6장 고급 상태 관리

지난 두 장에서는 카프카 스트림즈의 스테이트풀 처리를 논의하였다. 집계, 조인 및 윈도우 연산 수행 방법을 배웠기 때문에 스테이트풀 처리를 시작하는 것이 꽤 쉽다는 것이 명백해졌다.

그러나 이전에 암시했듯이 상태 저장소는 추가적인 운영 복잡성을 야기한다. 애플리케이션을 확장하고 고장을 경험하며 일상적인 유지보수를 함에 따라 애플리케이션이 시간이 지나도 계속 동작하는지를 보장하기 위해 스테이트풀 처리는 기본 역학의 이해가 필요로 함을 배울 것이다.

이 장의 목표는 스테이트풀 스트림 처리 애플리케이션을 구축할 때 보다 높은 수준의 신뢰성을 확보할 수 있도록 상태 저장소를 보다 깊게 살펴보는 것이다. 이 장의 대부분은 작업이 컨슈머 그룹에 대해 재분배될 필요가 있을 때 발생하는 리밸런싱이라는 주제에 할당되어 있다. 리밸런싱은 스테이트풀 애플리케이션에 특히 큰 영향을 미칠 수 있으며 따라서 여러분의 애플리케이션에서 이를 다룰 준비가 되어 있게 이를 이해하도록 할 것이다.

우리가 답할 일부 질문은 다음과 같다:

- 영구 상태 저장소가 디스크에서 어떻게 표현되는가?
- 스테이트풀 애플리케이션이 어떻게 내고장성을 달성하는가?
- 내장 상태 저장소를 어떻게 설정할 수 있는가?
- 어떤 종류의 이벤트가 스테이트풀 애플리케이션에 가장 영향을 미치는가?
- 상태 저장소가 무한정 늘어나지 않음을 어떻게 보장하는가?
- 다운스트림 업데이트의 속도를 제한하기 위해 DSL 캐시가 어떻게 사용될 수 있는가?
- 상태 저장소 리스너를 사용하여 상태 복구 과정을 어떻게 추적하는가?
- 리밸런스를 탐지하기 위해 상태 리스너가 어떻게 사용될 수 있는가?

영구 상태 저장소의 디스크 구조를 살펴봄으로써 시작해보자.

영구 상태 저장소 구조

카프카 스트림즈는 인메모리와 영구 상태 저장소 모두를 포함한다. 후자 유형의 상태 저장소는 상태가 재초기화될 필요가 있을 때 애플리케이션의 복구 시간을 줄일 수 있기 때문에 일반적으로 선호된다 (예, 고장 또는 태스크 이관의 경우).

기본적으로 영구 상태 저장소는 /tmp/kafka-streams 디렉토리 내에 존재한다. StreamsConfig.STATE_DIR_CONFIG 특성 설정을 통해 이를 재정의할 수 있으며 /tmp 디렉토리의 일시적인 특성을 고려한다면 (이 디렉토리 내용은 시스템 재부팅/충돌 시 삭제된다) 애플리케이션 상태 저장을 위해 다른 위치를 선택해야 한다.

영구 상태 저장소가 디스크에 존재하기 때문에 매우 쉽게 파일들을 조사할 수 있다[1]. 이를 통해 디렉토리와 파일 이름 자체만으로도 놀랄 만한 양의 정보를 얻을 수 있다. 예제 6-1의 파일 트리는 이전 장에서 만들었던 환자 모니터링 애플리케이션에서 생성된 것으로 주석은 중요 디렉토리 및 파일에 대해 추가적인 세부 정보를 제공한다.

예제 6-1. 디스크 상 영구 상태 저장소 구조 예

```
.
└── dev-consumer ❶
    ├── 0_0
    │   ├── .lock
    │   └── pulse-counts
    ├── 0_1
    │   ├── .lock
    │   └── pulse-counts
    ├── 0_2
    │   ├── .lock
    │   └── pulse-counts
    ├── 0_3 ❷
    │   ├── .checkpoint ❸
    │   ├── .lock ❹
    │   └── pulse-counts ❺
    │       └── ...
    ├── 1_0
    │   ├── ...
```

① 최상위 수준 디렉토리는 애플리케이션 ID를 포함한다. 이는 워크로드가 많은 수의 노드에서 스케줄링될 수 있을 때 (예, 쿠버네티스 클러스터) 특히 공유 환경에서 서버 상에 어떤 애플리케이션이 동작 중임을 이해하는데 도움이 된다.

② 2번째 수준 디렉토리 각각은 하나의 카프카 스트림즈 태스크에 해당한다. 디렉토리 이름은 태스크 ID 형태로 태스크 ID는 두 부분으로 구성된다: <서브토폴로지-ID>_<파티션>. "서브 토폴로지"에서 논의했듯이 서브 토폴로지는 프로그램 로직에 따라 하나 이상의 토픽으로부터 데이터를 처리할 수 있음을 주목하기 바란다.

③ 체크포인트 파일은 changelog 토픽의 오프셋을 저장한다 ("체인지로그 토픽" 참조). 이들은 로컬 상태 저장소 내에 어떤 데이터가 읽혀 졌는지를 카프카 스트림즈에 알려주며 뒤에 볼 것이지만 상태 저장소 복구에 매우 중요한 역할을 한다.

④ 잠금 파일은 상태 디렉토리에 대한 잠금을 얻기 위해 카프카 스트림즈가 사용한다. 이는 동시성 문제를 예방하는데 도움이 된다.

⑤ 실제 데이터는 이름이 부여된 상태 디렉토리에 저장된다. 여기서 pulse-counts는 상태 저장소를

[1] 비고: 파일을 절대로 수정하면 안된다.

구체화할 때 설정했던 명시적 이름에 해당한다.

상태 저장소가 디스크 상에서 어떤 구조를 가지는 지를 앎으로써 이들의 동작 방법에 대한 신비를 제거할 수 있다. 더구나 잠금 파일과 체크포인트 파일은 특히 중요한데 특정 에러 로그에서 참조되며 (예를 들어 체크포인트 파일에 대한 쓰기 실패로 권한 문제가 나올 수 있으며 동시성 문제로 카프카 스트림즈가 잠금을 얻을 수 없다는 에러를 야기할 수 있다) 따라서 이들의 위치와 유용성을 이해하는 것은 많은 도움이 된다.

체크포인트 파일은 상태 저장소 복구에 중요한 역할을 한다. 우선 스테이트풀 애플리케이션의 내고장성 특징을 살펴봄으로써 이를 좀 더 살펴보고 오프셋 체크포인트가 복구 시간을 줄이는데 어떻게 사용되는지를 살펴보자.

내고장성

카프카 스트림즈의 내고장성 특성은 카프카의 저장 계층과 그룹 관리 프로토콜에 기인한다. 예를 들어 파티션 수준에서 데이터 복제는 브로커가 오프라인이 되더라도 다른 브로커 상의 복제된 파티션으로부터 데이터가 소비될 수 있음을 의미한다. 더구나 컨슈머 그룹을 사용함으로써 애플리케이션의 단일 인스턴스가 다운되더라도 정상 인스턴스들 중 하나로 작업이 재분배될 수 있다.

그러나 스테이트풀 애플리케이션의 경우 카프카 스트림즈는 애플리케이션이 고장에 탄력적임을 보장하기 위해 추가 조치를 취한다. 이는 상태 저장소 지원을 위한 체인지로그 토픽과 상태 손실 시 재초기화 시간 최소화를 위한 대기 복제본 사용을 포함한다. 다음 절에서 카프카 스트림즈 특정적인 내고장성 특징을 보다 세부적으로 논의할 것이다.

체인지로그 토픽

명시적으로 비활성화되어 있지 않다면 상태 저장소는 카프카 스트림즈에 의해 생성 및 관리되는 체인지로그 토픽에 의해 지원을 받는다. 이 토픽은 저장소 내 모든 키에 대한 상태 업데이트를 수집하며 애플리케이션 상태 재구축을 위해 고장 시 재생될 수 있다[2]. 전체 상태 손실의 경우 (또는 새로운 인스턴스를 생성할 때) 체인지로그 토픽은 처음부터 재생된다. 그러나 (예제 6-1) 체크포인트 파일이 존재한다면 이 파일에서 발견되는 체크포인팅된 오프셋으로부터 상태가 재생될 수도 있다. 왜냐하면 이 오프셋이 상태 저장소 내에 이미 어떤 데이터가 읽어졌는지를 나타내기 때문이다. 단지 상태의 일부만을 복구하는 것이 전체 상태를 복구하는 것보다 시간이 덜 소요되기 때문에 후자가 보다 빠르다.

체인지로그 토픽은 DSL 내 Materialized 클래스를 사용하여 설정가능하다. 예를 들어 이전 절에서 다음 코드를 사용하여 pulse-counts 라는 상태 저장소를 구체화하였다.

```
pulseEvents
```

[2] 상태 저장소가 재초기화될 필요가 있을 때 체인지 로그 토픽을 재생하기 위해 복구 컨슈머라는 전용 컨슈머가 사용된다.

```
        .groupByKey()
        .windowedBy(tumblingWindow)
        .count(Materialized.as("pulse-counts"));
```

Materialized 클래스에는 체인지로그 토픽을 더욱 맞춤화할 수 있도록 하는 추가적인 메소드가 존재한다. 예를 들어 체인지 로깅을 완전히 비활성화하기 위한 일시 저장소라는 것을 생성하기 위해 다음 코드를 사용할 수 있다 (고장 시 복구될 수 없는 상태 저장소):

```
    Materialized.as("pulse-counts").withLoggingDisabled();
```

체인지 로깅을 비활성화하는 것은 상태 저장소가 더 이상 내고장적이지 않음을 의미하기 때문에 좋은 생각은 아니며 대기 복제를 사용할 수 없도록 한다. 체인지로그 토픽 설정의 경우 withRetention 메소드 (추후 "윈도우 유지"에서 다룬다)를 사용하여 윈도우 또는 세션 저장소의 유지를 재정의하거나 또는 체인지로그 토픽에 대해 특정 토픽 설정을 전달할 것이다. 예를 들어 insync 복제본의 수를 2 개로 늘리려면 다음 코드를 사용할 수 있다:

```
Map<String, String> topicConfigs =
  Collections.singletonMap("min.insync.replicas", "2"); ①
KTable<Windowed<String>, Long> pulseCounts =
  pulseEvents
  .groupByKey()
  .windowedBy(tumblingWindow)
  .count(
    Materialized.<String, Long, WindowStore<Bytes, byte[]>>
      as("pulse-counts")
      .withValueSerde(Serdes.Long())
      .withLoggingEnabled(topicConfigs)); ②
```

① 토픽 설정을 저장하기 위해 맵을 생성한다. 엔트리는 어떤 유효한 토픽 설정 및 값도 포함할 수 있다.

② 토픽 설정을 Materialized.withLoggingEnable 메소드에 전달함으로써 체인지로그 토픽을 설정한다.

이제 토픽을 자세히 살펴보면(--describe) 토픽이 이에 따라 설정되었음을 볼 수 있다:

```
$ kafka-topics ₩
    --bootstrap-server localhost:9092 ₩
    --topic dev-consumer-pulse-counts-changelog ₩ ①
    --describe
# output
Topic: dev-consumer-pulse-counts-changelog
PartitionCount: 4
ReplicationFactor:1
```

```
Configs: min.insync.replicas=2
...
```

① 체인지로그 토픽이 다음 네이밍 스킴을 가짐을 주목하기 바란다:

<application_id>-<internal_store_name>-changelog.

한 가지 주목할 사실은 이 책 작성 시점에 체인지로그 생성 이후 이 메소드를 사용하여 체인지로그 토픽을 재구성할 수 없다는 것이다[3]. 기존 체인지로그 토픽에 대해 토픽 설정을 업데이트해야 한다면 카프카 컨솔 스크립트를 사용하여 작업을 해야 한다. 체인지로그 토픽 생성 후 이를 수동으로 업데이트하는 방법 예는 다음과 같다:

```
$ kafka-configs ₩ ①
    --bootstrap-server localhost:9092 ₩
    --entity-type topics ₩
    --entity-name dev-consumer-pulse-counts-changelog ₩
    --alter ₩
    --add-config min.insync.replicas=1 ②
# output
Completed updating config for topic dev-consumer-pulse-counts-changelog
```

① kafka-topics 컨솔 스크립트를 사용할 수 있다. 컨플루언트 플랫폼이 아닌 바닐라 카프카를 사용하고 있다면 파일 확장자 (.sh)를 추가하길 바란다.

② 토픽 설정 업데이트

이제 상태 저장소를 지원하는 체인지로그 토픽의 목적과 기본 토픽 설정을 재정의하는 방법을 이해했는데 카프카 스트림즈를 매우 고가용성으로 만드는 특징인 대기 복제본을 살펴보자.

대기 복제본

스테이트풀 애플리케이션 고장 시 다운타임을 줄이는 한 가지 방법은 복수의 애플리케이션 인스턴스에 대해 태스크 상태 복제본을 만들어 유지하는 것이다.

카프카 스트림즈는 NUM_STANDBY_REPLICAS_CONFIG 특성에 대해 양의 값을 설정했다면 이를 자동적으로 다룬다. 예를 들어 2 개의 대기 복제본을 생성하기 위해 다음과 같이 애플리케이션을 설정할 수 있다:

```
props.put(StreamsConfig.NUM_STANDBY_REPLICAS_CONFIG, 2);
```

대기 복제본이 설정될 때 카프카 스트림즈는 실패한 스테이트풀 태스크를 상시 대기 인스턴스에 재할당하려고 할 것이다. 이는 처음부터 상태 저장소를 재초기화해야 할 필요성을 제거함으로써 큰 상태를

---

[3] 내부 토픽이 재설정될 수 있도록 하는 티켓이 존재하는데 https://oreil.ly/OoKBV에서 추적할 수 있다.

갖는 애플리케이션에 대한 다운타임을 현저하게 줄일 것이다.

또한 이 장 말미에서 볼 것이지만 카프카 스트림즈 신규 버전은 리밸런스 동안 상태 저장소에 쿼리할 때 대기 복제본으로 폴백할 수 있도록 한다. 그러나 이를 논의하기 전에 리밸런싱이 무엇이고 리밸런싱이 왜 스테이트풀 카프카 스트림즈 애플리케이션의 가장 큰 적인지를 논의해보자.

리밸런싱 : 상태 (저장소)의 적

체인지로그 토픽과 대기 복제본이 스테이트풀 애플리케이션 실패의 영향을 줄이는데 도움이 된다고 배웠다. 전자는 상태 손실 시 카프카 스트림즈가 이를 재구축할 수 있도록 하며 후자는 상태 저장소를 재초기화하는데 걸리는 시간을 최소화할 수 있도록 한다.

그러나 카프카 스트림즈가 실패를 투명하게 처리하는 반면 일시적이라도 상태 저장소 손실은 믿을 수 없게 파괴적일 수 있다는 사실은 변하지 않는다 (특히 heavily 스테이트풀 애플리케이션의 경우). 왜일까? 상태를 백업하기 위한 체인지 로깅 기법은 토픽 내 각 메시지를 재생해야 하기 때문에 그 토픽이 거대한 경우 각 레코드를 다시 읽는데 몇 분 또는 심한 경우 몇 시간이 걸릴 수도 있다.

상태 재초기화를 야기하는 가장 큰 요인은 리밸런싱이다. "컨슈머 그룹"에서 컨슈머 그룹을 논의할 때 최초로 이 용어를 접했다. 간단하게 말하자면 카프카가 자동으로 컨슈머 그룹의 활성 멤버에 작업을 분배하지만 때때로 특정 이벤트에 따라 작업이 재분배될 필요가 있다는 것이다 – 가장 주목할 만한 이벤트는 그룹 멤버십 변경이다[4]. 전체 리밸런스 프로토콜을 세부적으로 다루지는 않을 것이며 논의할 세부 수준은 용어 검토를 필요로 한다:

- 그룹 조정자는 컨슈머 그룹의 멤버십 유지를 담당하는 브로커이다 (예, 하트비트를 수신하고 멤버십 변경이 탐지될 때 리밸런스를 트리거)
- 그룹 리더는 파티션 재할당 결정을 담당하는 컨슈머이다.

다음 절에서 리밸런싱을 보다 논의할 때 이 용어들을 만날 것이다. 그러나 현재 가장 중요한 요점은 리밸런스가 스테이트풀 태스크를 대기 복제본을 갖지 못한 다른 인스턴스로 이관되도록 할 때 이것이 매우 비싸다는 것이다. 리밸런싱이 야기할 수 있는 문제들을 다루기 위해 여러 전략이 존재한다.

- 가능한 경우 상태 이동을 방지한다
- 상태가 이동 또는 재생되어야 하는 경우 가능한 복구를 빨리 한다.

두 전략을 사용하여 리밸런스의 영향을 최소화하기 위해 카프카 스트림즈가 자동적으로 취하는 그리고 우리가 직접 취할 수 있는 조치가 있다. 첫번째 전략인 상태 이관 방지로 시작하여 이를 보다 세부적으로 논의할 것이다.

---

[4] 소스 토픽에 파티션을 추가 또는 삭제하는 것 또한 리밸런스를 야기할 수 있다.

상태 이관 방지

스테이트풀 태스크가 실행 중인 다른 인스턴스에 재할당될 때 상태 또한 이관된다. 큰 상태를 갖는 애플리케이션의 경우 목적지 노드에서 상태 저장소를 재구축하는데 오랜 시간이 걸릴 수 있으며 따라서 가능한 피해야 한다.

컨슈머 중 하나인 그룹 리더는 활성 컨슈머 간에 작업이 분배되는 법을 결정하는 일을 책임지기 때문에 불필요한 상태 저장소 이관을 방지하기 위해 (로드 밸런싱 로직을 구현하는) 카프카 스트림즈 라이브러리에 일부 부담이 있다. 이를 달성하는 한 가지 방법은 고정(sticky) 할당자라는 것을 통해서로 카프카 스트림즈를 사용할 때 무료로 제공되는 것이다. 다음 절에서 이를 논의할 것이다.

고정 할당

스테이트풀 태스크가 재할당되는 것을 방지하기 위해 카프카 스트림즈는 이전에 태스크를 소유했던 (따라서 아직까지 기본 상태 저장소의 복제본을 갖고 있는) 인스턴스에 태스크를 재할당하려는 맞춤형 파티션 할당 전략[5]을 사용하다. 이 전략을 고정 할당이라고 부른다.

고정 할당자로 카프카 스트림즈가 해결하려는 문제를 이해하기 위해 다른 카프카 클라이언트 유형에 대한 기본 리밸런싱 전략을 살펴보자. 그림 6-1은 리밸런싱 발생 시 스테이트풀 태스크가 다른 애플리케이션 인스턴스로 잠재적으로 이관될 수 있음을 보여주는데 이는 매우 값비쌀 것이다.

카프카 스트림즈에 포함되어 있는 고정 파티션 할당은 각 파티션 (과 연관된 상태 저장소)를 어떤 태스크가 소유했는지 추척하고 스테이트풀 태스크를 이전 소유자에 재할당함으로써 이 문제를 다룬다. 그림 6-2에서 볼 수 있듯이 이는 아마도 큰 상태 저장소의 불필요한 재초기화를 방지하도록 돕기 때문에 애플리케이션의 가용성을 극적으로 향상시킨다.

고정 할당자가 태스크의 이전 소유자에 태스크를 재할당하는 것을 돕는 반면 상태 저장소는 카프카 스트림즈 클라이언트가 일시적으로 오프라인인 경우에도 이관될 수 있다. 이제 카프카 스트림즈 개발자가 일시적 다운타임 동안 리밸런스를 피하는 것을 도울 수 있는 방법을 논의해보자.

---

[5] 내부 클래스는 StickyTaskAssignor로 카프카 스트림즈에서 기본 파티셔너를 재정의하는 것은 불가능함을 주목하기 바란다.

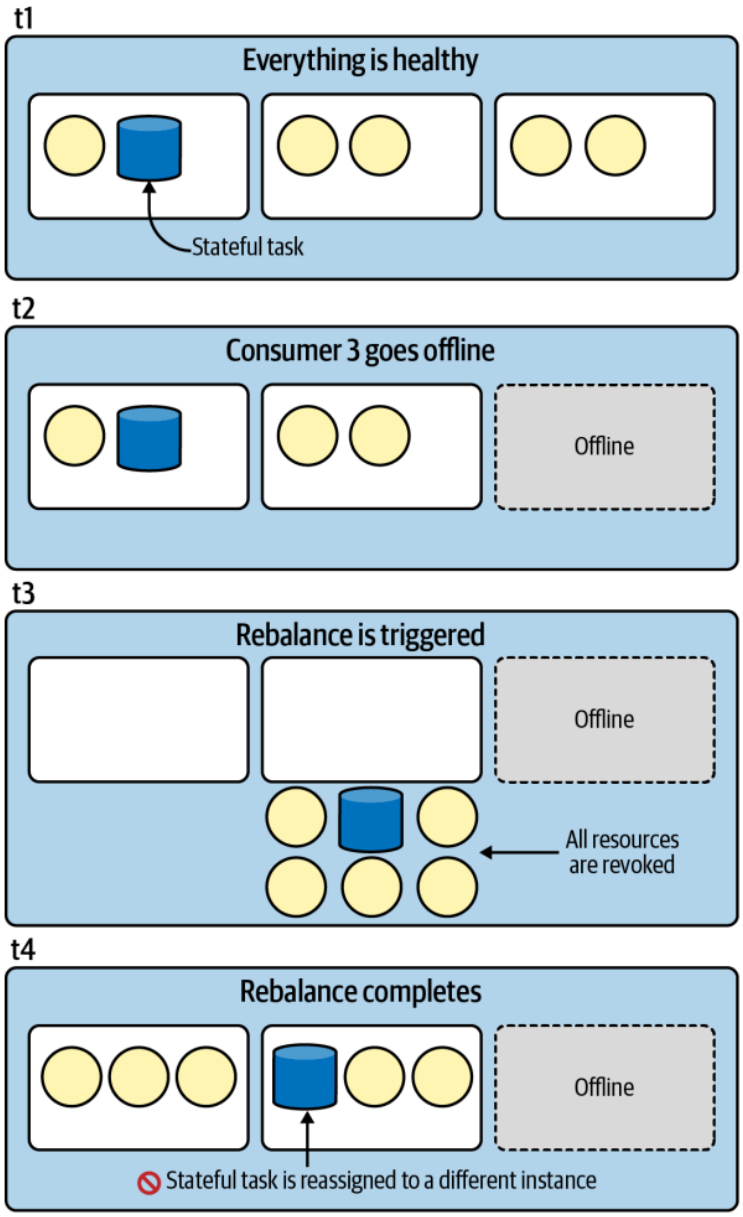


그림 6-1. 비고정 파티션 할당

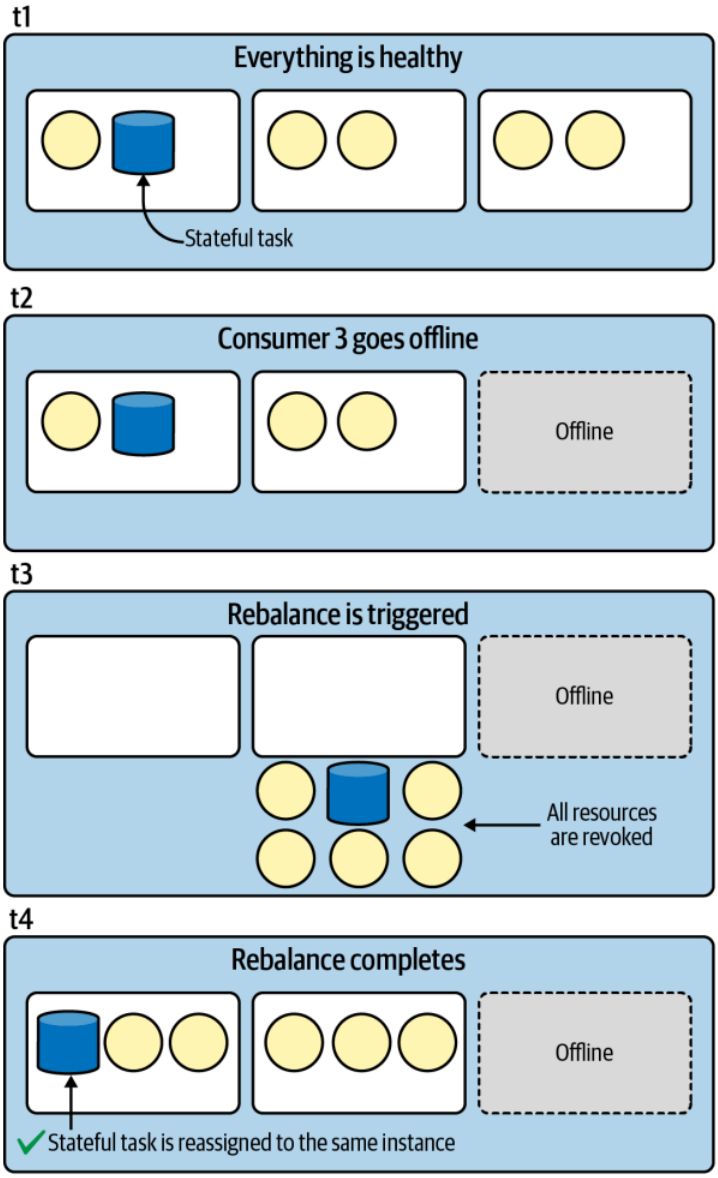


그림 6-2. 카프카 스트림즈의 내장 파티션 할당자를 사용한 고정 파티션 할당

정적 멤버십

상태가 이리저리 이동되도록 야기할 수 있는 문제 중 하나는 불필요한 리밸런스이다. 종종 롤링 바운스와 같은 건강한 이벤트도 일부 리밸런스를 야기할 수 있다. 그룹 조정자가 이러한 짧은 비가용 기간 동안 멤버십 변경을 탐지한다면 리밸런스를 트리거하고 즉시 작업을 나머지 애플리케이션 인스턴스에 재할당할 것이다.

인스턴스가 짧은 다운타임 이후 온라인으로 돌아올 때 이의 멤버십 ID (컨슈머가 등록될 때 조정자에 의해 할당된 고유 식별자)가 삭제되어 있기 때문에 이를 인지하지 못하며 따라서 인스턴스는 새로운 멤버로 취급되고 작업이 재할당될 수 있다.

이를 방지하기 위해 정적 멤버십을 사용할 수 있다. 정적 멤버십은 일시적인 다운타임으로 인한 리밸

런스 수를 줄이는데 목적이 있다. 이는 각각 고유한 애플리케이션 인스턴스를 식별하기 위해 하드코딩된 인스턴스 ID를 사용함으로써 얻어진다. 다음 설정 특성을 통해 ID를 설정할 수 있다:

Group.instance.id = app-1 ①

① 이 경우 app-1이라는 ID를 설정하였다. 다른 인스턴스를 생성하는 경우 또한 app-2와 같이 고유한 ID를 할당할 것이다. 이 ID는 전체 클러스터를 통해 고유해야 한다 (application.id과 상관없는 다른 카프카 스트림즈 애플리케이션 간에도 그리고 group,.id와 상관없는 다른 컨슈머 간에도).

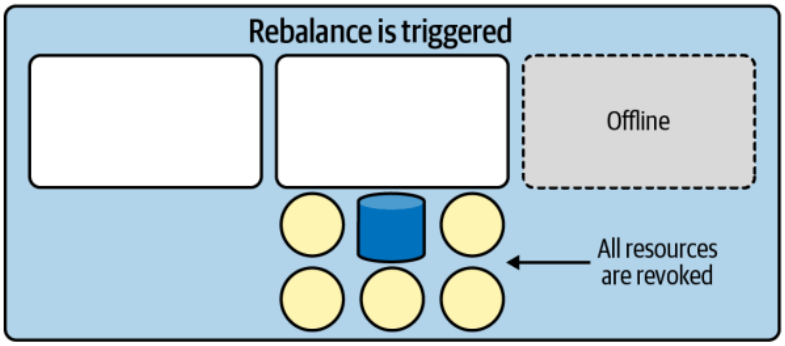
하드코딩된 인스턴스 ID는 일반적으로 보다 큰 세션 타임아웃[6]과 함께 사용되는데 이는 애플리케이션이 재시작하는데 더욱 많은 시간이 걸리며 따라서 조정자는 컨슈머가 짧은 기간 동안 오프라인일 때 컨슈머 인스턴스가 죽었다고 생각하지 않는다.

정적 멤버십은 카프카 버전 2.3 이상에만 사용가능하며 따라서 클라이언트 또는 브로커가 이전 버전이라면 우선 업그레이드해야 할 것이다. 세션 타임 증가가 양날의 검임을 알기 바란다. 인스턴스가 짧은 기간 동안 죽었다고 조정자가 가정하는 것을 방지하는 반면 트레이드 오프는 실제 고장을 보다 늦게 탐지하도록 야기할 수 있다는 것이다.

이제 정적 멤버십이 어떻게 불필요한 리밸런스를 방지하는데 도움이 될 수 있는지를 배웠는데 리밸런스 발생 시 이의 영향을 줄이는 방법을 살펴보자. 다시 한번 강조하면 이 영향을 줄이기 위해 카프카 스트림즈가 우리 대신 취할 수 있는 조치와 우리 스스로 취할 수 있는 행동이 존재한다. 다음 절에서 이들 모두를 논의할 것이다.

리밸런스 영향 줄이기

정적 멤버십이 불필요한 리밸런스를 방지하는데 사용될 수 있지만 종종 리밸런스를 방지할 수 없다. 결국 분산 시스템에서는 고장이 예상된다. 역사적으로 리밸런싱은 매우 고비용으로 이는 각각의 리밸런싱 라운드 시작 시 각 클라인트가 모든 리소스를 포기하기 때문이다. 그림 6-2에서 이를 보았는데 그림 6-3에 리밸런싱 단계에 대한 세부적인 뷰를 나타내었다.



[6] Session.timeout.ms 컨슈머 설정 참조

그림 6-3. 긴급 리밸런스 동안 모든 리소스를 취소하는 것은 극히 비효율적이다.

이 리밸런싱 전략은 긴급 리밸런싱이라 부르는데 2 가지 이유로 매우 영향이 크다:

- 모든 클라이언트가 그들의 리소스를 포기할 때 세상 정지 (Stop the world) 효과가 발생하며 이는 처리가 중지되기 때문에 애플리케이션이 매우 빨리 작업에 뒤쳐질 수 있음을 의미한다.
- 스테이트풀 태스크가 새로운 인스턴스에 재할당된다면 처리를 시작하기 전에 상태가 재생/재구축되어야 할 것이다. 이는 추가적인 다운타임을 야기한다.

카프카 스트림즈의 경우 맞춤형 고정 파티션 할당자를 사용함으로써 우리를 위해 두번째 문제 (스테이트풀 이관)를 완화하려고 함을 기억하기 바란다. 그러나 리밸런스 프로토콜 자체 대신 태스크 고정성을 얻기 위해 역사적으로 각자의 고유 구현에 의존해왔다. 2.4 버전에서 리밸런스 프로토콜의 업데이트는 리밸런스의 영향을 줄이는데 도움이 되는 추가적인 조치를 도입하였다. 다음 절에서 증분적 협업 리밸런싱이라는 새로운 프로토콜을 논의할 것이다.

증분적 협업 리밸런싱

증분적 협업 리밸런싱은 긴급 리밸런싱보다 효율적인 리밸런싱 프로토콜로 버전 2.4 이상에서는 기본적으로 활성화되어 있다. 이전 카프카 스트림즈 버전을 사용한다면 이전 긴급 리밸런싱 프로토콜에 비해 다음 장점을 갖고 있기 때문에 이러한 특징을 이용하기 위해 업그레이드하길 원할 것이다:

- 하나의 글로벌 리밸런싱 라운드가 몇몇의 보다 작은 라운드로 대체되었다 (증분적).
- 클라이언트가 소유권 변경이 필요치 않은 리소스 (태스크)를 계속 보유하며 이관 중인 태스크 처리만 중지한다 (협업적)

그림 6-4는 애플리케이션 인스턴스가 장기간 (조정자가 컨슈머 실패를 탐지하는데 사용하는 타임아웃인 session.timeout.ms 설정치를 초과하는 기간) 오프라인일 때 고차원에서 증분적 협업 리밸런싱 동작 원리를 보여준다.

그림에서 보듯이 (스테이트풀 태스크를 갖는 인스턴스를 포함하여) 정상 애플리케이션 인스턴스는 리밸런스 시작 시 자신들의 리소스를 포기할 필요가 없다. 이는 애플리케이션이 리밸런스 동안 계속 처리를 할 수 있도록 하기 때문에 긴급 리밸런싱 전략에 비해 큰 개선이다.

증분적 협업 리밸런싱에 대해 여기서 다루지 않을 많은 세부 정보가 존재하는데 신규 버전의 카프카 스트림즈가 내부적으로 이 프로토콜을 구현하고 있음을 아는 것이 중요하며 따라서 이를 지원하는 버전의 카프카 스트림즈를 실행하고 있음을 확인할 필요가 있다[7].

증분적 협업 리밸런싱이 어떻게 리밸런스의 영향을 줄이는데 도움이 될 수 있는지를 살펴보았으며 리

---

[7] 버전 2.4 이상에서 개선된 리밸런싱 전략을 사용한다. 증분적 협업 리밸런싱에 대한 더욱 자세한 정보는 https://oreil.ly/P3iVG 를 참조하기 바란다.

밸런스가 보다 덜 고통스러움을 보장하기 위해 애플리케이션 개발자가 취할 수 있는 보다 적극적인 역할을 살펴보자.

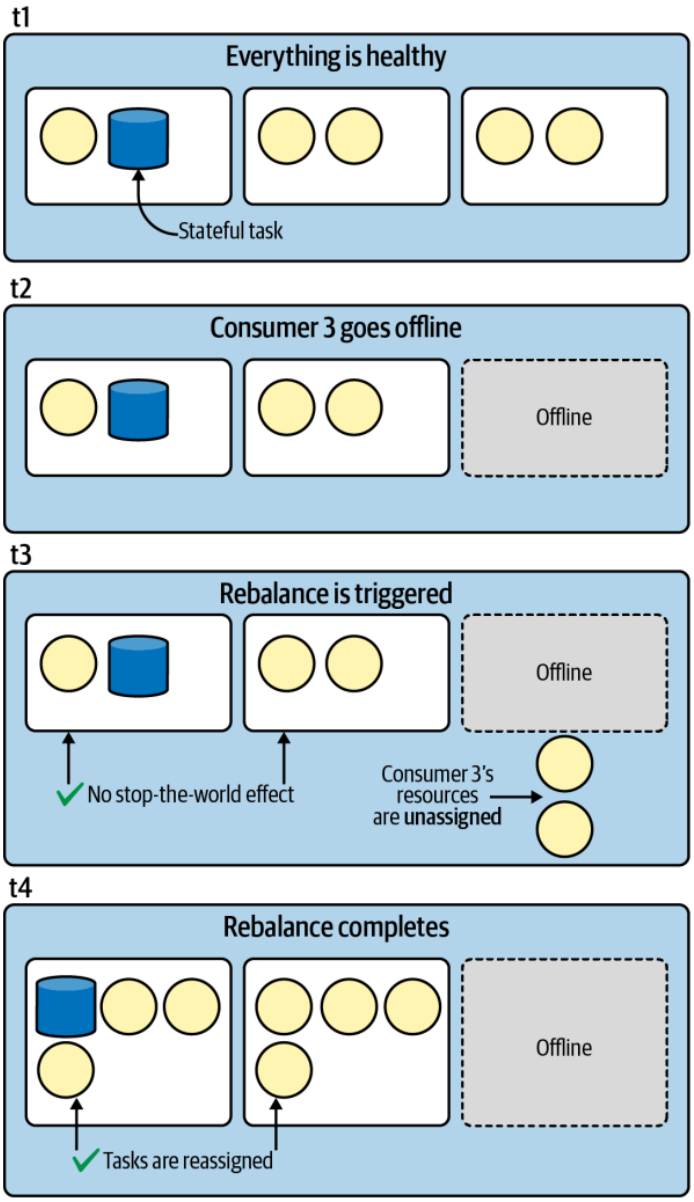


그림 6-4. 세상 정지 효과는 증분적 협업 리밸런싱에서 방지된다.

상태 크기 제어하기

여러분이 주의하지 않는 경우 상태 저장소는 무한정 커질 수 있고 운영 상 문제를 야기할 수 있다. 예를 들어 압축된 체인지로그 토픽 내 키 스페이스가 매우 크고 (가령 십억개의 키) 애플리케이션 상태가 10 개의 물리 노드에 균등하게 분배되어 있다고 가정해보자[8]. 노드 중 하나가 오프라인이 되면 상

[8] 실제 시나리오에서 키 스페이스기 정확히 균등하게 분리될 것 같지는 않으며 이는 논의를 위한 것으로 요점은 동일하게 유지된다.

태를 재구축하기 위해 최소 1억개의 레코드를 재생해야 할 것이다. 이는 많은 시간이 소요되며 가용성 문제를 야기할 수 있다. 또한 필요 이상의 많은 데이터를 보유하는 것은 계산 또는 저장 리소스에 유용하지 않다.

여러분의 유스케이스에 따라 전체 애플리케이션 상태를 무한정 보유할 필요가 없을 수 있다. 아마도 각 레코드의 가치는 고정된 라이프타임을 갖는다. 예를 들어 Mailchimp에서는 아직 전송되지 못한 활성 이메일 메시지의 수를 추적하여 이 메시지에 대한 다양한 통계를 보여주기 위해 key 당 집계를 수행한다. 그러나 결국 이러한 이메일 메시지는 비활성이 되며 (즉, 전달되거나 반송) 더 이상 이들을 추적할 필요는 없다. 이와 같은 유스케이스는 상태 저장소의 적극적인 정리를 필요로 한다. 상태 저장소를 작게 유지하기 위해 불필요한 데이터를 삭제하는 것은 리밸런싱의 영향을 상당히 줄인다. 스테이트풀 태스크가 이관될 필요가 있는 경우 불필요하게 큰 상태 저장소보다 작은 상태 저장소를 재구축하는 것이 보다 쉽다. 그렇다면 카프카 스트림즈에서 불필요한 상태를 어떻게 삭제하는가? 툼스톤(tombstone)을 사용한다.

툼스톤

툼스톤은 어떤 상태가 삭제될 필요가 있음을 나타내는 특수 레코드이다. 이들은 종종 삭제 마커로 간주되며 항상 키와 null 값을 갖는다. 이전에 언급했듯이 상태 저장소는 키 기반으로 툼스톤 레코드의 키는 상태 저장소로부터 어떤 레코드가 삭제되어야 하는지를 나타낸다.

툼스톤 생성 방법 예은 다음 코드 블록에 나타냈다. 이 가상 시나리오에서 병원 환자에 대한 집계를 수행하지만 환자의 체크아웃 이벤트를 보는 경우 이 환자에 대해 더 이상 추가적인 이벤트를 예상하지 않으므로 상태 저장소에서 이들의 데이터를 삭제한다:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream("patient-events");
stream
  .groupByKey()
  .reduce(
    (value1, value2) -> {
      if (value2.equals(PATIENT_CHECKED_OUT)) {
        // create a tombstone
        return null; ①
      }
      return doSomething(value1, value2); ②
      });
```

① 환자가 체크아웃할 때마다 툼스톤 (삭제 마커) 생성을 위해 null을 반환한다 이는 관련 키가 상태 저장소로부터 제거되도록 할 것이다.

② 환자가 체크아웃하지 않은 경우 집계 로직을 수행한다.

툼스톤은 키-값 저장소를 작게 유지하는데 유용하며 윈도우 키-값 저장소로부터 불필요한 데이터를 제거하기 위해 사용할 수 있는 다른 방법도 있다. 이를 다음 절에서 논의할 것이다.

윈도우 유지

윈도우 저장소는 상태 저장소를 작게 유지하기 위해 설정가능한 유지 기간을 갖고 있다. 예를 들어 이전 장에서 원시 펄스 이벤트를 심박수로 변환하기 위해 환자 모니터링 애플리케이션에 윈도우 저장소를 생성하였다. 관련 코드는 다음과 같다:

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

① 이 값이 명시적으로 재정의되지 않았기 때문에 기본 유지 기간 (1일)을 갖는 윈도우 저장소를 구체화한다.

그러나 Materialized 클래스는 카프카 스트림즈가 윈도우 저장소에 레코드를 얼마나 유지할 것인지를 지정하는데 사용할 수 있는 withRetention 메소드를 갖고 있다. 다음 코드는 윈도우 저장소 유지를 지정하는 방법을 보여준다:

```
TimeWindows tumblingWindow =
    TimeWindows.of(Duration.ofSeconds(60)).grace(Duration.ofSeconds(5));
KTable<Windowed<String>, Long> pulseCounts =
  pulseEvents
    .groupByKey()
    .windowedBy(tumblingWindow)
    .count(Materialized.<String, Long, WindowStore<Bytes, byte[]>>
      as("pulse-counts"))
    .withRetention(Duration.ofHours(6))) ①
    .suppress(
        Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull()));
```

① 6 시간의 유지 기간을 갖는 윈도우 저장소를 구체화한다.

유지 기간이 윈도우 크기와 유예 기간의 합보다 항상 커야 함을 주목하기 바란다. 이전 예에서 유지 기간은 65초보다 커야 한다 (텀블링 윈도우 크기 60초 + 유예 기간 5초). 기본 윈도우 유지 기간은 1일이며 따라서 이 값을 낮춤으로써 윈도우 상태 저장소 (와 체인지로그 토픽)의 크기를 줄일 수 있고 복구

시간을 빠르게 할 수 있다.

지금까지 상태 저장소를 작게 유지하기 위해 애플리케이션 코드에서 사용할 수 있는 2 가지 방법 (툼스톤 생성과 윈도우 저장소의 유지 기간 설정)을 논의하였다. 이제 체인지로그 토픽을 작게 유지하기 위한 다른 방법인 공격적 토픽 압축을 살펴보자.

공격적 토픽 압축

기본적으로 체인지로그는 압축된다. 이는 각 키에 대해 단지 최신 값만 유지되고 툼스톤 사용 시 관련 키의 값이 전적으로 삭제됨을 의미한다. 그러나 상태 저장소가 즉시 압축된 또는 삭제된 값을 반영하지만 토픽은 장기간 압축되지 않은/삭제된 값을 유지함으로써 필요 이상으로 크게 유지될 수 있다.

이는 카프카가 디스크 상에 토픽을 표현하는 방법과 관련이 있다. 우리는 이미 토픽이 어떻게 파티션으로 분리되고 카프카 스트림즈에서 이 파티션이 어떻게 단일 작업 단위로 변환되는지에 대해 논의했다. 그러나 파티션이 애플리케이션 측에서 일반적으로 다루는 가장 저수준의 토픽 추상화지만 (우리는 일반적으로 얼마나 많은 쓰레드가 필요한지 고려할 때 그리고 데이터가 어떻게 전달되고 관련 데이터와 어떻게 같이 파티션되어야 하는지를 알 때 파티션에 대해 고려한다) 카프카 브로커 측면에서 보다 낮은 수준의 추상화가 존재한다: 세그먼트.

세그먼트는 해당 토픽 파티션에 대해 메시지 부분집합을 포함하는 파일이다. 언제라도 항상 파티션에 현재 써지고 있는 파일인 활성 세그먼트가 존재한다. 시간이 지남에 따라 활성 세그먼트는 크기 임계치에 도달하고 비활성화될 것이다. 일단 세그먼트가 비활성화된다면 정리 대상이 될 것이다.

> 압축되지 않은 레코드의 경우 dirty하다고 간주된다. 로그 클리너는 dirty 로그에 대해 압축을 수행하는 프로세스로 이는 사용가능 디스크 공간을 늘림으로써 브로커에 도움을 주고 상태 저장소를 재구축하기 위해 재생되어야 하는 레코드 수를 줄임으로써 카프카 스트림즈 클라이언트에 도움을 준다.

활성 세그먼트는 정리 대상이 아니고 따라서 상태 저장소 재초기화 시 재생되어야 할 매우 많은 수의 비압축 레코드 및 툼스톤을 포함할 수 있기 때문에 보다 공격적인 토픽 압축을 위해 세그먼트 크기를 줄이는 것이 도움이 될 것이다[9]. 더구나 로그 클리너의 경우 로그 중 50% 이상이 정리/압축되었다면 로그 정리를 피할 것이다. 이는 설정가능하며 로그 정리 발생 주기를 늘리기 위해 조정될 수 있다.

표 6-1에 열거된 토픽 설정이 보다 공격적인 압축을 활성화하는데 유용하며, 상태 저장소가 재초기화될 필요가 있는 경우 재생되어야 할 레코드 수를 줄인다[10].

---

[9] 이에 대해서는 Levani Kokhreidze의 Achieving High Availability with Stateful Kafka Streams Applications을 참조하기 바란다.

[10] 설정 정의는 공식 카프카 문서를 따른다.

표 6-1. 보다 빈번한 로그 정리/압축을 위해 사용될 수 있는 토픽 설정

설정	기본	정의
Segment.bytes	1073741824 (1 GB)	로그를 위한 세그먼트 파일 크기를 제어. 정리는 항상 한 번에 한 파일로 세그먼크 크기가 클수록 파일 수가 적으며 유지에 대해 덜 미세한 제어를 한다.
Segment.ms	604800000 (7 일)	오래된 데이터가 압축 또는 삭제될 수 있음을 보장하기 위해 세그먼트 파일이 가득 차지 않더라도 이 기간 이후 카프카가 강제로 로그를 롤링할 기간을 제어
min.cleanable.dirty.ratio	0.5	로그 컴팩터가 얼마나 자주 로그 정리를 시도할 것인지를 제어 (로그 압축이 활성화되어 있다고 가정). 기본적으로 로그 중 50% 이상이 압축된 로그 정리를 피할 것이다. 이 비율은 중복에 의해 로그에서 낭비되는 최대 공간을 제한한다 (기본 설정의 경우 많아야 로그 중 50%가 중복을 포함할 수 있다). 비율이 높을수록 덜 빈번하고 더 효율적인 정리를 의미하지만 로그에 더 많이 공간이 낭비됨을 의미할 것이다. Max.compaction.lag.ms 또는 min.compaction.lag.ms 설정이 지정된다면 (i) dirty 비율 임계치가 충족되고 로그가 적어도 min.compaction.lag.ms 기간 동안 dirty (비압축) 레코드를 갖는다면 또는 (ii) 로그가 많아야 max.compaction.lag.ms 기간 동안 dirty (비압축) 레코드를 갖고 있다면 로그 컴팩터는 로그를 정리 대상으로 고려할 것이다.
max.compaction.lag.ms	Long.Max_Value -1	로그 압축에 대해 메시지가 비대상으로 유지될 최대 시간. 압축 중인 로그에만 적용가능
min.compaction.lag.ms	0	로그 압축에 대해 메시지가 비대상으로 유지될 최소 시간. 압축 중인 로그에만 적용가능

구체화된 저장소에서 이들 설정 중 둘을 변경하는 방법 예를 다음 코드에 나타냈다. 이들 토픽 설정은 세그먼트 크기와 또한 최소 정리가능 dirty 비율을 줄임으로써 더욱 빈번하게 로그 정리를 트리거하는 데 도움을 줄 수 있다.

```
Map topicConfigs = new HashMap<>();
topicConfigs.put("segment.bytes", "536870912"); ①
topicConfigs.put("min.cleanable.dirty.ratio", "0.3"); ②
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream("patient-events");
KTable<byte[], String> counts =
  stream
    .groupByKey()
    .count( Materialized.<byte[], Long, KeyValueStore<Bytes, byte[]>as("counts")
      .withKeySerde(Serdes.ByteArray())
      .withValueSerde(Serdes.Long())
      .withLoggingEnabled(topicConfigs));
```

① 세그먼트 크기를 1MB로 줄인다.

② 최소 정리가능 dirty 비율을 30%로 줄인다.

기본 저장 매체가 이론적으로 제한되지 않기 때문에 토픽 압축은 필요하다. 상태 저장소 크기를 최소화하는 다른 방법은 대신 고정 크기 데이터 구조를 사용하는 것이다. 이 방법에는 단점이 있지만 카프카 스트림즈는 이 측면에서 문제를 해결하는 상태 저장소를 포함한다. 다음 절에 이를 논의할 것이다.

고정 크기 LRU (Least Recently Used) 캐시

상태 저장소가 무한정 커지지 않음을 보장하기 위한 덜 일반적인 방법은 인메모리 LRU 캐시를 사용하는 것이다. 이는 상태가 설정된 크기를 초과 시 가장 오래된 엔트리를 자동적으로 삭제하는 설정가능하고 고정된 용량 (최대 엔트리 수로 지정)을 갖는 단순한 키-값 저장소이다. 더구나 엔트리가 인메모리 저장소에서 제거될 때마다 기본 체인지로그 토픽에 툼스톤이 자동적으로 전송된다.

인메모리 LRU 맵을 사용하는 방법은 다음과 같다:

```
KeyValueBytesStoreSupplier storeSupplier = Stores.lruMap("counts", 10); ①
StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> stream = builder.stream("patient-events");
stream
  .groupByKey()
  .count(
    Materialized.<String, Long>as(storeSupplier) ②
  .withKeySerde(Serdes.String())
  .withValueSerde(Serdes.Long()));
return builder.build();
```

① 최대 10개의 엔트리를 갖는 인메모리 LRU 저장소인 counts 생성

② 저장소 서플라이어를 사용하여 인메모리 LRU 저장소를 구체화한다.

프로세서 API로 이를 하기 위해서는 다음과 같이 저장소 빌더를 사용할 수 있다:

```
StreamsBuilder builder = new StreamsBuilder();
KeyValueBytesStoreSupplier storeSupplier = Stores.lruMap("counts", 10);
StoreBuilder<KeyValueStore<String, Long>> lruStoreBuilder =
  Stores.keyValueStoreBuilder(storeSupplier, Serdes.String(), Serdes.Long());
builder.addStateStore(lruStoreBuilder);
```

"공격적 토픽 압축"에서 언급했듯이 LRU 캐시를 지원하는 체인지로그 토픽의 경우 압축과 삭제가 즉시 일어나지 않으며 따라서 고장의 경우 상태 저장소를 재초기화하기 위해 전체 토픽이 재생되어야 하기 때문에 복구 시간이 영구 저장 상태소보다 길 수 있다 (LRU 맵에서 단지 10 개의 레코드만 확인하더라도). 이는 영구 대 인메모리 저장소에서 초기에 논의했던 인메모리 저장소 사용의 주요 단점이며 따라서 이 옵션은 트레이드 오프를 완전히 이해한 경우 사용되어야 한다.

이것으로 상태 저장소와 체인지로그 토픽을 불필요한 레코드없이 유지하는 것에 대한 논의를 마친다.

이제 상태 저장소가 읽기 지연 또는 쓰기 양에 의해 병목 현상이 발생하는 경우 추구할 수 있는 전략을 논의해보자.

레코드 캐시로 쓰기 중복 제거하기

"억제"에서 논의했듯이 윈도우 저장소에서 업데이트의 속도를 제한하기 위한 일부 DSL 메소드 (버퍼 설정과 함께 suppress, 표 5-2 참조)가 존재한다. 또한 상태 업데이트가 상태 저장소와 다운스트림 프로세서 모두에 쓰여지는 빈도를 제어하는 운영 파라미터도 존재한다[11]. 표 6-2는 이들 파라미터를 나타낸다.

표 6-2. 상태 저장소와 다운스트림 프로세서에 대한 쓰기를 줄이는데 사용될 수 있는 토픽 설정

원시 설정	StreamsConfig 특성	기본	정의
Cache.max.bytes.buffering	CACHE_MAX_BYTES_BUFFERING_CONFIG	1048576 (10 MB)	모든 쓰레드에 대해 버퍼링에 사용되는 바이트 단위의 최대 메모리 양
Commit.interval.ms	COMMIT_INTERVAL_MS_CONFIG	30000 (30 초)	프로세서의 포지션을 저장하는 주기

캐시 크기가 더 크고 커밋 간격이 더 길수록 동일 키에 대한 연속 업데이트의 중복을 제거하는데 도움이 될 수 있다. 이는 다음을 포함하여 장점을 갖는다:

- 읽기 지연 줄이기
- 상태 저장소, 체인지 로그 토픽 (활성화된 경우) 및 다운스트림 프로세서에 대한 쓰기 볼륨 줄이기

따라서 상태 저장소에 대한 읽기/쓰기 또는 (체인지로그 토픽에 대한 빈번한 업데이트의 결과일 수 있는) 네트워크 I/O에 병목 현상이 발생하는 경우 이들 파라미터의 조정을 고려해야 한다. 물론 레코드 캐시가 커질수록 몇몇 트레이드 오프가 존재한다:

- 메모리 사용량 증가
- 지연 증가 (레코드가 덜 자주 방출된다).

첫번째와 관련하여 (cache.max.bytes.buffering 파라미터에 의해 제어되는) 레코드 캐시에 할당되는 총 메모리는 모든 스트리밍 쓰레드에 대해 공유된다. 메모리 풀이 균등하게 나뉠 것이고 따라서 (다른 파티션에 비해 비교적 많은 데이터 양을 갖는 파티션인) hot 파티션을 처리하는 쓰레드는 캐시를 더욱 빈번하게 비울 것이다. 캐시 크기와 커밋 간격에 상관없이 최종 스테이트풀 계산은 동일할 것이다.

---

[11] 운영 파라미터와 suppress가 제공하는 비즈니스 로직 유형 방법 간 차이는 컨플루언트 블로그의 Eno Thereska et al.의 Watermarks, Tables, Event Time, and the Dataflow Model 내에 논의되어 있다.

보다 큰 커밋 간격을 사용할 때에도 트레이드 오프가 존재한다. 가령 고장 후 재실행해야 할 작업량은 이 설정 값을 증가시킴에 따라 증가할 것이다.

마지막으로 종종 어떤 캐시도 없이 각각의 중간 상태 변경을 보는 것이 바람직할 수도 있다. 사실 카프카 스트림즈에 처음인 사람들은 특정 수의 레코드를 소스 토픽에 생산했지만 (아마도 즉시가 아닌 몇 초의 지연 후에) 상태 변경의 일부만 플러시되는 것을 보기 때문에 캐시 플러싱 시 중복제거 또는 지연을 관측할 것이고 무언가 잘못되었다고 생각할 것이다. 따라서 개발 환경에서 캐시를 전적으로 비활성화하고 보다 작은 커밋 간격을 사용할 것이다. 이는 성능에 영향을 미칠 수 있기 때문에 실제 운영 환경에서는 주의하기 바란다.

상태 저장소 모니터링

애플리케이션을 실제 운영에 배치하기 전에 애플리케이션을 적절히 지원하기 위해 이들에 대해 충분한 가시성을 가지고 있는지를 보장하는 것은 중요하다. 이 절에서는 운영 상의 수고를 덜고 에러 발생 시 이를 디버깅하는데 충분한 정보를 가질 수 있도록 스테이트풀 애플리케이션 모니터링에 대한 일반적인 방법을 논의한다.

상태 리스너 추가하기

카프카 스트림즈 애플리케이션은 많은 상태 중 하나에 놓일 수 있다 (상태 저장소와 혼동하지 말길). 그림 6-5는 이들 상태들 각각과 유효한 전이를 보여주고 있다.

이전에 언급했듯이 리밸런싱 상태는 스테이트풀 카프카 스트림즈 애플리케이션에 특히 영향을 미칠 수 있으며 따라서 애플리케이션이 언제 리밸런싱 상태로 전이되고 얼마나 자주 발생하는지를 추적할 수 있다는 것은 모니터링 목적에 유용할 수 있다. 운좋게도 카프카 스트림즈는 상태 리스너를 사용하여 언제 애플리케이션 상태가 변경되는지 모니터링하는 것을 쉽게 한다. 상태 리스너는 애플리케이션 상태 변경 시마다 호출되는 콜백 메소드이다.

애플리케이션에 따라 리밸런스 발생 시 특정 조치를 취하기를 원할 수도 있다. 예를 들어 Mailchimp에서는 리밸런스가 트리거될 때마다 증분되는 카프카 스트림즈 내 특수 지표를 생성하였다. 지표는 모니터링 시스템 (Prometheus)에 보내지며 여기서 쿼리를 하거나 알림 생성을 위해 사용될 수 있다.

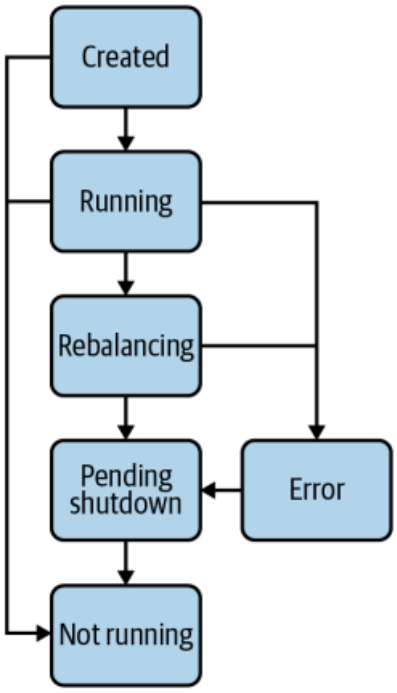


그림 6-5. 카프카 스트림즈 내 애플리케이션 상태와 유효 전이

다음 코드는 특별히 리밸런싱 상태로의 전이를 리스닝하는 상태 리스너를 카프카 스트림즈 토폴로지에 추가하는 방법을 보여준다:

```
KafkaStreams streams = new KafkaStreams(...);
streams.setStateListener( ①
    (oldState, newState) -> { ②
        if (newState.equals(State.REBALANCING)) { ③
        // do something
        }
    });
```

① 애플리케이션 상태 변경 시마다 메소드를 호출하기 위해 사용되는 KafkaStreams.setStateListener 메소드

② StateListener 메소드의 메소드 시그니처는 과거와 새로운 상태 모두를 포함한다.

③ 애플리케이션이 리밸런싱 상태로 들어갈 때마다 조건부로 조치를 수행한다.

상태 리스너가 매우 유용하지만 카프카 스트림즈 애플리케이션에 활용할 수 있는 유일한 리스너는 아니다. 다음 절에서는 스테이트풀 애플리케이션의 가시성을 향상시키기 위해 사용될 수 있는 다른 방법을 논의할 것이다.

상태 복구 리스너 추가하기

이전 절에서는 카프카 스트림즈 애플리케이션 내 리밸런스 트리거를 리스닝하는 방법을 배웠다. 그러

나 리밸런스는 상태 저장소 재초기화를 야기할 때 주된 관심 대상이다. 카프카 스트림즈는 상태 저장소가 재초기화될 때마다 호출될 수 있는 상태 복구 리스너를 포함하고 있다. 다음 코드는 카프카 스트림즈 애플리케이션에 상태 복구 리스너를 추가하는 방법을 보여준다.

```
KafkaStreams streams = new KafkaStreams(...);
streams.setGlobalStateRestoreListener(new MyRestoreListener());
```

MyRestoreListener 클래스는 StateRestoreListener의 인스턴스로 다음 코드 블록에서 구현된다. 이전절에서 보았던 상태 리스너와 달리 상태 복구 리스너는 상태 복구 프로세스 라이프 사이클 중 특정 부분에 연결되는 3 가지 메소드를 구현해야 한다. 다음 코드의 주석은 각 메소드가 어디에 사용되는지를 설명한다:

```
class MyRestoreListener implements StateRestoreListener {
    private static final Logger log =
        LoggerFactory.getLogger(MyRestoreListener.class);
    @Override
    public void onRestoreStart(  ①
        TopicPartition topicPartition,
        String storeName,
        long startingOffset,
        long endingOffset) {
        log.info("The following state store is being restored: {}", storeName);
    }
    @Override
    public void onRestoreEnd(  ②
        TopicPartition topicPartition,
        String storeName, l
        ong totalRestored) {
        log.info("Restore complete for the following state store: {}", storeName);
    }
    @Override
    public void onBatchRestored( ③
        TopicPartition topicPartition,
        String storeName,
        long batchEndOffset,
        long numRestored) {
        // this is very noisy. don't log anything
    }
}
```

① onRestoreStart 메소드는 상태 재초기화 시작에 호출된다. startingOffset 파라미터가 특별히 관심 대상인데 전체 상태가 재생되어야 하는지를 나타내기 때문이다 (이는 가장 영향이 큰 재초기화

유형으로 인메모리 저장소를 사용하거나 영구 저장소를 사용하고 이전 상태가 손실될 때 발생한다). startingOffset이 0으로 설정된 경우 완전한 재초기화가 요구된다. 0 보다 큰 값인 경우에는 부분적인 복구만이 필요하다.

② onRestoreEnd 메소드는 복구 완료 시 호출된다.

③ onBatchRestored 메소드는 단일 레코드 배치가 복구될 때 호출된다. 최대 배치 사이즈는 MAX_POLL_RECORED 설정과 동일하다. 이 메소드의 경우 잠재적으로 여러 번 호출될 수 있지만 복구 프로세스를 느리게 할 수 있기 때문에 이 메소드로 동기성 처리를 할 때는 주의를 하기 바란다. 일반적으로 이 메소드로 어떤 것도 하지 않는다 (로깅이 극히 noisy하더라도).

내장 지표

카프카 스트림즈 애플리케이션 모니터링에 대한 논의는 12 장으로 미룰 것이다. 그러나 카프카 스트림즈가 일련의 내장 JMX 지표를 포함하고 있음을 주목하는 것은 중요하다. 많은 지표들이 상태 저장소와 관련이 있다.

예를 들어 특정 상태 저장소 연산 및 쿼리 (예, get, put, delete, all, range) 속도, 연산에 대한 평균 및 최대 실행 시간과 억제 버퍼의 크기에 액세스할 수 있다. 또한 RocksDB 지원 저장소에 대해 많은 지표가 존재하며 바이트 수준에서 I/O 트래픽을 살펴볼 때 매우 유용한 bytes-written-rate와 bytes-read-rate을 예로 들 수 있다.

이들 지표의 세부 분석은 컨플루언트의 모니터링 문서에서 찾을 수 있다. 실제로 알림 목적으로 애플리케이션의 건전도 (예, 컨슈머 래그)에 대해 보다 고수준의 측정치를 사용하고 있으며 특정 문제해결 시나리오에 대해 이러한 세부 상태 저장소 지표를 갖는 것은 매우 좋다.

상호대화형 쿼리

카프카 스트림즈 버전 2.5 이전에는 리밸런스가 상호대화형 쿼리를 사용해 상태를 보여주는 애플리케이션에 대해 특히 고통스러웠다. 이전 버전의 라이브러리에서 오프라인 또는 파티션 리밸런싱은 고장난 상태 저장소에 대해 상호대화형 쿼리를 하도록 할 것이다. 정상 리밸런스 (예, 롤링 업데이트)라도 가용성 문제를 야기할 수 있기 때문에 고가용성을 필요로 하는 마이크로 서비스의 경우 거래 차단기였다.

그라나 카프카 스트림즈 2.5부터 새롭게 이관된 상태 저장소가 재초기화되는 중이라도 대기 복제본이 오래된 결과(stale results)를 나타내도록 사용될 수 있다. 이는 애플리케이션이 리밸런싱 상태로 들어가더라도 API를 매우 고가용한 상태로 유지시킨다. 예제 4-11에서 상태 저장소 내 해당 키의 메타 데이터 검색 방법을 배웠음을 주목하기 바란다. 최초 예에서 우리는 활성 카프카 스트림즈 인스턴스를 추출하였다:

```
KeyQueryMetadata metadata =
    streams.queryMetadataForKey(storeName, key, Serdes.String().serializer()); ①
```

```
String remoteHost = metadata.activeHost().host(); ②
int remotePort = metadata.activeHost().port(); ③
```

① 지정된 키에 대한 메타데이터를 얻는데 존재하는 경우 지정된 키가 존재해야 하는 호스트 및 포트 쌍을 포함한다.

② 활성 카프카 스트림즈 인스턴스의 호스트네임을 추출한다.

③ 활성 카프카 스트림즈 인스턴스의 포트를 추출한다.

버전 2.5에서 다음 코드를 사용하여 대기 호스트를 검색할 수 있다:

```
KeyQueryMetadata metadata = s
    treams.queryMetadataForKey(storeName, key, Serdes.String().serializer()); ①
if (isAlive(metadata.activeHost())) { ②
    // route the query to the active host }
else {
    // route the query to the standby hosts
    Set standbys = metadata.standbyHosts(); ③
}
```

① 해당 키에 대해 활성 및 대기 호스트 모두를 얻기 위해 KafkaStreams.queryMetadataForKey 메소드를 사용한다.

② 활성 호스트가 살아있는지를 알기 위해 검사한다. 여러분 스스로 이를 구현해야 할 것이며 아마도 애플리케이션의 현재 상태를 나타내기 위해 상태 리스너와 RPC 서버에 해당 API 엔드 포인트를 추가할 수도 있다. 애플리케이션이 실행 중인 상태일 때마다 isAlive는 true여야 한다.

③ 활성 호스트가 살아있지 않은 경우 복제된 상태 저장소에 쿼리할 수 있도록 대기 호스트를 검색한다. 비고: 대기 호스트가 설정되어 있지 않은 경우 이 메소드는 빈 집합을 반환할 것이다.

위에서 볼 수 있듯이 대기 복제본에 쿼리할 수 있는 능력은 활성 인스턴스가 다운되었거나 쿼리에 응답할 수 없을 때라도 애플리케이션이 매우 고가용성임을 보장한다. 이것으로 리밸랜스의 영향을 완화하는 방법에 대한 논의를 마치며 다음으로 맞춤형 상태 저장소를 논의할 것이다.

맞춤형 상태 저장소

여러분의 맞춤형 상태 저장소를 구현하는 것도 가능한다. 이를 위해 StateStore 인터페이스를 구현해야 한다. 이를 직접 구현하거나 또는 KeyValueStore, WIndowStore 또는 SessionStore와 같은 고수준 인터페이스 중 하나를 사용할 수 있으며, 이는 저장소가 어떤 목적으로 사용할지에 대해 특정적인 추가 인터페이스 메소드를 추가한다[12].

---

[12] 예를 들어 KeyValueStore 인터페이스는 기본 저장소가 스토리지 엔진에 키-값 쌍을 쓸 필요가 있음을 알기 때문에 void put(K key, V value) 메소드를 추가하고 있다.

StateStore 인터페이스 외에 맞춤형 저장소의 새로운 인스턴스 생성 로직을 포함할 StoreSupplier 인터페이스도 구현하길 원할 것이다. 내장 RocksDB 기반 상태 저장소의 성능 특성과 일치시키는 것은 어렵기 때문에 일반적으로 매우 지루하고 에러가 발생하기 쉬운 맞춤형 저장소 구현 작업을 진행하는 것인 반드시 필요치는 않다. 이러한 이유와 매우 기본적인 맞춤형 저장소를 구현하는데 필요한 방대한 양의 코드를 고려한다면 Github 상의 맞춤형 저장소의 일부 예 중 하나를 당신에게 알려줄 것이다.

마지막으로 맞춤형 저장소를 구현하기로 결정했다면 네트워크 호출을 필요로 하는 어떤 저장 솔루션이라도 성능에 큰 영향을 미칠 수 있음을 알기 바란다. RocksDB 또는 로컬 인메모리 저장소가 좋은 이유 중 하나는 스트림 태스크와 함께 배치되기 때문이다. 물론 여러분의 마일리지는 프로젝트 요건에 기반하여 변할 것이며 따라서 궁극적으로 성능 목표를 사전에 정의하고 이에 따라 상태 저장소를 선택하기 바란다.

요약

여러분은 상태 저장소가 내부적으로 카프카 스트림즈에 의해 어떻게 관리되는지 그리고 스테이트풀 애플리케이션이 시간이 지나도 원활히 실행됨을 보장하기 위해 개발자로서 사용가능한 옵션에 대해 깊이 이해하였다. 이는 툼스톤, 공격적 토픽 압축 및 상태 저장소에서 오래된 데이터를 삭제하는 (따라서 상태 재초기화 시간을 줄이는) 다른 기법의 사용을 포함한다. 또한 대기 복제본을 사용함으로써 스테이트풀 태스크에 대한 장애 조치 시간을 줄이고 리밸런스가 일어나도 애플리케이션을 매우 고가용하게 유지할 수 있다. 마지막으로 영향이 큰 리밸런싱의 경우 정적 멤버십을 사용하여 어느 정도 방지할 수 있으며 증분적 협업 리밸런싱이라는 개선된 리밸런스 프로토콜을 지원하는 카프카 스트림즈 버전을 사용함으로써 영향을 최소화할 수 있다.