9 장 ksqlDB를 사용한 데이터 통합

ksqlDB를 사용해 스트림 처리 애플리케이션을 구축하기 위한 첫 번째 단계는 현재 처리하려는 데이터가 존재하는 장소와 보강/변환 데이터가 최종적으로 저장될 장소를 고려하는 것이다. ksqlDB는 내부적으로 카프카 스트림즈를 활용하기 때문에 구축하는 애플리케이션에 대한 직접적인 입력 및 출력은 카프카 토픽일 것이다. ksqlDB는 Elasticsearch, PostgreSQL, MySQL, Google Pub/Sub, Amazon Kinesis, MongoDB 외 수백가지의 인기있는 제3자 시스템을 포함하여 다른 데이터 소스와의 통합을 쉽게 한다.

물론 데이터가 카프카에 존재하고 외부 시스템에 결과를 처리할 계획이 없다면 (카프카 커넥트에 의해 구동되는) ksqlDB 데이터 통합 기능을 이용한 작업은 필요치 않다. 그러나 외부 시스템에 쓰고 외부 시스템으로부터 읽어야 한다면 이 장은 ksqlDB와 카프카 커넥트를 사용하여 적절한 데이터 소스와 싱크의 연결을 도울 수 있는 필수 토대를 제공할 것이다.

이 장은 카프카 생태계의 별도 API인 카프카 커넥트와 이를 통해 무엇을 할 수 있고 커넥트가 어떻게 작성되었는지에 대한 주제에 대한 완전한 가이드는 아니다. 시작하기에 충분한 배경지식을 제공할 것이고 커넥트 API로 작업하기 위한 ksqlDB의 고수준 추상화를 살펴볼 것이다. 이 장에서 탐구할 주제는 다음을 포함한다:

- 빠른 카프카 커넥트 개요
- 카프카 커넥트 통합 모드
- 카프카 커넥트 워커 구성
- 소스 및 싱크 커넥터 설치
- ksqlDB에서 소스 및 싱크 커넥터 생성, 삭제 및 검사
- 카프카 커넥트 API를 사용한 소스 및 싱크 커넥터 검사
- 컨플루언트 스키마 레지스트리에서 커넥터 스키마 검사

이 장 말미에는 ksqlDB를 사용하여 스트리밍 ETL (추출, 변환 및 적재) 작업을 수행하기 위한 3 가지 태스크 중 2 가지를 이해할 것이다.

- 외부 시스템의 데이터를 카프카 내로 추출하기
- 카프카로부터 외부 시스템에 데이터 적재하기

ETL의 나머지인 변환은 데이터 변환이 데이터 통합보다 스트림 처리와 더욱 밀접한 관계를 갖고 있기 때문에 다음 두 장에서 다룰 것이다. ksqlDB의 데이터 통합 기능을 실제로 보강하는 기술에 대해 배우기 위해 카프카 커넥트의 빠른 개요로 시작할 것이다.

카프카 커넥트 개요

카프카 커넥터는 카프카 생태계[1]에서 5 가지 API 중 하나로 외부 데이터 저장소, API 및 파일 시스템을 카프카와 연결하는데 사용된다. 데이터가 카프카에 있다면 ksqlDB를 사용하여 이를 처리, 변환 및 보강할 수 있다. 카프카 커넥트의 주요 컴포넌트는 다음과 같다:

커넥터

커넥터는 (잠시 후 논의할) 워커에 설치될 수 있는 패키징된 코드로 카프카와 다른 시스템 간 데이터 흐름을 용이하게 하며 2 가지로 나눌 수 있다:

- 소스 커넥터는 외부 시스템으로부터 카프카로 데이터를 읽어 들인다.
- 싱크 커넥터는 카프카로부터 데이터를 외부 시스템에 쓴다.

태스크

태스크는 커넥터 내 작업 단위이다. 태스크 수는 구성가능하며 단일 워커 인스턴스가 수행할 수 있는 작업 수를 제어할 수 있도록 한다.

워커

워커는 커넥터를 실행하는 JVM 프로세서이다. 다중 워커가 작업 병렬화/분산을 돕고 부분적인 고장 (예, 한 워커가 오프라인이 되는 경우) 시 내고장성을 얻을 수 있도록 배치될 수 있다.

컨버터

컨버터는 커넥트에서 데이터의 직렬화/역직렬화를 다루는 코드이다. 기본 컨버터 (예, AvroConverter)가 워커 수준에서 지정되어야 하며 커넥터 수준에서 이를 재정의할 수도 있다.

커넥트 클러스터

커넥트 클러스터는 함께 그룹으로 작업하여 데이터를 카프카 내외부로 이동시키는 하나 이상의 카프카 커넥트 워커이다.

그림 9-1은 이들 컴포넌들이 함께 동작하는 방법의 시각화를 보여준다.

[1] 컨슈머, 프로듀서, 스트림, Admin API가 나머지 4 가지이다.

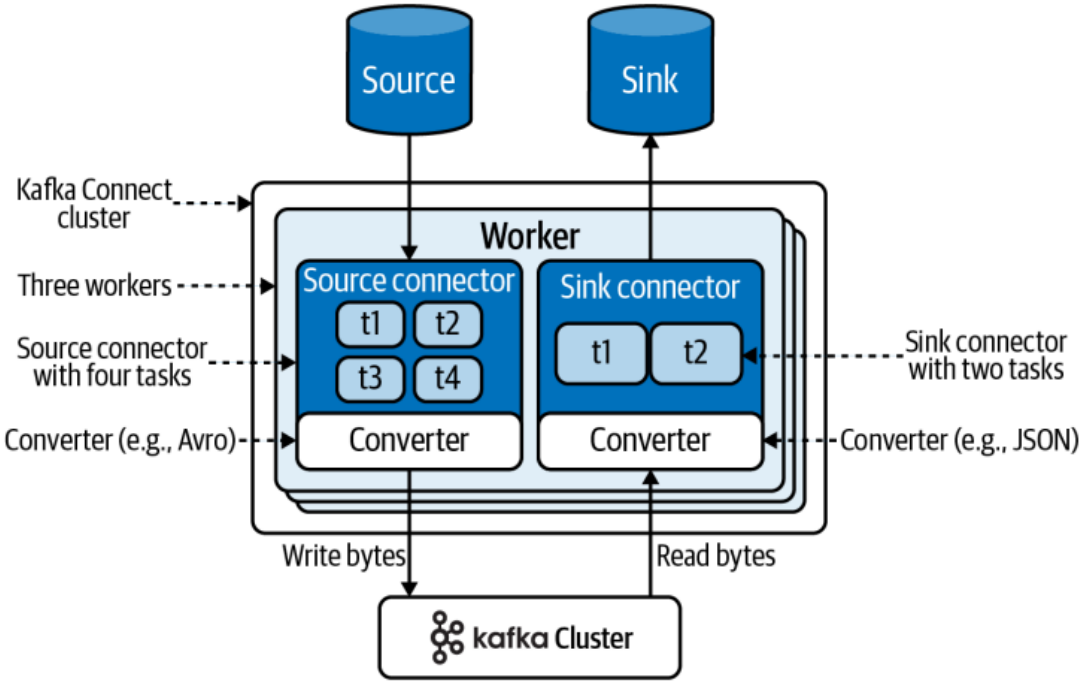

그림 9-1. 카프카 커넥트 아키텍처

지금 추적해야 할 것이 많은 것처럼 보일 수 있지만 이 장을 진행함에 따라 ksqlDB가 카프카 커넥트를 통해 작업하기 위한 정신 모델을 단순화함을 볼 수 있을 것이다. 이제 ksqlDB와의 사용을 위해 카프카 커넥트 배치에 대한 옵션을 살펴보자.

외부 및 내장 커넥트

ksqlDB에서 카프카 커넥트 통합은 2 가지 다른 모드로 동작될 수 있다. 이 절에서는 각 모드와 어느 경우 사용하는지를 설명한다. 외부 모드를 살펴봄으로써 시작할 것이다.

외부 모드

카프카 커넥트가 이미 동작 중 이거나 ksqlDB 외부에서 카프카 커넥트를 별도로 생성 및 관리하고자 한다면 외부 모드로 카프카 커넥트 통합을 동작시킬 수 있다. 이는 ksql.connect. url 특성을 사용하여 ksqlDB에서 카프카 커넥트 클러스터의 URL을 가리키도록 하는 것을 포함한다. ksqlDB는 커넥터 생성 및 관리를 위해 외부 카프카 커넥트 클러스터에 직접 요청을 할 것이다. 다음 코드는 외부 모드 활성화를 위한 구성 예를 나타낸다 (이는 ksqlDB server properties 파일에 저장될 것이다).

```
ksql.connect.url=http://localhost:8083
```

외부 모드로 동작할 때 애플리케이션이 필요로 하는 어떤 소스/싱크 커넥터라도 외부 워커 자체에 설치되어야 할 것이다. 외부 모드로 동작 시 이 모드 사용 장점 중 하나가 ksqlDB에 머신 리소스를 공유할 필요가 없다는 것이기 때문에 일반적으로 워커가 ksqlDB 서버와 함께 배치되지 않음을 주목하기 바란다. 그림 9-2는 외부 모드로 동작 중인 카프카 커넥트를 나타낸다.

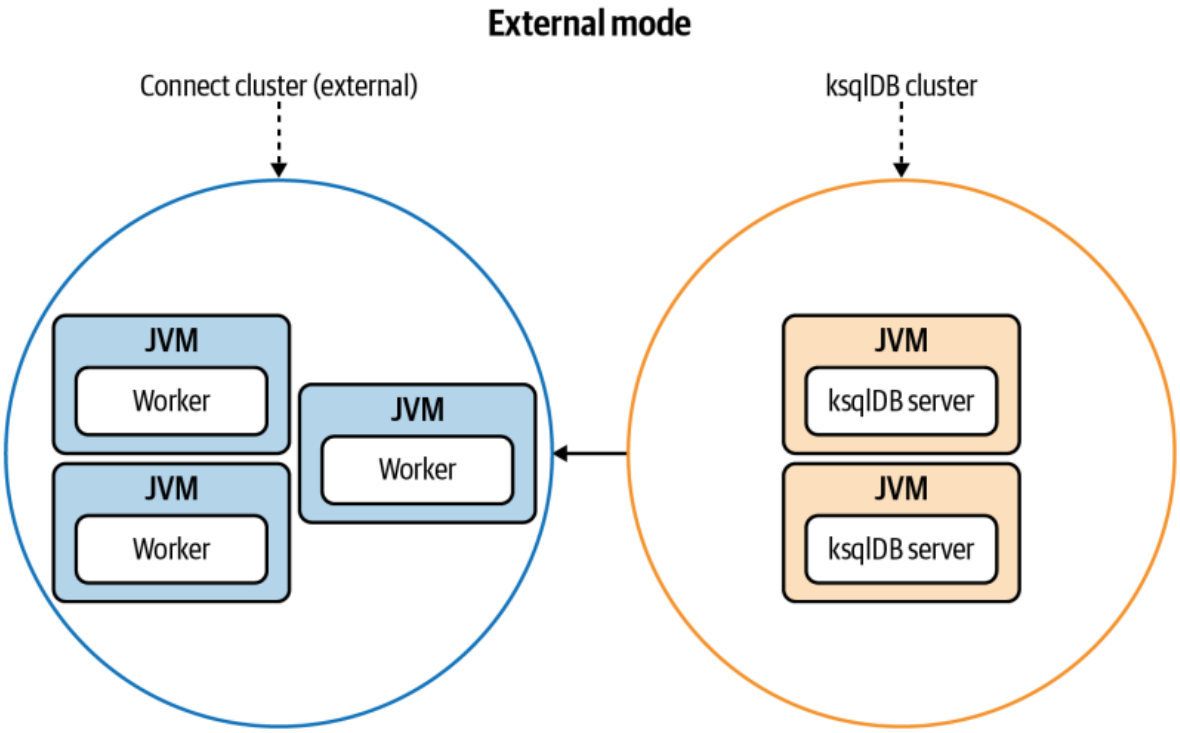

그림 9-2. 외부 모드로 동작 중인 카프카 커넥트

카프카 커넥트를 외부 모드로 동작시키는 이유 중 일부는 다음을 포함한다:

- 스트림 처리 및 데이터 수신/송신 워크로드를 독립적으로 확장/축소하거나 이들 다른 유형의 워크로드에 대해 자원 격리를 원하는 경우
- 소스/싱크 토픽에 대해 높은 처리량을 예상하는 경우
- 기존 카프카 커넥트 클러스터를 이미 갖고 있는 경우

다음으로 이 책 튜토리얼에서 사용할 내장 모드를 살펴보자.

내장 모드

내장 모드에서 카프카 커넥트 워커는 ksqlDB 서버와 동일한 JVM에서 실행된다. 워커는 카프카 커넥트 분산 모드[2]에서 동작하며 이는 작업이 다중 작업 워커 인스턴스를 통해 분산될 수 있음을 의미한다. 카프카 커넥트 워커의 수는 ksqlDB 클러스터 내 ksqlDB 서버 수와 일치한다. 다음 경우에 내장 모드를 사용해야 한다;

[2] 카프카 커넥트는 자체 배치 모드 (분산 및 독립형)를 갖고 있는데 ksqlDB의 카프카 커넥트 통합 모드 (외부 및 내장)와 혼동되지 않아야 한다.

- 스트림 처리 및 데이터 수신/송신 워크로드를 함께 확장/축소하길 원하는 경우
- 소스/싱크 토픽에 대해 낮음에서 중간 정도의 처리량을 예상하는 경우
- 매우 간단하게 데이터 통합을 원하고, 별도의 카프카 커넥트 배치를 원하지 않으며 데이터 통합/변환 워크로드를 독립적으로 확장/축소할 필요가 없는 경우
- ksqlDB 서버 재시작 시 카프카 커넥트의 재시작을 신경쓰고 싶지 않은 경우
- ksqlDB와 카프카 커넥트 간 계산/메모리 자원 공유를 신경쓰고 싶지 않은 경우[3]

내장 모드에서는 ksqlDB 서버와 카프카 커넥트가 함께 배치되기 때문에 ksqlDB가 동작 중인 동일 노드에 애플리케이션이 필요로 하는 소스/싱크 커넥터를 설치해야 할 것이다. 그림 9-3은 내장 모드로 동작 중인 카프카 커넥트를 나타낸다.

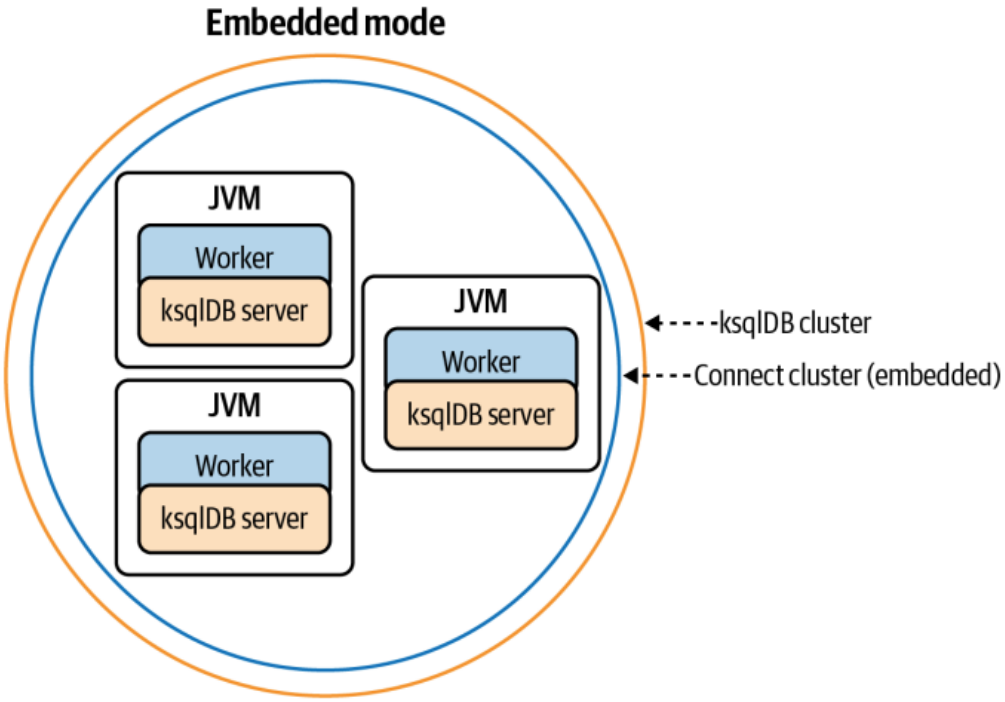

그림 9-2. 내장 모드로 동작 중인 카프카 커넥트

내장 모드로 동작시키기 위해 ksqlDB 서버 구성에서 ksql.connect.worker.config 특성을 설정해야 할 것이다. 이 특성의 값은 카프카 커넥트 워커 구성 경로이다 (워커가 실제 소스 및 싱크 커넥터를 실행시키는 카프카 커넥트 프로세스임을 기억하기 바란다). ksqlDB 서버 특성 파일에서 이 특성을 구성하는 방법 예는 다음과 같다:

```
ksql.connect.worker.config=/etc/ksqldb-server/connect.properties
```

[3] 더욱 자세한 정보는 ksqlDB 문서를 참고하기 바란다.

어디에 ksql.connect.worker.config 특성이 참조하는 워커 구성 파일이 존재하는가? 다음에 살펴볼 것이다.

커넥트 워커 구성

카프카 커넥트는 고도로 구성가능한데 공식 아파치 카프카 문서에서 사용가능한 구성 특성을 세부적으로 다룬다. 그러나 이 절에서는 카프카 커넥트 워커 구성 예를 봄으로써 보다 중요한 파라미터 중 일부를 다룰 것이다. 내장 모드로 동작시키고 있다면 이는 파일 (connect.properties)에 저장되어야 하고 ksqlDB 서버 구성에서 ksql.connect.worker.config 특성에서 참조되어야 한다. 외부 모드로 동직시키고 있다면 카프카 커넥트 시작 시 워커 구성이 인수로 전달되어야 한다. 다음 코드 블록은 구성 예를 보여준다.

```
bootstrap.servers=localhost:9092 ①
group.id=ksql-connect-cluster ②
key.converter=org.apache.kafka.connect.storage.StringConverter ③
value.converter=org.apache.kafka.connect.storage.StringConverter ④
config.storage.topic=ksql-connect-configs ⑤
offset.storage.topic=ksql-connect-offsets
status.storage.topic=ksql-connect-statuses
errors.tolerance=all ⑥
plugin.path=/opt/confluent/share/java/ ⑦
```

① 카프카 클러스터와의 초기 연결 확립을 위해 사용되어야 하는 카프카 브로커의 호스트/포트 쌍 목록

② 이 워커가 속하는 커넥트 클러스터에 해당하는 문자열 식별자. 동일한 group.id로 구성되는 워커는 동일 클러스터에 속하며 실행 커넥터에 대해 워크로드를 공유할 수 있다.

③ 카프카 커넥트 포맷과 카프카에 쓰여진 직렬화된 포맷 간 변환에 사용되는 컨버터 클래스. 이는 어떤 커넥터도 어떤 직렬화 포맷으로 동작할 수 있도록 하기 때문에 카프카에 써진 또는 카프카로부터 읽는 메시지 내 키의 포맷을 제어한다. 일반적인 포맷은 JSON과 Avro를 포함한다.

④ 카프카 커넥트 포맷과 카프카에 쓰여진 직렬화된 포맷 간 변환에 사용되는 컨버터 클래스. 이는 어떤 커넥터도 어떤 직렬화 포맷으로 동작할 수 있도록 하기 때문에 카프카에 써진 또는 카프카로부터 읽는 메시지 내 값의 포맷을 제어한다. 일반적인 포맷은 JSON과 Avro를 포함한다.

⑤ 카프카 커넥트는 커넥터와 태스크 구성 관련 정보를 저장하기 위해 몇 개의 토픽을 사용한다. 여기서는 (워커가 ksqlDB 서버 인스턴스와 동일한 JVM에서 실행될 것임을 의미하는) 내장 모드로 동작시킬 것이기 때문에 접두사 ksql이 붙은 이들 토픽에 대한 표준적인 이름을 사용한다.

⑥ 이 특성을 통해 카프카 커넥트에서 기본 에러 처리 정책을 구성할 수 있다. 유효한 값은 none (에러 발생 시 즉시 실패한다)과 all (에러를 전적으로 무시하거나 errors.deadletterque.topic.name 특성과 함께 사용 시 선택한 토픽으로 모든 에러를 전달한다)이다.

⑦ 플러그인 (커넥터, 컨버터, 변환)이 설치될 것으로 예상되는 콤마 구분 파일 시스템 경로. 이 장 후반부에 커넥터 설치 방법을 살펴볼 것이다.

위에서 볼 수 있듯이 워커 구성 대부분은 매우 간단하다. 그러나 데이터 직렬화라는 중요한 태스크와 관련된 구성이기 때문에 더욱 자세히 살펴볼 구성 중 하나는 컨버터 특성 (key.converter과 value.converter)이다. 다음 절에서 컨버터와 직렬화를 보다 자세히 살펴볼 것이다.

컨버터와 직렬화 포맷

카프카 커넥트에서 사용하는 컨버터 클래스는 카프카 커넥트와 ksqlDB 모두에서 데이터가 어떻게 직렬화 및 역직렬화되는지에 중요한 역할을 한다. 이전 장의 "헬로우 월드" 튜토리얼에서는 ksqlDB에서 스트림을 생성하기 위해 예제 9-1의 statement를 사용했다.

예제 9-1. Users 토픽으로부터 읽은 스트림 생성

```
CREATE STREAM users (
 ROWKEY INT KEY,
 USERNAME VARCHAR
) WITH (
 KAFKA_TOPIC='users',
 VALUE_FORMAT='JSON'
);
```

이전 문장은 사용자 토픽 (KAFKA_TOPIC='users')이 JSON 직렬화된 레코드 값 (VALUE_FORMAT='JSON')을 포함한다는 것을 ksqlDB에 알려준다. JSON 포맷의 데이터를 토픽에 쓰고 있는 프로듀서를 갖고 있는 경우 포맷에 대해 추론하는 것은 매우 쉽다. 그러나 가령 PostgreSQL 데이터베이스에서 카프카로 데이터를 스트리밍하기 위해 카프카 커넥트를 사용하고 있다면? PostgreSQL 데이터가 카프카에 써질 때 어떤 포맷을 직렬화되는가?

이때 컨버터 구성이 동작한다. 카프카 커넥트가 처리하는 레코드 키와 값 모두의 직렬화 포맷을 제어하기 위해 key.converter과 value.converter 특성을 적절한 컨버터 클래스로 설정할 수 있다. 표 9-1은 가장 일반적인 컨버터 클래스와 해당 ksqlDB 직렬화 포맷을 보여준다 (예, 예제 9-1에서 보듯이 스트림 또는 테이블 생성 시 VALUE_FORMAT 특성에 제공할 값).

표 9-1은 또한 보다 간결한 메시지 포맷을 원할 때 유용한 레코드 스키마를 저장하기 위해 컨플루언트 스키마 레지스트리에 어떤 컨버터가 의존하는지를 보여준다.

표 9-1. 카프카 커넥트와 사용할 수 있는 가장 일반적인 클래스와 ksqlDB 내 해당 직렬화

타입	컨버터 클래스	스키마 레지스트리 필요	ksqlDB 직렬화 타입
Avro	io.confluent.connect.avro.AvroConverter	예	AVRO

Protobuf	io.confluent.connect.protobuf.ProtobufConverter	예	PROTOBUF
JSON (스키마 레지스트리)	io.confluent.connect.json.JsonSchemaConverter	예	JSON_SR
JSON	org.apache.kafka.connect.json.JsonConverter[a]	아니오	JSON
String	org.apache.kafka.connect.storage.StringConverter	아니오	KAFKA[b]
DoubleConverter	org.apache.kafka.connect.converters.DoubleConverter	아니오	KAFKA
IntegerConverter	org.apache.kafka.connect.converters.IntegerConverter	아니오	KAFKA
LongConverter	org.apache.kafka.connect.converters.LongConverter	아니오	KAFKA

[a] 카프카 커넥트와 함께 제공되는 JsonConverter를 사용할 때 커넥터 워커 구성 내 다음 구성을 설정하길 원할 수 있다: value.converter.schemas.enable. 이 값을 true로 설정하는 경우 커넥트에 JSON 레코드에 자체 스키마를 내장하라고 알려줄 것이다 (스키마 레지스트리를 사용하지 않지만 각 레코드에 스키마를 포함함으로써 메시지 크기가 매우 클 수 있다). 반대로 이 값을 false로 설정하는 경우 ksqlDB는 스트림 또는 테이블 생성 시 정의한 데이터 타입 힌트를 사용하여 필드 타입을 결정할 것이다. 이는 뒤에 탐구할 것이다.

[b] KAFKA 포맷은 카프카 내장 Serdes 중 하나를 사용하여 레코드 키 또는 값이 직렬화되었음을 나타낸다 (표 3-1 참조).

스키마 레지스트리를 필요로 하는 표 9-1의 각 컨버터에 대해 추가적인 구성 특성인 { key | value }.converter.schema.registry.url 을 추가해야 할 것이다. 예를 들어 이 책에서는 Avro 데이터로 작업할 것이며 따라서 커넥터가 이 포맷으로 값을 쓰길 원한다면 예제 9-2에 보듯이 워커 구성을 업데이트할 수 있다.

예제 9-2. 레코드 값에 대해 AvroConverter를 사용하는 워커 구성

```
bootstrap.servers=localhost:9092
group.id=ksql-connect-cluster
key.converter=org.apache.kafka.connect.storage.StringConverter ①
value.converter=org.apache.kafka.connect.storage.StringConverter ②
config.storage.topic=ksql-connect-configs
offset.storage.topic=ksql-connect-offsets
status.storage.topic=ksql-connect-statuses
errors.tolerance=all
plugin.path=/opt/confluent/share/java/
```

① 카프카 커넥트가 Avro 포맷으로 값을 직렬화함을 보장하기 위해 AvroConverter을 사용한다.

② AvroConverter는 레코드 스키마를 저장하기 위해 컨플루언트 스키마 레지스트리를 필요로 하며 따라서 value.converter.schema.registry.url 특성을 사용하여 스키마 레지스트리가 동작 중인 URL을 지정해야 한다[4].

이제 카프카 커넥트에서 데이터 직렬화 포맷 지정 방법을 배웠고 카프카 커넥트 워커 구성 (예제 9-2)을 배웠기 때문에 실제 커넥터를 설치 및 사용하기 위한 튜토리얼을 진행해보자.

튜토리얼

이 튜토리얼에서는 PostgreSQL에서 카프카로의 데이터 스트리밍을 위해 JDBC 소스 커넥터를 사용할 것이다. 그 후 카프카에서 Elasticsearch로 데이터를 쓸 Elasticsearch 싱크 커넥터를 생성할 것이다. 환경 설정 (PostgreSQL과 Elasticsearch 인스턴스 포함) 을 위해 Github 상의 튜토리얼 코드와 지침을 참고하기 바란다.

커넥터 설치를 시작할 것이다.

커넥터 설치

소스와 싱크 커넥터를 설치하는 2 가지 기본 방법이 있다:

- 수동 설치
- 컨플루언트 허브를 통한 자동 설치

수동 설치는 커넥터 구현에 따라 다를 수 있고 관리자가 선택한 아티팩트 (커넥터 아티팩트는 보통하나 이상의 JAR 파일을 포함한다) 배포 방법에 커넥트가 의존한다. 그러나 이 방법의 경우 보통 웹 사이트로부터 또는 Maven Central or Artifactory와 같은 아티팩트 저장소로부터 아티팩트를 직접 다운로드한다. 커넥터가 다운로드되면 JAR 파일이 plugin.path 구성 특성에 의해 지정된 위치에 놓인다.

커넥터를 다운로드하는 보다 쉬운 방법으로 이 책에서 사용할 방법은 컨플루언트 개발 CLI 툴을 사용하여 커넥터를 설치할 수 있도록 한다. Confluent-hub라는 CLI는 컨플루언트 문서의 지침을 사용하여 설치될 수 있다. 컨플러언트 허브를 설치한 경우 매우 쉽게 커넥터를 다운로드할 수 있다. 커넥터 설치 명령 구문은 다음과 같다:

```
confluent-hub install <owner>/<component>:<version> [options]
```

예를 들어 Elasticsearch 싱크 커넥터를 설치하려면 다음 명령을 실행할 수 있다:

```
confluent-hub install confluentinc/kafka-connect-elasticsearch:10.0.2 ₩
```

---

[4] 레코드 키에 대해서도 스키마 레지스트리 종속적인 컨버터를 사용하는 것도 가능하다. 이 경우 워커 구성에서 key.converter.schema.registry.url을 지정해야 한다.

```
 --component-dir /home/appuser ₩ ①
 --worker-configs /etc/ksqldb-server/connect.properties ₩ ②
 --no-prompt ③
```

① 커넥터가 설치되는 디렉토리

② 워커 구성 위치. 설치 위치 (--component-dir로 지정)는 이미 포함되지 않은 경우 plugin.path에 추가될 것이다.

③ CLI가 권고/기본 값으로 처리할 수 있도록 함으로써 상호대화식 단계 (예, 설치 확인, 소프트웨어 라이선스 협약 등)를 건너뛴다. 이는 설치 스크립트에 유용하다.

비슷하게 PostgreSQL 소스 커넥터는 다음 명령을 사용하여 설치될 수 있다:

```
confluent-hub install confluentinc/kafka-connect-jdbc:10.0.0 ₩
 --component-dir /home/appuser/ ₩
 --worker-configs /etc/ksqldb-server/connect.properties ₩
 --no-prompt
```

내장 모드로 실행시키는 경우 ksqlDB 서버 인스턴스 시작 후 설치된 커넥터를 사용하려고 한다면 ksqlDB를 재시작해야 함을 주목하기 바란다. 애플리케이션이 필요한 커넥터를 설치한 경우 ksqlDB에서 커넥터 인스턴스를 생성 및 관리할 수 있다. 다음 절에서 이를 논의할 것이다.

ksqlDB를 통한 커넥터 생성

커넥터 생성 구문은 다음과 같다:

```
CREATE { SOURCE | SINK } CONNECTOR [ IF NOT EXISTS ] <identifier> WITH(
 property_name = expression [, ...]);
```

PostgreSQL 인스턴스를 postgres:5432에서 동작시킨다고 가정하면 ksqlDB에서 다음 명령을 실행하여 titles 테이블로부터 읽어 들이는 소스 커넥터를 설치할 수 있다:

```
CREATE SOURCE CONNECTOR `postgres-source` WITH( ①
 "connector.class"='io.confluent.connect.jdbc.JdbcSourceConnector', ②
 "connection.url"=
'jdbc:postgresql://postgres:5432/root?user=root&password=secret', ③
 "mode"='incrementing', ④
 "incrementing.column.name"='id', ⑤
 "topic.prefix"='', ⑥
 "table.whitelist"='titles', ⑦
 "key"='id'); ⑧
```

① 커넥터 구성을 전달하기 위해 WITH 절이 사용된다 (이는 커넥터에 따라 다르며 따라서 사용가능한 구성 특성 목록에 대한 커넥터 문서를 살펴봐야 할 것이다).

② 커넥터 Java 클래스

③ JDBC 소스 커넥터는 데이터 저장소 (이 경우 PostgreSQL 데이터베이스)에 연결하기 위한 접속 URL을 필요로 한다.

④ JDBC 소스 커넥터를 동작시킬 수 있는 여러 모드가 존재한다. 추가되는 어떤 새로운 레코드도 Titles 테이블에 스트리밍하길 원하고 자동 증분 칼럼을 갖고 있기 때문에 모드를 incrementing로 설정할 것이다. 이 모드와 커넥터가 지원하는 다른 모드는 커넥터 문서에 자세히 설명되어 있다.

⑤ 소스 커넥터가 이미 봤던 행들을 결정하기 위해 사용할 자동 증분 칼럼명

⑥ 각 테이블은 전용 토픽 (예, titles 테이블은 titles 토픽으로 스트리밍될 것이다)으로 스트리밍된다. 선택적으로 토픽명에 접두사를 붙일 수 있다 (예, ksql- 접두사는 ksql-titles 토픽으로 titles 데이터를 스트리밍할 것이다). 이 튜토리얼에서는 접두사를 사용하지 않는다.

⑦카프카로 스트리밍할 테이블 목록

⑧레코드 키로 사용될 값

CREATE SOURCE CONNECTOR 문을 실행시키는 경우 다음과 같은 메시지를 봐야 한다:

```
 Message
-----------------------------------
 Created connector postgres-source
-----------------------------------
```

이제 애플리케이션 출력을 Elasticsearch로 쓸 싱크 커넥터를 생성해보자. 이는 소스 커넥터 생성과 매우 비슷하다:

```
CREATE SINK CONNECTOR `elasticsearch-sink` WITH(
 "connector.class"=
 'io.confluent.connect.elasticsearch.ElasticsearchSinkConnector',
 "connection.url"='http://elasticsearch:9200',
 "connection.username"='',
 "connection.password"='',
 "batch.size"='1',
 "write.method"='insert',
 "topics"='titles',
 "type.name"='changes',
 "key"='title_id');
```

위에서 볼 수 있듯이 커넥터 특정적인 구성은 커넥터에 따라 다르다. 이전 구성 대부분은 자명한데 독자가 Elsaticsearch 싱크 커넥터 구성 참조에서 찾을 수 있는 ElasticsearchSinkConnector 구성에 대해 배우기 위한 연습으로 이를 남겨둘 것이다. 다시 CREATE SINK CONNECTOR 문 실행 후 이전과 비슷한 확인 정보를 봐야 한다:

```
 Message
----------------------------------------
 Created connector elasticsearch-sink
----------------------------------------
```

커넥터 인스턴스가 ksqlDB에 설치된 경우 다양한 방법으로 이들과 상호작용할 수 있다. 다음 절에서 일부 유스케이스를 살펴볼 것이다.

커넥터 보기

상호대화식 모드에서 동작 중인 커넥터와 상태 모두를 열거하는 것은 때때로 도움이 된다. 커넥터를 열거하는 구문은 다음과 같다:

```
{ LIST | SHOW } [ { SOURCE | SINK } ] CONNECTORS
```

다른 말로 모든 커넥터, 소스 커넥터만 또는 싱크 커넥터만을 볼 수 있다. 현재 소스와 싱크 커넥터 모두를 생성했기 때문에 둘 모두를 열거하기 위해 다음 변형을 사용해보자:

```
SHOW CONNECTORS ;
```

두 커넥터 모두 출력에 열거되는 것을 봐야 한다:

```
 Connector Name     | Type   | Class    | Status
---------------------------------------------------------------------
 postgres-source    | SOURCE | ...      | RUNNING (1/1 tasks RUNNING)
 elasticsearch-sink | SINK   | ...      | RUNNING (1/1 tasks RUNNING)
---------------------------------------------------------------------
```

SHOW CONNECTORS 명령은 활성 커넥터와 이들의 상태를 포함하여 유용한 정보를 출력한다. 이 경우 두 커넥터 모두 RUNNING 상태의 단일 태스크를 갖고 있다. 다른 상태로는 UNASSIGNED, PAUSED, FAILED와 DESTROYED가 있다. FAILED와 같은 상태를 본다면 검사할 수 있다. 예를 들어 postgres-source 커넥터 (PostgreSQL 인스턴스를 단순히 죽여 시뮬레이션할 수 있음)가 구성된 PostgreSQL 데이터베이스와의 연결이 끊긴다면 다음과 같은 것을 볼 것이다:

```
 Connector Name  | Type   | Class | Status
-----------------------------------------------------------
 postgres-source | SOURCE | ...   | FAILED
-----------------------------------------------------------
```

커넥터에 대해 더욱 많은 정보를 어떻게 얻을까? 예를 들어 실패한 태스크를 검사하고자 한다면. 이는 커넥터를 설명하는 ksqlDB 능력이 유용한 경우이다. 다음에 이를 살펴보자.

커넥터 설명

ksqlDB는 DESCRIBE CONNECTOR 명령을 사용하여 커넥터 상태를 검색하는 것을 쉽게 한다. 예를 들어 postgres-source 커넥터가 이전 절에서 논의했듯이 기본 상태 저장소와의 연결이 끊긴다면 상태에 대한 추가적인 정보를 얻기 위해 커넥터를 설명할 수 있다. 예를 들어;

```
DESCRIBE CONNECTOR 'postgres-source';
```

에러가 존재한다면 출력에 에러 추적을 봐야 한다. 생략된 버전은 다음과 같다:

```
Name : postgres-source
Class : io.confluent.connect.jdbc.JdbcSourceConnector
Type : source
State : FAILED
WorkerId : 192.168.65.3:8083
Trace : org.apache.kafka.connect.errors.ConnectException ①
 Task ID | State | Error Trace
---------------------------------------------------------------------
 0 | FAILED | org.apache.kafka.connect.errors.ConnectException ②
```

① 이 예에서 스택 추적은 생략되어 있으며 실제 실패의 경우 예외에 대한 전체 에러 추적을 볼 수 있다.

② 태스크 특정적인 고장. 태스크는 다른 상태에 존재할 수 있다 (RUNNINIG, UNASSIGNED, FAILED 등)

보통 정상 상태의 태스크를 본다. DESCRIBE CONNECTOR 명령에 대한 출력 예는 다음과 같다:

```
Name : postgres-source
Class : io.confluent.connect.jdbc.JdbcSourceConnector
Type : source
State : RUNNING
WorkerId : 192.168.65.3:8083
 Task ID | State | Error Trace
---------------------------------
 0 | RUNNING |
---------------------------------
```

이제 커넥터 생성 및 설명 방법을 살펴보았기 때문에 이들을 제거하는 버리는 방법을 배워보자.

커넥터 버리기

이전에 등록했던 커넥터를 재구성하거나 커넥터를 영구히 삭제하기 위해서는 커넥터 버리기가 필요하다. 커넥터를 버리기 위한 구문은 다음과 같다:

```
DROP CONNECTOR [ IF EXISTS ] <identifier>
```

예를 들어 PostgreSQL 커넥터를 버리려면 다음 명령을 실행할 수 있다:

```
DROP CONNECTOR 'postgres-source';
```

커넥터를 버릴 때마다 커넥터가 실제 버려졌다는 확인을 볼 수 있다. 예를 들어:

```
 Message
------------------------------------
 Dropped connector "postgres-source"
------------------------------------
```

소스 커넥터 확인하기

PostgreSQL 소스 커넥터가 데이터를 PostgreSQL 데이터베이스에 쓰고 있는지 확인하기 위한 가장 쉬운 방법은 토픽의 내용을 출력하는 것이다. 예를 들어 Postgres 인스턴스에 titles 테이블을 생성한 후 일부 데이터로 이 테이블을 사전에 채워보자.

```
CREATE TABLE titles (
 id SERIAL PRIMARY KEY,
 title VARCHAR(120)
);
INSERT INTO titles (title) values ('Stranger Things');
INSERT INTO titles (title) values ('Black Mirror');
INSERT INTO titles (title) values ('The Office');
```

이전 문장은 ksqlDB 문이 아닌 PostgreSQL 문이다.

PostgreSQL 소스 커넥터는 이 테이블의 데이터로 titles 토픽을 자동적으로 채워야 한다. 이를 확인하기 위해 ksqlDB의 PRINT 문을 사용할 수 있다;

```
PRINT 'titles' FROM BEGINNING;
```

ksqlDB는 다음과 비슷한 출력은 보여줄 것이다:

```
Key format: JSON or KAFKA_STRING
Value format: AVRO or KAFKA_STRING
rowtime: 2020/10/28 ..., key: 1, value: {"id": 1, "title": "Stranger Things"}
rowtime: 2020/10/28 ..., key: 2, value: {"id": 2, "title": "Black Mirror"}
rowtime: 2020/10/28 ..., key: 3, value: {"id": 3, "title": "The Office"
```

출력의 첫번째 두 줄에서 ksqlDB가 titles 토픽의 레코드에 대해 키와 값 포맷 모두를 추론함을 주목하기 바란다. 레코드 키와 레코드 값에 대해 각각 StringConverter와 AvroConverter을 사용했기 때문에 (예제 9-2) 이는 예상된 출력이다.

비슷하게 싱크 커넥터 확인은 싱크 토픽에 데이터를 생산하고 그후 다운스트림 데이터 저장소에 쿼리하는 것을 필요로 한다. 독자를 위한 연습으로 이를 남겨둘 것이다 (수행 방법은 코드 저장소 참조). 이제 카프카 커넥트 클러스터와 직접 상호작용하는 방법을 살펴보고 왜 그렇게 하고 싶은지에 대한 유스케이스를 탐구할 것이다.

카프카 커넥트 클러스터와 직접 상호작용하기

어떤 경우 ksqlDB 외부에서 카프카 커넥트 클러스터와 직접 상호작용하길 원할 수 있다. 예를 들어 카프카 커넥트 엔드포인트가 종종 ksqlDB에서 사용가능하지 않은 정보를 노출하며 이을 통해 실패한 태

스크 재시작과 같은 일부 중요한 동작 태스크를 수용할 수 있다. 커넥트 API에 대한 완벽한 가이드 제공은 이 책의 범위를 벗어난 것이지만 커넥트 클러스터에 대해 실행하길 원할 수 있는 일부 예는 다음 테이블에서 볼 수 있다.

유스케이스	쿼리 예
커넥터 열거	curl -XGET localhost:8083/connectors
커넥터 설명	curl -XGET localhost:8083/connectors/elasticsearch-sink
태스크 열거	curl -XGET -s localhost:8083/connectors/elasticsearch-sink/tasks
태스크 상태 얻기	curl -XGET -s localhost:8083/connectors/elasticsearch-sink/ tasks/0/status
실패한 태스크 재시작	curl -XPOST -s localhost:8083/connectors/elasticsearch-sink/ tasks/0/restart

마지막으로 컨플루언트 스키마 레지스트리를 활용한 직렬화 포맷을 사용할 때 스키마를 검사하는 방법을 배워보자.

관리형 스키마 검사

표 9-1에서 스키마 저장을 위해 컨플루언트 스키마 레지스트리가 필요한 일부 직렬화 포맷을 보았다. 이제 카프카 커넥트는 컨플루언트 스키마 레지스트리에 스키마를 자동적으로 저장할 것이다. 표 9-2는 관리형 스키마를 검사하기 위해 스키마 레지스트리 엔드포인트에 할 수 있는 일부 쿼리를 보여준다.

표 9-2. 스키마 레지스트리 쿼리 예

유스케이스	쿼리 예
스키마 타입 열거	curl -XGET localhost:8081/subjects/
스키마 버전 열거	curl -XGET localhost:8081/subjects/titles-value/versions
스키마 버전 얻기	curl -XGET localhost:8081/subjects/titles-value/ versions/1
스키마 최신 버전 얻기	curl -XGET localhost:8081/subjects/titles-value/ versions/latest

전체 API 참조는 스키마 레지스트리 API 참조에서 찾을 수 있다.

요약

ksqlDB와 외부 시스템을 통합하는 방법을 배웠다. 커넥터 관리를 위한 다양한 ksqlDB 문 지식과 함께 카프카 커넥트에 대한 이해를 바탕으로 ksqlDB를 사용하여 데이터 처리, 변환 및 보강 방법을 배울 준비를 마쳤다. 다음 장에서는 넷플릭스의 스트림 처리 유스케이스를 살펴보고 SQL을 사용하여 스트림 처리 애플리케이션을 구축하기 위한 추가적인 ksqlDB 문을 탐구할 것이다.