11 장 ksqlDB를 사용한 중간 및 고급 스트림 처리

이전 장에서는 ksqlDB를 사용하여 기본적인 데이터 전처리 및 변환을 수행하는 방법을 배웠다. 논의한 SQL 문은 스테이트리스로 이들을 통해 데이터 필터링, 복합 또는 중첩 구조 편평화, 데이터 재구성을 위한 투영 등을 할 수 있었다. 이 장에서는 데이터 보강 및 집계 유스케이스를 논의함으로써 ksqlDB에 대한 이해를 심화할 것이다. 논의할 대부분의 SQL 문은 스테이트풀 (예, 조인과 집계가 필요한 다중 레코드 포함) 및 시간 기반 (예, 윈도우 연산)으로 내부적으로 더욱 복잡하지만 더욱 강력하다.

이 장에서 다룰 주제는 다음을 포함할 것이다:

- 데이터 결합 및 보강을 위해 조인 사용
- 집계 수행
- CLI를 사용하여 구체화 뷰에 대해 풀 쿼리 실행 (예, 포인트 조회)
- 내장 ksqlDB 함수로 작업 (scalar, aggregate와 table 함수)
- Java를 사용하여 사용자 정의 함수 생성

이들 개념의 많은 부분을 소개하기 위해 이전 장의 넷플릭스 변경 추적 튜토리얼을 사용할 것이다. 그러나 ksqlDB 함수를 포함하여 일부 주제는 이 장 말미에서 독립적으로 논의될 것이다.

프로젝트 환경 설정 단계를 검토함으로써 시작해보자.

프로젝트 환경 설정

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-11
```

이전 장에서 논의했듯이 이 튜토리얼에 필요한 각 컴포넌트를 구동시키기 위해 다음을 포함한 도커 컴포즈를 사용할 것이다 (예, 카프카, ksqlDB, CLI 등). 각 컴포넌트를 구동시키기 위해 저장소를 복제한 후 다음 명령을 실행한다:

```
docker-compose up
```

달리 명시하지 않는 한 이 장에서 논의하는 SQL 문은 ksqlDB에서 실행될 것이다. 다음 명령을 사용하여 ksqlDB CLI에서 로그인할 수 있다:

```
docker-compose exec ksqldb-cli ₩
    ksql http://ksqldb-server:8088 --config-file /etc/ksqldb-cli/cli.properties
```

지금까지 환경 설정은 10 장과 거의 비슷하다. 그러나 이 장에서는 이전 장의 튜토리얼에 기반하여 구축하며 따라서 지난 장이 완료된 이후 ksqlDB 환경을 설정해야 한다. 이는 다음 절에서 논의할 매우 중

요한 ksqlDB 문으로 이끈다.

SQL 파일로부터 환경 부트스트래핑

넷플릭스 변경 추적 애플리케이션을 구축하기 위해 튜토리얼의 일부를 통해 작업을 진행했으며 따라서 완료한 부분부터 다시 시작하기 위해 실행해야 할 일련의 쿼리를 갖고 있다. 이는 이 튜토리얼에 특정적일 수 있지만 환경 설정을 위해 일련의 쿼리를 실행하는 것은 일반적인 개발 워크플로우로 ksqlDB는 이를 쉽게 하는 특수 문을 갖고 있다:

이 문의 구문은 다음과 같다:

```
RUN SCRIPT <sql_file> ①
```

① SQL 파일은 실행할 여러 쿼리를 포함할 수 있다.

예를 들어 이전 장의 모든 쿼리를 /etc/sql/init.sql 파일에 놓고 이전에 작업했던 모음과 쿼리를 재생성하기 위해 다음 명령을 실행할 수 있다:

```
ksql> RUN SCRIPT '/etc/sql/init.sql' ;
```

RUN SCRIPT 가 실행될 때 SQL 파일에 포함된 모든 문의 출력이 클라이언트에 반환될 것이다. CLI의 경우 화면에 출력된다. 10 장 쿼리에 대한 생략된 출력 예는 다음과 같다.

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

----------------------------
 Registered custom type ...
----------------------------

CREATE TABLE titles ...

---------------
 Table created
---------------

CREATE STREAM production_changes ...

----------------
 Stream created
----------------

CREATE STREAM season_length_changes ...
-------------------------------------------------
 Created query with ID CSAS_SEASON_LENGTH_CHANGES_0
-------------------------------------------------
```

위에서 볼 수 있듯이 RUN SCRIPT는 ksqlDB 애플리케이션에 대해 반복하기 위해 유용한 시간 절약 메커니즘이다. CLI에서 문을 실험하고 진행 상황을 추후 실행될 수 있는 파일에 체크포인트할 수 있다. 운영 단계로 쿼리를 배치할 준비가 될 때 헤드리스 모드 (그림 8-9 참조)에서 ksqlDB를 동작시키기 위

해 동일한 SQL 파일을 사용할 수 있다.

RUN SCRIPT가 어떤 단계로 실행되는지를 시각화하고 이 장에서 해결할 추가 단계에 대한 의미를 얻기 위해 그림 11-1을 참조하기 바란다.

해결해야 할 다음 단계는 3 단계로 season_length_changes 스트림의 필터링 및 변환된 데이터를 보강해야 한다. 다음 절에서 이를 살펴볼 것이다.

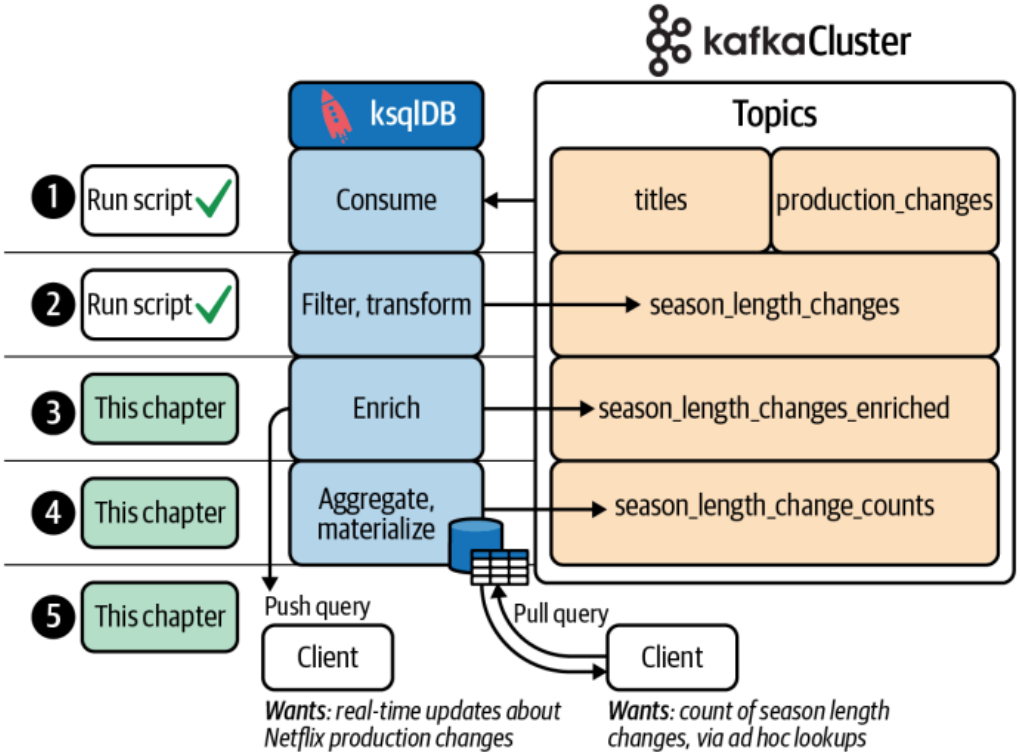

그림 11-1. 넷플릭스 변경 추적 애플리케이션 아키텍처 개요; 이전 장에서 1 및 2 단계는 완료되었고 RUN SCRIPT를 사용하여 재생성된다.

데이터 보강

데이터 보강은 원시 데이터를 개선 또는 보강하는 프로세스와 관련이 있다. 이는 보통 데이터 포맷 또는 구조 변경에 중점을 두는 간단한 데이터 변환을 넘어선다. 대신 보강은 데이터에 정보를 추가하는 것을 포함하며 데이터베이스에서 가장 널리 사용되는 보강 기법 중 하나는 조인이다.

애플리케이션 3 단계에서 season_length_changes 스트림을 titles 테이블 정보로 보강할 것이다. 다음 절에서 이를 수행하는 방법을 탐구할 것이다.

조인

데이터 조인은 join 서술부 (관련 레코드가 발견될 때 true가 아닌 경우 false가 되는 부울식)를 사용하여 복수의 데이터 소스로부터 관련 레코드를 결합시키는 것을 포함한다. 데이터가 종종 많은 소스에 걸쳐 흩어져 있고 처리 및 분석을 위해 함께 가져와야 하기 때문에 조인은 관계형 및 스트림 세계 모

두에서 일반적이다.

ksqlDB에는 많은 유형의 조인이 있으며 변형은 2 차원을 사용하여 표현될 수 있다.

- 사용되는 조인식 (INNER JOIN, LEFT JOIN과 FULL JOIN)
- 조인되는 모음 타입 (STREAM, TABLE)

첫번째 조인식으로 시작해보자. 표 11-1은 ksqlDB에서 사용가능한 각각의 조인 타입을 설명한다.

표 11-1. ksqlDB에서 사용가능한 조인식

SQL 식	설명
INNER JOIN	조인 양쪽의 입력 레코드가 동일 키를 공유할 때 트리거
LEFT JOIN	조인 왼쪽의 레코드가 수신될 때 트리거. 조인 오른쪽에 동일 키를 갖는 일치되는 레코드가 없는 경우 오른쪽 값은 null이 된다.
FULL JOIN	조인 양쪽 중 한쪽의 레코드가 수신될 때 트리거. 조인 반대쪽에 동일 키를 갖는 일치되는 레코드가 없는 경우 해당 값은 null이 된다.

과거 전통적인 데이터베이스로 작업을 했다면 이전 조인식은 익숙해야 한다. 그러나 전통적인 데이터베이스가 테이블 조인을 할 수 있는 반면 ksqlDB는 테이블과 스트림이라는 2 가지 유형의 모음에 대한 조인을 지원한다. 또한 사용할 수 있는 조인식 유형 (INNER JOIN, LEFT JOIN 또는 FULL JOIN)은 어떤 모음이 조인 대상인지에 따른다. 표 11-2는 사용가능한 결합에 대한 요약으로 윈도우 칼럼은 조인이 시간 제한적이어야 하는지 여부를 나타낸다.

표 11-2. ksqlDB의 조인 유형

조인 타입	지원 표현식	윈도우
스트림-스트림	INNER JOIN LEFT JOIN FULL JOIN	예
스트림-테이블	INNER JOIN LEFT JOIN	아니오
테이블-테이블	INNER JOIN LEFT JOIN FULL JOIN	아니오

위에서 볼 수 있듯이 스트림-스트림 조인은 윈도우가 있다. 스트림이 무제한이기 때문에 관련 레코드의 검색을 사용자 정의 시간 범위로 제한해야 한다. 그렇지 않은 경우 관련 레코드에 대해 2 개 이상의 연속적인 스트림을 검색하는 실행불가능한 태스크가 남겨질 것이며, 이것이 이 조인 타입에 대해 ksqlDB가 윈도우 요건을 부과한 이유이다.

모음에 대한 데이터 조인 선결조건

조인 쿼리를 작성하기 전에 모음에 대해 데이터 조인을 하기 위한 일부 선결조건이 있음을 주목하는 것은 중요하다. 이는 다음을 포함한다:

- 조인식에 참조된 모든 칼럼은 동일 데이터 타입이어야 한다 (STRING, INT, LONG 등)
- 조인 양쪽의 파티션 수가 동일해야 한다[1].
- 토픽 내 데이터가 동일한 파티션 전략을 사용해 써져야 한다 (보통 이는 프로듀서가 입력 레코드 키에 기반한 해시를 생성하는 기본 파티셔너를 사용할 것임을 의미한다).

이 요건을 염두에 두고 조인 쿼리를 작성해보자. 튜토리얼에서 전처리된 데이터 스트림 season_length_changes는 title_id 칼럼을 갖고 있다. 타이틀에 대해 더 많은 정보를 조회하기 위해 이 값을 사용할 것이다 (titles 테이블에 저장된 타이틀 이름, 예 Stranger Things 또는 Black Mirror 등). 이를 이너 조인식으로 표현하려면 예제 11-1의 SQL 문을 실행할 수 있다.

예제 11-1. 두 모음을 조인하는 SQL 문

```
SELECT
 s.title_id,
 t.title,
 s.season_id,
 s.old_episode_count,
 s.new_episode_count,
 s.created_at
FROM season_length_changes s
INNER JOIN titles t ①
ON s.title_id = t.id ②
EMIT CHANGES ;
```

① titles 테이블의 해당 레코드가 발견될 수 있을 때 조인이 트리거되길 원하기 때문에 INNER JOIN을 사용한다.

② 스트림은 조인에 사용할 어떤 칼럼도 지정할 수 있다. 테이블은 PRIMARY KEY로 지정된 칼럼에 대해서만 조인되어야 한다. 후자 요건은 테이블이 키-값 저장소임을 고려할 때 이해될 수 있다. 스트림으로부터 새로운 레코드가 들어옴에 따라 관련 레코드가 있는지 확인하기 위해 테이블의 키-값 저장소에 포인트 조회를 실행해야 한다. 레코드가 이미 PRIMARY KEY 값으로 상태 저장소에 저장되어 있기 때문에 이를 사용하는 것이 가장 효율적인 방법이다. 이는 키가 테이블에 대한 고유한 제약이기 때문에 또한 단지 테이블 내 한 레코드가 스트림 레코드에 일치될 것임을 보장한다.

이전 문의 출력은 타이틀 이름을 포함하는 새로운 보강된 레코드를 보여준다 (TITLE 칼럼 참조).

---

[1] 이는 조인을 포함한 SQL 문을 실행할 때 검사된다. 파티션 수가 일치하지 않는 경우 Can't join S with T since the number of partitions don't match와 유사한 에러를 볼 것이다.

```
+---------+----------------+-----------+------------------+------------------+-------------+
|TITLE_ID |TITLE           |SEASON_ID  |OLD_EPISODE_COUNT |NEW_EPISODE_COUNT |CREATED_AT   |
+---------+----------------+-----------+------------------+------------------+-------------+
|1        |Stranger Things |1          |12                |8                 |2021-02-08...|
```

이전 조인은 부분적으로 모든 조인 선결 조건이 튜토리얼에서 이미 충족되었기 때문에 매우 단순했다. 그러나 실제 애플리케이션의 조인 선결 조건은 그렇게 편리하게 만족될 수 없다. 조인 수행을 위해 추가적인 단계가 필요한 약간 보다 더 복잡한 상황을 탐구해보자.

새로운 타입으로 칼럼 캐스팅

어떤 경우 조인하는 데이터 소스가 조인 속성에 다른 데이터 타입을 지정할 수도 있다. 예를 들어 s.title_id가 실제 VARCHAR로 인코딩되어 있지만, INT로 인코딩되어 있는 ROWKEY에 대해 조인해야 한다면? 이 경우 내장 CAST 표현식을 사용하여 s.title_id를 INT 타입으로 변환하면 데이터 조인에 대한 첫번째 선결조건을 충족시킬 수 있다.

```
SELECT ... ①
FROM season_length_changes s
INNER JOIN titles t
ON CAST(s.title_id AS INT) = t.id ②
EMIT CHANGES ;
```

① 간결함을 위해 칼럼명을 생략했다.

② 이 가상 예에서 s.title_id가 VARCHAR로 인코딩되어 있다면 첫번째 조인 선결조건을 충족시키기 위해 s.title_id를 t.id (INT)와 동일 타입으로 캐스팅해야 한다.

데이터 리파티셔닝

조인 실행 전에 데이터를 리파티션해야 하는 상황을 고려해보자. 이는 "모음에 대한 데이터 조인 선결조건"에 기술한 마지막 두 조인 선결조건 중 하나가 충족되지 않는 경우 발생할 수 있다. 예를 들어 titles 테이블이 8 개의 파티션을 갖고 있고 season_length_changes 테이블이 4 개의 파티션을 갖는다면?

이 경우 조인을 수행하기 위해 모음 중 하나를 리파티셔닝해야 한다. 이는 예제 11-2의 SQL 문과 같이 PARTITIONS 특성으로 새로운 모음을 생성함으로써 달성될 수 있다.

예제 11-2. PARTITIONS 특성을 사용하여 데이터를 리파티셔닝하는 방법 예

```
CREATE TABLE titles_repartition
WITH (PARTITIONS=4) AS ①
SELECT * FROM titles
EMIT CHANGES;
```

① 기본적으로 모음과 동일한 이름을 갖는 새로운 토픽을 생성할 것이다 (TITLES_REPARTITION). 리파티셔닝된 토픽에 맞춤형 이름을 지정하고 싶다면 KAFKA_TOPIC 특성을 제공할 수 있다.

조인 양쪽에 대해 파티션 수가 일치됨을 보장하기 위해 데이터를 리파티셔닝했다면 리파티셔닝된 모음에 대해 조인을 할 수 있다.

영구 조인

이제 데이터 조인을 위한 다양한 선결조건을 충족시키는 방법을 배웠기 때문에 튜토리얼의 3 단계 구현을 계속 진행하자. 넷플릭스 변경 추적 애플리케이션에 대해 조인 선결조건은 이미 충족되었기 때문에 조인 쿼리의 시작점은 예제 11-1이다.

이전 장에서 논의했듯이 SELECT로 시작하는 쿼리는 영구적이지 않으며 이는 서버 재시작 시 살아있지 않고 결과가 카프카에 써지지 않음을 의미한다. 따라서 출력을 단순히 CLI 컨솔에 출력하는 대신 조인 쿼리 결과를 실제 영속화하려는 경우 CREATE STREAM AS SELECT (CSAS) 문을 사용해야 한다.

다음 SQL 문은 보강된 레코드를 season_length_changes_enriched라는 새로운 토픽에 쓸 조인 쿼리의 영구 버전을 보여준다 (그림 11-1의 3 단계):

```
CREATE STREAM season_length_changes_enriched ①
WITH (
 KAFKA_TOPIC = 'season_length_changes_enriched',
 VALUE_FORMAT = 'AVRO',
 PARTITIONS = 4,
 TIMESTAMP='created_at', ②
 TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
) AS
SELECT ③
 s.title_id,
 t.title,
 s.season_id,
 s.old_episode_count,
 s.new_episode_count,
 s.created_at
FROM season_length_changes s
INNER JOIN titles t
ON s.title_id = t.id
EMIT CHANGES ;
```

① 원래 쿼리를 영구 쿼리로 만들기 위해 CREATE STREAM을 사용한다.

② 이는 created_at 칼럼이 윈도우 집계 및 조인을 포함하여 ksqlDB가 시간 기반 연산에 사용하는 타임스탬프를 포함함을 알려준다.

③ 기존 SELECT 문이 여기서 시작한다.

이제 ksqlDB를 사용하여 데이터를 조인하는 방법을 배웠고 튜토리얼의 3 단계를 완료했다. 조인 논의

를 완료하기 전에 스트림 처리 애플리케이션에서 일반적으로 사용되는 다른 유형의 조인에 대해 논의해보자.

윈도우 조인

이 튜토리얼에서는 윈도우 조인을 수행할 필요는 없다. 그러나 스트림-스트림 조인에 필요하기 때문에 이 책에서 최소한 이를 언급하지 않는다면 후회할 것이다. 윈도우 조인은 추가적인 조인 속성인 시간을 포함하기 때문에 비윈도우 조인 (예, 이전 절에 생성했던 조인)과는 다르다.

윈도우와 시간에 한 장을 할애했는데 따라서 이 주제를 자세히 논의하기 위해서는 5 장을 다시 보기 바란다. 5 장에 언급되어 있듯이 윈도우 조인은 그림 11-2와 같이 내부적으로 설정된 시간 경계 내에 속하는 레코드를 그룹화하는 슬라이딩 윈도우를 사용한다.

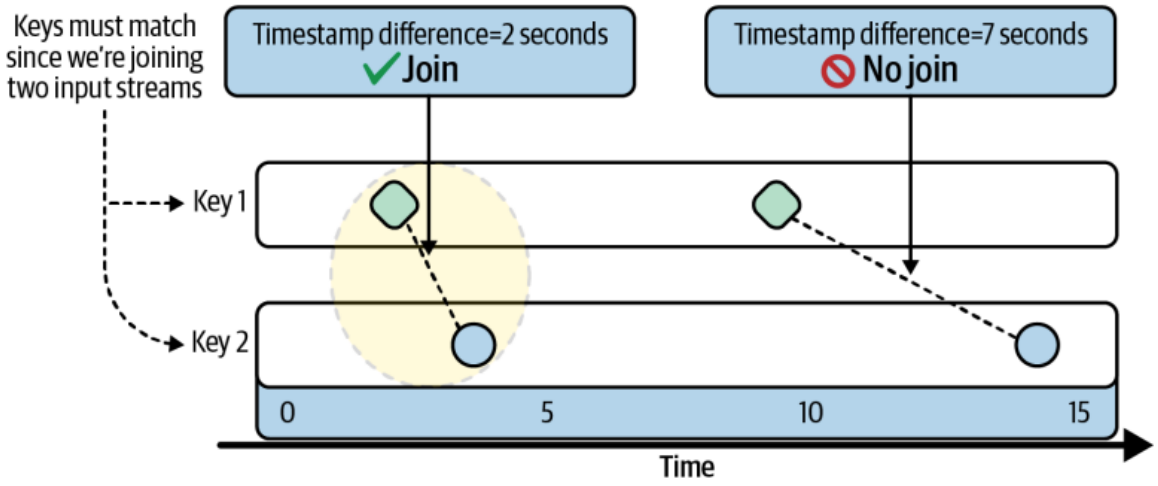

그림 11-2. 슬라이딩 윈도우

윈도우 조인을 생성하기 위해서는 조인 절에 WITHIN 표현식을 포함해야 한다. 구문은 다음과 같다:

```
WITHIN <number> <time_unit>
```

단수 및 복수 형태 모두를 포함하여 지원되는 시간 단위는 다음과 같다:

- DAY, DAYS
- HOUR, HOURS
- MINUTE, MINUTES
- SECOND, SECONDS
- MILLISECOND, MILLISECONDS

예를 들어 잠시 (윈도우 조인이 필요없는) 튜토리얼에서 벗어나 다른 유스케이스를 고려해보자. 넷플

릭스에서 일하고 있고 사용자가 시청 종료 전에 2 분 미만으로 시청한 모든 영화 또는 TV 프로그램을 포착하기로 결정했다. 가상 예에서 시청 시작 및 시청 종료 이벤트는 별도의 카프카 토픽에 써진다. 이는 session_id (시청 세션을 식별)와 이벤트 타임을 사용하여 각 레코드를 조인해야 하기 때문에 윈도우 조인에 대해 훌륭한 유스케이스이다. 이 예를 SQL로 옮기기 위해 예제 11-3과 같이 DDL을 사용하여 우선 2 개의 소스 스트림을 생성해보자.

예제 11-3. 시청 시작 및 종료 이벤트를 위한 2 개의 별도 스트림 생성

```
CREATE STREAM start_watching_events ( ①
 session_id STRING, ②
 title_id INT,
 created_at STRING
)
WITH (
 KAFKA_TOPIC='start_watching_events',
 VALUE_FORMAT='JSON',
 PARTITIONS=4,
 TIMESTAMP='created_at',
 TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss' ③
);
CREATE STREAM stop_watching_events ( ④
 session_id STRING,
 title_id INT,
 created_at STRING
)
WITH (
 KAFKA_TOPIC='stop_watching_events',
 VALUE_FORMAT='JSON',
 PARTITIONS=4,
 TIMESTAMP='created_at',
 TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
);
```

① 시청 시작 이벤트에 대한 소스 스트림을 생성한다. 이 스트림은 사용자가 새로운 프로그램 또는 영화 시청을 시작할 때마다 새로운 레코드를 볼 것이다.

② session_id는 조인 속성 중 하나로 하나의 시청 세션을 나타내면 두 소스 스트림 모두에서 참조된다.

③ 선택사항인 TIMESTAMP_FORMAT를 통해 카프카 토픽의 타임스탬프 포맷을 나타내는 포맷 문자열을 지정할 수 있다. Java의 DateTimeFormatter 클래스가 지원하는 값이다.

④ 시청 종료 이벤트에 대한 소스 스트림을 생성한다. 이 스트림은 사용자가 새로운 프로그램 또는

영화 시청을 종료할 때마다 새로운 레코드를 볼 것이다.

이제 2 개의 다른 시청 세션에 대해 각각 2 개의 시청 시작 및 중지 이벤트를 삽입해보자. (2명의 시청자가 2 개의 다른 프로그램을 보는 것으로 생각할 수 있다). 첫번째 세션 session_123은 총 90 초의 시청 시간을 가질 것이고 두번째 세션 session_456은 총 25분의 시청 시간을 가질 것이다.

```
INSERT INTO start_watching_events
VALUES ('session_123', 1, '2021-02-08 02:00:00');
INSERT INTO stop_watching_events
VALUES ('session_123', 1, '2021-02-08 02:01:30');
INSERT INTO start_watching_events
VALUES ('session_456', 1, '2021-02-08 02:00:00');
INSERT INTO stop_watching_events
VALUES ('session_456', 1, '2021-02-08 02:25:00');
```

마지막으로 윈도우 조인을 사용하여 2 분 미만의 시청 세션을 포착해보자. 고수준에서는 쿼리를 통해 다음 질문을 던지는 것이다: (시청 시작 타임스탬프에 의해 나타남) 시작 시간에 대해 2 분 내에 어떤 시청 세션이 종료되었는가 (시청 종료 타임스탬프에 의해 나타남)? 다음 SQL 문은 ksqlDB에서 이를 질문하는 방법을 보여준다.

```
SELECT
 A.title_id as title_id,
 A.session_id as session_id
FROM start_watching_events A
INNER JOIN stop_watching_events B
WITHIN 2 MINUTES ①
ON A.session_id = B.session_id
EMIT CHANGES ;
```

① 2 분 간격 미만의 타임스탬프를 갖고 session_id가 두 스트림에 대해 동일한 start_watch_events 및 stop_watching_events 스트림으로부터 레코드를 조인한다 (후자의 조건은 다음 라인에 표현되어 있다). 이는 2 분 미만인 모든 시청 세션을 포착할 것이다.

Session_123의 시청 시간이 단지 90 초이기 때문에 다음 출력을 봐야 한다.

```
+------------+-------------+
|TITLE_ID    |SESSION_ID   |
+------------+-------------+
|1           |session_123  |
```

이제 윈도우 및 비윈도우 조인에 대해 잘 파악했기 때문에 ksqlDB에서 데이터 집계 방법을 배움으로써 튜토리얼을 계속 진행하자.

집계

ksqlDB에서 한 번에 하나의 레코드로 작업하는 것은 데이터 필터링, 데이터 구조 변환 또는 한 번에 한 이벤트 보강 수행 등 특정 태스크에 유용하다. 그러나 데이터로부터 얻을 수 있는 가장 유용한 통찰 중 일부는 관련 레코드를 그룹화하고 집계하는 것이다. 예를 들어 기간별로 넷플릭스에서 시즌 길이 변경 수를 세는 것은 새로운 TV 프로그램 또는 영화에 대한 계획 프로세스 개선과 운영상 문제를 겪고 있는 프로젝트에 대한 지표 제공에 도움을 줄 수 있다.

집계는 스트림과 테이블 모두에 대해 계산될 수 있지만 항상 테이블을 반환한다. 이는 집계 함수가 관련 레코드 그룹에 적용되고 집계 함수 (예, COUNT)의 결과가 새로운 레코드가 들어올 때마다 쉽게 검색 및 업데이트될 수 있는 가변 구조에 저장되어야 하기 때문이다[2].

2 가지 넓은 범위의 집계가 있다: 윈도 및 비윈도우. 이 튜토리얼에서는 (그림11-1의 4 단계) season_length_change_counts 테이블을 계산하기 위해 윈도우 집계를 사용할 것이다. 우선 다음 절에서 데이터 집계의 기본을 살펴보자.

집계 기본

ksqlDB에서 데이터 집계를 위한 2 가지 기본 단계가 있다[3]:

- 집계 함수를 활용하는 SELECT 표현식을 생성한다.
- GROUP BY 절을 사용하여 관련 레코드를 그룹화한다. 집계 함수는 각 그룹에 적용될 것이다.

예제 11-4는 집계 쿼리 예를 보여준다.

표 11-4. 집계 함수 (COUNT)와 GROUP BY 절 사용 예

```
SELECT
 title_id,
 COUNT(*) AS change_count, ①
 LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count ②
FROM season_length_changes_enriched
GROUP BY title_id ③
EMIT CHANGES ;
```

① 내장 COUNT 집계 함수를 사용하여 각 title_id에 대한 레코드 수를 센다.

② 내장 LATEST_BY_OFFSET 집계 함수를 사용하여 각 title_id에 대해 최신 에피소드 수를 얻는다.

---

[2] COUNT 연산의 경우 주어진 키의 값은 0으로 초기화되고 동일 키를 갖는 새로운 레코드가 들어올 때마다 1씩 증가할 것이다. 이전에 언급했듯이 이런 유형의 가변 시맨틱은 스트림 대신 내부적으로 상태 기반 테이블 구조를 필요로 한다.

[3] 다음 절에서 볼 것인데 집계가 계산되는 타임 윈도우 지정을 포함하는 선택적인 3 단계도 있다.

③ title_id 칼럼별로 이 스트림의 레코드를 그룹화한다.

예제 11-4의 집계는 내장 COUNT 함수를 사용하지만 ksqlDB는 AVG, COUNT_DISTINCT, MAX, MIN, SUM 등 많은 집계 함수를 포함하고 있다. 집계는 항상 수학적인 것은 아니다. 2 가지 두드러진 예로는 EARLIEST_BY_OFFSET과 LATEST_BY_OFFSET 집계 함수로 이들은 각각 칼럼에 대해 (offset에 의해 계산된) 가장 오래된 값과 최신 값을 반환한다. 이 장 말미에 ksqlDB의 모든 사용가능한 집계 함수를 열거하는 방법을 나타내고 필요한 경우 스스로 함수를 작성하는 방법을 논의할 것이다. 그러나 지금은 ksqlDB에서 선택할 많은 다양한 내장 함수가 존재한다고 말하는 것으로 충분하다.

두번째 요건은 GROUP BY 절을 사용하여 레코드 그룹화를 포함한다. 이는 레코드를 이 절에 지정한 칼럼에 기반하여 다른 버킷에 놓는 것이다. 예를 들어 예제 11-4의 쿼리는 동일 title_id를 갖는 모든 항목을 동일 그룹으로 그룹화한다.

마지막으로 SELECT 표현식은 추가적인 칼럼을 포함할 수 있다. 예를 들어 쿼리에 title_id와 season_id를 추가해보자:

```
SELECT
 title_id,
 season_id, ①
 COUNT(*) AS change_count,
 LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
GROUP BY title_id, season_id ②
EMIT CHANGES ;
```

① 쿼리에 비집계 칼럼 (season_id)를 추가한다.

② GROUP BY 절에 season_id를 추가한다.

집계 수행 시 SELECT 표현식에 비집계 칼럼을 추가하는 경우 GROUP BY 절에도 포함되어야 한다. 예를 들어 다음 쿼리를 실행한다고 해보자:

```
SELECT
 title_id,
 season_id, ①
 COUNT(*) AS change_count,
 LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
GROUP BY title_id ②
EMIT CHANGES ;
```

① SELECT 표현식에 season_id를 포함했다

② 그러나 GROUP BY 절에는 이를 포함하지 않았다.

다음과 같은 에러 메시지를 볼 것이다:

```
Non-aggregate SELECT expression(s) not part of GROUP BY: SEASON_ID
Either add the column to the GROUP BY or remove it from the SELECT
```

에러 메시지에도 불구하고 이를 해결하기 위한 세번째 옵션이 존재한다: GROUP BY 절에 포함되지 않은 칼럼에 집계 함수 적용.

> GROUP BY 절에 포함한 각 필드는 테이블에서 키의 일부가 된다. 예를 들어 GROUP BY title_id는 각각 고유한 title_id를 갖는 테이블을 생성할 것이다. GROUP BY title_id, season_id와 같이 복수 칼럼으로 그룹화한다면 테이블은 각 칼럼 값이 |+| (예, 1|+|2)로 분리되는 복합 키를 가질 것이다.
>
> 이는 복합 키를 갖는 테이블에 쿼리하는 경우 쿼리에 분리자를 포함해야 하기 때문에 구체화 테이블에 대해 풀 쿼리를 실행할 때 기억하는 것이 중요하다 (예, SELECT * FROM T WHERE COL='1|+|2'). ksqlDB 향후 버전에서는 복합 키에 대한 쿼리 방법을 개선할 것이며 여기서 기술한 동작은 최소한 0.14.0 버전에 존재한다.

이전 집계 각각은 비윈도우 집계의 예이다. 이들은 GROUP BY 절 내 필드로 버킷될 뿐 별도의 시간 기반 윈도우 조건을 사용하지 않기 때문에 비윈도우로 고려된다. 그러나 ksqlDB는 다음 절에서 탐구할 윈도우 집계도 지원한다.

윈도우 집계

종종 특정 기간에 대해 집계를 계산하길 원할 수도 있다. 예를 들어 24 시간 간격으로 시즌 길이 변경이 얼마나 많이 발생했는지 알고 싶어할 수 있다. 이는 집계에 다른 차원을 추가시킨다: 시간. 운좋게도 ksqlDB는 이 유스케이스를 위해 설계된 윈도우 집계를 지원한다.

5 장에서 카프카 스트림즈의 3 가지 다른 유형의 윈도우를 논의하였다. 동일 윈도우 유형이 ksqlDB에서도 사용가능하며 표 11-3은 해당 ksqlDB 표현식을 보여준다.

표 11-3. ksqlDB 윈도우 타입

윈도우 타입	예
텀블링 윈도우	WINDOW TUMBLING (SIZE 30 SECONDS)
호핑 윈도우	WINDOW HOPPING (SIZE 30 SECONDS, ADVANCE BY 10 SECONDS)
세션 윈도우	WINDOW SESSION (60 SECONDS)

쿼리에 윈도우 표현식을 통합하기 위해서 다음과 같이 GROUP BY 절 전에 윈도우 표현식을 포함하면 된다.

```
SELECT
 title_id,
 season_id,
 COUNT(*) AS change_count,
 LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count
FROM season_length_changes_enriched
WINDOW TUMBLING (SIZE 1 HOUR) ①
GROUP BY title_id, season_id ②
EMIT CHANGES ;
WINDOW <window_type> ( ①
 <window_properties>, ②
 GRACE PERIOD <number> <time_unit> ③
)
```

① 레코드를 한 시간 간격 버킷으로 그룹화한다.

② title_id와 session_id로 레코드를 그룹화한다.

이전 쿼리는 다음 결과를 산출할 것이다:

```
+-----------+----------+-------------+---------------------+
|TITLE_ID   |SEASON_ID |CHANGE_COUNT |LATEST_EPISODE_COUNT |
+-----------+----------+-------------+---------------------+
|1          |1         |1            |8
```

물론 ksqlDB 윈도우 표현식을 사용하여 윈도우 집계를 하는 것이 매우 쉽지만 시간 차원 추가로 인해 다음과 관련하여 추가적인 고려를 해야 한다:

- 데이터가 언제 다운스트림 프로세서로 방출되는가?
- 지연된/순서가 뒤바뀐 데이터를 어떻게 처리할 수 있는가?
- 각 윈도우를 얼마나 오래 유지해야 하는가?

다음 절에서 이들 각각을 다룰 것이다.

지연 데이터

ksqlDB 또는 카프카 스트림즈에서 데이터의 지연 도착에 대해 논의할 때 wall clock time에 따라 지연되는 이벤트에 대해서는 전적으로 논의하지 않았다. 예를 들어 이벤트가 오전 10시 2분에 발생하였고 10시 15분까지 스트림 처리 애플리케이션에 의해 처리되지 않았다면 전통적인 의미에서 이를 지연되었다고 고려할 수 있다.

그러나 5 장에서 배웠듯이 카프카 스트림즈 (그리고 연장선상에서 ksqlDB)은 스트림 타임이라는 내부 시각을 유지하고 있으며 이는 소비된 레코드 타임스탬프로부터 추출된 항상 증가하는 값이다. 타임스탬프는 레코드 메타 데이터 또는 예제 11-3에서 보듯이 TIMESTAMP와 TIMESTAMP_FORMAT 특성을

사용하여 레코드 내 내장 값으로부터 추출될 수 있다.

레코드가 타임스탬프 순서를 벗어나 소비되면 현재 스트림 타임보다 이전 타임스탬프를 갖고 도착한 레코드는 지연되었다고 고려된다. 그리고 윈도우 집계 시 이들이 윈도우에 허용되는지 여부 (집계 시 계산될 수 있는지) 또는 특정 시간 경과 후 (이 시점에서 늦었다고 고려되어 윈도우에 허용되지 않음) 단순히 무시되는지를 제어할 수 있기 때문에 지연된 레코드는 특히 관심대상이다. 허용된 지연을 유예 기간으로 부르는데 ksqlDB에서 유예 기간을 정의하는 구문은 다음과 같다:

```
WINDOW <window_type> (①
 <window_properties>, ②
 GRACE PERIOD <number> <time_unit> ③
)
```

① 윈도우 타입은 HOPPING, TUMBLING 또는 SESSION일 수 있다.

② 각각의 윈도우 타입 생성을 위해 필요한 윈도우 특성에 대해서는 표 11-3을 참조하기 바란다.

③ 유예 기간을 정의한다. 슬라이딩 윈도우에 대해 논의했던 동일 시간 단위가 유예 기간에 대해서도 지원된다. "윈도우 조인"을 참조하기 바란다.

예를 들어 최대 10분까지 지연을 허용하고 이 임계치 이후 도착하는 어떤 레코드도 무시하는 윈도우 집계를 적용하려면 다음 쿼리를 사용할 수 있다:

```
SELECT
 title_id,
 season_id,
 COUNT(*) AS change_count,
 LATEST_BY_OFFSET(new_episode_count) AS episode_count
FROM season_length_changes_enriched
WINDOW TUMBLING (SIZE 1 HOUR, GRACE PERIOD 10 MINUTES) ①
GROUP BY title_id, season_id
EMIT CHANGES ;
```

① 1 시간 간격의 버킷으로 레코드를 그룹화하고 최대 10분까지 지연을 허용한다.

10 MINUTES로 정의된 이전 유예 기간을 통해 스트림 타임에 미치는 영향과 레코드가 오픈 윈도우에 허용될 것인지를 명시적으로 확인하기 위해 (이전 쿼리가 실행 중인 동안) 다른 탭에서 일부 레코드를 생산할 수 있다. 예제 11-5는 이들 레코드를 생산하기 위해 실행할 문장을 보여준다.

예제 11-5. 다른 CLI 세션에서 스트림 타임에 미치는 영향을 관찰하기 위해 다양한 타임스탬프를 갖는 레코드를 생상한다.

```
INSERT INTO production_changes VALUES (
 '1', 1, 1, 'season_length',
 STRUCT(season_id := 1, episode_count := 12),
```

```
 STRUCT(season_id := 1, episode_count := 8),
 '2021-02-24 10:00:00' ①
);
INSERT INTO production_changes VALUES (
 '1', 1, 1, 'season_length',
 STRUCT(season_id := 1, episode_count := 8),
 STRUCT(season_id := 1, episode_count := 10),
 '2021-02-24 11:00:00' ②
);
INSERT INTO production_changes VALUES (
 '1', 1, 1, 'season_length',
 STRUCT(season_id := 1, episode_count := 10),
 STRUCT(season_id := 1, episode_count := 8),
 '2021-02-24 10:59:00' ③
);
INSERT INTO production_changes VALUES (
 '1', 1, 1, 'season_length',
 STRUCT(season_id := 1, episode_count := 8),
 STRUCT(season_id := 1, episode_count := 12),
 '2021-02-24 11:10:00' ④
);
INSERT INTO production_changes VALUES (
 '1', 1, 1, 'season_length',
 STRUCT(season_id := 1, episode_count := 12),
 STRUCT(season_id := 1, episode_count := 8),
 '2021-02-24 10:49:00' ⑤
```

① 스트림 타임이 오전 10시로 설정되어 있고 이 레코드는 (오전 10시 – 오전 11시) 윈도우에 추가될 것이다.

② 스트림 타임이 오전 11시로 설정되어 있고 이 레코드는 (오전 11시 – 오후 12시) 윈도우에 추가될 것이다.

③ 이 레코드의 타임스탬프가 현재 스트림 타임보다 이전이기 때문에 스트림 타임은 변경되지 않는다. 레코드는 유예 기간이 아직 active이기 때문에 (오전 10시 – 오전 11시) 윈도우에 추가될 것이다.

④ 스트림 타임이 오전 11시로 설정되어 있고 이 레코드는 (오전 11시 – 오후 12시) 윈도우에 추가될 것이다.

⑤ 이 레코드는 11 분 늦었기 때문에 윈도우에 추가되지 않는다. 다른 말로 stream_time (오전 11시) – current_time (오전 10시 49분) = 11 분으로 유예 기간 10분 보다 크다.

늦게 도착하여 (예를 들어 예제 11-5의 마지막 레코드) 레코드를 건너뛸 때 다음과 같이 ksqlDB 서버 로그에 포함되는 일부 도움이 되는 정보를 봐야 한다:

```
WARN Skipping record for expired window.
key=[Struct{KSQL_COL_0=1|+|1}]
topic=[...]
partition=[3]
offset=[5]
timestamp=[1614164340000] ❶
window=[1614160800000,1614164400000) ❷
expiration=[1614164400000] ❸
streamTime=[1614165000000] ❹
```

① 이 레코드와 연관된 현재 타임스탬프는 2021-02-24 10:49:00이다. (이 부분 책이 틀림)

② 이 레코드가 들어가는 그룹의 윈도우는 2021-02-24 10:00:00 – 2021-02-24 11:00:00 범위를 갖는다.

③ ksqDB가 보는 가장 최신 타임스탬프인 스트림 타임은 2021-02-24 11:10:00이 된다.

추천되지만 유예 기간 설정은 선택사항이며 따라서 이는 실제 순서가 뒤바뀐/지연된 데이터 처리 방법에 의존한다. 유예 기간을 설정하지 않는 경우 윈도우 유지가 만료되어 윈도우가 닫힐 때까지 계속 열려 있을 것이다.

윈도우 유지는 구성가능하며 윈도우 데이터가 얼마나 오래 쿼리될 수 있는지를 제어하기 때문에 ksqlDB에서 중요한 역할을 한다. 다음 절에서 윈도우 유지를 제어하는 방법을 살펴보자.

윈도우 유지

윈도우 집계 결과를 쿼리하려는 경우 ksqlDB의 오래된 윈도우의 유지 기간을 제어하길 원할 수 있다. 윈도우가 제거된다면 더 이상 이에 쿼리를 할 수 없다. 윈도우 유예 기간을 명시적으로 설정하는 것에 대한 다른 동기는 상태 저장소를 작게 유지하기 위한 것이다. 더욱 많은 윈도우를 유지할 수로록 상태 저장소는 더욱 커지게 될 것이다. 6 장에서 배웠듯이 상태 저장소를 작게 유지함으로써 리밸런싱의 영향을 줄이고 또한 애플리케이션의 자원 이용율도 줄일 수 있다[4].

윈도우 유지를 설정하기 위해 WINDOW 표현식에 RETENTION 특성을 지정하면 된다. 구문은 다음과 같다:

```
WINDOW { HOPPING | TUMBLING | SESSION } ( ①
  <window_properties>, ①
```

[4] 기본적으로 상태 저장소는 데이터를 인메모리에 저장하고 선택적으로 키 공간이 클 때 디스크로 넘길 수 있는 RocksDB라는 내장 키-값 저장소를 활용한다. 따라서 큰 상태 저장소를 유지할 때 디스크와 메모리 이용률을 알아야 한다.

```
 RETENTION <number> <time_unit> ②
)
```

① 각각의 윈도우 타입 생성을 위해 윈도우 특성에 대해서는 표 11-3을 참조하기 바란다.

② 윈도우 유지 기간을 정의한다. 슬라이딩 윈도우에 대해 논의했던 동일 시간 단위가 윈도우 유지 설정에 대해서도 지원된다. "윈도우 조인"을 참조하기 바란다.

> 윈도우 유지 기간은 윈도우 크기 + 유예 기간 이상이어야 한다. 또한 유지 기간은 윈도우가 얼마나 오래 유지되는지에 대한 하한이다. 윈도우는 유지 기간이 만료되는 정확한 시간에 제거되지 않을 가능성이 높다.

예를 들어 2 일 동안 윈도우가 유지되도록 설정하기 위해 쿼리를 리팩토링해보자. 표 11-6의 SQL 문은 이를 위한 방법을 보여준다.

예제 11-6. 윈도우 유지 기간을 명시적으로 설정하는 ksqlDB 문

```
SELECT
 title_id,
 season_id,
 LATEST_BY_OFFSET(new_episode_count) AS episode_count,
 COUNT(*) AS change_count
FROM season_length_changes_enriched
WINDOW TUMBLING (
 SIZE 1 HOUR,
 RETENTION 2 DAYS, ①
 GRACE PERIOD 10 MINUTES
 GROUP BY title_id, season_id
EMIT CHANGES ;
```

① 2 일의 윈도우 유지 기간을 설정한다.

여기서 주목해야 할 한 가지 중요한 사실은 유지 기간이 카프카 스트림즈와 ksqlDB 내부 시각인 스트림 타임에 의해 구동된다는 것이다. 이는 wall clock time을 참조하지 않는다.

예제 11-6은 튜토리얼 4 단계를 완료할 수 있게 해준다. 해야 할 마지막은 이 쿼리로부터 구체화 뷰를 생성하는 것이다. 다음 절에서 이를 탐구할 것이다.

구체화 뷰

구체화 뷰는 데이터베이스 세계에서 오랫동안 존재했으며 쿼리의 결과 (구체화)를 저장하는데 사용된다. 미리 계산된 쿼리 결과는 이후 쿼리에 사용가능하며 전통적인 데이터베이스에서 배치같은 방식으로 한 번에 많은 로우에 동작하는 값비싼 쿼리의 성능을 개선하는데 적합하다.

ksqlDB 또한 전통적인 데이터베이스와 동일한 특성 중 일부를 유지하는 구체화 뷰의 개념을 갖고 있

다.

- 이들은 다른 모음에 대해 쿼리로부터 유도된다.
- 이들은 조회 스타일 방식으로 쿼리될 수 있다 (ksqlDB에서는 풀 쿼리라고 한다).

ksqlDB에서 구체화 뷰는 일부 중요한 방법에서 차이가 있다:

이 책 작성 시점에 ksqlDB의 구체화 뷰는 집계 쿼리로부터만 계산될 수 있다.

이들은 새로운 데이터가 들어옴에 따라 자동으로 갱신된다. 이를 갱신이 예약되고, 온디맨드로 발행되며 또는 존재하지 않을 수 있는 전통적인 시스템과 비교해보기 바란다.

ksqlDB에서 구체화 뷰에 대해 논의할 때 실제로는 풀 쿼리를 실행할 수 있는 특정 유형의 TABLE에 대해 논의하는 것이다. 이 책에서 이미 보았듯이 테이블은 카프카 토픽 기반으로 직접 (소스 모음 생성하기) 또는 비집계 쿼리부터 생성될 수 있다 (예, 예제 11-2). 이 책 작성 시점에 ksqlDB는 이러한 유형의 테이블에 대해서는 키를 통한 조회 (풀 쿼리)를 지원하지 않는다 (이 제한이 향후 버전에서는 없을 수도 있다).

그러나 집계 쿼리로부터 생성된 TABLE 객체인 구체화 뷰에 대해서는 풀 쿼리를 실행할 수 있다[5].

예를 들어 예제 11-6에서 생성했던 윈도우 집계 쿼리로부터 구체화 뷰를 생성해보자. 에제 11-7의 SQL 문은 이를 위한 방법을 보여준다:

```
CREATE TABLE season_length_change_counts
WITH (
 KAFKA_TOPIC = 'season_length_change_counts',
 VALUE_FORMAT = 'AVRO',
 PARTITIONS = 1
) AS
SELECT
 title_id,
 season_id,
 COUNT(*) AS change_count,
 LATEST_BY_OFFSET(new_episode_count) AS episode_count
FROM season_length_changes_enriched
WINDOW TUMBLING (
 SIZE 1 HOUR,
 RETENTION 2 DAYS,
 GRACE PERIOD 10 MINUTES
```

[5] 뷰 명명법은 전통적인 시스템에서 채택되었지만 ksqlDB에 별도의 뷰 객체는 존재하지 않는다. 따라서 ksqlDB 관련 뷰를 들을 때 조회 스타일 풀 쿼리에 사용될 수 있는 테이블 모음으로 생각해야 한다.

```
)
GROUP BY title_id, season_id
EMIT CHANGES ;
```

이제 쿼리할 수 있는 season_length_changes 라는 구체화 뷰가 존재한다. 이는 튜토리얼의 4 단계를 완료하며 이제 마지막 단계를 진행할 준비를 마쳤다. 다양한 클라이언트를 사용하여 데이터에 대한 쿼리 실행하기

클라이언트

튜토리얼의 마지막 단계는 ksqlDB로 처리했던 데이터를 다양한 클라이언트로 쿼리할 수 있음을 보장하는 것이다. 구체화 뷰 (season_length_change_counts)에 대한 쿼리와 보강 스트림 (season_length_changes_enriched)에 대해 푸시 쿼리를 실행하는 프로세스를 진행해보자. CLI와 curl 모두를 사용하여 각각의 쿼리 타입을 탐구할 것이다.

또한 curl을 사용하는데 이 커맨드 라인 유틸리티를 사용하여 운영 단계 클라이언트를 구축할 것이라는 기대가 아니라 다양한 언어로 RESTful 클라이언트를 구현할 때 유용한 참조점일 것이기 때문이다. 이는 curl의 간단한 구문때문인데 이는 요청에 사용되는 HTTP 메소드, HTTP 클라이언트 헤더와 요청 페이로드를 전달한다. 또한 ksqlDB에 의해 반환되는 응답 포맷의 명확한 이해를 위해 curl이 반환한 원시 응답을 분석할 수 있다.

> 이 책 작성 시점에 Java 클라이언트의 초기 버전 또한 출시되었다. 이 책의 소스 코드는 이 클라이언트 사용법의 예 또한 포함하고 있다. 여기서는 가까운 미래에 Java 클라이언트 인터페이스가 어떻게 변경될 것인지에 대한 불확실성 때문에 여기서는 생략했다. 또한 이 책 작성 시점에 Java 클라이언트, ksqlDB REST API와 파이썬 또는 Go로 작성된 새로운 공식적인 클라이언트의 가능한 소개를 언급했던 ksqlDB 개선 제안 (KLIP)이 논의되고 있었다.

더욱 자세하게 풀 쿼리를 탐구해보자.

풀 쿼리

구체화 뷰가 준비되었으며 뷰에 대해 풀 쿼리를 실행할 수 있다. 풀 쿼리 실행 구문은 다음과 같다:

```
SELECT select_expr [, ...]
FROM from_item
WHERE condition
```

위에서 볼 수 있듯이 풀 쿼리는 매우 단순하다. 이 책 작성 시점에 풀 쿼리 자체에서는 조인도 집계도 지원되지 않았다 (예제 11-7에서 보듯이 구체화 뷰를 생성할 때 이들 연산 모두를 수행할 수 있지만). 대신 풀 쿼리는 키 칼럼을 참조하는 단순한 조회로 간주될 수 있으며 윈도우 뷰의 경우 주어진 윈도우 타임 범위에 대한 하한을 포함하는 WINDOWSTART 의사 칼럼을 선택적으로 참조할 수 있다 (상한은 다른 의사 칼럼인 WINDOWEND에 저장되며 이 칼럼은 쿼리할 수 없다).

그렇다면 조회를 수행할 수 있는 키 칼럼의 이름은 무엇인가? 때에 따라 다르다. GROUP BY 절 내 단

일 필드로 그룹화할 때 키 칼럼의 이름은 그룹화된 필드의 이름과 일치할 것이다. 예를 들어 다음 표현식을 포함한 쿼리를 가정해보자:

```
GROUP BY title_id
```

Title_id 칼럼에 대해 조회를 수행했던 풀 쿼리를 수행할 수 있다.

그러나 우리 쿼리는 GROUP BY 절에 2 개의 칼럼을 참조하고 있다.

```
GROUP BY title_id, season_id
```

이 경우 ksqlDB는 KSQL_COL? 포맷의 키 칼럼을 생성할 것이다 (이는 향후 변경될 수 있는 ksqlDB의 현재 구현 아티팩트이다). DESCRIBE 출력으로부터 칼럼 이름을 검색할 수 있다:

```
ksql> DESCRIBE season_length_change_counts ;

Name                 : SEASON_LENGTH_CHANGE_COUNTS
 Field         | Type
--------------------------------------------------------------------------
 KSQL_COL_0    | VARCHAR(STRING)  (primary key) (Window type: TUMBLING)
 EPISODE_COUNT | INTEGER
 CHANGE_COUNT  | BIGINT
--------------------------------------------------------------------------
```

이 경우 예제 11-8의 풀 쿼리를 사용하여 season_length_change_counts 뷰에 대해 조회를 수행할 수 있다.

예제 11-8. 풀 쿼리 예

```
SELECT *
FROM season_length_change_counts
WHERE KSQL_COL_0 = '1|+|1' ; ①
```

① 복수 필드로 그룹화할 때 키는 각 칼럼의 값이 |+|로 분리된 값이다. 하나의 필드로만 그룹화한다면 WHERE title_id=1을 사용할 것이다.

> IN 서술어 또한 복수의 가능한 키를 일치시키기 위해 풀 쿼리에서 사용될 수 있다 (예, WHERE KSQL_COL_0 IN ('1|+|1', '1|+|2')).

CLI에서 이전 쿼리를 실행하면 다음 출력을 볼 것이다:

```
+------------+---------------+--------------+--------------+--------------+
|KSQL_COL_0  |WINDOWSTART    |WINDOWEND     |CHANGE_COUNT  |EPISODE_COUNT |
+------------+---------------+--------------+--------------+--------------+
|1|+|1       |1614160800000  |1614164400000 |2             |8             |
|1|+|1       |1614164400000  |1614168000000 |2             |12            |
```

위에서 볼 수 있듯이 2 개의 의사 칼럼 WINDOWSTART와 WINDOWEND가 출력에 포함되어 있다. 또한 유닉스 타임스탬프 또는 보다 가독성있는 datetime 문자열을 사용하여 WINDOWSTART 칼럼을 쿼

리할 수 있다. 다음 2 개의 문은 두 유형의 쿼리를 보여준다:

```
SELECT * FROM
season_length_change_counts
WHERE KSQL_COL_0 = '1|+|1'
AND WINDOWSTART=1614164400000;
SELECT *
FROM season_length_change_counts
WHERE KSQL_COL_0 = '1|+|1'
AND WINDOWSTART = '2021-02-24T10:00:00';
```

이 경우 이전 쿼리의 출력은 동일하다:

```
+-----------+-------------+-------------+-------------+-------------+
|KSQL_COL_0 |WINDOWSTART  |WINDOWEND    |CHANGE_COUNT |EPISODE_COUNT |
+-----------+-----------------------------+-----------------------------+
|1|+|1      |1614160800000 |1614164400000 |2            |8            |
```

구체화 뷰에서 데이터를 쿼리할 때 CLI 외에 다른 클라이언트를 사용하여 쿼리를 발행하고 싶을 수 있다. Curl 예는 파이썬, Go 또는 다른 언어로 작성된 맞춤형 클라이언트로 쉽게 확장될 수 있기 때문에 curl을 사용하여 풀 쿼리를 발행하는 방법을 논의해보자.

Curl

아마도 CLI 외부에서 ksqlDB 서버에 쿼리하는 가장 쉬운 방법은 널리 사용되고 있는 커맨드 라인 유틸리티인 curl을 사용하는 것이다. 다음 명령은 예제 11-8에서 생성했던 동일 풀 쿼리를 커맨드 라인에서 실행하는 방법을 보여준다:

```
curl -X POST "http://localhost:8088/query" ₩ ①
 -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" ₩ ②
 --data $'{ ③
 "ksql":"SELECT * FROM season_length_change_counts WHERE KSQL_COL_0=₩'1|+|1₩';",
 "streamsProperties": {}
}'
```

① /query 엔드포인트에 POST 요청을 제출하여 풀 쿼리를 실행할 수 있다.

② 컨텐츠 타입은 API 버전 v1과 직렬화 포맷 (json)을 포함한다.

③ POST 데이터로 풀 쿼리를 전달한다. 쿼리는 ksql 키 하에 JSON 객체로 중첩된다.

이전 명령의 출력은 다음과 같다. 이는 칼럼 이름 (header.schema 참조)과 하나 이상의 로우 모두를 포함한다.

```
[
  {
    "header": {
      "queryId": "query_1604158332837",
      "schema": "`KSQL_COL_0` STRING KEY, `WINDOWSTART` BIGINT KEY, `WINDOWEND`
        BIGINT KEY, `CHANGE_COUNT` BIGINT, `EPISODE_COUNT` INTEGER"
    }
  },
  {
    "row": {
      "columns": [
        "1|+|1",
        1614160800000,
        1614164400000,
        2,
        8
      ]
    }
  },
  ... ❶
]
```


① 위에서 볼 수 있듯이 CLI 외부에서도 ksqlDB 서버에 쿼리하는 것은 매우 간단하다. 이제 푸시 쿼리 실행에 대해 보다 배워보자.

푸시 쿼리

이 책에서 이미 많은 푸시 쿼리 예를 보았다. 예를 들어 CLI에서 season_length_change_coutns에 대해 쿼리를 실행하는 것은 다음 SQL 문 실행과 같이 간단하다:

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

따라서 이 절에서는 curl로 푸시 쿼리를 실행하는 것에 중점을 둔다.

Curl을 사용한 푸시 쿼리

Curl을 사용한 푸쉬 쿼리 실행은 풀 쿼리 실행과 비슷하다. 예를 들어 다음 명령을 사용하여 푸시 쿼리를 발행할 수 있다:

```
curl -X "POST" "http://localhost:8088/query" ₩
 -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" ₩
 -d $'{
 "ksql": "SELECT * FROM season_length_changes_enriched EMIT CHANGES ;",
 "streamsProperties": {}
}'
```

응답 예는 다음과 같다:

```
[{"header":{"queryId":"none","schema":"`ROWTIME` BIGINT, `ROWKEY` INTEGER,
  `TITLE_ID` INTEGER, `CHANGE_COUNT` BIGINT"}},


{"row":{"columns":[1,"Stranger Things",1,12,8,"2021-02-24 10:00:00"]}},
{"row":{"columns":[1,"Stranger Things",1,8,10,"2021-02-24 11:00:00"]}},
{"row":{"columns":[1,"Stranger Things",1,10,8,"2021-02-24 10:59:00"]}},
{"row":{"columns":[1,"Stranger Things",1,8,12,"2021-02-24 11:10:00"]}},
{"row":{"columns":[1,"Stranger Things",1,12,8,"2021-02-24 10:59:00"]}},

]
```

비고: 어떤 출력도 보지 못한다면 컨슈머 오프셋이 latest로 설정되어 있을 가능성이 높으며 따라서 확인을 위해 예제 1-5의 문을 재실행할 필요가 있다.

풀 쿼리와 달리 쿼리 실행 후 연결이 즉시 종료되지 않는다. 새로운 데이터가 사용가능할 때마다 계속 응답을 방출할 장기 연결을 통해 출력이 일련의 청크로 클라이언트에 스트리밍된다. 이는 HTTP의 청크 전달 인코딩을 통해 가능하다.

이제 ksqlDB에서 보강 및 집계된 데이터에 쿼리하는 방법을 배웠기 때문에 넷플릭스 변경 추적 애플리케이션의 마지막 단계를 완료하였다. 이 장을 마무리하기 전에 최종 주제를 논의해보자: 함수와 연산자

함수와 연산자

ksqlDB는 데이터 작업에 사용될 수 있는 풍부한 일련의 함수 및 연산자를 포함하고 있다. 우선 연산자를 빠르게 살펴보자.

연산자

ksqlDB는 SQL 문에 포함될 수 있는 여러 연산자를 포함하고 있다.

- 산술 연산자(+,-,/,*,%)
- 문자열 연결 연산자(+,||)
- 배열 인덱스 또는 맵 키에 액세스하기 위한 첨자 연산자([])
- 구조체 역참조 연산자(->)

이미 이들 중 일부가 동작하는 것을 보았는데 이 책에서는 연산자를 살펴보는데 많은 시간을 소비하지 않을 것이며 전체 연산자 참조와 사용 예는 공식 ksqlDB 문서에서 찾을 수 있다. 이제 보다 흥미로운 주제로 돌아가자: 함수

함수 보기

ksqlDB의 가장 흥미로운 기능 중 하나는 내장 함수 라이브러리이다. 이미 함수 중 하나인 COUNT를

보았는데 더욱 많은 함수가 존재하며 라이브러리 또한 계속 증가하고 있다. 사용가능한 함수를 열거하기 위해 SHOW FUNCTIONS 문을 사용할 수 있다.

예를 들어:

```
ksql> SHOW FUNCTIONS ;
```

SHOW FUNCTIONS 명령의 출력은 매우 많은데 생략된 예는 다음과 같다:

```
 Function Name         | Type
-----------------------------------
 ...

 AVG                   | AGGREGATE
 CEIL                  | SCALAR
 CHR                   | SCALAR
 COALESCE              | SCALAR
 COLLECT_LIST          | AGGREGATE
 COLLECT_SET           | AGGREGATE
 CONCAT                | SCALAR
 CONCAT_WS             | SCALAR
 COUNT                 | AGGREGATE
 COUNT_DISTINCT        | AGGREGATE
 CUBE_EXPLODE          | TABLE
 DATETOSTRING          | SCALAR
 EARLIEST_BY_OFFSET    | AGGREGATE
 ELT                   | SCALAR
 ENCODE                | SCALAR
 ENTRIES               | SCALAR
 EXP                   | SCALAR
 EXPLODE               | TABLE
 ...
```

내장 ksqlDB 함수 목록을 볼 때 TYPE 칼럼에서 3 가지 다른 범주를 알아차릴 것이다. 다음 표에 각 타입을 설명하였다.

함수 타입	설명
SCALAR	한 번에 하나의 로우에 동작하고 하나의 출력값을 반환하는 스테이트리스 함수
AGGREGATE	데이터 집계에 사용되는 스테이트풀 함수. 하나의 출력값을 반환한다.
TABLE	하나의 입력을 취해 0 이상의 출력을 산출하는 스테이트리스 함수. 이는 카프카 스트림즈에서 flatMap 동작 방식과 비슷하다.

함수 라이브러리를 살펴보았는데 특정 함수에 대해 더욱 많은 정보를 원할 수 있다. 다음 절에서 이를 위한 방법을 살펴볼 것이다

함수 설명

ksqlDB 함수에 대해 더 많은 정보를 원한다면 항상 ksqlDB 웹사이트의 공식 문서를 볼 수 있다. 그러나 함수를 설명하기 위해 CLI에서 벗어나 탐색할 필요는 없다. 왜냐하면 원하는 정보를 제공할 특수 문을 포함하고 있기 때문이다. 구문은 다음과 같다:

DESCRIBE FUNCTION <identifier>

예를 들어 내장 EARLIEST_BY_OFFSET 함수에 대해 더욱 많은 정보를 원한다면 예제 11-9의 SQL 문을 실행할 수 있다.

예제 11-9. ksqlDB의 함수 설명 방법 예

```
ksql> DESCRIBE FUNCTION EARLIEST_BY_OFFSET ;

Name        : EARLIEST_BY_OFFSET
Author      : Confluent
Overview    : This function returns the oldest value for the column,
              computed by offset. ❶
Type        : AGGREGATE ❷
Jar         : internal ❸
Variations  : ❹

        Variation   : EARLIEST_BY_OFFSET(val BOOLEAN)
        Returns     : BOOLEAN
        Description : return the earliest value of a Boolean column

        Variation   : EARLIEST_BY_OFFSET(val INT)
        Returns     : INT
        Description : return the earliest value of an integer column

        ...
```

① 함수 설명

② 함수 타입 (사용가능한 함수 타입 목록은 표 11-3 참조)

③ ksqlDB에 포함된 내장 함수를 볼 때 Jar 값은 내부로 보일 것이다. 다음 절에서 실제 디스크 상의 Jar 경로를 포함할 자신의 함수를 구축하는 방법을 살펴볼 것이다.

④ 변형 부분은 함수에 대해 모든 유효한 메소드 시그니처 목록을 포함할 것이다. 이 부분은 간결함을 위해 생략되었지만 이 함수에 대해 최소한 2 개의 변형이 있음을 볼 수 있다: 한 변형은 BOOLEAN 인수를 받아 BOOLEAN 값을 반환하고 다른 변형은 INT를 받아 INT를 반환한다.

ksqlDB는 인상적인 수의 내장 함수를 갖고 있지만 또한 필요 시 스스로 함수를 구현할 수 있는 자유도 제공한다. 따라서 SHOW FUNCTIONS 출력을 스크롤해서 유스케이스에 맞는 어떤 것도 찾지 못하더라도 초조해지지 말기 바란다. 다음 절에서 자신의 함수를 생성하는 방법을 설명할 것이다.

맞춤형 함수 생성

때때로 ksqlDB 쿼리에 사용할 맞춤형 함수를 생성하길 원할 수 있다. 예를 들어 칼럼에 특수한 수학 함수를 적용하거나 데이터 스트림의 입력을 사용하여 머신러닝 모델을 호출하길 원할 수 있다. 유스케이스의 복잡성에 상관없이 ksqlDB는 여러분의 사용자 정의 함수 셋을 통해 내장 함수 라이브러리를 확장할 수 있는 능력을 제공하는 Java 인터페이스를 포함하고 있다.

ksqlDB에는 3 가지 다른 유형의 사용자 정의 함수가 존재하는데 각각은 이전 절에서 논의했던 내장 함수 타입과 관련이 있다 (scalar, aggregate와 table 함수). 다음 표는 사용자 정의 함수 타입의 요약을 보여준다.

타입	설명
사용자 정의 함수, UDF	맞춤형 스칼라 함수. 스테이트리스로 정확히 하나의 값을 반환
사용자 정의 집계 함수, UDAF	맞춤형 집계 함수, 스테이트풀로 정확히 하나의 값을 반환
사용자 정의 테이블 함수, UDTF	맞춤형 테이블 함수. 스테이트리스로 0 개 이상의 값을 반환

맞춤형 함수 동작 원리에 대한 이해를 위해 텍스트 문자열에서 불용어를 제거하는 UDF를 구현할 것이다. UDAF와 UDTF 예도 이 장의 소스 코드에서 사용가능하지만 어떤 함수를 구현하든 개발 및 배치 프로세스가 대체로 동일하기 때문에 생략했다. UDF, UDAF와 UDTF 구현 방법에는 약간 미묘한 차이가 있는데 소스 코드에서 이들 차이를 강조했다.

이제 사용자 정의 함수 생성 방법을 배워보자.

불용어 제거 UDF

공통적인 데이터 전처리 태스크는 텍스트 문자열에서 소위 불용어를 제거하는 것이다. 불용어는 기반 텍스트에 많은 의미를 부여하지 않는 일반적인 단어이다 (예, a, and, are, but, or, the 등). 텍스트로부터 정보를 추출하려는 머신러닝 또는 자연어 모델을 갖고 있다면 우선 모델 입력에서 불용어를 제거하는 것이 일반적이다. 따라서 REMOVE_STOP_WORDS라는 UDF를 생성할 것이다. 어떤 타입이든 사용자 정의 함수를 생성하는 단계는 간단하며 다음 단계를 포함한다:

1. 함수 코드를 포함할 Java 프로젝트를 생성한다.
2. 프로젝트에 io.confluent.ksql:ksql-udf 의존성을 추가한다. 이는 UDF 클래스에 대한 주석을 포함한다.
3. 함수 구현에 필요한 다른 의존성을 추가한다. 예를 들어 코드에서 활용하려는 제3자 Maven 의존성이 있다면 이를 빌드 파일에 추가한다.
4. 적절한 주석을 사용하여 UDF 로직을 작성한다 (이를 뒤에서 다룰 것이다).
5. 함수 소스 코드와 의존하는 모든 제3자 코드를 하나의 JAR 파일로 결합하는 우버 JAR로 코드를 빌드 및 패키징한다.
6. 우버 JAR를 ksqlDB의 확장 디렉토리에 복사한다. 이는 ksqlDB 서버의 구성 파일 경로로 ksql.extension,dir 구성 특성을 사용하여 정의될 수 있다.
7. ksqlDB 서버를 재시작한다. ksqlDB 서버가 다시 동작할 때 SQL 문에서 사용할 수 있도록 새로운 함수를 적재할 것이다.

이제 동작 원리를 보기 위해 이들 지침을 사용하여 함수를 구현해보자. 우선 Gradle의 init 명령을 사용하여 새로운 Java 프로젝트를 생성해보자. 다음 코드는 이를 수행하는 방법을 보여준다:

```
mkdir udf && cd udf
gradle init ₩
 --type java-library ₩
 --dsl groovy ₩
 --test-framework junit-jupiter ₩
 --project-name udf ₩
 --package com.magicalpipelines.ksqldb
```

빌드 파일 (build.gradle)에 ksql-udf 의존성을 추기해보자. 이 의존성은 컨플루언트의 Maven 저장소에 있으며 따라서 repositories 블록을 업데이트해야 한다:

```
repositories {
 // ...
 maven {
 url = uri('http://packages.confluent.io/maven/') ①
 }
}
dependencies {
 // ...
 implementation 'io.confluent.ksql:ksqldb-udf:6.0.0' ②
}
```

① ksql-udf 의존성이 존재하는 컨플루언트 Maven 저장소를 추가한다.

② 프로젝트에 ksql-udf 의존성을 추가한다. 이 아티팩트는 뒤에 볼 것이지만 코드에 추가되어야 하는 모든 주석을 포함한다.

우리 UDF는 어떠한 제3자 코드도 필요로 하지 않는데 그런 경우 UDF가 필요로 하는 아티팩트를 갖는 dependencies 블록을 업데이트할 수 있다[6]. OK, 함수의 비즈니스 로직을 구현할 시점이다. 이 경우 RemoveStopWordsUdf.java 파일을 생성하여 다음 코드를 추가할 수 있다:

```
public class RemoveStopWordsUdf {
 private final List<String> stopWords =
 Arrays.asList(
 new String[] {"a", "and", "are", "but", "or", "over", "the"});
 private ArrayList<String> stringToWords(String source) { ①
 return Stream.of(source.toLowerCase().split(" "))
 .collect(Collectors.toCollection(ArrayList<String>::new));
 }
 private String wordsToString(ArrayList<String> words) { ②
```

[6] 의존성은 com.github.jonrengelman.shadow 플러그인과 같은 플러그인을 사용하여 우버 JAR로 패키징되어야 할 것이다. 더욱 자세한 정보는 UDF 문서를 참조하기 바란다.

```
 return words.stream().collect(Collectors.joining(" "));
 }
 public String apply(final String source) { ③
 ArrayList<String> words = stringToWords(source);
 words.removeAll(stopWords);
 return wordsToString(words);
 }
}
```

① 이 메소드는 텍스트 문자열을 단어 목록으로 변환한다.

② 이 메소드는 단어 목록을 다시 문자열로 변환한다.

③ 이 메소드는 소스 문자열로부터 불용어 목록을 제거하는 비즈니스 로직을 구현한다. 이 메소드 이름은 원하는 데로 만들 수 있지만 비정적으로 public 가시성을 가져야 한다.

UDF에 대한 비즈니스 로직을 작성했기 때문에 ksql-udf 의존성에 포함되는 주석을 추가해야 한다. 이들 주석은 JAR 적재 시 ksqlDB가 읽으며 ksqlDB에 이름, 설명 및 파라미터를 포함하여 많은 정보를 제공한다. 뒤에 볼 것인데 주석에 제공하는 값은 DESCRIBE FUNCTION 출력에서 볼 수 있다:

```
@UdfDescription( ①
 name = "remove_stop_words", ②
 description = "A UDF that removes stop words from a string of text",
 version = "0.1.0",
 author = "Mitch Seymour")
public class RemoveStopWordsUdf {
 // ...
 @Udf(description = "Remove the default stop words from a string of text") ③
 public String apply(
 @UdfParameter(value = "source", description = "the raw source string") ④
 final String source
 ) { ... }
}
```

① UdfDescription 주석은 구동 시 이 클래스가 ksqlDB 라이브러리 내로 적재되어야 하는 ksqlDB 함수를 포함함을 ksqlDB에 알려준다. UDAF 또는 UDTF를 구현한다면 대신 UdafDescription/UdtfDescription가 사용되어야 한다.

② 함수 라이브러리에 나타나는 함수 이름. 특성 나머지 (설명, 버전 및 저자)는 DESCRIBE FUNCTION 출력에 나타날 것이다.

③ Udf 주석이 ksqlDB에 의해 호출될 수 있는 공개 메소드에 적용되어야 한다. 단일 Udf 내에 Udf 주석을 갖는 여러 메소드를 가질 수 있으며 EARLIEST_BY_OFFSET 함수를 논의할 때 논의했듯이 각각은 함수의 별도 변형으로 고려될 것이다. UDTF와 UDAF는 다른 주석을 사용함을 주목하기 바란다

(Udtf와 UdafFactory). 자세한 사항은 이 장의 소스 코드를 참조하기 바란다.

④ UdfParameter은 UDF의 각 파라미터에 대해 도움이 되는 정보를 제공하는데 사용될 수 있다. 이는 DESCRIBE FUNCTION 출력에서 VARIATIONS 부분에 나타난다.

UDF에 대한 비즈니스 로직을 작성했고 클래스 수준 및 메소드 수준 주석을 추가했다면 우버 JAR를 빌드할 시간이다. Gradle을 통해 다음 명령을 사용하여 수행될 수 있다:

```
./gradlew build --info
```

이전 명령은 Java 프로젝트의 루트 디렉토리에 대해 다음 경로에 JAR를 생성할 것이다.

```
build/libs/udf.jar
```

새로운 UDF에 대해 ksqlDB에 알려주기 위해 이 JAR를 ksql.extension.dir 구성 파라미터에 의해 지정된 경로에 놓아야 한다. 이 파라미터를 ksqlDB 서버 구성 파일에 정의할 수 있다. 예를 들아:

```
ksql.extension.dir=/etc/ksqldb/extensions
```

ksqlDB 확장 디렉토리의 경로를 알았다면 다음과 같이 JAR를 이 경로에 복사하고 ksqlDB 서버를 재시작할 수 있다.

```
cp build/libs/udf.jar /etc/ksqldb/extensions
ksql-server-stop
ksql-server-start
```

재시작 시 SHOR FUNCTIONS 명령의 출력에서 UDF를 볼 수 있어야 한다.

```
ksql> SHOW FUNCTIONS ;

 Function Name          | Type
-----------------------------------
...
...
REMOVE_STOP_WORDS      | SCALAR ❶
...
...
```

① UdfDescription 주석에 의해 정의된 함수 이름이 알파벳 순서의 함수 목록에 나타날 것이다.

또한 내장 함수를 설명하는데 사용된 동일 SQL 문을 사용하여 맞춤형 함수를 설명할 수도 있다.

```
ksql> DESCRIBE FUNCTION REMOVE_STOP_WORDS ;

Name        : REMOVE_STOP_WORDS
Author      : Mitch Seymour
Version     : 0.1.0
Overview    : A UDF that removes stop words from a string of text
Type        : SCALAR
Jar         : /etc/ksqldb/extensions/udf.jar ❶
Variations  :

        Variation   : REMOVE_STOP_WORDS(source VARCHAR)
        Returns     : VARCHAR
Description : Remove the default stop words from a string of text
source      : the raw source string
```

① Jar 값이 디스크 상의 UDF JAR의 물리적 경로를 지정함을 주목하기 바란다.

마지막으로 ksqlDB의 내장 함수와 같이 함수를 사용할 수 있다. Model_inpts이라는 스트림을 생성한 후 함수를 실행시킬 수 있는 테스트 데이터를 삽입해보자:

```
CREATE STREAM model_inputs (
 text STRING
)
WITH (
 KAFKA_TOPIC='model_inputs',
 VALUE_FORMAT='JSON',
 PARTITIONS=4
);
INSERT INTO model_inputs VALUES ('The quick brown fox jumps over the lazy dog');
```

이제 함수를 적용해보자:

```
SELECT
 text AS original,
 remove_stop_words(text) AS no_stop_words
FROM model_inputs
EMIT CHANGES;
```

다음 출력에서 볼 수 있듯이 새로운 함수가 예상한데로 동작한다:

```
+-------------------------------------------+-------------------------------+
|ORIGINAL                                   |NO_STOP_WORDS                  |
+-------------------------------------------+-------------------------------+
|The quick brown fox jumps over the lazy dog |quick brown fox jumps lazy dog |
```

맞춤형 ksqlDB 함수에 대한 추가 리소스

맞춤형 ksqlDB 함수를 생성하는 것은 큰 주제로 이에 여러 장을 할당할 수 있다. 그러나 다른 곳에서 ksqlDB 함수에 대해 말하고 쓰는데 많은 시간을 보냈으며 공식 문서 또한 이 기능에 대해 많은 정보를

포함하고 있다. 맞춤형 ksqlDB 함수에 대한 더욱 자세한 정보를 다음 리소스를 확인하기 바란다.

- "The Exciting Frontier of Custom KSQL Functions" (Mitch Seymour, Kafka Summit 2019)
- "ksqlDB UDFs and UDAFs Made Easy" (Mitch Seymour, Confluent Blog)
- Official ksqlDB Documentation

요약

단순한 인터페이스를 가짐에도 불구하고 ksqlDB는 다른 모음 내 데이터 조인, 데이터 집계, 키 조회를 사용하여 쿼리될 수 있는 구체화 뷰 생성 등을 포함하여 많은 중급에서 고급 스트림 처리 유스케이스를 지원한다. 또한 ksqlDB는 다양한 범위의 데이터 처리 및 보강 유스케이스를 해결하는데 사용할 수 있는 광범위한 내장 함수 라이브러리를 제공한다. 일반적인 수학 함수 (AVG, COUNT_DISTINCT, MAX, MIN, SUM 등), 문자열 함수 ((LPAD, REGEXP_EXTRACT, REPLACE, TRIM, UCASE 등) 또는 지리공간 함수 (GEO_DISTANCE)를 찾고 있는지에 상관없이 ksqlDB는 이를 다룰 수 있다.

그러나 ksqlDB가 실제 빛을 발하는 것은 개발자에게 맞춤형 Java 함수를 통해 내장 함수 라이브러리를 확장하기 위한 옵션을 제공하면서 고수준 인터페이스를 제공할 수 있다는 것이다. 이는 애플리케이션이 맞춤형 비즈니스 로직을 필요로 하더라도 간단한 SQL 통용어를 사용하여 스트림 처리 애플리케이션을 계속 구축할 수 있도록 하기 때문에 엄청나게 중요하다. 이는 Java 지식을 필요로 하기 때문에 더욱 고급 사용으로 전환하지만 이 장에서 보여줬듯이 이 프로세스는 ksqlDB에 의해 간단해졌다.

이제 ksqlDB를 통해 기본, 중간 및 고급 스트림 처리 유스케이스를 해결하는 방법을 배웠기 때문에 이 책의 마지막으로 이동해보자. 여기서는 카프카 스트림즈와 ksqlDB 애플리케이션 테스트, 배치 및 모니터링 방법을 배울 것이다.