10 장 ksqlDB를 사용한 스트림 처리 기본

이 장에서는 일반적인 스트림 처리 태스크를 수행하기 위한 ksqlDB 사용법을 배울 것이다. 탐구할 주제는 다음을 포함한다:

- 스트림 및 테이블 생성
- ksqlDB 데이터 타입 활용
- 간단한 부울 조건, 와일드카드 및 범위 필터를 사용한 데이터 필터링
- 복잡한 또는 중첩 구조 편평화(flattening)를 통한 데이터 재구성
- 사용가능한 필드 중 서브셋을 선택하기 위한 투영(projection) 사용
- NULL 값 처리를 위한 조건 표현 사용
- 파생 스트림 및 테이블 생성과 결과를 다시 카프카에 쓰기

이 장 말미에는 ksqlDB SQL 통용어를 사용하여 기본적인 데이터 전처리 및 변환 태스크를 처리할 준비가 잘 되어 있을 것이다. 더구나 이전 장에서 논의했던 SQL 문은 내부적으로 ksqlDB의 카프카 커넥트 통합을 활용했지만 이 장에서 사용할 모든 SQL 문은 ksqlDB의 카프카 스트림즈 통합을 활용할 것이다. 이는 ksqlDB가 얼마나 강력한지를 실제 확인하기 시작할 것이다. 왜냐하면 단지 SQL 몇 줄을 사용해 본격적인 스트림 처리 애플리케이션을 구축할 수 있을 것이기 때문이다. 평소와 같이 관련된 언어 문장을 활용하는 애플리케이션을 구축함으로써 이 주제를 탐구할 것이다. 더 이상 고민하지 말고 튜토리얼을 시작해보자.

튜토리얼: 넷플릭스에서 변경 모니터링

넷플릭스는 매년 비디오 컨텐츠에 수십억 달러를 투자하고 있다. 한 번에 많은 영화와 TV 프로그램을 제작하면서 제작 변경 발생 시 (예, 출시일 변경, 재무 변경, 출연진 일정 등) 다양한 시스템에 업데이트를 전달하는 것은 원활한 운영을 유지하기 위해 중요하다. 역사적으로 넷플릭스는 이러한 스트림 처리 유스케이스에 아파치 Flink를 사용해왔다. 그러나 넷플릭스 엔지니어 NitinSahrma와 협업을 통해 대신 ksqlDB를 사용하여 이 문제를 해결하기로 결정하였다[1].

이 애플리케이션의 목표는 단순하다. 제작 변경 스트림을 소비하고, 처리를 위해 데이터를 필터링 및 변환하며 리포팅 목적으로 데이터를 보강 및 집계하여 최종적으로 다운스트림 시스템에 처리된 데이터를 사용가능 하도록 하는 것이다. 일이 많은 것같아 보이지만 ksqlDB를 사용하는 경우 구현은 매우 간단하다.

---

[1] 비고: 이는 넷플릭스에서 운영 단계 애플리케이션이 아니고 ksqlDB를 사용하여 넷플릭스 유스케이스를 모델링하는 방법을 보여주기 위한 것이다.

이 튜토리얼에서 중점을 둘 변경 유형은 프로그램의 시즌 길이에 대한 변경일 것이다 (예를 들어, Stranger Things, 시즌 4는 원래 12 개의 에피소드로 계획되었지만 8 개 에피소드로 구성된 시즌으로 재작업될 수 있으며, 이는 출연진 일정, 현금 계획 등을 포함하여 다양한 시스템에 파급 효과를 야기한다). 이는 실세계 문제를 모델링 할 뿐만 아니라 ksqlDB 애플리케이션에서 해결해야 할 가장 일반적인 태스크를 다룰 것이기 때문에 이 예가 선택되었다.

그림 10-1은 구축할 변경 추적 애플리케이션의 아키텍처 개요를 나타낸다. 각 단계에 대한 설명은 그림 다음에서 찾을 수 있다.

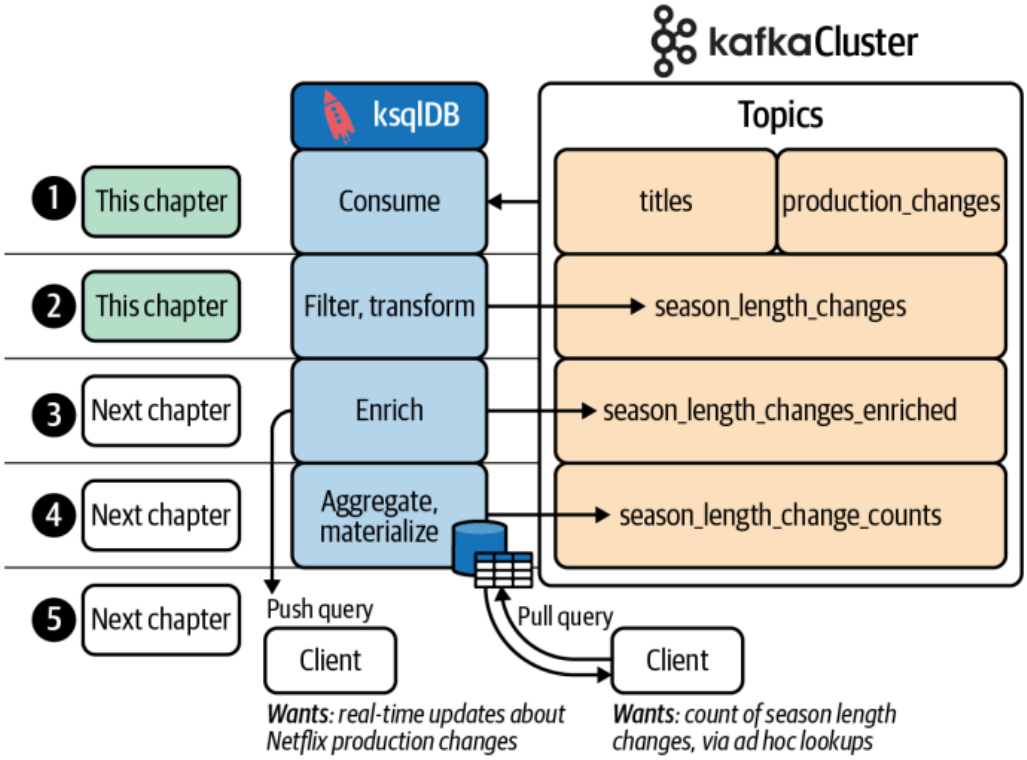

그림 10-1. 넷플릭스 변경 추적 애플리케이션 아키텍처 개요

① 애플리케이션이 2 개의 토픽으로부터 읽어 들일 것이다.

- Titles 토픽은 넷플릭스 서비스에서 호스팅되고 있는 영화와 TV 프로그램 (이 튜토리얼에서는 집합적으로 타이틀로 간주)에 대한 메타 데이터 (타이틀명, 출시일 등)를 포함하는 압축된 토픽이다.
- 현재 제작 중인 타이틀에 대한 출연진 일정, 예산, 출시일 또는 시즌 길이 변경이 발생할 때마다 Production_changes 토픽에 써진다.

② 소스 토픽에서 데이터를 소비한 후 보강을 위한 production_changes 데이터를 준비하기 위해 기본적인 전처리 (예, 필터링 및 변환)를 수행해야 한다. 필터링된 후 타이틀의 에피소드 수/시즌 길이에 대한 변경만을 포함할 전처리 스트림이 season_length_changes 토픽에 써질 것이다.

③ 이후 전처리된 데이터에 대해 데이터 보강을 수행할 것이다. 구체적으로 season_length_changes 스트림과 titles 데이터를 조인하여 다중 소스와 차원으로부터 결합된 레코드를 생성할 것이다.

④ 다음 5 분 주기로 변경 수를 세는 윈도우 및 비윈도우 집계를 수행할 것이다. 이에 따라 테이블이 구체화되고 조회 스타일의 풀 쿼리를 위해 사용가능해질 것이다.

⑤ 마지막으로 보강 및 집계된 데이터를 2 가지 다른 유형의 클라이언트에 사용가능하게 할 것이다. 첫번째 클라이언트는 ksqlDB와의 장기 연결을 사용한 푸시 쿼리를 통해 연속적인 업데이트를 수신할 것이다. 두번째 클라이언트는 전통적인 데이터베이스 조회와 더욱 유사한 단기 풀 쿼리를 사용하여 포인트 조회를 수행할 것이다.

이 예에서 소개할 개념의 수를 고려하여 튜토리얼은 2 장으로 나뉠 것이다. 이 장에서는 스트림 및 테이블 생성과 기본적인 데이터 전처리와 변환 단계 (1-2 단계)에 중점을 둘 것이다. 다음 장에서는 데이터 보강, 집계 및 푸시/풀 쿼리에 중점을 둘 것이다 (이 장에서도 푸시 쿼리가 2 단계에 필요하기 때문에 이를 논의할 것이지만).

프로젝트 환경 설정

이 장의 코드는 https://github.com/mitch-seymour/mastering-kafka-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-10
```

이 튜토리얼은 애플리케이션에 필요한 각 컴포넌트를 구동시키기 위해 다음을 포함한 도커 컴포즈를 사용한다.

- 카프카
- 스키마 레지스트리
- ksqlDB 서버
- ksqlDB CLI

각 컴포넌트를 구동시키기 위해 저장소를 복제한 후 다음 명령을 실행한다:

```
docker-compose up
```

이 장에서 논의하는 SQL 문은 ksqlDB CLI에서 실행될 것이다.

다음 명령을 사용하여 ksqlDB CLI 에서 로그인할 수 있다:

```
docker-compose exec ksqldb-cli ₩ ①
```

Ksql http://ksqldb-server:8088 ②

① ksqldb-cli 컨테이너에서 명령을 실행하기 위해 docker-compose를 사용한다. 실행할 명령은 다음 라인에 있다.

② ksql은 CLI 실행 파일 이름이다. http://ksqldb-server:8088 인수는 ksqlDB 서버 URL 이다.

CLI가 동작 중이며 튜토리얼을 시작해보자.

소스 토픽

이제 카프카 토픽에 데이터가 존재하고 ksqlDB를 사용하여 이를 처리하고자 한다. 훌륭하다! 어디서 시작하나?

시작할 가장 논리적인 장소는 소스 토픽의 데이터를 살펴보는 것이며 이후 ksqlDB에서 데이터 모델링 방법을 결정하는 것이다. 이 튜토리얼에서는 2 개의 기본 소스 토픽이 존재한다: titles와 production_changes. 표 10-1은 각 소스 토픽의 레코드 예를 보여준다.

표 10-1. 각 소스 토픽의 레코드 예

소스 토픽	레코드 예

Titles	{
	"id": 1,
	"title": "Stranger Things",
	"on_schedule": false
	}

Production_changes	{
	"uuid": 1,
	"title_id": 1,
	"change_type": "season_length",
	"before": {
	"season_id": 1,
	"episode_count": 12
	},
	"after": {
	"season_id": 1,
	"episode_count": 8
	},
	"created_at": "2021-02-08 11:30:00"
	}

이전 장에서 ksqlDB에서 지원되는 레코드 수준 직렬화 포맷을 논의했다. 이들은 AVRO, JSON, PROTOBUF, DELIMITED와 단일 값 프리미티브 (예, String, Double, Integer과 Long 타입으로 KAFKA 포맷 하에 집합적으로 그룹화됨)를 포함한다. 예를 들어 titles 토픽의 데이터가 JSON 포맷이기 때문에 레코드 직렬화 포맷으로 JSON을 사용할 수 있음을 알고 있다.

아직 다루지 않은 것은 필드 수준 데이터 타입이다. 예를 들어 표 10-1의 titles 레코드는 3 개의 필드 (id, title, on_schedule)를 갖고 있으며 각각은 다른 값 유형이다 (정수, 문자열 및 부울). 토픽 내 데이터를 모델링하기 위해 스트림 또는 테이블을 생성하기 전에 각 필드가 어떤 데이터 유형과 연관되어 있는지를 고려해야 한다. ksqlDB는 사용할 많은 내장 데이터 타입을 포함하고 있으며 다음 절에서 논의할 것이다.

데이터 타입

표 10-2는 ksqlDB 내 사용가능한 데이터 타입을 보여준다.

타입	설명
ARRAY<element-type>	동일 유형 요소 모듬 (예, ARRAY<STRING>)
BOOLEAN	부울 값
INT	32 비트 부호있는 정수
BIGINT	64비트 부호있는 정수
DOUBLE	배정밀도(64비트) IEEE 754 부동 소수점 숫자
DECIMAL(precision, scale)	구성 가능한 총 자릿수(정밀도) 및 소수점 오른쪽 자릿수(스케일)가 있는 부동 소수점 숫자
MAP<key-type, element-type>	각각이 데이터 유형(예: MAP<STRING,INT>)과 일치하는 키와 값을 포함하는 객체
STRUCT<file-name field-type [, ...]>	구조화된 필드 모음(예: STRUCT<FOO INT, BAR BOOLEAN>)
VARCHAR 또는 STRING	유니코드 문자 시퀀스(UTF8)

ksqlDB 데이터 타입에 대해 한 가지 흥미로운 사실은 일부 직렬화 포맷의 경우 데이터 타입이 선택사항이라는 것이다. 이는 AVRO, PROTOBUF와 JSON-SR을 포함하여 컨플루언트 스키마 레지스트리에 의존하는 직렬화 포맷을 포함한다. 왜냐하면 스키마 레지스트리가 이미 필드 이름과 타입을 이미 저장하고 있기 때문이며 따라서 CREAT 문에서 데이터 타입을 다시 지정하는 것은 불필요하다 (ksqlDB가 스키마 레지스트리로부터 스키마 정보를 가져올 수 있다).

이에 대한 예외는 키 칼럼을 지정해야 할 때이다. ksqlDB에서는 ksqlDB에 메시지 키에 대해 어떤 키를 읽을 지 ksqlDB에 알려주기 위해 PRIMARY KEY (테이블의 경우) 또는 KEY (스트림의 경우) 식별자를 사용할 수 있다.

따라서, PRIMARY KEY 칼럼을 포함하는 부분적인 스키마를 지정하기만 하면 되며 이는 ksqlDB에 메시지 키에 대해 이 칼럼을 읽으라고 알려주는 것이다. 그러나 값 칼럼 (예, title)은 스키마로부터 유도될 수 있다. 예를 들어 titles 데이터가 AVRO 포맷인 경우이고 관련 Avro 스키마가 스키마 레지스트리에 저장되어 있다면 titles 테이블을 생성하기 위해 다음 CREATE 문 중 하나를 사용할 수 있다.

명시적 타입	추론 타입[a]

CREATE TABLE titles (
  id INT PRIMARY KEY,
  title VARCHAR
	CREATE TABLE titles (
	  id INT PRIMARY KEY
	) WITH (

) WITH (KAFKA_TOPIC='titles', VALUE_FORMAT='AVRO', PARTITIONS=4);	KAFKA_TOPIC='titles', VALUE_FORMAT='AVRO', PARTITIONS=4);

[a] 스키마 레지스트리를 사용하는 직렬화 포맷에 대해서만 사용가능

이 책에서는 짧은 형태 버전으로 충분할 때에도 예의 명확성 향상을 위해 명시적인 버전을 사용할 것이다. 또한 CREATE { STREAM | TABLE } 구문을 곧 자세히 설명할 것이지만 ksqlDB에서 맞춤형 타입 생성 방법을 배움으로써 데이터 타입에 대한 논의를 계속해보자.

맞춤형 타입

맞춤형 타입 (PostgreSQL의 복합 타입과 유사)을 통해 필드 이름과 연관된 데이터 타입의 그룹을 지정하고 추후 선택한 이름을 사용하여 필드에 대한 동일 모음을 참조할 수 있다. 맞춤형 타입은 특히 복잡한 타입 정의 재사용에 유용하다. 예를 들어 애플리케이션이 시즌 길이에 대한 변경을 포착해야 하며 이는 구조적으로 동일한 before 및 after 상태를 가질 수 있다. 다음 레코드가 이를 보여준다:

```
{
 "uuid": 1,
 "before": { ①
 "season_id": 1,
 "episode_count": 12
 },
 "after": {
 "season_id": 1,
 "episode_count": 8
 },
 "created_at": "2021-02-08 11:30:00"
}
```

① before 필드의 구조와 after 필드의 구조는 동일하다. 이는 맞춤형 타입의 좋은 유스케이스이다.

Before와 after 필드에 대해 별도의 STRUCT 정의를 지정할 수도 있으며 또는 재사용할 수 있는 맞춤형 타입을 간단히 생성할 수 있다. 이 튜토리얼에서는 SQL 문의 가독성 향상을 위해 후자를 선택할 것이다. 다음 표는 맞춤형 타입으로 작업하기 위한 다양한 연산과 관련 SQL 구문을 보여준다.

연산	구문
맞춤형 타입 생성	CREATE TYPE <type name> AS <type>;
모든 등록된 맞춤형 타입 보기	{ LIST | SHOW } TYPES
맞춤형 타입 버리기	DROP TYPE <type name>

다음 문장을 사용하여 season_length라는 맞춤형 타입을 생성해보자:

```
ksql> CREATE TYPE season_length AS STRUCT<season_id INT, episode_count INT> ;
```

타입이 생성된 경우 이를 보기 위해 SHOW TYPES 쿼리를 사용할 수 있다:

```
ksql> SHOW TYPES ;
Type Name | Schema
----------------------------------------------------------------
SEASON_LENGTH | STRUCT <SEASON_ID INTEGER, EPISODE_COUNT INTEGER
----------------------------------------------------------------
```

맞춤형 타입을 버리고 싶다면 다음 문장을 실행할 것이다:

```
ksql> DROP TYPE season_length;
```

이 튜토리얼에서는 season_length 맞춤형 타입을 사용할 것이며 따라서 이를 버렸다면 진행 전에 재생성함을 확인하기 바란다. 이제 ksqlDB에서 다양한 데이터 타입에 대해 배웠기 때문에 애플리케이션이 필요한 스트림과 테이블을 생성할 수 있다. 다음 절에서 이 주제를 탐구할 것이다.

모음

스트림과 테이블은 카프카 스트림즈와 ksqlDB의 핵심인 2 가지 기본 추상화이다. ksqlDB에서는 이들을 모음(collection)으로 간주한다. "스트림과 테이블"에서 스트림과 테이블을 처음 살펴보았는데 이들을 생성하기 위한 ksqDB 구문을 살펴보기 전에 차이를 빨리 검토할 것이다.

테이블은 카프카 토픽 내 각 고유 키에 대해 최신 상태 또는 (집계의 경우) 계산이 기본 모음[2]에 저장되는 연속적으로 업데이트되는 데이터셋의 스냅샷으로 간주될 수 있다. 이들은 압축된 토픽에 의해 지원되며 내부적으로 카프카 스트림즈 상태 저장소를 활용한다.

테이블에 대한 인기 높은 유스케이스는 스트림을 통해 들어오는 이벤트에 대한 추가적인 컨텍스트를 제공하기 위해 조회 테이블이 참조될 수 있는 조인 기반 데이터 보강이다. 이들은 다음 장에서 논의할 집계에서 특수한 역할을 담당한다.

반면 스트림은 불변의 이벤트 시퀀스로 모델링된다. 가변 특성의 테이블과는 달리 스트림에서 각각 이벤트는 모든 다른 이벤트와 무관하다고 고려된다. 스트림은 스테이트리스로 이는 각 이벤트가 소비되고 처리된 후 잊혀진다는 것을 의미한다.

차이를 시각화하기 위해 다음 이벤트 시퀀스를 고려해보자 (키와 값은 <key, value>로 나타냈다).

이벤트 시퀀스
<K1, V1>
<K1, V2>
<K1, V3>
<K2, V1>

스트림은 이벤트에 대한 전체 이력을 모델링하는 반면 테이블은 각 고유 키에 대한 최신 상태만 포착

[2] 이들 모음은 "상태 저장소"에서 논의한 바와 같이 RocksDB 상태 저장소에 의해 지원된다.

한다. 이전 시퀀스의 스트림과 테이블 표현은 다음과 같다:

스트림	테이블
<K1, V1> <K1, V2> <K1, V3> <K2, V1>	<K1, V3> <K2, V1>

ksqlDB에서 스트림과 테이블을 생성하는 2 가지 방법이 존재한다. 이들은 카프카 토픽에 기반하여 직접 생성되거나 (이 책에서는 소스 모음이라고 간주한다) 또는 다른 스트림과 테이블로부터 파생될 수 있다 (파생 모음으로 간주한다). 이 장에서 2 가지 유형의 모음을 살펴볼 것이지만 ksqlDB에서 모든 스트림 처리 애플리케이션의 시작점인 소스 모음으로 논의를 시작할 것이다.

소스 모음 생성하기

ksqlDB로 작업을 시작하기 전에 카프카 토픽에 기반한 소스 모음을 생성해야 한다. 이는 ksqlDB에서 카프카 토픽에 직접 쿼리하지 못하기 때문이다. 스트림과 테이블에는 쿼리할 수 있다. 스트림과 테이블 생성을 구문은 다음과 같다.

```
CREATE [ OR REPLACE ] { STREAM | TABLE } [ IF NOT EXISTS ] <identifier> (
 column_name data_type [, ... ]
) WITH (
 property=value [, ... ]
)
```

이 튜토리얼에서는 스트림과 테이블 모두를 생성해야 한다. 영화와 TV 프로그램 (앞으로 이들 모두를 titles로 간주)에 대한 메타 데이터를 포함하는 titles 토픽은 각 타이틀과 연관된 현재 메타 데이터만 관심이 있기 때문에 테이블로 모델링될 것이다. 다음 문장을 사용하여 테이블을 사용할 수 있다:

```
CREATE TABLE titles (
 id INT PRIMARY KEY, ①
 title VARCHAR
) WITH (
 KAFKA_TOPIC='titles',
 VALUE_FORMAT='AVRO',
 PARTITIONS=4 ②
);
```

① PRIMARY KEY는 테이블의 키 칼럼을 지정하며 이는 레코드 키로부터 유도된다. 테이블이 가변의

업데이트 스타일 시맨틱을 가짐을 기억하기 바란다. 따라서 동일한 기본 키를 갖는 복수의 레코드를 본다면 테이블에는 단지 가장 최신 레코드만 저장될 것이다. 이에 대한 예외는 레코드가 설정되어 있지만 값이 NULL 인 경우이다. 이 경우 레코드는 툼스톤으로 고려되며 관련 키의 삭제를 트리거할 것이다. 테이블의 경우 ksqlDB는 키가 NULL로 설정된 어떤 레코드라고 무시할 것임을 주목하기 바란다 (이는 스트림의 경우에는 해당되지 않는다).

② WITH 절에서 PARTITIONS 특성을 지정하고 있기 때문에 ksqlDB는 이미 존재하는 경우 토픽을 생성할 것이다 (이 경우 토픽은 4 개의 파티션을 갖고 생성될 것이다). 또한 REPLICAS 특성을 사용한다면 토픽의 복제 수를 설정할 수도 있다. 다음 절에서 WITH 절을 보다 세부적으로 탐구할 것이다.

또한 production_changes 토픽은 스트림으로 모델링할 것이다. 이는 각 변경 이벤트의 최신 상태를 추적할 필요가 없기 때문에 이 토픽에 적합한 모음 타입이다.; 단순히 각 값을 계속 소비하고 처리해야 한다. 따라서 다음 문장을 사용하여 소스 스트림을 생성할 수 있다.

```
CREATE STREAM production_changes (
 rowkey VARCHAR KEY, ①
 uuid INT,
 title_id INT,
 change_type VARCHAR,
 before season_length,
 after season_length, ②
 created_at VARCHAR
) WITH (
 KAFKA_TOPIC='production_changes',
 PARTITIONS='4',
 VALUE_FORMAT='JSON',
 TIMESTAMP='created_at', ③
 TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
```

① 테이블과 달리 스트림은 기본 키 칼럼을 갖지 않는다. 스트림은 가변의 insert 스타일 시맨틱을 가지며 따라서 레코드를 고유하게 식별하는 것이 불가능하다. 그러나 KEY 식별자는 레코드 키 칼럼의 별칭을 지정하기 위해 사용될 수 있다 (이는 카프카 레코드 키에 해당한다).

② 여기서는 before와 after 필드 모두에 대한 타입 정의를 재사용하기 위해 맞춤형 타입인

season_length를 사용하고 있다. 두 필드는 구조적으로 동일한 복합 객체이다[3].

③ 이는 윈도우 집계와 조인 (다음 장에서 논의)을 포함하여 시간 기반 연산에 ksqlDB가 사용하는 타임스탬프를 created_at 칼럼이 포함하고 있음을 ksqlDB에 알려준다. 다음 줄의 TIMESTAMP_FORMAT는 레코드 타임스탬프의 포맷을 적절히 지정한다. TIMESTAMP와 TIMESTAMP_FORMAT 특성에 대한 자세한 정보는 다음 절에서 다룰 것이다.

CREATE { STREAM | TABLE } 문에서 사용가능한 WITH 절은 아직 논의하지 않은 여러 특성을 지원한다. 스트림과 테이블에 대한 논의를 계속하기 전에 다음 절에서 이들을 탐구해보자.

With 절

스트림 또는 테이블을 생성할 때 ksqlDB의 WITH 절을 사용하여 여러 특성을 구성할 수 있다. 표 10-3은 여러분이 사용할 것 같은 보다 중요한 특성 중 일부를 보여준다.

표 10-3. WITH 절에서 지원되는 특성

특성 이름	설명	필수
KAFKA_TOPIC	소비하길 원하는 데이터를 포함하는 카프카 토픽	예
VALUE_FORMAT	소스 토픽 내 데이터의 직렬화 포맷 (예, AVRO, PROTOBUF, JSON, JSON_SR, KAFKA)	예
PARTITIONS	파티션 수를 지정하여 소스 토픽을 생성하는 경우 이를 지정한다.	아니오
REPLICAS	복제 수를 지정하여 소스 토픽을 생성하는 경우 이를 지정한다.	아니오
TIMESTAMP	ksqlDB가 시간 기반 연산 (예, 윈도우 조인)에 사용하는 타임스탬프를 포함하는 칼럼 이름. 이 특성을 지정하지 않는 경우 레코드 메타 데이터의 타임스탬프를 사용할 것이다. 칼럼이 BIGINT 타입인 경우 ksqlDB는 추가 구성없이 타임스탬프를 파싱할 방법을 알고 있다. 칼럼이 VARCHAR 타입인 경우 TIMESTAMP_FORMAT 특성을 적절히 설정해야 한다.	아니오
TIMESTAMP_FORMAT	타임스탬프 포맷. Java.time.format.DateTimeFormatter가 지원하는 모든 포맷이 유효하다 (예, yyyy-MM-dd HH:mm:ss)	아니오
VALUE_DELIMITER	필드 구분자 역할을 하는 문자. 콤마(,)가 기본 구분자로 스페이스와 탭도 유효한 값이다.	아니오

스트림 또는 테이블을 생성했다면 이들을 검사하는 방법을 배움으로써 논의를 계속해보자.

스트림과 테이블로 작업하기

상호대화식 모드에서 생성한 모음을 보고 설명하는 것은 유용하다. 일부 경우 이들이 필요치 않거나 동일한 이름으로 스트림 또는 테이블을 재생성하고 싶기 때문에 모음을 삭제하길 원할 수도 있다. 이 절에서는 이러한 유스케이스를 논의하고 관련 SQL 문에 대한 응답 예를 보일 것이다. 우선 애플리케

[3] 여기서 복합이라는 것은 non-primitive 타입과 관련이 있다. 이 경우 before와 after 필드 모두 맞춤형 타입을 사용하여 별칭을 지정한 STRUCT를 사용하여 표현된다. MAP과 ARRAY는 복합 타입이다.

이션에 정의된 활성 스트림 및 테이블을 보는 방법을 살펴보자.

스트림과 테이블 보기

종종 현재 등록된 스트림과 테이블 모두에 대한 정보를 보고 싶을 수 있다. ksqlDB는 이를 위해 2 개의 문을 갖고 있으며 구문은 다음과 같다.

```
{ LIST | SHOW } { STREAMS | TABLES } [EXTENDED];
```

LIST와 SHOW는 교환가능하며 이 책에서는 SHOW를 사용할 것이다.

예를 들어 튜토리얼에 필요한 소스 스트림과 테이블 모드를 생성하였는데 ksqlDB CLI 내 이들 모음에 대한 정보를 보기 위해 SHOW 문을 사용해보자. 각 문의 출력은 다음 코드 블록의 명령 다음에서 바로 볼 수 있다.

```
ksql> SHOW TABLES ;
 Table Name | Kafka Topic | Format | Windowed
-------------------------------------------
 TITLES | titles | AVRO | false
-------------------------------------------
ksql> SHOW STREAMS ;
 Stream Name | Kafka Topic | Format
-----------------------------------------------------------
 PRODUCTION_CHANGES | production_changes | JSON
-----------------------------------------------------------
```

위에서 볼 수 있듯이 스트림과 테이블 열거의 출력은 매우 최소한이다. 모음에 대해 더욱 많은 정보가 필요한 경우 다음과 같이 SHOW 문의 변형인 EXTENDED를 사용할 수 있다.

```
ksql> SHOW TABLES EXTENDED; ①
Name : TITLES
Type : TABLE
Timestamp field : Not set - using <ROWTIME>
Key format : KAFKA
Value format : AVRO
Kafka topic : titles (partitions: 4, replication: 1)
Statement : CREATE TABLE titles (...) ②
 Field | Type
----------------------------------------
 ID | INTEGER (primary key)
 TITLE | VARCHAR(STRING)
----------------------------------------
Local runtime statistics                    ③
------------------------
```

messages-per-sec: 0.90 total-messages: 292 last-message: 2020-06-12...

① SHOW STREMS EXTENDED 또한 지원되는데 출력 포맷이 비슷하기 때문에 스트림 특정적인 예는 생략했다.

② 전체 DDL 문을 실제 출력에서 볼 수 있지만 여기서는 간략함을 위해 생략했다.

③ 런타임 통계는 여기서는 나타내지 않았지만 consumer-total-message-bytes, failed-messages-per-ser, last-failed 등을 포함하여 많은 필드를 포함하고 있다. 이는 스트림과 테이블의 처리량과 오류율 (예, 역직렬화 에러)에 대략적으로 살펴보는데 유용하다. 아무 것도 사용가능하지 않은 경우 이들 통계가 생략될 수 있음을 주목하기 바란다 (스트림 또는 테이블에 활동이 없는 경우).

SHOW 명령이 ksqlDB 클러스터 내 모든 스트림/테이블에 대한 데이터를 보는데 사용되기 때문에 이전 출력은 현재 등록된 각 스트림/테이블에 대해 반복될 것이다.

이제 클러스터 내 모든 스트림과 테이블을 보는 방법을 배웠으며 한 번에 하나의 특정 스트림과 테이블을 기술하는 방법을 살펴보자.

스트림과 테이블 기술하기

모음을 기술하는 것은 SHOW 명령과 비슷하지만 한 번에 하나의 스트림 또는 테이블 인스턴스에 대해 동작한다. 스트림 또는 테이블을 기술하기 위한 구문은 다음과 같다:

```
DESCRIBE [EXTENDED] <identifier> ①
```

① <identifier>은 스트림 또는 테이블의 이름이다. 이 명령에 STREAM 또는 TABLE 키워드가 없음을 주목하기 바란다. ksqlDB는 동일 이름의 스트림과 테이블 생성을 허용하지 않기 때문에 이 명령에서 모음 타입을 구별할 필요가 없다. 단순히 기술하기 원하는 스트림 또는 테이블의 고유 식별자만 알면 된다.

예를 들어 다음 문장을 사용하여 titles 테이블을 기술할 것이다:

```
ksql> DESCRIBE titles ;
Name : TITLES
 Field | Type
----------------------------------------
 ID | INTEGER (primary key)
 TITLE | VARCHAR(STRING)
----------------------------------------
```

더욱 많은 정보가 필요한 경우, 예를 들어 런타임 통계, 다음 코드 스니펫에서 보듯이 DESCRIBE EXTENDED 변형을 사용할 수 있다. 여기서는 지정된 스트림/테이블의 출력만 보여주고 ksqlDB 클러스터에 등록된 모든 스트림과 테이블을 보여주지 않는다는 것을 제외하고는 현 시점에서 SHOW { STREAMS | TABLE } EXTENDED 출력과 동일하기 때문에 DESCRIBE EXTENDED 명령의 출력은 생략했다.

```
ksql> DESCRIBE EXTENDED titles
```

보다 전통적인 데이터베이스에서 다른 일반적인 태스크는 더 이상 필요치 않은 데이터베이스 객체를 버리는 것이다. 비슷하게 다음 절에서 볼 것이지만 ksqlDB를 통해 스트림과 테이블을 버릴 수 있다.

스트림과 테이블 변경하기

종종 기존 모음을 변경할 수도 있다. ksqlDB는 이를 위해 ALTER 문을 포함하고 있다. 구문은 다음과 같다:

```
ALTER { STREAM | TABLE } <identifier> alterOption [,...]
```

ksqlDB 버전 0.14.0에서 ALTER 문을 통해 수행될 수 있는 연산은 단지 칼럼을 추가하는 것이며, 이는 향후 확장될 수 있다. ALTER 문을 사용해 칼럼을 추가하는 방법 예는 다음과 같다:

```
ksql> ALTER TABLE titles ADD COLUMN genre VARCHAR; ①
 Message
----------------------
 Table TITLES altered.
----------------------
```

① titles 테이블에 이름이 genre인 VARCHAR 칼럼 추가

스트림과 테이블 버리기

부모님께서 "내가 너를 이 세상에 데려왔으니 너를 세상밖으로 데리고 나갈 수 있다" 라고 말씀하시곤 했다. 비슷하게 여러분이 스트림과 테이블을 생성했고 이를 버리길 원한다면 DROP { STREAM | TABLE } 명령을 사용할 수 있다. 전체 구문은 다음과 같다.

```
DROP { STREAM | TABLE } [ IF EXISTS ] [DELETE TOPIC]
```

예를 들어 production_changes 스트림과 해당 토픽을 버리고 싶다면 다음 문장을 실행할 것이다[4]:

```
ksql> DROP STREAM IF EXISTS production_changes DELETE TOPIC ;
 Message
------------------------------------------------------------------------
 Source `PRODUCTION_CHANGES` (topic: production_changes) was dropped.
------------------------------------------------------------------------
```

선택사항인 DELETE TOPIC 절에 주의하기 바란다. 이 절이 포함된 경우 ksqlDB가 토픽을 지울 것이다. 토픽이 아니라 단지 테이블만 삭제하는 경우 DELETE TOPIC을 배제할 수 있다.

이제 고수준에서 모음으로 작업하는 방법에 대해 많은 부분을 다루었다. 예를 들어 ksqlDB의 SHOW

---

[4] 튜토리얼을 따라 이 문장을 실행한다면 튜토리얼 나머지를 진행하기 전에 production_changes를 재생성하기 바란다.

와 DESCRIBE 문을 사용하여 스트림 또는 테이블의 존재를 확인하고 메타 데이터를 검사할 수 있으며 CREATE과 DROP 문을 사용하여 생성 및 삭제할 수 있다. 이제 기본 스트림 처리 패턴과 관련 SQL 문을 탐구함으로써 스트림과 테이블로 작업하는 추가적인 방법을 살펴보자.

기본 쿼리

이 절에서는 ksqlDB에서 데이터 필터링 및 변환하는 기본적인 방법을 살펴볼 것이다. 이 시점에서 변경 추적 애플리케이션이 2 개의 다른 카프카 토픽 titles와 production_changes에서 데이터를 소비하고 있음을 상기하기 바란다. 이들 토픽 각각에 대해 소스 모음 (titles 테이블과 production_changes 스트림)을 생성함으로써 애플리케이션의 1 단계를 이미 완료했다 (그림 10-1 참조). 다음으로 애플리케이션의 2 단계를 다룰 것인데 이는 시즌 길이 변경에 대해서만 production_changes 스트림을 필터링하고 데이터를 보다 간단한 포맷으로 변환한 후 필터링 및 변환된 스트림을 season_length_changes라는 새로운 토픽에 작성하는 것을 필요로 한다.

개발 목적으로 특히 유용한 문장을 살펴보면서 시작해보자: INSERT VALUES 문.

INSERT VALUES

스트림 또는 테이블에 값을 삽입하는 것은 모음을 데이터로 사전에 채워야 할 때 매우 유용하다. 전통적인 데이터베이스에 익숙하다면 알고 있었던 insert 시맨틱과는 약간 다르다. 스트림과 테이블 모두에 대해 레코드는 카프카 토픽에 추가되고 테이블은 단지 각 키의 최신 표현만 저장하며 따라서 테이블에 대해 upsert 연산의 역할을 한다 (ksqlDB에 별도의 UPDATE 문이 없는 이유).

이 튜토리얼에서는 다양한 유형의 SQL 문을 통한 실습을 용이하게 하기 위해 테스트 데이터로 titles 테이블과 production_changes 스트림을 사전에 채울 것이다. 모음에 값을 삽입하는 구문은 다음과 같다:

```
INSERT INTO <collection_name> [ ( column_name [, ...]] ) ]
VALUES (
 value [,...]
);
```

애플리케이션은 타이틀의 시즌 길이에만 관심이 있는데, 따라서 다음 레코드를 삽입해보자:

```
INSERT INTO production_changes (
 uuid,
 title_id,
 change_type,
 before,
 after,
 created_at
) VALUES (
 1,
```

```
  1,
  'season_length',
  STRUCT(season_id := 1, episode_count := 12),
  STRUCT(season_id := 1, episode_count := 8),
  '2021-02-08 10:00:00'
);
```

모든 다른 유형의 변경을 필터링해야 하기 때문에 출시일 변경도 삽입해보자. 이는 뒤에 필터 조건을 테스트할 때 유용할 것이다. ksqlDB에 의해 자동으로 생성되는 특수 의사 칼럼인 ROWKEY와 ROWTIME에 대한 값을 또한 지정함으로써 INSERT 문의 약간 다른 변형을 사용할 것이다.

```
INSERT INTO production_changes (
  ROWKEY,
  ROWTIME,
  uuid,
  title_id,
  change_type,
  before,
  after,
  created_at
) VALUES (
  '2',
  1581161400000,
  2,
  2,
  'release_date',
  STRUCT(season_id := 1, release_date := '2021-05-27'),
  STRUCT(season_id := 1, release_date := '2021-08-18'),
  '2021-02-08 10:00:00'
);
```

마지막으로 적절한 평가를 위해 타이틀에 일부 데이터도 삽입해보자. 여기서는 칼럼 이름이 생략된 INSERT INTO VALUES 변형을 사용할 것이다. 대신 정의된 순서대로 각 칼럼에 대해 값을 제공할 것이다 (예, 첫번째 값은 id를 지정하고 두번째 값은 title에 해당할 것이다).

```
INSERT INTO titles VALUES (1, 'Stranger Things');

INSERT INTO titles VALUES (2, 'Black Mirror');

INSERT INTO titles VALUES (3, 'Bojack Horseman');
```

이제 테이블과 스트림을 일부 데이터로 미리 채웠기 때문에 모음에 대해 쿼리를 실행시키는 재미있는 부분으로 넘어가 보자.

단순한 SELECT (일시적인 푸시 쿼리)

실행시킬 수 있는 간단한 쿼리 형태는 일시적인 (영구적이지 않은) 푸시 쿼리이다. 이는 문장 끝에 EMIT CHANGES 절을 갖는 간단한 SELECT 문이다. 일시적인 푸시 쿼리 구문은 다음과 같다:

```
SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN join_collection ON join_criteria ]
[ WINDOW window_expression ]
[ WHERE condition ]
[ GROUP BY grouping_expression ]
[ PARTITION BY partitioning_expression ]
[ HAVING having_expression ]
EMIT CHANGES
[ LIMIT count ];
```

Production_changes 스트림에서 모든 레코드를 선택하여 시작해보자:

```
ksql> SET 'auto.offset.reset' = 'earliest'; ①
ksql> SELECT * FROM production_changes EMIT CHANGES ; ②
```

① 이를 통해 토픽의 처음부터 읽을 수 있으며 이는 데이터를 이미 삽입했기 때문에 유용하다.

② 일시적인 쿼리를 실행시킨다.

이 장 후반에 논의할 영구 쿼리와 달리 푸시 쿼리는 ksqlDB 재시작 시 살아있지 않을 것이다.

이전 쿼리를 실행시키면 화면에서 최초 쿼리 출력이 나오는 것을 볼 것이며 새로운 데이터가 들어오길 기다리면서 쿼리가 계속 진행될 것이다. 예제 10-1은 쿼리 출력을 보여준다.

표 10-1. 일시적인 푸시 쿼리 출력

```
+------+----+--------+-------------+--------------------+--------------------+-------------+
|ROWKEY|UUID|TITLE_ID|CHANGE_TYPE  |BEFORE              |AFTER               |CREATED_AT   |
+------+----+--------+-------------+--------------------+--------------------+-------------+
|2     |2   |2       |release_date |{SEASON_ID=1,       |{SEASON_ID=1,       |2021-02-08...|
                                     EPISODE_COUNT=null}  EPISODE_COUNT=null}

|null  |1   |1       |season_length|{SEASON_ID=1,
                                     EPISODE_COUNT=12}  |{SEASON_ID=1,       |2021-02-08...|
                                                          EPISODE_COUNT=8}
```

다른 CLI 세션을 열어 production_changes 스트림에 다른 INSERT VALUES 문을 실행한다면 결과 출력이 자동으로 업데이트될 것이다.

테이블에 대한 일시적인 푸시 쿼리도 동일하게 동작한다. Titles 테이블에 대해 SELECT 문을 실행시켜 확인할 수 있다:

```
ksql> SELECT * FROM titles EMIT CHANGES ;
```

다음 코드에서 보듯이 테이블의 최초 컨텐츠가 방출될 것이다. 또한 새로운 데이터가 도착 시 출력이 이에 따라 업데이트될 것이다:

```
+----+-----------------+
|ID  |TITLE            |
+----+-----------------+
|2   |Black Mirror     |
|3   |Bojack Horseman  |
|1   |Stranger Things  |
```

이 절에서 논의했던 것과 같이 단순한 SELECT 문은 스트림 처리 애플리케이션 구축에 유용한 시작점이지만 데이터 처리는 종종 다른 방식으로 데이터를 변환하는 것을 필요로 한다. 다음 절에서 기본적인 데이터 변환 태스크를 탐구할 것이다.

투영

아마도 가장 간단한 데이터 변환 형태는 스트림 또는 테이블에서 사용가능한 칼럼 중 일부만을 선택하여 이를 통해 다운스트림 연산에 대한 데이터 모델을 단순화하는 것을 포함한다. 이는 투영으로 불리며 SELECT * 구문을 원하는 명시적인 칼럼명으로 대체하는 것을 필요로 한다. 예를 들어 이 튜토리얼에서는 production_changes 스트림의 title_id, before, after와 created_at 칼럼으로 작업해야 하며 따라서 이들 칼럼을 새로운 단순화된 스트림으로 투영하는 다음 문장을 작성할 수 있다.

```
SELECT title_id, before, after, created_at
FROM production_changes
EMIT CHANGES ;
```

이 쿼리의 출력은 단순화된 스트림을 보여준다.

```
+---------+--------------------+----------------------------------+
|TITLE_ID |BEFORE              |AFTER               |CREATED_AT   |
+---------+--------------------+--------------------+-------------+
|2        |{SEASON_ID=1,       |{SEASON_ID=1,       |2021-02-08...|
           EPISODE_COUNT=null}   EPISODE_COUNT=null}

|1        |{SEASON_ID=1,       |{SEASON_ID=1,       |2021-02-08...|
           EPISODE_COUNT=12}     EPISODE_COUNT=8}
```

위에서 볼 수 있듯이 스트림은 애플리케이션을 위해 실제 필요하지 않은 일부 레코드도 아직까지 포함하고 있다. 단지 season_lentgh 변경에서만 관심이 있지만 TITLE_ID=2인 레코드는 예제 10-1에서 보듯이 release_date 변경이다. 이 유스케이스를 다룰 수 있는 방법을 보기 위해 ksqlDB의 필터링 기능을 탐구해보자.

필터링

ksqlDB SQL 통용어는 스트림과 테이블 필터링을 위해 사용될 수 있는 WHERE 절을 포함하고 있다. 애플리케이션이 넷플릭스에서 특정 유형의 제작 변경에만 관심이 있기 때문에 (season_length 변경) 관련 레코드를 필터링하기 위해 에제 10-2의 문장을 사용할 수 있다.

예제 10-2. 레코드 필터링을 위해 WHERE 절을 사용한 ksqlDB 문

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

예제 10-3은 이 쿼리의 출력을 보여준다.

예제 10-3. 필터링된 결과

```
+---------+--------------------+----------------------------------+
|TITLE_ID |BEFORE              |AFTER               |CREATED_AT   |
+---------+--------------------+--------------------+-------------+
|1        |{SEASON_ID=1,       |{SEASON_ID=1,       |2021-02-08...|
           EPISODE_COUNT=12}    EPISODE_COUNT=8}
```

이제 더 진행하기 전에 WHERE 문의 변형을 탐구해보자.

> 뒤에 논의할 WHERE 문 변형 외에 풀 쿼리에 대해 현재 지원되는 IN 서술어도 존재한다. IN 서술어는 향후 언젠가는 푸시 쿼리에서도 지원되리라 예상한다. IN 서술어의 목적은 복수의 값을 일치시키는 것이다 (예, WHERE id IN (1, 2, 3)). 0.14.0 이상의 ksqlDB 버전을 사용하고 있다면 그 버전이 푸시 쿼리에 대해 IN 서술어를 지원하는지 확인하기 위해 ksqlDB 체인지로그를 확인할 수 있다.

와일드카드

ksqlDB는 와일드 카드 필터링도 지원하는데 칼럼 값의 일부만을 일치시키려고 할 때 유용하다. 이 경우 0 이상의 문자를 표현하는 % 문자를 사용하는 LIKE 연산자는 다음과 같이 더욱 강력한 필터링에 사용될 수 있다:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type LIKE 'season%' ①
EMIT CHANGES ;
```

① 단어 season으로 시작하는 change_type 칼럼이 있는 레코드를 일치시킨다.

논리 연산자

AND/OR 논리 연산자를 사용하여 다중 필터 조건을 제공할 수 있다. 또한 다중 필터 조건을 단일 논리 표현으로 그룹화하기 위해 괄호 또한 사용할 수 있다. 다음 SQL 문은 이들 개념 각각을 보여준다:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE NOT change_type = 'release_date' ①
AND ( after->episode_count >= 8 OR after->episode_count <=20 ) ②
```

```
EMIT CHANGES ;
```

① change_type이 release_date을 제외한 어떤 값으로 설정되어 있는 레코드만 포함한다

② 괄호 내 두 조건을 함께 평가한다. 이 경우 새로운 에피소드 카운트가 8과 20 (둘 다 포함) 사이인 변경을 필터링한다.

마지막으로 두번째 필터링 조건이 에피소드 카운트가 특정 범위 내에 있는 레코드를 포착하지만 범위 필터를 구현하는 것이 더 나은 옵션이다.

Between (범위 필터)

특정 숫자 또는 문자 범위 내에 있는 레코드를 필터링하려면 BETWEEN 연산자를 사용할 수 있다. 범위는 inclusive로 BETWEEN 8 AND 20은 8과 20을 포함하여 8과 20 사이에 값을 일치시킬 것이다. SQL 문 예는 다음과 같다:

```
SELECT title_id, before, after, created_at
FROM production_changes
WHERE change_type = 'season_length'
AND after->episode_count BETWEEN 8 AND 20
EMIT CHANGES ;
```

튜토리얼의 경우 예제 10-2의 필터링 조건으로 충분할 것이지만 다른 유형의 필터링 조건을 사용하는 방법을 알고 있어야 한다.

이제 스트림이 필터링되었고 예제 10-3의 출력에서 볼 수 있듯이 두 칼럼 (before와 after)은 복합 다변수 구조를 갖고 있다 (예, {SEASNON_ID=1, EPISODE_COUNT=12}). 이는 데이터 변환 및 전처리의 다른 일반적인 유스케이스를 강조한다: 복합 구조를 보다 단순한 구조로 편평화 및 풀기. 다음에 이 유스케이스를 살펴볼 것이다.

복잡 구조 편평화/풀기

값 편평화는 복합 구조 (예, STRUCT) 내 중첩 필드를 최상위 단일 값 칼럼으로 분해하는 것이다. 투영과 같이 이는 다운스트림 연산을 위한 데이터 모델을 단순화하는 유용한 방법이다.

예를 들어 이전 절에서 단일 레코드에 대해 after 칼럼에서 다음과 같은 복합 값을 보았다:

```
{SEASON_ID=1, EPISODE_COUNT=8}
```

->연산자를 사용하여 중첩 구조 필드에 액세스할 수 있으며 이들 값을 다음 쿼리를 사용하여 별도의 칼럼으로 가져올 수 있다:

```
SELECT
 title_id,
 after->season_id,
 after->episode_count,
```

```
 created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

다음 출력에서 볼 수 있듯이 이는 다변수 데이터 포인트 ({SEASON_ID=1, EPISODE_COUNT=8})가 2개의 별도 칼럼에 존재하기 때문에 편평화 효과를 갖는다.

```
+----------+----------+-------------+-------------------+
|TITLE_ID  |SEASON_ID  |EPISODE_COUNT |CREATED_AT         |
+----------+----------+-------------+-------------------+
|1         |1          |8             |2021-02-08 10:00:00|
```

Production_changes 스트림에 대해 추후 보다 고급 연산을 위해 사용할 수 있는 변환된 버전을 생성하는 중에 있는데, 결과를 카프카에 역으로 쓰기 전에 기본적인 스트림 처리 유스케이스에 유용한 다른 유형의 식을 살펴보자.

조건식

ksqlDB는 몇몇 조건식도 지원한다. 이들 식에 대해 많은 다른 사용이 있지만 한 가지 일반적인 유스케이스는 NULL 칼럼에 대체 값을 제공함으로써 스트림 또는 테이블의 잠재적인 데이터 무결성 문제를 다루는 것이다.

예로 before 또는 after 칼럼 중 하나에서 season_id가 종종 NULL인 production_changes 스트림의 잠재적 데이터 무결성 문제를 다룬다고 하자. 이 경우 이러한 에러 케이스를 다루기 위해 3 가지 다른 유형의 조건식 중 하나를 사용하고 필요 시 대체 season_id 값으로 폴백할 수 있다. COALESCE 함수를 살펴봄으로써 시작해보자.

COALESCE

COALESCE 함수는 값 목록에서 첫번째 non-NULL 값을 반환하는데 사용될 수 있다. COALESCE의 함수 시그니처는 다음과 같다:

```
COALESCE(first T, others T[])
```

예를 들어 non-NULL season_id를 선택하기 위한 폴백 로직을 구현하기 위해 다음과 같이 SELECT 문을 업데이트할 수 있다.

```
SELECT COALESCE(after->season_id, before->season_id, 0) AS season_id
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

이 경우 after->season_id가 NULL이라면 before->season_id로 폴백하며, before->season_id도 NULL이라면 기본 값 0으로 폴백할 것이다.

IFNULL

IFNULL 함수는 단지 단지 하나의 폴백 값만 갖는다는 것을 제외하고는 COALESCE와 비슷하다.

IFNULL의 함수 시그니처는 다음과 같다:

```
IFNULL(expression T, altValue T)
```

After->season_id가 NULL일 때 before->season_id로 폴백하려면 다음과 같이 SELECT 문을 업데이트 할 수 있다.

```
SELECT IFNULL(after->season_id, before->season_id) AS season_id
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

CASE 문

ksqlDB의 모든 조건식 중에서 CASE 문은 가장 강력하다. 이를 통해 여러 부울 조건을 평가하고 조건이 참인 최초 값을 반환할 수 있다. COALESCE와 IFNULL과 달리 CASE 문에서 평가된 조건들은 단순한 NULL 검사로 제한되지 않는다:

CASE 문의 구문은 다음과 같다:

```
CASE expression
 WHEN condition THEN result [, ...]
 [ELSE result]
END
```

예를 들어 다음 코드는 여러 폴백 조건을 포함하고 after->season_id와 before->season-id 값 모두가 NULL인 경우 0을 반환하는 CASE 문 사용법을 보여준다:

```
SELECT
 CASE
 WHEN after->season_id IS NOT NULL THEN after->season_id
 WHEN before->season_id IS NOT NULL THEN before->season_id
 ELSE 0
 END AS season_id
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

단순한 NULL 검사를 수행한다면 COALESCE 또는 IFNULL을 사용하는 것이 더욱 합리적이다. 이 튜토리얼에서는 IFNULL 변형을 사용할 것이다.

이제 필터링 및 변환된 production_changes 스트림을 역으로 카프카에 쓸 준비를 마쳤다.

결과를 카프카에 다시 쓰기 (영구 쿼리)

지금까지 일시 쿼리로 작업해왔다. 이 쿼리는 SELECT로 시작하고 출력은 클라이언트에 반환되며 카프카에 다시 써지지 않는다. 또한 일시 쿼리는 서버 재시작시 살지 못한다.

ksqlDB는 결과를 카프카에 쓰고 서버 재시작 시 살아있는 소위 영구 쿼리를 생성할 수 있도록 한다. 이는 필터링된, 변환된 및/또는 보강된 스트림을 다른 클라이언트에 사용가능하도록 할 때 유용하다. 쿼리 결과를 역으로 카프카에 쓰기 위해 파생 모음을 생성할 수 있다. 다음 절에서 이들에 대해 논의할 것이다.

파생 모음 생성하기

파생 모음은 다른 스트림과 테이블로부터 스트림과 테이블을 생성한 결과이다. 칼럼 스키마를 지정하지 않고 AS SELECT 절이 추가되기 때문에 소스 모음을 생성한 방식과 구문은 약간 차이가 있다. 파생 모음 생성을 위한 전체 구문은 다음과 같다:

```
CREATE { STREAM | TABLE } [ IF NOT EXISTS ] <identifier>
WITH (
 property=value [, ... ]
)
AS SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN join_collection ON join_criteria ]
[ WINDOW window_expression ]
[ WHERE condition ]
[ GROUP BY grouping_expression ]
[ PARTITION BY partitioning_expression ]
[ HAVING having_expression ]
EMIT CHANGES
[ LIMIT count ];
```

파생 스트림 생성을 위한 쿼리는 종종 2 개의 약어 중 하나와 관련이 있다:

- 파생 스트림 생성에는 CSAS 쿼리 (CREATE STREAM AS SELECT)가 사용된다.
- 파생 테이블 생성에는 CTSAS 쿼리 (CREATE TABLE AS SELECT)가 사용된다.

Season_length_changes 파생 스트림을 생성하기 위해 지금까지 배웠던 필터 조건, 데이터 변환과 조건식을 적용해보자. 이는 애플리케이션에 필요한 변경 유형만을 포함할 것이다.

```
CREATE STREAM season_length_changes
WITH (
 KAFKA_TOPIC = 'season_length_changes',
 VALUE_FORMAT = 'AVRO',
```

```
  PARTITIONS = 4,
  REPLICAS = 1
) AS SELECT
  ROWKEY,
  title_id,
  IFNULL(after->season_id, before->season_id) AS season_id,
  before->episode_count AS old_episode_count,
  after->episode_count AS new_episode_count,
  created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

① 파생 모음 생성은 소스 모음 생성과 구문적으로 비슷하다 (예, CREATE STREAM과 CREATE TABLE).

② 파생 모음 생성 시 WITH 절도 지원된다.

③ 파생 스트림을 season_length_changes 토픽에 쓴다.

④ AS SELECT 부분은 파생 스트림 또는 테이블을 채우기 위한 쿼리를 정의한 부분이다.

⑤ 투영에는 키 칼럼을 포함해야 한다. 이는 카프카 레코드에서 키에 어떤 값이 사용되는지를 ksqlDB에 알려준다.

⑥ 개별 칼럼을 지정함으로써 투영을 사용하여 원래 스트림 또는 테이블을 재구성한다.

⑦ IFNULL 조건식 사용을 통해 특정 데이터 무결성 문제 유형을 처리할 수 있다. 또한 칼럼에 명시적인 이름을 제공하기 위해 AS 문을 사용한다.

⑧ 중첩 또는 복합 값을 편평화함으로써 다운스트림 프로세서와 클라이언트가 작업하기 쉽게 만든다.

⑨ 스트림 편평화를 통해 다른 일반적인 스트림 처리 유스케이스인 레코드의 특정 서브셋을 포착할 수 있다.

파생 모음을 생성했다면 쿼리가 생성되었다는 확인을 봐야 한다:

```
 Message
----------------------------------------------------
 Created query with ID CSAS_SEASON_LENGTH_CHANGES_0
----------------------------------------------------
```

위에서 볼 수 있듯이 이전 CSAS 문을 실행시켰을 때 ksqlDB는 연속적인/영구 쿼리를 생성했다. ksqlDB가 생성한 쿼리 (예를 들어 CSAS_SEASON_LENGTH_CHANGES_0)는 제공된 SQL 문을 처리하기 위해 ksqlDB에 의해 동적으로 생성된 카프카 스트림즈 애플리케이션이다. 이 장 완료 전에 쿼리와 상호작용하기 위해 실행시킬 수 있는 일부 문을 빨리 논의해보자.

쿼리 보기

파생 모음을 생성했든 또는 간단한 일시 쿼리를 실행했든 ksqlDB 클러스터 내 활성 쿼리와 이들의 현재 상태를 보는 것은 때로 유용하다.

활성 쿼리를 보기 위한 구문은 다음과 같다:

```
{ LIST | SHOW } QUERIES [EXTENDED];
```

이제 필터링 및 변환된 season_length 변경을 포함하는 파생 스트림을 생성했기 때문에 다음 문을 실행하여 쿼리에 대한 정보를 볼 수 있다.

```
ksql> SHOW QUERIES;

 Query ID                      | Query Type | Status    | .... ❶
-------------------------------------------------------------
 CSAS_SEASON_LENGTH_CHANGES_0 | PERSISTENT | RUNNING:1 | .... ❷
-------------------------------------------------------------
```

① SHOW_QUERIES 출력의 일부 칼럼은 간결함을 위해 생략했다. 그러나 출력에는 영구 쿼리가 쓰고 있는 카프카 토픽, 원래 쿼리 문자열 등을 포함하여 보다 많은 정보가 나타난다.

② 특정 연산 (예, 쿼리 종료 및 설명)을 위해 쿼리 ID (CSAS_SEASON_LENGTH_CHANGES_0)가 필요하다. CSAS 또는 CTAS 문을 실행한 경우 쿼리 타입은 PERSISTENT로 쿼리가 일시적인 경우 (예, CREATE 대신 SELECT로 시작) PUSH로 보일 것이다. 마지막으로 이 쿼리의 상태는 RUNNING 상태임을 나타낸다. 다른 유효한 쿼리 상태는 ERROR과 UNRESPONSIVE가 있다.

쿼리 설명

활성 쿼리에 대해 SHOW QUERIES 문이 제공하는 것보다 많은 정보가 필요한 경우 쿼리를 설명할 수 있다. 쿼리 설명을 위한 구문은 다음과 같다:

```
EXPLAIN { query_id | query_statement }
```

예를 들어 이전 절에서 생성했던 쿼리를 설명하기 위해서는 다음 문을 실행시킬 수 있다:

```
ksql> EXPLAIN CSAS_SEASON_LENGTH_CHANGES_0 ;
```

또한 실제 동작 중이지 않은 가상 쿼리 또한 설명할 수도 있다. 예를 들어:

```
ksql> EXPLAIN SELECT ID, TITLE FROM TITLES ;
```

각 경우에서 출력은 매우 장황한데 따라서 여러분을 위한 연습으로 남길 것이다. 포함된 정보에는 쿼리에 포함된 필드 이름과 타입, 쿼리를 실행 중인 ksqlDB 서버의 상태 (가상 쿼리를 설명하는 경우가 아닌 경우)와 실제 쿼리 수행에 사용 중인 카프카 스트림즈 토폴로지 설명이 있다.

쿼리 종료하기

이 장 초반부에 실행 중인 스트림과 테이블을 버리는 방법을 배웠다. 그러나 새롭게 생성된 season_length_changes 스트림을 버리려고 하는 경우 다음과 같은 에러를 볼 것이다:

```
ksql> DROP STREAM season_length_changes ;

Cannot drop SEASON_LENGTH_CHANGES.
The following queries read from this source: [].
The following queries write into this source: [CSAS_SEASON_LENGTH_CHANGES_0].
You need to terminate them before dropping SEASON_LENGTH_CHANGES.
```

에러 문장은 매우 자명하다: 쿼리가 현재 액세스하고 있는 경우 모음을 버릴 수 없다 (예, 이에 쓰거나 이로부터 읽는 중). 따라서 우선 쿼리를 종료해야 한다.

쿼리를 종료하기 위한 구문은 다음과 같다:

```
TERMINATE { query_id | ALL }
```

다중 쿼리가 존재한다면 TERMINATE ALL을 실행시켜 한 번에 모두를 중지시킬 수 있다. 또한 초반에 생성했던 (ID가 CSAS_SEASON_LENGTH_CHANGES_0) 쿼리만을 중지하고자 한다면 다음 구문을 실행시킬 수 있다:

```
TERMINATE CSAS_SEASON_LENGTH_CHANGES_0 ;
```

쿼리가 종료되었고 어떤 쿼리도 스트림 또는 테이블에 액세스하고 있지 않기 때문에 DROP { STREAM | TABLE } 문을 사용하여 모음을 버릴 수 있다.

함께 모으기

이 장에서 많은 다양한 스트림 처리 유스케이스와 SQL 문을 탐구하였는데 스트림 처리 애플리케이션의 1 및 2 단계 구축에 필요한 실제 코드는 매우 간결하다. 예제 10-4는 튜토리얼의 이 부분을 완료하는데 필요한 전체 쿼리 셋을 보여준다.

예제 10-4. 넷플릭스에서 영감을 받은 변경 추적 애플리케이션의 1-2 단계 쿼리 셋

```
CREATE TYPE season_length AS STRUCT<season_id INT, episode_count INT> ;
CREATE TABLE titles (
 id INT PRIMARY KEY,
 title VARCHAR
) WITH (
 KAFKA_TOPIC='titles',
 VALUE_FORMAT='AVRO',
 PARTITIONS=4
);
CREATE STREAM production_changes (
 rowkey VARCHAR KEY,
```

```
  uuid INT,
  title_id INT,
  change_type VARCHAR,
  before season_length,
  after season_length,
  created_at VARCHAR
) WITH (
  KAFKA_TOPIC='production_changes',
PARTITIONS='4',
  VALUE_FORMAT='JSON',
  TIMESTAMP='created_at',
  TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
CREATE STREAM season_length_changes
WITH (
  KAFKA_TOPIC = 'season_length_changes',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 4,
  REPLICAS = 1
) AS SELECT
  ROWKEY,
  title_id,
  IFNULL(after->season_id, before->season_id) AS season_id,
  before->episode_count AS old_episode_count,
  after->episode_count AS new_episode_count,
  created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

다음 장에서는 ksqlDB를 사용하여 데이터를 보강, 집계 및 구체화하는 방법을 배우면서 튜토리얼을 계속 진행할 것이다 (그림 10-1의 3-5 단계)

요약

지금까지 데이터 필터링, 복합 구조 편평화, 조건식 사용 등을 포함하여 ksqlDB를 통해 많은 중요한 스트림 처리 태스크를 수행하는 방법을 배웠다. 또한 스트림, 테이블과 쿼리의 상태를 검사하는데 특히 도움이 되는 몇몇 SQL 문과 이들 엔티티 각각을 생성 및 버리기 위한 관련 문도 배웠다.

이 장에서는 데이터 전처리 및 변환에 중점을 두었지만 아직까지 ksqlDB에서 알아야 할 다른 것들도 존재한다. 구체적으로 데이터 보강과 집계에 중점을 둘 것이다. 다음 장에서는 이들 주체를 자세히 탐구할 것이며 도중에 ksqlDB에서 사용가능한 더욱 강력한 일련의 SQL 문과 구문에 대해 배울 것이다.