 (즉, 중간) 결과를 견딜 수 있다[9].

그러나 환자 모니터링 애플리케이션에서 이는 바람직하지 않다. 우리는 60초 미만 데이터를 사용하여 심박수를 계산할 수 없으며 따라서 윈도우의 최종 결과만 방출하면 된다. 여기서 suppress 연산자가 작동한다. Suppress 연산자는 윈도우에 대해 최종 계산만 방출하고 모든 다른 이벤트를 억제 (즉 메모리에 중간 계산을 일시적으로 저장)하는데 사용될 수 있다. Suppress 연산자를 사용하기 위해 3 가지를 결정해야 한다:

---

[9] 이는 윈도우가 닫힐 때에만 처리가 발생하는 많은 다른 스트리밍 시스템이 동작하는 것과 다르다.

- 중간 윈도우 계산을 억제하기 위해 사용해야 하는 억제 전략
- 억제된 이벤트를 버퍼링하기 위해 사용해야 하는 메모리 (Buffer Config를 사용하여 설정)
- 이 메모리 한계가 초과될 때 취하는 조치 (이는 Buffer Full 전략을 사용하여 제어)

우선 2 가지 억제 전략을 살펴보자. 표 5-1은 카프카 스트림즈에서 사용가능한 각 전략을 설명한다. 각각의 전략이 Suppressed 클래스의 메소드로 사용가능함을 주목하기 바란다.

표 5-1. 윈도우 억제 전략

전략	설명
Suppressed.untilWindowCloses	윈도우에 대해 최종 결과만 방출
Suppressed.untilTimeLimit	마지막 이벤트 수신 후 설정가능한 시간이 경과한 후 윈도우에 대한 결과를 방출. 동일 키의 다른 이벤트가 이 시간 제한 도달 전에 도착하는 경우 이 이벤트가 버퍼 내 최초 이벤트를 대체한다 (발생 시 타이머가 재시작되지 않는다). 이는 속도 제한 업데이트의 효과가 있다.

환자 모니터링 애플리케이션에서는 전체 60초가 경과된 경우에만 심박수 윈도우 결과를 방출하기를 원한다. 따라서 Suppressed.untilWindowCloses 억제 전략을 사용할 것이다. 그러나 프로세서 토폴로지에서 이 전략을 사용하기 전에 방출되지 않은 결과를 메모리에 버퍼링하는 방법을 카프카 스트림즈에 지시해야 한다. 결국 억제된 레코드는 버려지지 않는다; 대신 주어진 윈도우에서 각 키의 최신 미방출 레코드는 결과를 방출할 때까지 메모리에 유지되며, 따라서 카프카 스트림즈는 이 잠재적으로 메모리 집약적인 억제 업데이트 태스크에 대해 어떻게 사용되는지 명시해야 한다[10]. 버퍼링 전략을 정의하기 위해 Buffer Config를 사용해야 한다. 표 5-2는 카프카 스트림즈에서 사용가능한 각 Buffer Config를 설명한다.

표 5-2. Buffer Configs

Buffer Config	설명
BufferConfig.maxBytes()	인메모리 버퍼가 이 바이트 수까지 억제된 이벤트를 저장
BufferConfig.maxRecords()	인메모리 버퍼가 이 키 수까지 억제된 이벤트를 저장
BufferConfig.unbounded()	인메모리 버퍼가 억제된 이벤트를 저장하기 위해 윈도우 내 억제된 레코드 유지를 위해 필요한 만큼 힙 스페이스를 사용. 애플리케이션에 힙 부족이 발생하는 경우 OutOfMemoryError (OOM) 예외가 발생

마지막으로 레코드 억제를 위한 마지막 요건은 카프카 스트림즈에 버퍼가 가득 찼을 때 무엇을 할지를 지시하는 것이다. 카프카 스트림즈는 표 5-3과 같이 Buffer Full 전략을 갖고 있다.

표 5-3. Buffer Full 전략

---

[10] 카프카 스트림즈는 suppress 연산자를 사용할 때 각 키의 최신 레코드를 유지하기 때문에 요구되는 메모리양은 키 스페이스의 함수로 입력 스트림에 대해 보다 작은 키 스페이스는 큰 키 스페이스보다 적은 메모리를 필요로 한다. 유예 기간 또한 메모리 사용에 영향을 미칠 수 있다.

Buffer Full 전략	설명
shutDownWhenFull	버퍼가 가득 찼을 때 애플리케이션을 우아하게 셧다운. 이 전략 사용시 중간 윈도우 결과를 절대 볼 수 없음
emitEarlyWhenFull	버퍼가 가득 찼을 때 애플리케이션 셧다운 대신 가장 오래된 결과를 방출. 이 전략을 사용하여 중간 윈도우 계산을 볼 수도 있음

이제 어떤 억제 전략, Buffer Config와 Buffer Full 전략이 사용가능한지를 이해했고 이들 3 가지의 어떤 조합이 동작하는 지를 결정해보자. 우리의 경우 속도 제한 업데이트를 원하지 않으며 따라서 untilTimeLimit을 사용하지 않을 것이다. 대신 심박수 윈도우의 최종 결과만을 방출하고 싶기 때문에 untilWindowCloses가 더욱 적합하다. 두번째 키 스페이스가 비교적 작을 것으로 예상하며 따라서 unbounded Buffer Config를 선택할 것이다. 마지막으로 결과를 초기에 방출하는 것은 부정확한 심박수 계산을 야기할 것이기 때문에 이를 사용하지 않을 것이다 (예, 단지 20 초 경과 후 심박수를 방출하는 경우). 따라서 Buffer Full 전략으로 shutDownWhenFull 을 사용할 것이다.

이를 종합하여 예제 5-4와 같이 suppress 연산자를 사용하여 환자 모니터링 토폴로지를 업데이트할 수 있다.

예제 5-4. 심박수 윈도우 (tumblingWIndow)의 최종 결과만을 방출하기 위해 suppress 연산자 사용

```
TimeWindows tumblingWindow =
 TimeWindows
  .of(Duration.ofSeconds(60))
 .grace(Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
 pulseEvents
  .groupByKey()
  .windowedBy(tumblingWindow)
  .count(Materialized.as("pulse-counts"))
  .suppress(
    Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull())); ①
```

① 단지 최종 계산만 방출하도록 윈도우 결과를 억제

이제 (그림 5-1)의 프로세서 토폴로지의 4 단계를 완료했고 단계 5 및 6을 진행해보자: pulse-events와 body-temp-events 데이터 필터링 및 키재생성.

윈도우 KTables 필터링 및 키재생성

예제 5-4의 KTable을 자세히 살펴보면 윈도잉을 통해 키가 String 타입에서 Windowed<String> 타입으로 변경되었음을 알 수 있다. 이전에 언급했듯이 이는 윈도잉을 통해 레코드가 추가 차원인 윈도우 범위로 그룹화되기 때문이다. 따라서 body-temp-events 스트림과 조인하기 위해서는 pulse-events의 키를 재생성해야 한다. 또한 사전 정의된 임계치를 초과하는 레코드에만 관심이 있기 때문에 두 데이터 스트림 모두를 필터링해야 한다.

어떤 순서든 필터 및 키재생성 연산을 수행해야 하는 것처럼 보이며 사실이다. 그러나 한 가지 충고는 가능한 초기에 필터링을 수행하는 것이다. 레코드 키재생성은 토픽 재분배를 필요로 하며 따라서 우선 필터링한다면 이 토픽에 대한 읽기/쓰기 수를 줄이고 애플리케이션 성능을 보다 우수하게 할 것이다.

필터링 및 키재생성 모두 이전 장에 논의했기 때문에 이 개념에 대해 더 깊이 논의하지는 않는다. 그러다 다음 코드 블록에서 환자 모니터링 애플리케이션의 필터링 및 키재생성 코드를 볼 수 있다:

```
KStream<String, Long> highPulse =
  pulseCounts
    .toStream() ①
    .filter((key, value) -> value >= 100)  ②
    .map(
      (windowedKey, value) -> {
        return KeyValue.pair(windowedKey.key(), value); ③
        });
KStream<String, BodyTemp> highTemp =
  tempEvents.filter((key, value) -> value.getTemperature() > 100.4); ④
```

① 레코드 키재생성을 위한 map 연산자를 사용할 수 있도록 스트림으로 변환

② 사전 정의된 임계치 100 bpm 초과 심박수에 대해서만 필터링

③ WindowedKey.key()에서 사용가능한 원래 키를 사용하여 스트림 키재생성. windowedKey가 org.apache.kafka.streams.kstream.Windowed의 인스턴스임을 주목하기 바란다. 이 클래스는 원래 키(windowedKey.key())와 타임 윈도우 (windowedKey.window())에 액세스하는 방법을 포함한다

④ 사전 정의된 임계치 100.4°F를 초과하는 체온 값에 대해서만 필터링

프로세서 토폴로지의 5 및 6 단계를 완료했고 윈도우 조인을 수행할 준비가 되어 있다.

윈도우 조인

"슬라이딩 조인 윈도우"에서 논의했듯이 윈도우 조인은 슬라이딩 조인 윈도우를 필요로 한다. 슬라이딩 조인 윈도우는 어떤 레코드가 함께 조인되어야 하는지를 결정하기 위해 조인 양측의 이벤트의 타임스탬프를 비교한다. 스트림은 무제한이기 때문에 KStream-KStream 조인에 대해 윈도우 조인이 요구된다. 따라서 관련 값의 빠른 조회를 수행하기 위해 데이터는 로컬 상태 저장소로 구체화되어야 한다.

다음과 같이 조인 윈도우를 구축하고 펄스 수와 체온 스트림을 조인할 수 있다:

```
StreamJoined<String, Long, BodyTemp> joinParams =
  StreamJoined.with(Serdes.String(), Serdes.Long(), JsonSerdes.BodyTemp()); ①
JoinWindows joinWindows =
  JoinWindows
```

```
      .of(Duration.ofSeconds(60)) ②
      .grace(Duration.ofSeconds(10)); ③
ValueJoiner<Long, BodyTemp, CombinedVitals> valueJoiner = ④
  (pulseRate, bodyTemp) -> new CombinedVitals(pulseRate.intValue(), bodyTemp);
KStream<String, CombinedVitals> vitalsJoined =
  highPulse.join(highTemp, valueJoiner, joinWindows, joinParams); ⑤
```

① 조인에 사용되는 Serdes를 지정

② 1분 이하로 차이가 나는 타임스탬프를 갖는 레코드가 동일 윈도우에 들어가며조인될 것이다.

③ 최대 10 초까지 지연을 견딘다.

④ 심박수와 체온을 CombinedVitals 객체로 결합한다.

⑤ 조인을 수행한다

조인은 시간과 관련해 특히 흥미로운데 조인 양측이 다른 속도로 레코드를 수신할 수 있고 따라서 이벤트가 적시에 조인되는 것을 보장하기 위해 어떤 추가적인 동기화가 필요하기 때문이다. 운좋게도 카프카 스트림즈는 조인 양측의 타임스탬프가 프로세서 토폴로지를 통해 데이터가 어떻게 흐르는지를 결정할 수 있도록 함으로써 이러한 시나리오를 다룰 수 있다. 다음 절에서 이 아이디어를 세부적으로 논의할 것이다.

시간 기반 데이터 플로우

이미 윈도우 조인 및 집계를 포함하여 시간이 특정 연산의 동작에 어떤 영향을 미칠 수 있음을 보았다. 그러나 시간은 또한 스트림을 통한 데이터 흐름을 제어한다. 특히 복수 소스로부터 이력 데이터를 처리할 때 정확성을 보장하기 위해 스트림 처리 애플리케이션이 입력 스트림을 동기화하는 것이 중요하다.

이러한 동기화를 용이하게 하기 위해 카프카 스트림즈는 각각 스트림 태스크에 대해 단일 파티션 그룹을 생성한다. 파티션 그룹은 우선순위 큐를 사용하여 해당 태스크가 다루는 각 파티션에 대해 큐잉된 레코드를 버퍼링하며 처리를 위한 다음 레코드 (모든 입력 파티션에 대해) 선택을 위한 알고리즘을 포함하고 있다. 가장 낮은 타임스탬프를 갖는 레코드가 처리를 위해 선택된다.

> 스트림 타임은 특정 토픽-파티션에 대해 관측된 가장 최신 타임스탬프이다. 초기에는 알려져 있지 않으며 단지 증가되거나 동일하게 유지될 수만 있다. 이는 새로운 데이터가 들어오는 경우에만 증가된다. 이는 카프카 스트림즈 자체에 대해 내부적이기 대문에 논의했던 다른 시간 개념과는 다르다.

단일 카프카 스트림즈 태스크가 (조인과 같이) 하나 이상의 파티션으로부터 데이터를 소비할 때 카프카 스트림즈는 각 파티션 (레코드 큐)에서 (헤드 레코드라는) 다음의 미처리 레코드들의 타임스탬프를 비교하고 처리를 위해 가장 낮은 타임스탬프를 갖는 레코드를 선택할 것이다. 선택된 레코드는 토폴로지의 적절한 소스 프로세서로 전달된다. 그림 5-8은 어떻게 동작되는지를 보여준다.

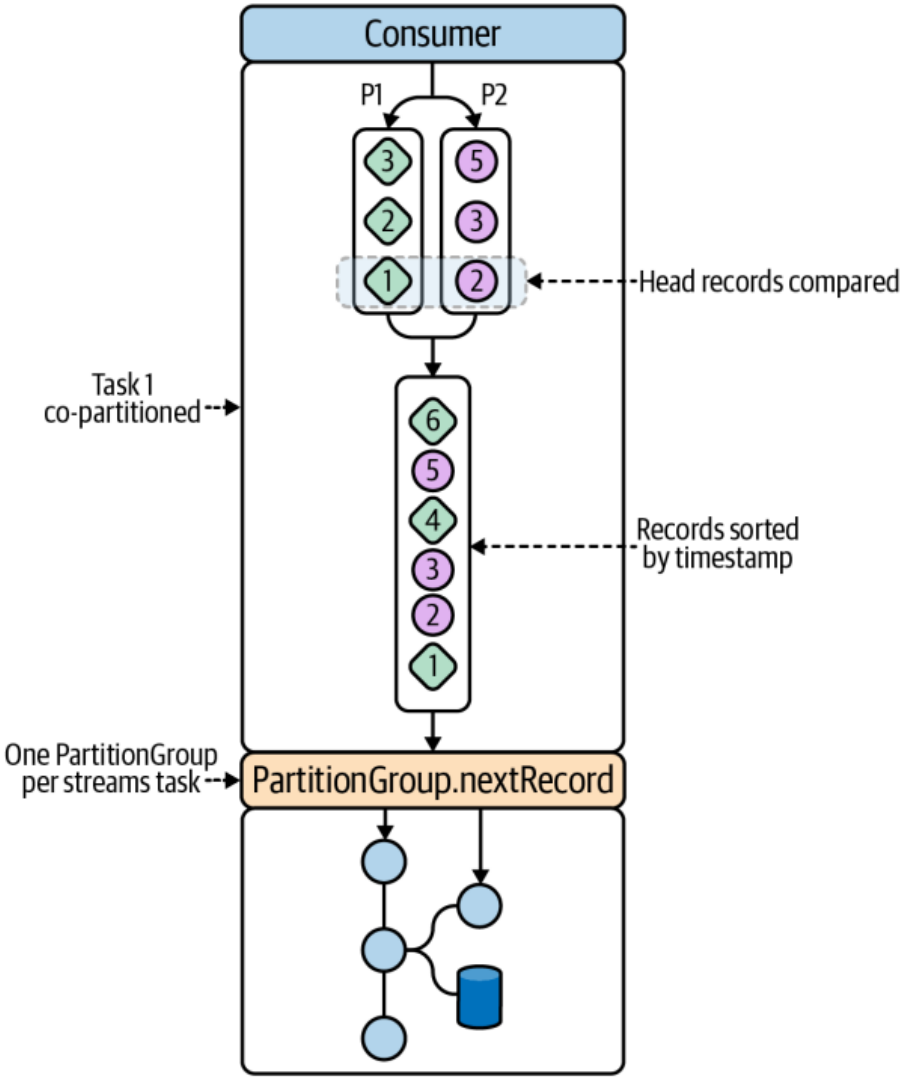

그림 5-8. 데이터가 카프카 스트림즈 애플리케이션을 통해 어떻게 흐르는지를 결정하기 위해 레코드 타임스탬프가 비교된다.

조인이 준비되었고 카프카 스트림즈가 시간 기반 플로우 제어 메커니즘[11]을 사용하여 타임스탬프 순서대로 레코드를 처리하려고 할 것임을 확인하였으며 이제 환자 모니터링 토폴로지의 마지막 단계를 다룰 준비가 되어 있다. 토폴로지에 싱크 프로세서를 추가하는 방법을 간단히 요약하고 그 후 윈도우 키-값 저장소에 쿼리하는 방법을 배울 것이다.

알림 싱크

---

[11] 입력 스트림 중 하나가 비게 될 때 이 방법이 최선임을 참고하는 것이 중요하다. 그러나 이를 예방하기 위해 데이터가 빈 입력 스트림에 새로운 레코드가 도착하길 얼마나 오래 기다릴지를 제어하는 max.task.idle.ms를 사용할 수 있다. 기본 값은 0이며 이 값을 증가시킴으로써 더 오래 기다려 시간 동기화를 향상시킬 수 있도록 할 것이다.

조인 결과를 다운스트림 컨슈머가 사용할 수 있도록 하기 위해 카프카에 보강 데이터를 다시 쓸 필요가 있다. 이전 장에서 보았듯이 카프카 스트림즈에서 싱크를 추가하는 것은 매우 쉽다. 다음 코드는 알림 싱크를 추가하는 방법을 보여준다. 임계치와 윈도우 조인에 의해 결정된 바와 같이 환자가 SIRS에 대해 위험에 처해 있다고 애플리케이션이 결정할 때마다 이 싱크에 쓰여질 것이다:

```
vitalsJoined.to(
    "alerts",
    Produced.with(Serdes.String(), JsonSerdes.CombinedVitals())
);
```

소스 스트림을 등록할 때 타임스탬프 추출기를 사용할 때의 장점은 (예제 5-2 참조) 출력 레코드가 추출된 타임스탬프와 연관될 것이라는 것이다. 조인되지 않은 스트림/테이블의 경우 타임스탬프는 소스 프로세서를 등록할 때 생성되었던 최초 타임스탬프 추출로부터 전파된다. 그러나 환자 모니터링 애플리케이션과 같이 조인을 수행한다면 카프카 스트림즈는 조인에 포함된 각 레코드의 타임스탬프를 살펴보고 출력 레코드에 대해 최대 값을 선택할 것이다[12].

알림 싱크를 등록함으로써 alerts 토픽의 실시간 컨슈머에 환자 모니터링 데이터를 제공한다. 이제 (환자의 심박수를 계산하는) 윈도우 집계 결과를 노출함으로써 윈도 키-값 저장소에 쿼리하는 방법을 배워보자.

윈도우 키-값 저장소 쿼리하기

"비윈도우 키-값 저장소 쿼리하기"에서 비윈도 키-값 저장소에 쿼리하는 법을 배웠다. 그러나 윈도우 키-값 저장소는 레코드 키가 다차원이고 (비윈도우 키-값 저장소에서 보았던) 원래 레코드 키와는 달리 원래 키와 윈도우 범위 모두로 구성되어 있기 때문에 일련의 다른 쿼리를 지원한다. 키와 윈도우 스캔을 살펴보면서 시작할 것이다.

키+윈도우 범위 스캔

윈도우 키-값 저장소에 사용될 수 있는 2 가지 다른 유형의 스캔이 존재한다. 첫번째는 해당 윈도우 범위에서 특정 키를 검색하며 따라서 3 개의 파라미터를 필요로 한다.

- 검색할 키 (환자 모니터링 애플리케이션의 경우 환자 ID에 해당한다. 예, 1)
- 이포크[13]로부터의 밀리초로 표현되는 윈도우 범위의 하한 경계 (예, 1605171720000은 2020-11-12T09:02:00.00Z).

---

[12] 2.3 버전 이전에서는 출력 레코드의 타임스탬프는 무엇이든 레코드가 조인을 트리거했을 때의 타임스탬프로 설정되었다. 최신 버전에서 최대 값을 사용함으로써 데이터 순서 여부에 상관없이 결과 타임스탬프가 동일하기 대문에 이전보다 개선을 이루었다.

[13] 1970-01-01T00:00:00Z (UTC)

- 이포크로부터의 밀리초로 표현되는 윈도우 범위의 상한 경계 (예, 1605171780000는 2020-11-12T09:03:00Z)

이러한 범위 스캔 타입 실행 방법과 결과로부터 적절한 특성을 추출하는 방법은 다음과 같다:

```
String key = 1;
Instant fromTime = Instant.parse("2020-11-12T09:02:00.00Z");
Instant toTime = Instant.parse("2020-11-12T09:03:00Z");
WindowStoreIterator<Long> range = getBpmStore().fetch(key, fromTime, toTime); ①
while (range.hasNext()) {
  KeyValue<Long, Long> next = range.next(); ②
  Long timestamp = next.key; ③
  Long count = next.value; ④
    // do something with the extracted values
}
 range.close(); ⑤
```

① 선택된 시간 범위에서 각 키를 통해 반복하는데 사용될 수 있는 반복자를 반환

② 반복에서 다음 요소를 얻는다.

③ 레코드의 타임스탬프는 key 특성으로 사용가능하다.

④ 값은 value 특성을 사용하여 추출될 수 있다.

⑤ 메모리 누수 예방을 위해 반복자를 닫는다.

윈도우 범위 스캔

윈도우 키-값 저장소에 대해 수행될 수 있는 두번째 유형의 범위 스캔은 해당 시간 범위 내 모든 키를 검색하는 것이다. 이러한 유형의 쿼리는 2 개의 파라미터를 필요로 한다.

- 이포크[14]로부터의 밀리초로 표현되는 윈도우 범위의 하한 경계 (예, 1605171720000은 2020-11-12T09:02:00.00Z).
- 이포크로부터의 밀리초로 표현되는 윈도우 범위의 상한 경계 (예, 1605171780000는 2020-11-12T09:03:00Z)

다음 코드 블록은 이러한 범위 스캔 타입 실행 방법과 결과로부터 적절한 특성을 추출하는 방법을 보여준다:

```
Instant fromTime = Instant.parse("2020-11-12T09:02:00.00Z");
```

[14] 1970-01-01T00:00:00Z (UTC)

```
Instant toTime = Instant.parse("2020-11-12T09:03:00Z");
KeyValueIterator<Windowed<String>, Long> range =
  getBpmStore().fetchAll(fromTime, toTime);
while (range.hasNext()) {
  KeyValue<Windowed<String>, Long> next = range.next();
  String key = next.key.key();
  Window window = next.key.window();
  Long start = window.start();
  Long end = window.end();
  Long count = next.value;
  // do something with the extracted values
}
range.close();
```

모든 엔트리

범위 스캔 쿼리와 비슷하게 all() 쿼리는 로컬 상태 저장소에서 사용가능하는 모든 윈도우 키-값 쌍에 대해 반복자를 반환한다[15]. 다음 코드 스니펫은 로컬 윈도우 키-값 저장소에 대해 all() 쿼리를 실행하는 방법을 보여준다. 결과를 통한 반복은 범위 스캔 쿼리와 동일하며 따라서 편의를 위해 그 로직은 생략하였다:

```
KeyValueIterator<Windowed<String>, Long> range = getBpmStore().all();
```

> 메모리 누수를 예방하기 위해 반복자 작업이 완료된 경우 이를 닫는 것이 중요하다. 예를 들어 이전 코드 스니펫을 보면 반복자 작업 완료 시 range.close()을 호출하였다.

이러한 쿼리 유형을 사용하여 이전 장에서 논의한 바와 동일한 방식으로 상호대화형 쿼리 서비스를 구축할 수 있다. 애플리케이션에 RPC 서비스와 클라이언트를 추가하고 원격 애플리케이션 인스턴스 발견을 위한 카프카 스트림즈 인스턴스 발견 로직을 활용하며 RPC 또는 RESTful 엔드 포인트에 이전 윈도우 키-값 저장소 쿼리를 수행하면 된다 (“원격 쿼리 참조).

요약

이 장에서는 시간이 보다 고급 스트림 처리 유스케이스에 사용되는 방법을 배웠다. 토폴로지 정의에서 사용한 타임 시맨틱을 신중히 고려하여 카프카 스트림즈에서 보다 결정론적인 처리를 달성할 수 있다. 시간은 윈도우 집계, 윈도우 조인 및 다른 시간 기반 연산의 동작을 야기할 뿐만 아니라 카프카 스트림즈가 시간 기반으로 입력 스트림을 동기화하려고 하기 때문에 애플리케이션을 통해 데이터가 어떻게 그리고 언제 흐를지를 제어한다.

데이터 윈도잉은 이벤트 간 시간 관계를 유도할 수 있도록 한다. 데이터 집계 (원시 펄스 이벤트를 시

---

[15] 상태 저장소 내 키 수에 따라 헤비급 호출일 수 있다.

간 기반 심박수 집계로 변환) 또는 데이터 조인 (심박수 스트림과 체온 스트림을 조인)에 이러한 관계를 활용하든 아니든 시간은 보다 의미있는 데이터 보강에 대한 문을 열어준다.

마지막으로 윈도우 키-값 저장소에 쿼리하는 방법을 배움으로써 윈도우 상태 저장소에 대한 중요한 사실을 배웠다: 키는 다차원 (원래 키와 윈도우의 시간 범위를 포함)으로 윈도우 범위 스캔을 포함하여 일련의 다른 쿼리를 지원한다.

다음 장에서는 고급 상태 관리 태스크를 살펴봄으로써 스테이트풀 카프카 스트림즈 애플리케이션에 대한 논의를 마칠 것이다.