8 장 ksqlDB 시작하기

ksqlDB에 대한 스토리는 단순화와 진화에 대한 스토리이다. ksqlDB는 스트림 처리 애플리케이션 구축 과정 단순화라는 카프카 스트림즈와 같은 목표를 갖고 개발되었다. 그러나 ksqlDB가 진화함에 따라 목표가 카프카 스트림즈보다 더욱 엄청났음이 명백해졌다. 이는 ksqlDB가 스트림 처리 애플리케이션을 구축하는 방법 뿐만 아니라 이들 애플리케이션을 다른 시스템 (카프카 외부 시스템 포함)과 통합하는 방법을 단순화하기 때문이다. ksqlDB는 SQL 인터페이스를 통해 이 모든 것을 하며 초보자와 전문가 모두 카프카의 힘을 활용하는 것을 쉽게 만든다.

이제 여러분이 생각하고 있는 것을 알고 있다: 왜 카프카 스트림즈와 ksqlDB 모두 알아야 하는지, 그리고 이 책의 한 부분을 찢고 투자의 일부분을 복구하기 위해 Graigs-list에서 이를 팔 수 있는지? 실제로 카프카 스트림즈와 ksqlDB 모두 스트림 처리 툴벨트에 갖고 있어야 할 훌륭한 도구이고 서로를 잘 보완한다. SQL로 표현될 수 있는 스트림 처리 애플리케이션을 위해 그리고 단일 툴을 사용하여 종단간 데이터 처리 파이프라인을 생성하기 위한 데이터 소스 및 싱크를 쉽게 설정하기 위해 ksqlDB를 사용할 수 있다. 반면 보다 복잡한 애플리케이션에 카프카 스트림즈를 사용할 수 있으며 이들 라이브러리에 대한 지식은 ksqlDB가 실제로 카프카 스트림즈 위에 구축되어 있기 때문에 ksqlDB에 대한 이해를 깊게 할 것이다.

카프카 스트림즈 위에 구축된 ksqlDB에 대한 부분은 이 장 말미에 공개할 예정이었는데, 이는 아마도 물을 벌컥 마시는 도중에 여러분의 주의를 끌었을 것이다. 그러나 이를 공유하기 전에 두 단락을 만들 수 없었다. 이것이 큰 셀링 포인트이고 ksqlDB를 사용한 작업을 좋아할 것이라고 생각한 이유이다. 대부분의 데이터베이스는 내부를 들여다보기 시작하면 엄청 복잡해지며 이는 몇 달 동안의 지루한 내부 공부없이는 기술 전문성 확보를 어렵게 한다. 그러나 카프카 스트림즈를 핵심 수단으로 한다는 것은 ksqlDB가 잘 설계되고 쉽게 이해되는 추상화 계층 위에 구축되어 있음을 의미하며 이는 내부에 깊이 들어가 재미있고 접근 가능한 방식으로 이 기술의 힘을 완전히 활용하는 법을 배울 수 있도록 한다. 사실 이 책의 카프카 스트림즈 부분을 ksqlDB 내부의 초기 모습으로 볼 수도 있다.

몇 장을 ksqlDB의 연구에 할애할 만큼 ksqlDB에는 많은 훌륭한 기능이 존재한다. 이 첫번째 장에서 이 기술을 처음으로 살펴보고 일부 중요한 질문에 답을 할 것이다:

- ksqlDB가 정확히 무엇인지
- ksqlDB를 언제 사용해야 하는지
- ksqlDB의 기능이 시간에 따라 어떻게 진화했는지
- ksqlDB가 어떤 분야에서 단순화를 제공하는지
- ksqlDB 아키텍터의 주요 구성요소는 무엇인지
- ksqlDB를 어떻게 설치 및 실행시킬 수 있는지

따라서 더 이상 고민하지 말고 ksqlDB가 실제 무엇이고 무엇을 할 수 있는지에 대해 보다 컨텍스트를

제공함으로써 시작해보자.

ksqlDB가 무엇인지

ksqlDB는 (카프카 생태계에 카프카 스트림즈가 도입된 후 1년이 약간 지난) 2017년 컨플루언트에서 출시한 오프 소스 이벤트 스트리밍 데이터베이스이다. ksqlDB는 카프카 생태계의 두 특수 컴포넌트를 단일 시스템으로 통합하고 이들 컴포넌트와 상호작용하기 위한 고수준 SQL 인터페이스를 제공함으로써 스트림 처리 애플리케이션의 구축, 배치 및 유지보수 방식을 단순화한다. ksqlDB로 할 수 있는 것은 다음을 포함한다:

- SQL을 사용하여 데이터를 스트림 또는 테이블 (이들 각각은 ksqlDB에서 모음(collection)으로 간주)로 모델링한다.
- Java 코드 작성없이 데이터에 대한 새로운 파생 표현을 생성하기 위해 많은 SQL 구문 (예, 데이터 조인, 집계, 변환, 필터링 및 윈도우)을 적용한다.
- 연속적으로 실행되어 새로운 데이터가 사용가능할 때마다 클라이언트에 결과를 방출/푸시하는 푸시 쿼리를 사용하여 스트림과 테이블에 쿼리한다. 내부적으로 푸시 쿼리는 카프카 스트림즈 애플리케이션으로 컴파일되며 이벤트를 관찰하여 빨리 대응해야 하는 이벤트 기반 마이크로 서비스에 이상적이다.
- 스트림과 테이블의 구체화된 뷰를 생성하고 풀 쿼리를 사용하여 이들 뷰에 쿼리한다. 풀 쿼리는 전통적인 SQL 데이터베이스의 키 조회 방식과 유사하며 내부적으로 카프카 스트림즈와 상태 저장소를 활용한다. 풀 쿼리는 동기성/주문형(on demand) 워크 플로우에서 ksqlDB를 사용하여 작업을 해야 하는 클라이언트에 의해 사용될 수 있다.
- 외부 데이터 저장소와 ksqlDB를 통합하는 커넥터를 정의하며 이는 다양한 데이터 소스와 싱크로부터 데이터를 읽어 와 쓸 수 있도록 한다. 커넥터를 테이블 및 스트림과 결합하여 양단간 스트리밍 ETL 파이프라인을 생성할 수도 있다[1].

다음 장에서 이러한 능력 각각을 세부적으로 탐구할 것이다. ksqlDB가 카프카 커넥터와 카프카 스트림즈라는 성숙한 기반 위에 구축되어 있기 때문에 자유롭게 이들 툴의 안정성과 파워뿐만 아니라 보다 친화적인 인터페이스의 장점을 얻을 수 있다. 다음 절에서 ksqlDB 사용 시 장점을 논의할 것이며 이는 ksqlDB를 언제 사용할지를 알려줄 것이다.

언제 ksqlDB를 사용할지

보다 고수준 추상화가 보다 낮은 추상화보다 작업하는 것이 보다 쉽다는 것은 놀랍지 않다. 그러나 가령 "SLQ이 Java보다 작성하기 쉽다"라고 말한다면 ksqlDB 사용 시 보다 단순한 인터페이스와 아키텍

[1] 추출(Extract), 변환(Transform), 적재(Load)

처에 기인하는 많은 장점에 대해 설명하고 있는 것이다. 이들 장점은 다음을 포함한다:

- 쿼리 제출을 위한 CLI 및 REST 서비스를 사용하여 주문형 스트림 처리 애플리케이션을 구성하고 해체할 수 있는 관리형 런타임에 기인한 보다 상호대화형 워크 플로우
- 스트림 처리 토폴로지가 JVM 언어 대신 SQL을 사용해 표현되기 때문에 유지할 코드가 더 적다.
- 진입 장벽이 보다 낮고, 배울 새로운 개념이 더 적음, 특히 전통적인 SQL 데이터베이스에 익숙하지만 스트림 처리가 처음인 사람의 경우. 이는 신규 개발자 대상 프로젝트 적응 프로그램과 유지보수 가능성을 향상시킨다.
- 단순화된 아키텍처. (외부 데이터 소스를 카프카 내로 통합하는) 커넥터 관리와 데이터 변환 인터페이스가 단일 시스템으로 결합하였기 때문이다. ksqlDB와 동일한 JVM에서 카프카 커넥트를 실행하기 위한 옵션 또한 존재한다[2].
- 개발자 생산성 향상. 스트림 처리 애플리케이션을 표현하기 위해 더 적은 코드를 작성하고 저수준 시스템의 복잡성이 새로운 추상화 계층 이면에 감춰지기 때문이다. 또한 상호대화형 워크 플로우는 보다 빠른 피드백 루프로 변환되며 쿼리 검증을 매우 간단하게 하는 테스트 도구가 포함되어 있다[3].
- 프로젝트 간 일관성. SQL의 선언적 구문 덕분에 SQL로 표현된 스트림 처리 애플리케이션은 종분화 (다른 유사한 프로젝트와 차별되도록 하는 고유 기능 개발)로 고통을 받을 가능성이 적다. 카프카 스트림즈는 이벤트 스트림 작업을 위한 일련의 표준 DSL 연산자를 도입함으로써 이미 이 작업을 훌륭하게 수행하고 있으며 애플리케이션 코드 나머지를 관리할 때 많은 자유가 존재하며 이는 소보다 애완 동물 같은 애플리케이션으로 이어질 수 있다 (애완 동물: 메인프레임, 단독 서버, 고가용성 로드 밸런서/방화벽, DB 시스템 등과 같이 다운되서는 안되는 시스템, 소: 자동화 툴을 사용하여 구축된 2 개 이상의 서버로 구성되어 내고장성 등이 보장되는 시스템).
- 공식적으로 지원되는 도커 이미지를 포함하여 여러 배포 옵션으로 인한 손쉬운 설정 및 턴키 배치. 실제 운영까지 간소화된 경로를 원하는 사람들을 위해 완전 관리형 클라우드 기반 ksqlDB 제품도 존재한다 (예, 컨플루언트 클라우드).
- 보다 우수한 데이터 탐구 지원. ksqlDB는 토픽의 내용 열거 및 출력을 손쉽게 하고 데이터에

---

[2] 운영 단계에서 이 설정을 사용할지 여부는 워크로드에 달려있다. 9 장에서 논의할 것인데 큰 작업 부하의 경우 컴포넌트들을 독립적으로 확장할 수 있도록 커넥트를 외부에서 실행시키고 싶을 것이다. 그러나 결합된 설정의 경우 개발 목적에는 매우 편리하다.

[3] 12 장에서 살펴볼 것인데 카프카 스트림즈 애플리케이션 테스팅은 매우 쉬움을 주목하기 바란다.

대한 구체화된 뷰 생성 및 쿼리를 빨리 할 수 있도록 한다. 이러한 유형의 데이터 탐구 유스케이스는 ksqlDB에 더욱 적합하다.

이제 ksqlDB 사용 시 장점의 일부를 논의했기 때문에 카프카 스트림즈 대신 언제 ksqlDB를 사용해야 하는지를 살펴보자. 들을 수 있는 가장 일반적인 대답은 SQL을 사용하여 스트림 처리 애플리케이션이 표현될 수 있을 때마다 ksqlDB를 사용한다는 것이다. 그러나 이러한 답은 완전하지 않으며 대신 프로젝트가 이전에 언급한 장점 중 어떤 것이라도 활용할 수 있고 스트림 처리 애플리케이션이 자연스럽고 간단하게 SQL을 사용하여 표현될 수 있을 때 ksqlDB를 사용할 것을 제안한다.

예를 들어 ksqlDB의 가장 크고 가장 강력한 기능 중 하나는 사용자 내장 함수를 맞춤형 Java 기반 함수로 확장할 수 있다는 것이다. 이 기능을 정기적으로 활용하길 원할 수 있으며 특정 애플리케이션/쿼리 셋에 대해 JVM 수준에서 빈번하게 동작하는 경우 적절한 추상화 수준에서 동작하는 지 여부를 평가하는 것은 가치가 있다 (예, 카프카 스트림즈를 사용하는 것이 더 합리적일 수 있다).

카프카 스트림즈에 더 적합한 추가적인 유스케이스도 존재한다. 예를 들어 애플리케이션 상태에 대해 보다 저수준의 액세스가 필요하고 데이터에 대해 주기적 함수를 실행시켜야 하며 ksqlDB에서 지원되지 않는 데이터 포맷으로 작업해야 하고, 애플리케이션 프로파일링/모니터링 (예, 분산 추적, 맞춤형 지표 수집 등)에 대해 보다 많은 유연성을 원하거나 SQL로 쉽게 표현되지 않는 많은 비즈니스 로직을 갖고 있다면 카프카 스트림즈가 더욱 적합하다.

언제 ksqlDB를 사용하고 어떤 장점을 제공하는지를 논의하였으며 따라서 ksqlDB가 어떻게 시간에 따라 진화했고 이들 통합을 개선했는지를 살펴봄으로써 개별 통합 (카프카 스트림즈와 카프카 커넥트)에 대해 보다 잘 이해하게 될 것이다.

새로운 유형의 데이터베이스의 진화

ksqlDB가 시간에 따라 어떻게 진화했고 그 과정에서 어떤 능력을 얻었는지를 이해하는 것은 도움이 된다. ksqlDB의 진화가 흥미롭지만 이 절은 단순히 역사를 살펴보는 것보다 큰 목적을 갖고 있다. ksqlDB가 다른 이름 (KSQL)으로 알려져 있기 때문에 언제 특정 기능이 도입되었는지를 앎으로써 이 기술에 대해 다른 세대를 구분할 수 있다.

ksqlDB가 시간이 지남에 따라 카프카 스트림즈 통합을 어떻게 발전시켰고 카프카 스트림즈가 ksqlDB의 가장 근본적인 기능 중 하나인 데이터 쿼리를 어떻게 지원하는 지로 이 절을 시작해보자.

카프카 스트림즈 통합

처음 2 년 동안 ksqlDB는 KSQL로 알려졌다. 초기 개발은 핵심 기능에 중점을 두었다; SQL 문을 분석하고 이를 본격적인 스트림 처리 애플리케이션으로 컴파일할 수 있는 스트리밍 SQL 엔진. 이러한 초기 진화 형태에서 KSQL은 관계형 데이터베이스 (RDBMS)로부터 기능을 차용하고 스트림 처리 계층에서 무거운 작업을 수행하기 위해 카프카 스트림즈를 사용하여 개념적으로 전통적인 SQL 데이터베이스와 카프카 스트림즈를 혼합한 것이었다. 그림 8-1은 이를 보여준다.

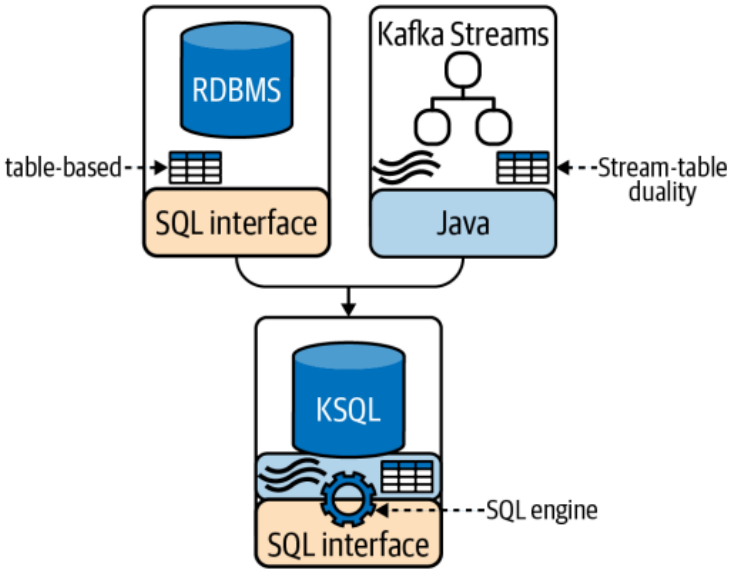

그림 8-1. ksqlDB 진화의 첫번째 단계에서는 카프카 스트림즈와 SQL 인터페이스를 포함하는 전통적인 SQL 데이터베이스의 기능을 결합하였다.

KSQL이 진화 트리의 RDBMS 분기에서 차용한 가장 주목할 만한 기능은 SQL 인터페이스이다. 사용자가 카프카 스트림즈를 사용하기 위해 Java 또는 Scala와 같은 JVM 언어를 더 이상 사용할 필요가 없기 때문에 이는 카프카 생태계에서 스트림 처리 애플리케이션 구축에 대한 언어 장벽을 제거하였다.

Why SQL

SQL 자체는 여러 차례의 제품 개발 및 단순화의 산물이다. 영구 데이터 저장소에서 데이터를 검색하는 것은 복잡한 언어로 긴 프로그램의 작성을 필요로 하곤 했다. 그러나 수학적 표기와 선언적 구문을 사용함으로써 관계형 언어는 보다 더 간결해지고 효율적이 되었다. 언어 최적화라는 추가 과정을 통해 새로운 언어인 SQL은 보다 자연스러운 영어처럼 읽을 수 있게 되었다[4]. 이러한 모든 단순화 과정을 거친 후 얻은 것은 오늘날 폭넓은 인기를 받고 있는 영구 저장소 쿼리를 위한 고도의 액세스 가능한 언어이다. 스트리밍 유스케이스에 SQL을 채택함으로써 ksqlDB는 고전적인 SQL 사용 시와 동일한 장점을 즉시 실현한다:

- 영어처럼 읽는 간결하고 표현적인 구문
- 선언적 프로그래밍 스타일
- 낮은 학습 곡선

---

[4] D. D. Chamberlin, "Early History of SQL," in IEEE Annals of the History of Computing, vol. 34, no. 4, pp. 78– 82, Oct.–Dec. 2012.

SQL 문법이 ANSI SQL에 고무되었지만 스트림과 테이블 내 모두의 데이터를 모델링하기 위해서는 특수한 SQL 통용어가 필요했다. 전통적인 SQL 데이터베이스가 기본적으로 후자와 관련되어 있으며 무제한 데이터셋 (스트림)은 기본적으로 지원하지 않는다. 이는 고전적인 SQL에서 2 가지 유형의 문으로 일반적으로 나타난다:

- 고전적인 DDL (Data Definition Language, 데이터 정의 언어) 문은 데이터베이스 객체 생성 및 파괴에 중점을 둔다 (보통 테이블이지만 때때로 데이터베이스, 뷰 등):

```
CREATE TABLE users ...;
DROP TABLE users;
```

- 고전적인 DML (Data Manipulation Language, 데이터 처리 언어) 문은 테이블 내 데이터 읽기 및 처리에 중점을 둔다:

```
SELECT username from USERS;
INSERT INTO users (id, username) VALUES(2, "Izzy");
```

KSQL (그리고 추후 ksqlDB)에 구현된 SQL 통용어는 스트림을 지원하기 위해 고전적인 SQL을 확장한다. 다음 장에서 확장된 DDL과 DML을 탐구할 것이며 예상할 수 있듯이 스트림에 대해 CREATE TABLE and DROP TABLE 문과 같은 동등한 문이 존재하며 확장된 DML은 스트림과 테이블 모두에 대한 쿼리를 지원한다.

ksqlDB 진화와 관련하여 이 기술의 초기 형태인 KSQL이 기본적으로 푸시 쿼리를 지원하기 위해 카프카 스트림즈를 사용했음을 주목하는 것은 중요하다. 이들은 연속적으로 실행되는 쿼리로 스트림 또는 테이블에 대해 실행될 수 있고 새로운 데이터가 사용가능해질 때마다 클라이언트에 결과를 방출 (또는 푸시)할 수 있다. 푸시 쿼리에 대한 데이터 흐름은 그림 8-2에 나타냈다.

> 당시 KSQL의 쿼리는 푸시 쿼리로 간주되지 않았다. 이 책에서는 KSQL 쿼리의 초기 형태를 기술하기 위해 아직까지 이 용어를 사용하지만 이 쿼리가 클라이언트에 결과를 푸시하는 방법에 대해 충분히 설명적이기 때문에 용어는 나중에 개발되었다.

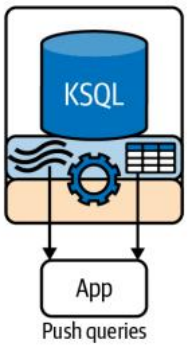

그림 8-2. 푸시 쿼리는 새로운 데이터가 사용가능해질 때마다 자동적으로 결과를 방출한다; 이는 애플리케이션/클라이언트가 새로운 데이터를 리스닝할 수 있도록 한다.

시간이 지남에 따라 SQL 엔진은 보다 진보하게 되었고 KSQL이 ksqlDB로 재명명되었을 때 이름 변경과 함께 중요한 기능이 도입되었다: 풀 쿼리. 풀 쿼리의 수명은 전통적인 데이터베이스에서 하는 쿼리의 수명과 비슷한데 이는 데이터의 키 조회를 수행하기 위해 사용되는 짧은 수명의 쿼리이기 때문이다. 내부에서 이는 카프카 스트림즈와 상태 저장소를 활용한다. 4 장에서 상기할 수 있듯이 상태 저장소는 보통 RocksDB에서 제공하는 로컬 내장형 키-값 저장소이다. 그림 8-3은 ksqlDB에서 사용가능한 푸시와 풀 쿼리 모두에 대한 데이터 흐름을 보여준다.

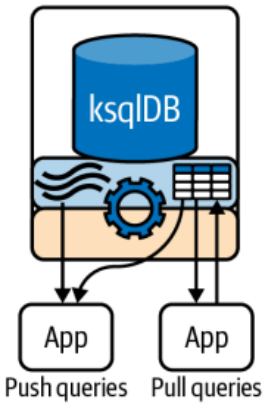

그림 8-3. ksqlDB는 푸시 및 풀 쿼리 모두를 지원한다.

두 유형의 쿼리 모두 카프카 스트림즈와 아키텍처에서 보다 깊게 살펴볼 ksqlDB 고유의 SQL 엔진에 크게 의존한다. 이제 카프카 스트림즈 통합을 살펴보았기 때문에 카프카 커넥트 통합을 살펴보고 카프카 스트림즈 통합과 다른 일련의 스트리밍 유스케이스를 어떻게 처리하는지를 배워보자.

커넥트 통합

2 장에서 배웠듯이 카프카 스트림즈 애플리케이션은 카프카 토픽에서 읽고 이에 쓴다. 따라서 처리할 데이터가 카프카 외부에 있거나 카프카 스트림즈 애플리케이션의 출력을 외부 데이터 저장소로 보내고자 한다면 적절한 시스템들로/로부터 데이터를 이동하기 위해 데이터 파이프라인을 구축해야 한다. 이 ETL 프로세스는 카프카 생태계의 별도 컴포넌트인 카프카 커넥트에 의해 보통 처리된다. 따라서 바닐라 카프카 스트림즈를 사용할 때 카프카 커넥트와 적절한 싱크/소스 커넥터를 스스로 배치해야 한다.

초기 형태인 KSQL은 바닐라 카프카 스트림즈에 동일한 제한을 부과하였다. 비카프카 데이터 소스와의 통합은 커넥터 관리가 별도 시스템에 의해 처리되어야 하기 때문에 추가적인 아키텍처 복잡성과 운영 오버헤드를 부담해야 했다. 그러나 KSQL이 보다 진보된 형태인 ksqlDB로 진화했을 때 이는 일부 새로운 ETL 기능을 가져왔다. 이는 카프카 커넥트 통합을 기능 목록에 추가함으로써 가능해졌다. 이 통합은 다음을 포함한다:

- 소스와 싱크 커넥터 정의를 위한 추가적 SQL 문. 다음 장의 튜토리얼에서 보다 많은 코드 예제를 제공할 것인데 증강된 DDL을 최초 모습은 다음과 같다:

```
CREATE SOURCE CONNECTOR `jdbc-connector` WITH (
 "connector.class"='io.confluent.connect.jdbc.JdbcSourceConnector',
 "connection.url"='jdbc:postgresql://localhost:5432/my.db',
 "mode"='bulk',
 "topic.prefix"='jdbc-',
 "table.whitelist"='users',
 "key"='username'
);
```

- 외부에 배치된 카프카 커넥트 클러스터에서 커넥터를 관리 및 실행하거나 보다 간단한 설정을 위해 ksqlDB와 함께 분산 카프카 커넥트 클러스터를 실행할 수 있는 기능

카프카 커넥트 통합을 통해 ksqlDB는 카프카 스트림즈 통합에 의해 처리되는 ETL 프로세스의 변환 부분 대신 이벤트 스트림의 전체 ETL 수명주기를 지원한다. 그림 8-4는 데이터 통합과 변환 기능 모두를 갖는 ksqlDB의 업데이트된 모습을 보여준다.

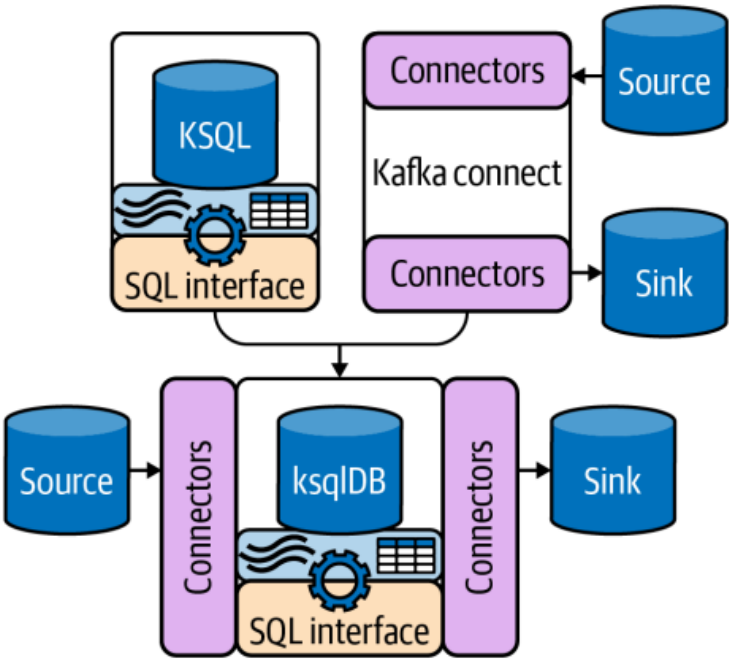

그림 8-4. ksqlDB는 데이터 변환과 통합 모두를 지원하는 시스템으로 진화하였다.

ksqlDB의 쿼리 기능을 구동하는 카프카 스트림즈 통합과 추가적인 데이터 소스를 쿼리에 사용가능하게 돕는 카프카 커넥트 통합 모두를 통해 ksqlDB의 강력함을 보기 시작할 수 있다.

ksqlDB가 전통적인 SQL 데이터베이스와 어떻게 비교되는지?

이제 ksqlDB가 어떻게 새로운 유형의 데이터베이스 (이벤트 스트리밍 데이터베이스)로 진화했는지를 살펴보았기 때문에 전통적인 SQL 데이터베이스와 어떻게 비교되는지를 탐구해보자. 결국 데이터베이스 관리자가 몇 달 동안 계속 실행될 쿼리를 실행하고 있음을 들을 때 답을 원하고 있다. 우선 ksqlDB와 전통적인 SQL 데이터베이스 간 유사성을 살펴봄으로써 시작해보자.

유사성

스트리밍에 중점을 두고 있지만 ksqlDB는 전통적인 SQL 데이터베이스의 많은 특성을 갖고 있다. 이러한 유사성은 다음을 포함한다:

SQL 인터페이스

전통적인 SQL 데이터베이스와 같이 ksqlDB는 SQL 문법, 파서와 실행 엔진을 포함한다. 이는 각 유형의 시스템에서 고수준 선언적 언어인 SQL을 사용하여 데이터와 상호작용할 수 있음을 의미한다. ksqlDB의 SQL 통용어는 투영을 위한 SELECT, 소스 정의를 위한 FROM, 필터링을 위한 WHERE, 조인을 위한 JOIN 등을 포함하여 예상할 수 있는 언어 문을 포함한다.

DDL과 DML 문

DDL과 DML은 전통적인 SQL 데이터베이스와 ksqlDB 모두에서 지원되는 2 가지 넓은 범주의 문이다. DDL 문은 데이터베이스 객체 (전통적인 데이터베이스 내 테이블, ksqlDB 내 테이블과 스트림) 생성 및 파괴를 담당하고 DML 문은 데이터 읽기 및 처리에 사용된다.

쿼리 제출을 위한 네트워크 서비스 및 클라이언트

전통적인 SQL 데이터베이스로 작업했다면 2 가지 사실을 예상할 것 같다: 네트워크를 통해 데이터베이스에 연결할 수 있고 쿼리 제출을 위한 기본 클라이언트 구현 (예, CLI)이 존재한다. ksqlDB 또한 이들 기능을 포함하며 네트워크 서비스가 REST API로 구현되어 있고 상호대화식으로 쿼리를 제출하기 위해 ksqlDB CLI와 UI가 구현되어 있다. ksqlDB 서버와 상호작용하기 위한 Java 클라이언트 또한 존재한다.

스키마

상호작용하는 모음은 필드 이름과 타입을 포함하는 스키마 정의를 포함한다. 또한 보다 유연한 데이터베이스 시스템 (예, Postgres)과 같이 ksqlDB는 사용자 정의 타입 또한 지원한다.

구체화 뷰

전통적인 데이터베이스에서 읽기 성능을 최적화하기 위해 사용자는 때때로 쿼리 결과를 포함하는 이름이 있는 객체인 구체화 뷰를 생성한다. 그러나 전통적인 시스템에서 이 뷰는 느리게 (뷰 업데이트가 나중을 위해 대기하거나 수동으로 적용해야 함) 또는 빠르게 (새로운 데이터가 도착할 때마다) 업데이트될 수 있다. 빠르게 유지되는 뷰는 ksqlDB가 데이터를 표현하는 방법과 유사하며 새로운 데이터가 사용가능해질 때마다 즉시 스트림과 테이블이 업데이트된다.

데이터 변환을 위한 내장 함수 및 연산자

많은 전통적인 SQL 데이터베이스와 같이 ksqlDB는 데이터 작업을 위해 풍부한 일련의 함수 및 연산자를 포함한다. "함수 및 연산자"에서 함수와 연산자를 세부적으로 논의할 것이며 여기서는 다양한 문자열 함수, 수학 함수, 시간 함수, 테이블 함수, 지리공간 함수 등이 있음을 말하는 것으로 충분하

다. 여러 연산자 (+,-,/,*,%,|| 등) 또한 포함되어 있으며 Java를 사용하여 고유 함수를 정의하기 위한 플러그형 인터페이스도 존재한다.

데이터 복제

대다수 전통적인 데이터베이스는 원장 기반 복제를 활용한다. 여기서 리더 노드에 쓰여진 데이터가 추종자 (또는 복제본)에 전파된다. ksqlDB는 (기본 토픽 데이터에 대해) 카프카와 ("대기 복제본"에서 논의한 대기 복제본을 통해 스테이트풀 테이블 데이터에 대해) 카프카 스트림즈 모두로부터 이러한 복제 전략을 상속한다. 상호대화형 모드에서 ksqlDB는 또한 쿼리를 command 토픽이라는 내부 토픽에 저장으로써 statement 기반 복제를 사용하며 이는 단일 ksqlDB 클러스터 내 다중 노드가 동일 쿼리를 처리 및 실행시킬 수 있음을 보장한다.

위에서 볼 수 있듯이 ksqlDB는 전통적인 SQL 데이터베이스의 많은 특성을 갖고 있다. 이제 ksqlDB와 다른 데이터베이스 시스템 간 차이점을 탐구해보자. 이는 무한 루프에서 데이터베이스를 폴링하는 것이 무제한 데이터셋을 최고로 지원하는 ksqlDB와 같은 이벤트 스트리밍 데이터베이스를 사용하는 것과는 같지 않다고 데이터베이스 관리자에 설명할 수 있도록 도울 것이다.

차이점

ksqlDB와 전통적인 SQL 데이터베이스 간 많은 유사성에도 불구하고 중요한 차이점이 존재한다. 이는 다음을 포함한다:

보강된 DDL 및 DML 문

전통적인 데이터베이스에서 지원되는 고전적인 DDL 및 DML 문은 테이블 내 데이터 모델링과 쿼리에 중점을 두고 있다. 그러나 이벤트 스트리밍 데이터베이스인 ksqlDB는 세상에 대해 다른 뷰를 갖고 있다. 이는 "스트림/테이블 이중성"에서 논의했듯이 이를 인식하며 따라서 SQL 통용어가 스트림과 테이블 내 데이터 모델링 및 쿼리를 지원한다. 이는 또한 다른 시스템에서 일반적으로 발견되지 않는 새로운 데이터베이스 객체인 커넥터를 도입하고 있다.

푸시 쿼리

대다수 전통적인 SQL 데이터베이스에서 쿼리 패턴은 데이터의 현재 스냅샷에 대해 쿼리를 발행하고 요청 완료 또는 에러 발생 시 바로 쿼리를 종료하는 것이다. 이렇게 짧고 조회 스타일의 쿼리가 ksqlDB에서 지원되지만 ksqlDB가 무제한의 이벤트 스트림을 운영하기 때문에 새로운 데이터가 수신될 때마다 결과를 방출하며 몇 달 또는 심지어 몇 년 동안 실행될 수 있는 연속적인 쿼리도 지원한다. 이는 외부로를 의미하며 ksqlDB는 데이터 변경을 구독하고자 하는 클라이언트에 대해 보다 우수한 지원을 제공한다.

간단한 쿼리 기능

ksqlDB는 푸시 쿼리를 통해 연속적으로 또는 풀 쿼리를 통해 상호대화식으로 빠르게 유지되는 구체화 뷰를 쿼리하는 고도의 전문화된 데이터베이스이다. 분석 저장소 (예, Elasticsearch), 관계형 시스

템 (예, Postgres, MySQL) 또는 다른 유형의 특수한 데이터 저장소[5]와 동일한 쿼리 기능을 제공하려는 것이 아니다. 쿼리 패턴은 스트리밍 ETL, 구체화 캐시와 이벤트 기반 마이크로 서비스를 포함하여 일련의 특정 유스케이스에 맞춰져 있다.

보다 복합한 스키마 관리 전략

스키마는 SQL 자체를 사용할 때 예상하는 것과 같이 정의될 수 있다. 그러나 별도의 스키마 레지스트리 (컨플루언트 스키마 레지스트리)에 저장될 수 있으며 이는 스키마 진화 지원/호환성 보장, (직렬화된 레코드에서 스키마를 스키마 식별자로 대체함으로써) 데이터 크기 감소, 자동적인 칼럼명/데이터 타입 추론과 (다운스트림 애플리케이션이 ksqlDB에 의해 처리된 데이터를 역직렬화하기 위해 레지스트리에서 레코드 스키마를 검색할 수 있기 때문에) 다른 시스템과의 보다 쉬운 통합 등 장점을 갖고 있다.

ANSI에 영감을 받았지만 완전히 호환되지 않음

스트리밍 SQL을 표준화하려는 시도는 아직까지는 비교적 새로우며 ksqlDB가 사용하는 SQL 통용어는 SQL 표준에 없는 구성을 도입하고 있다.

고가용성, 내고장성 및 장애 조치가 보다 원활하게 동작

일부 시스템의 경우와 같은 별도의 애드온 또는 기업형 기능이 존재하지 않는다. 이들은 ksqlDB DNA 내에 구축되어 있고 고도로 구성가능하다.[6]

로컬 및 원격 저장

ksqlDB가 다루는 데이터는 카프카에 존재하며 테이블을 사용할 때 로컬 상태 저장소에 구체화된다. 이는 몇 가지 흥미로운 특징을 갖고 있다. 예를 들어 동기화/커밋 acking은 카프카 자체에 의해 다루어지며 저장 계층이 SQL 엔진과 독립적으로 확장될 수 있다. 또한 보다 내구성있고 확장가능한 저장을 위해 카프카 고유의 분산 저장 계층을 이용하면서 계산과 데이터 (예, 상태 저장소)를 함께 배치하는 성능상의 장점도 얻는다.

일관성 모델

ksqlDB는 최종적으로 일관성있는 비동기 일관성 모델을 고수한다. 반면 많은 전통적인 시스템들은 원자성 (atomicity), 일관성 (consistency), 격리성 (isolation) 및 내구성 (durability) 모델을 보다 더 준수한다.

이제 ksqlDB와 전통적인 SQL 데이터베이스 간 유사성과 차이점 모두를 살펴보았는데 최종 판결은? 글쎄 ksqlDB가 보다 전통적인 데이터베이스의 많은 특징을 갖고 있지만 이를 대체하려는 목표는 없다.

---

[5] 더욱 자세한 정보는 Jay Kreps의 컨플루언트의 "ksqlDB 소개"를 참조하기 바란다.

[6] 예를 들어 카프카 스트림즈의 대기 복제 구성을 통해 상시 대기(hot standby가 지원된다.

이는 스트리밍 유스케이스에 사용될 수 있는 고도의 전문 툴이며 다른 시스템의 기능이 필요할 때는 ksqlDB의 카프카 커넥트 통합을 통해 보강된, 변형된 또는 다르게 처리된 데이터를 선택한 데이터 저장소에 이동시킬 수 있다.

ksqlDB 설치 전에 아키텍처를 개략적으로 살펴보자.

아키텍처

ksqlDB는 카프카 스트림즈 위에 구축되어 있기 때문에 카프카 스트림즈 동작 방법에 대한 저수준 이해를 위한 2 장의 스트리밍 아키텍처 논의를 검토할 수 있다. 이 절에서는 아키텍처의 ksqlDB 특정적인 컴포넌트인 ksqlDB 서버와 클라이언트에 중점을 둔다.

ksqlDB 서버

ksqlDB 서버는 스트림 처리 애플리케이션 구동을 담당한다 (ksqlDB 의미에서 비즈니스 문제 해결을 위해 함께 동작하는 일련의 쿼리일 것이다). 각 서버는 카프카 스트림즈 애플리케이션의 단일 인스턴스와 유사하며 (쿼리 셋에 의해 생성된) 워크로드는 동일한 ksql.service.id 구성을 갖는 다중 ksqlDB 서버로 분산될 수 있다. 카프카 스트림즈 애플리케이션과 같이 ksqlDB 서버는 카프카 클러스터와 별도로 배치된다 (보통 브로커 자체와 다른 별도의 머신/컨테이너 상).

협업하는 ksqlDB 서버 그룹을 ksqlDB 서버 클러스터로 부르며 일반적으로 클러스터 수준에서 단일 애플리케이션에 대한 워크로드를 격리시키는 것이 권고된다. 예를 들어 그림 8-5는 2 개의 ksqlSB 클러스터 각각이 다른 서비스 ID로 격리된 워크로드를 실행하고 있음을 보여준다. 클러스터는 서로에 상관없이 확장 및 관리될 수 있다.

ksqlDB 클러스터에 용량 추가가 필요한 경우 더욱 많은 ksqlDB 서버를 배치할 수 있다. 또한 언제라도 ksqlDB 서버를 제거함으로써 클러스터를 축소할 수도 있다. 동일한 서비스 ID를 갖는 ksqlDB 서버는 동일한 컨슈머 그룹 멤버이기 때문에 새로운 ksqlDB 서버가 추가 또는 삭제될 때 카프카는 자동적으로 작업을 재할당/분배한다 (삭제는 수동 또는 자동 (예, 시스템 고장)일 수 있다).

각 ksqlDB 서버는 SQL 엔진과 REST 서비스 두 컴포넌트로 구성된다. 다음 절에서 이들 별도의 컴포넌트를 각각 논의할 것이다.

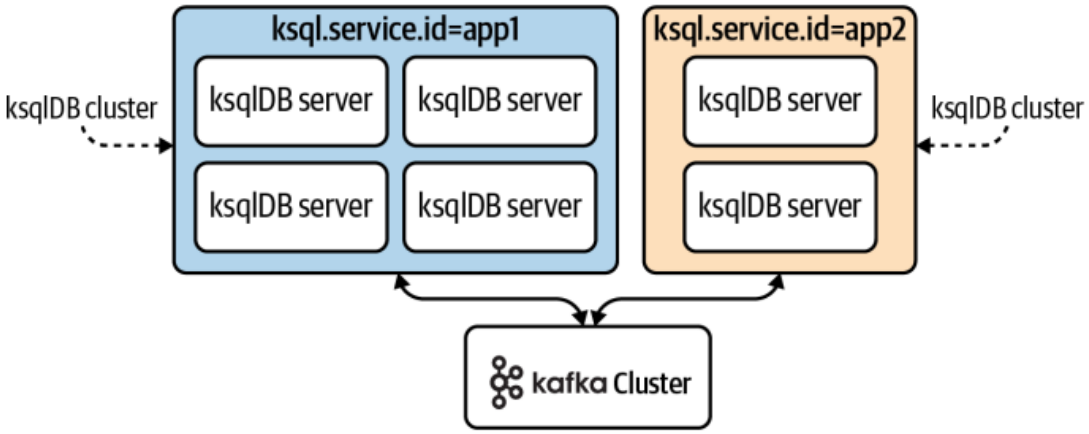

그림 8-5. 카프카와 독립적으로 데이터를 처리하는 2 개의 ksqlDB 클러스터

SQL 엔진

SQL 엔진은 SQL 문 파싱, 이를 하나 이상의 카프카 스트림즈 토폴로지로 변환 그리고 궁극적으로 카프카 스트림즈 애플리케이션 실행을 담당한다. 그림 8-6은 이 프로세스를 시각화한 것이다.

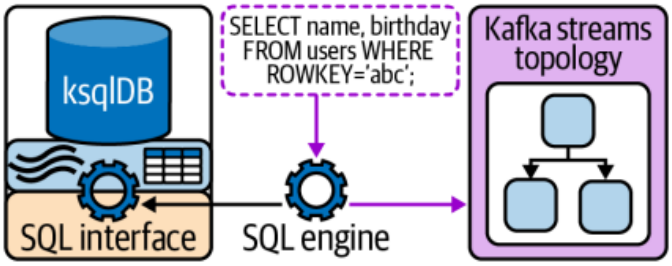

그림 8-6. SQL 엔진은 SQL 문을 카프카 스트림즈 토폴로지로 변환한다.

파서 자체는 ANTLR이라는 툴을 사용하는데 이는 SQL 문을 추상 구문 트리 (Abstract Syntax Tree, AST)로 변환하며 트리 내 각 노드는 입력 쿼리에서 인지된 문자 또는 토큰을 표현한다. ksqlDB는 파스 트리의 각 노드를 방문하고 발견한 토큰을 사용하여 카프카 스트림즈 토폴로지를 구축한다. 예를 들어 쿼리에 WHERE 절을 포함한 경우 ksqlDB는 쿼리에 대해 기본 카프카 스트림즈 토폴로지를 구축할 때 스테이트리스 filter 연산자를 사용해야 함을 알고 있다. 비슷하게 쿼리에 조인 조건 (예, LEFT JOIN)이 포함되면 ksqlDB는 토폴로지에 leftJoin 연산자를 추가할 것이다. 소스 프로세서는 FROM 값으로부터 결정되고 투영에는 SELECT 문이 사용된다.

엔진이 쿼리를 실행하는데 필요한 필수 프로세서 토폴로지를 생성했다면 실제로 카프카 스트림즈 애플리케이션을 실행시킬 것이다. 이제 쿼리를 SQL 엔진에 넘겨주기 위해 필요한 컴포넌트를 살펴보자: REST 서비스.

REST 서비스

ksqlDB는 클라이언트가 SQL 엔진과 상호작용할 수 있도록 하는 REST 인터페이스를 포함하고 있다. 이는 엔진에 쿼리를 제출하고 (예, SELECT 문으로 시작하는 DML 문), 다른 유형의 문을 실행하며 (예,

DDL 문), 클러스터 상태/건전성을 확인하는 등을 위해 ksqlDB CLI, ksqlDB UI와 다른 클라이언트에 의해 주로 사용된다. 기본적으로 8088 포트에서 수신 대기하고 HTTP를 통해 통신하지만 listeners 구성을 사용하여 엔드포인트를 변경할 수 있고 ssl 구성을 사용하여 HTTPS를 통한 통신을 활성화할 수 있다. 두 구성 집합은 다음과 같다:

```
listeners=http://0.0.0.0:8088
ssl.keystore.location=/path/to/ksql.server.keystore.jks
ssl.keystore.password=...
ssl.key.password=...
```

REST API는 옵션으로 동작 모드에 따라 전적으로 비활성화될 수도 있다. 그러나 ksqlDB와 상호대화식으로 작업할 때는 기본 클라이언트 (예, 다음 절에서 논의할 ksqlDB CLI와 UI) 또는 맞춤형 클라이언트 중 하나를 사용한 API 동작이 필요하다. 다음과 같이 curl을 사용하여 요청을 발행할 수도 있다.

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

API와 직접 상호작용할 때 ksqlDB 문서에서 가장 최신 REST API를 참조해야 한다. ksqlDB는 아직도 빠르게 진화하고 있으며 ksqlDB 프로젝트에 제출되었던 설계 제안에 기반하여 앞으로 변화가 예상되는 분야이다. 이 책에서 대다수 예제는 다음에 논의할 공식 지원 클라이언트를 통해 간접적으로 API를 사용할 것이다.

ksqlDB 클라이언트

이전 절에서 ksqlDB 서버가 쿼리 제출과 ksqlDB 클러스터에 대한 정보 검색을 위해 REST 인터페이스를 포함하고 있음을 배웠다. 또한 curl 또는 맞춤형 클라이언트를 사용하여 REST 서비스와 상호작용할 수 있음을 배웠지만 대부분의 경우 ksqlDB 서버와의 상호작용을 위해 공식 클라이언트 중 하나를 사용하고 싶을 것이다. 이 절에서 ksqlDB CLI로 시작하여 이들 클라이언트에 대해 배울 것이다.

ksqlDB CLI

ksqlDB CLI는 실행 중인 ksqlDB 서버와 상호작용할 수 있도록 하는 커맨드 라인 애플리케이션이다. 이는 쿼리 제출, 토픽 검사, ksqlDB 구성 변경 등을 상호대화식으로 할 수 있도록 하기 때문에 ksqlDB로 실험 시 훌륭한 툴이다. 이는 도커 이미지 (confluentinc/ksqldb-cli) 또는 컨플루언트 플랫폼 (컨플루언트 클라우드 상의 완전 관리형 또는 자기 관리형 배치를 통해)의 일부로 배포된다.

CLI 호출은 ksql 명령 실행과 ksqlDB 서버의 호스트/포트 조합 지정을 포함한다 (이는 REST 서비스에서 논의했던 listerners에 해당한다).

```
ksql http://localhost:8088
```

Ksql 명령 실행 시 다음과 비슷한 프롬프트가 표시될 것이다:

```
===========================================
=       _              _ ____  ____       =
=      | | _____  __ _| |  _ \| __ )      =
=      | |/ / __|/ _` | | | | |  _ \      =
=      |   <\__ \ (_| | | |_| | |_) |     =
=      |_|\_\___/\__, |_|____/|____/      =
=                   |_|                   =
=  Event Streaming Database purpose-built =
=        for stream processing apps       =
===========================================

Copyright 2017-2020 Confluent Inc.

CLI v0.14.0, Server v0.14.0 located at http://ksqldb-server:8088
Server Status: RUNNING

Having trouble? Type 'help' (case-insensitive) for a rundown of how things work!

ksql>
```

이 책에서는 이전 프롬프트가 SQL 통용어를 탐구하고 튜토리얼을 진행할 때 시작점일 것이다. CLI 사용을 시작하기 전에 다른 클라이언트 ksqlDB UI를 살펴보자.

ksqlDB UI

컨플루언트 플랫폼은 ksqlDB와 상호작용하기 위해 UI를 포함하고 있다. UI는 상용 기능이며 따라서 컨플루언트 플랫폼과 (완전 관리 클라우드 환경에서 컨플루언트 플랫폼을 실행하는) 컨플루언트 클라우드의 상업용 라이선스 버전에서는 볼 수 있다. 웹 기반 편집기를 통해 쿼리를 제출할 수 있을 뿐만 아니라 데이터 흐름 시각화, 웹 폼을 사용한 스트림과 테이블 생성, 실행 중인 쿼리 목록 보기 등을 할 수 있다. 그림 8-7은 ksqlDB UI 스크린샷을 보여준다.

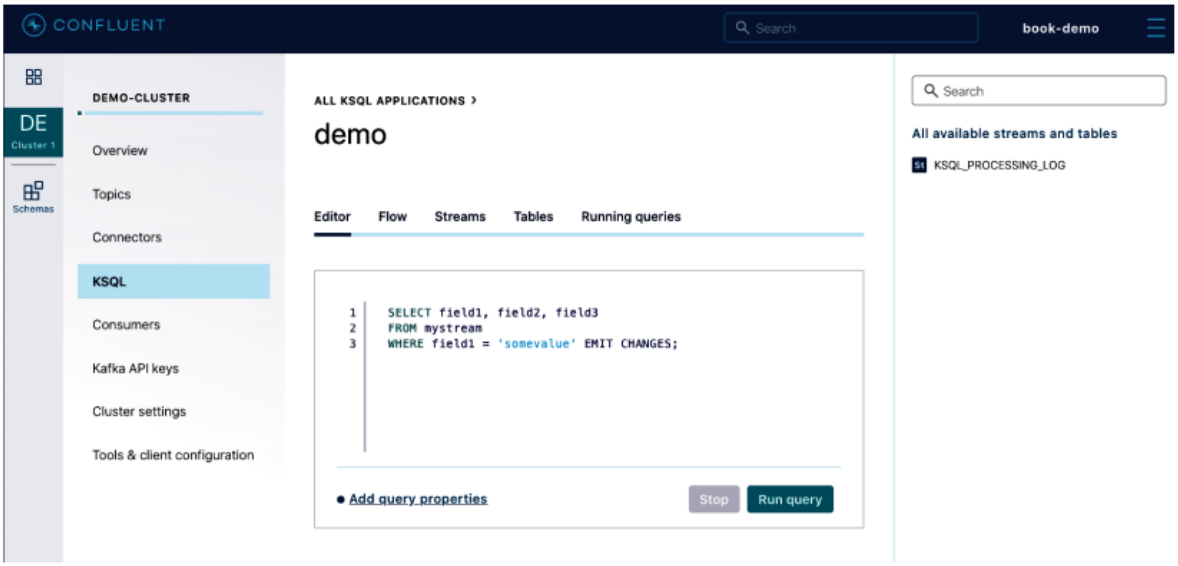

그림 8-7. 컨플루언트 클라우드에서 보여지는 ksqlDB UI

베치 모드

ksqlDB는 실행 중인 ksqlDB 서버와 허용할 상호작용 수준에 따라 2 가지 다른 배치 모드를 지원한다. 이 절에서는 이 배치 모드 각각을 설명하고 언제 사용할 수 있는지를 논의할 것이다.

상호대화형 모드

ksqlDB를 상호대화형 모드로 실행할 때 클라이언트는 REST API를 사용하여 언제라도 새로운 쿼리를 제출할 수 있다. 이름에서 알 수 있듯이 이는 상호대화형 경험으로 이끌며 ksqlDB가 마음대로 스트림, 테이블, 쿼리와 커넥터를 생성 및 파괴할 수 있도록 한다.

그림 8-8은 상호대화형 모드로 동wkr 중인 kslqDB에 대한 설명을 보여준다. 상호대화형 모드의 한 가지 주요 특징은 (REST API를 통해) SQL 엔진에 제출된 모든 쿼리가 command 토픽이라는 내부 토픽에 써진다는 것이다. 이 토픽은 ksqlDB에 의해 자동생성 및 관리되며 스테이트풀 복제를 달성하기 위해 사용되고 클러스터 내 모든 ksqlDB가 쿼리를 실행 및 동작시킬 수 있도록 한다.

상호대화형 모드는 ksqlDB의 기본 배치 모드로 이 모드로 동작시키기 위해 특별한 구성을 필요치 한다. 그러나 상호대화형 모드를 비활성화하고 싶다면 특별한 구성이 필요한 헤드리스 모드에서 동작시켜야 한다. 다음 절에서 헤드리스 모드를 논의할 것이다.

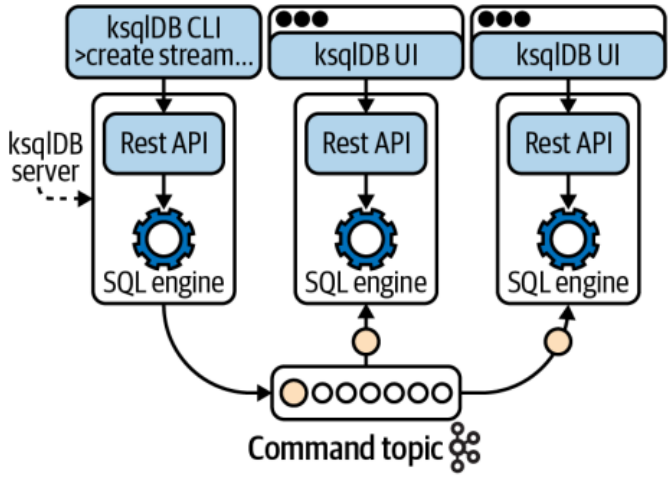

그림 8-8. 상호대화식 모드에서 동작 중인 ksqlDB. REST API를 통해 클라이언트 (예, ksqlDB CLI, UI, Java 클라이언트 또는 curl)가 쿼리를 제출할 수 있다.

헤드리스 모드

어떤 경우 ksqlDB 클러스터에 대해 클라이언트가 상호대화식으로 쿼리를 제출하는 것을 원하지 않을 수도 있다. 예를 들어 운영 중인 배치를 잠그고 싶다면 실행 중인 쿼리를 변경하지 못함을 보장하기 위해 이를 (REST API가 비활성화된) 헤드리스 모드로 실행할 수 있다. 헤드리스 모드로 동작시키기 위해서는 SQL 엔진이 실행할 영구 쿼리를 포함한 파일을 생성하고 queries.file ksqlDB 서버 구성을 사용하여 이 파일에 대한 경로를 지정하면 된다. 예를 들어

```
queries.file=/path/to/query.sql ①
```

① ksqlDB 서비에서 실행시키려는 쿼리를 포함한 파일 경로가 queries.file 특성을 사용해 지정된다.

그림 8-9는 헤드리스 모드로 동작 중인 ksqlDB에 대한 설명을 보여준다. 상호대화형 배치와는 달리 헤드리스 모드는 스테이트풀 복제를 위해 command 토픽을 활용하지 않음을 주목하기 바란다. 그러나 config 토픽에 일부 내부 메타 데이터를 쓴다.

이 책의 튜토리얼 대부분에 대해 상호대화형 모드로 실행시킬 것이다. 마지막으로 최초의 헬로우 월드 스타일 튜토리얼로 발을 적셔보자.

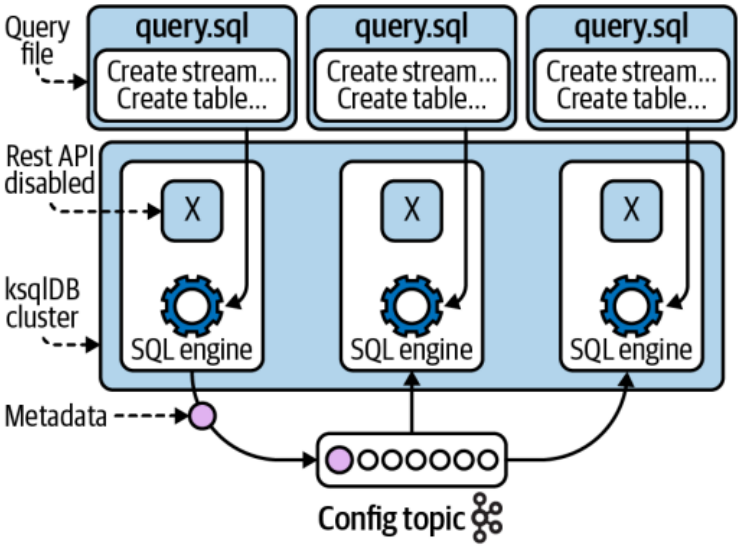

그림 8-9. 헤드리스 모드에서 동작 중인 ksqlDB

튜토리얼

이 절에서는 간단한 헬로우 월드 튜토리얼을 진행할 것이다. 이는 ksqlDB를 통해 users 토픽의 각사용자에게 헬로우라고 말하는 간단한 스트림 처리 애플리케이션 구축을 포함할 것이다. ksqlDB 설치 방법을 배우면서 시작해보자.

ksqlDB 설치하기

ksqlDB로 작업을 시작하기 위해 몇몇 다른 방법이 있다. 다음 표는 가장 인기있는 옵션을 나타낸다.

설치 방법	링크	비고
컨플루언트 플랫폼 다운로드	https://www.confluent.io/download	컨플루언트 플랫폼의 자기 관리형 버전에서 커뮤니티 라이센스를 갖는 소프트웨어 컴포넌트
컨플루언트 클라우드 사용	https://confluent.cloud/	다운로드 불필요. 계정 생성만 필요.
공식 도커 이미지 다운로드 및 실행	ksqlDB 서버와 CLI에 대한 이미지 https://hub.docker.com/r/confluentinc/ksqldb-server https://hub.docker.com/r/confluentinc/ksqldb-cli	모든 의존성과 관련 소프트웨어를 별도로 동작해야 한다 (카프카, 스키마 레지스트리, ksqlDB CLI)
Github 오픈소스 저장소 복제 및 소스로부터 빌드	https://github.com/confluentinc/ksql	가장 복잡한 옵션

표 내 가장 쉬운 방법은 공식 도커 이미지를 사용하는 것이다. 이 책의 코드 저장소는 적절한 도커 이미지를 사용하여 관련 서비스 각각을 구동하는 도커 컴포즈 배치를 포함하고 있다. Github의 전체 지침을 참조하기 바란다.

이 장에서 코드의 나머지는 ksqlDB 서버와 CLI를 실행하고 카프카 토픽을 미리 생성하기 위해 필요한 원시 명령을 참조할 것이다. 해당되는 경우 도커 컴포즈 기반 워크 플로우에 더욱 특정적이기 때문에 별도로 주석처리되어 있지만 도커 컴포즈에서 이 명령을 실행하는 방법에 대한 지침을 추가하였다.

ksqlDB가 설치되었고 ksqlDB 서버를 실행시킬 시간이다. 다음 절에서 실행 방법을 배울 것이다.

ksqlDB 서버 실행하기

ksqlDB를 설치했다면 ksqlDB 서버에 대한 구성을 생성해야 한다. 마음대로 사용할 수 있는 여러 구성이 있지만 지금은 ksql-server.properties 파일에 2 가지 가장 중요한 구성을 단지 저장해보자:

```
listeners=http://0.0.0.0:8088 ①
bootstrap.servers=kafka:9092 ②
```

① ksqlDB 서버의 REST API 엔드포인트. 이는 모든 IPv4 인터페이스에 바인딩될 것이다.

② 하나 이상의 카프카 브로커에 해당하는 호스트와 포트 쌍 목록. 카프카 클러스터와의 연결 확립을 위해 사용될 것이다.

ksqlDB 서버 구성을 파일로 저장한 후 다음 명령을 사용하여 ksqlDB 서버를 시작할 수 있다.

```
ksql-server-start ksql-server.properties ①
```

① 도커 컴포즈 워크 플로우에서 이 명령은 docker-compose.yml 파일 내에 설정되어 있다. 더욱 자세한 정보는 이 장의 코드 저장소를 참조하기 바란다.

이전 명령을 실행한 후 ksqlDB가 부팅됨에 따라 많은 정보가 컨솔로 출력되는 것을 봐야 한다. 출력되는 라인 내에서 다음과 유사한 무언가를 봐야 한다:

```
[2020-11-28 00:53:11,530] INFO ksqlDB API server listening on http://0.0.0.0:8088
```

"배치 모드"의 논의에서 상기할 수 있듯이 기본 배치 모드는 상호대화형 모드로 ksql-server.properties 파일 내 queries.file 을 설정하지 않았기 때문에 REST 서비스가 ksqlDB 서버 내부에서 동작 중이며 이제 ksqlDB 서버에 쿼리를 제출하기 위해 기본 클라이언트 중 하나를 사용할 수 있다. 이 튜토리얼에서는 다음에 볼 ksqlDB CLI를 통해 쿼리를 제출할 것이다.

토픽 미리 생성하기

다음 장에서는 SQL 문의 특정 파라미터 설정을 통해 ksqlDB가 토픽을 자동 생성하는 법을 배울 것이다. 이 튜토리얼에서는 다음 명령을 실행함으로써 users 토픽을 미리 생성할 것이다.

```
kafka-topics ₩ ①
 --bootstrap-server localhost:9092 ₩
 --topic users ₩
 --replication-factor 1 ₩
 --partitions 4 ₩
```

--create

① 도커 컴포즈에서 실행한다면 docker-compose exec kafka를 명령 앞에 추가한다.

이제 ksqlDB CLI를 사용해보자.

ksqlDB CLI 사용하기

ksqlDB가 실행할 일련의 쿼리를 생성함으로써 헬로우 월드 스트리밍 처리 애플리케이션을 구축할 시간이다. 쿼리 제출을 위해 CLI를 사용할 것이기 때문에 첫번째 할 일은 다음과 같이 ksqlDB 서버의 엔드포인트를 이용해 ksql 명령을 실행하는 것이다.

ksql http://0.0.0.0:8088 ①

① 도커 컴포즈를 실행한다면 docker-compose exec ksqldb-cli ksql http://ksqldb-server:8088을 실행한다.

이로써 CLI로 떨어지며 여기에서 다양한 쿼리와 문을 실행시킬 수 있다. 또한 CLI에서 다양한 ksqlDB 구성을 조정할 수도 있다. 예를 들어 쿼리가 기본 카프카 토픽의 시작부터 읽는 것을 보장하기 위해 다음 SET 문을 실행한다.

SET 'auto.offset.rest' = 'earliest';

Users 토픽을 미리 생성했기 때문에 이의 존재와 토픽의 기본 구성 (예, 파티션과 복제본 카운트) 중 일부를 보기 위해 SHOW TOPICS 명령을 사용할 수 있다.

다음 코드 블록은 이 명령을 보여준다:

```
ksql> SHOW TOPICS ;

 Kafka Topic | Partitions | Partition Replicas
---------------------------------------------
 users       | 4          | 1
---------------------------------------------
```

이제 ksqlDB의 CREATE STREAM DDL 문을 사용하여 users 토픽 내 데이터를 스트림으로 모델링해보자. 다른 무엇보다 이는 ksqlDB에 데이터 타입과 이 토픽에서 발견할 수 있을 것으로 예상되는 포맷에 대한 정보를 제공하며 쿼리할 수 있는 스트림을 생성한다.

```
CREATE STREAM users (
 ROWKEY INT KEY, ①
 USERNAME VARCHAR ②
) WITH ( ③
 KAFKA_TOPIC='users', ③
 VALUE_FORMAT='JSON' ⑤
);
```

① ROWKEY는 카프카의 레코드 키에 해당한다. 여기서는 타입을 INT로 지정한다.

② 여기서 users 토픽 내 레코드가 USERNAME 필드를 가질 것임을 지정한다. 데이터 타입은 VARCHAR이다.

③ WITH 절은 추가적인 특성을 전달하기 위해 사용된다. 추후 논의할 몇몇 추가적인 특성이 존재한다.

④ 스트림이 WITH 절 내 KAFKA-TOPIC 특성에 지정한 users 토픽에서 읽는다.

⑤ 카프카 내 레코드 값의 직렬화 포맷을 지정한다.

CREAT SREAM 문을 실행한다면 다음과 같이 컨솔에서 스트림이 생성되었다는 확인을 볼 수 있다.

```
 Message
----------------
 Stream created
----------------
```

Users 스트림에 쿼리하기 전에 ksqlDB의 INSERT INTO 문을 사용해 스트림에 일부 테스트 데이터를 삽입해보자.

```
INSERT INTO users (username) VALUES ('izzy');
INSERT INTO users (username) VALUES ('elyse');
INSERT INTO users (username) VALUES ('mitch');
```

이제 스트림을 생성하였고 테스트 데이터로 채웠기 때문에 users 스트림에 나타나는 모든 사용자에 대해 인사말을 생성할 푸시 쿼리를 생성할 수 있다. 연속적이고 일시적인[7] (영구적이 아님) 푸시 쿼리를 생성하기 위해 예제 8-1의 문을 실행한다.

예제 8-1. 헬로우 월드 스타일 푸시 쿼리

```
SELECT 'Hello, ' + USERNAME AS GREETING
FROM users
EMIT CHANGES; ①
```

① EMIT CHANGES를 포함함으로써 새로운 데이터가 사용가능할 때마다 변경을 자동적으로 클라이언트에 방출/푸시할 푸시 쿼리를 실행하라고 ksqlDB에 말하는 것이다.

Users 스트림에 일부 테스트 데이터를 삽입했기 때문에 컨솔에 다음 출력이 즉시 출력되는 것을 봐야 한다. 왜냐하면 auto.offset.rest을 earliest로 설정했고 토픽에 이미 사용가능한 데이터가 있기 때문이다.

---

[7] 일시적이라는 것은 이 쿼리 결과가 카프카에 다시 써지는 것이 아님을 의미한다. 다음 장에서 영구 쿼리를 탐구할 것이다.

```
+--------------------+
|GREETING            |
+--------------------+
|Hello, izzy         |
|Hello, elyse        |
|Hello, mitch        |
```

SELECT 쿼리가 최초 결과 셋이 방출된 후에도 계속 실행될 것임을 주목하기 바란다. 이전에 언급했듯이 이는 푸시 쿼리의 특성으로 쿼리를 종료할 때까지 (또는 쿼리에 LIMIT 절을 포함한다면 레코드 제한에 도달할 때까지) 계속 실행되어 결과를 방출할 것이다.

또한 다른 CLI 세션을 열어 users 스트림에 더 많은 데이터를 생성하기 위해 추가적인 INSERT INTO 문을 실행시킬 수도 있다. 새로운 레코드가 삽입되자 마자 푸시 쿼리 (이전에 발행된 SELECT 문으로 표현되는)가 컨솔에 계속 결과를 방출할 것이다.

이제 이 장의 간단한 튜토리얼을 마쳤으며 이는 ksqlDB가 할 수 있는 것의 빙산의 일각만을 보여준다. 다음 몇 장에 걸쳐 보다 세부적으로 쿼리 언어를 탐구하고 ksqlDB와 익숙해지도록 흥미로운 튜토리얼을 진행할 것이다.

요약

ksqlDB의 역사를 처음으로 살펴보았고 왜 구축되었는지와 언제 사용되는지를 배웠으며 전통적인 데이터베이스와 어떻게 비교되는지를 살펴보았고 아키텍처를 엿보았다. 다음 장에서는 이를 외부 데이터 소스와 통합하는 방법을 탐구함으로써 ksqlDB에 대해 배우는 여행을 계속할 것이다.