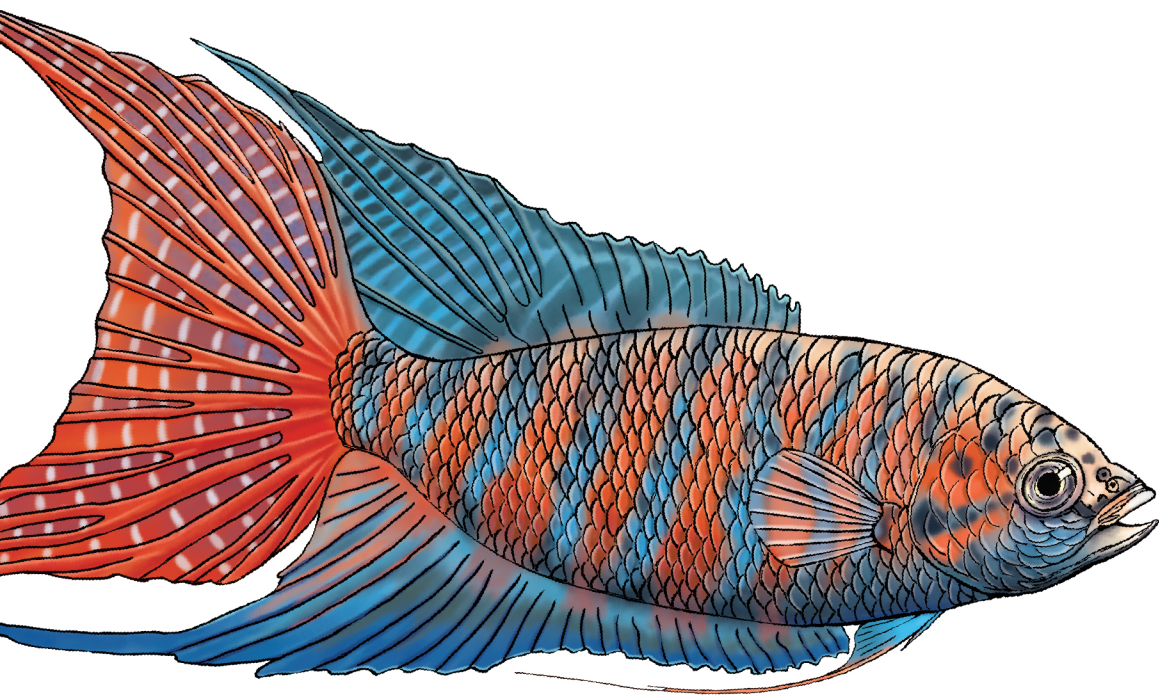


O'REILLY®

Kafka Streams и ksqlDB

данные в реальном времени



Митч Сеймур
Предисловие Джей Крепс

Mastering Kafka Streams and ksqlDB

Building Real-Time Data Systems by Example

Mitch Seymour

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

Kafka Streams и ksqlDB

данные в реальном времени

Митч Сеймур



Санкт-Петербург • Москва • Минск

2023

ББК 32.973.233.02
УДК 004.042
С28

Сеймур Митч

- C28 Kafka Streams и ksqlDB: данные в реальном времени. — СПб.: Питер, 2023. — 432 с.: ил. — (Серия «Библиотека программиста»).
ISBN 978-5-4461-3945-3

Работа с неограниченными и быстрыми потоками данных всегда была сложной задачей. Но Kafka Streams и ksqlDB позволяют легко и просто создавать приложения потоковой обработки. Из книги специалисты по обработке данных узнают, как с помощью этих инструментов создавать масштабируемые приложения потоковой обработки, перемещающие, обогащающие и преобразующие большие объемы данных в режиме реального времени.

Митч Сеймур, инженер службы обработки данных в Mailchimp, объясняет важные понятия потоковой обработки на примере нескольких любопытных бизнес-задач. Он рассказывает о достоинствах Kafka Streams и ksqlDB, чтобы помочь вам выбрать наиболее подходящий инструмент для каждого уникального проекта потоковой обработки. Для разработчиков, не пишущих код на Java, особенно ценным будет материал, посвященный ksqlDB.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.973.233.02
УДК 004.042

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1492062493 англ.

Authorized Russian translation of the English edition of Mastering Kafka Streams and ksqlDB, ISBN 9781492062493 © 2021 Mitch Seymour
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-3945-3

© Перевод на русский язык ООО «Прогресс книга», 2022
© Издание на русском языке, оформление ООО «Прогресс книга», 2022
© Серия «Библиотека программиста», 2022

Краткое содержание

https://t.me/it_boooks

Предисловие	15
Введение	17

ЧАСТЬ I. KAFKA

Глава 1. Краткое введение в Kafka.....	26
--	----

ЧАСТЬ II. БИБЛИОТЕКА KAFKA STREAMS

Глава 2. Начало работы с Kafka Streams	48
Глава 3. Обработка без сохранения состояния	86
Глава 4. Обработка с сохранением состояния.....	119
Глава 5. Окна и время.....	167
Глава 6. Расширенное управление состоянием.....	201
Глава 7. Processor API	231

ЧАСТЬ III. KSQLDB

Глава 8. Знакомство с ksqlDB.....	268
Глава 9. Интеграция данных в ksqlDB.....	295
Глава 10. Основы потоковой обработки с ksqlDB	314
Глава 11. Продвинутая обработка потоков с ksqlDB	345

ЧАСТЬ IV. ПУТЬ К ПРОМЫШЛЕННОЙ ЭКСПЛУАТАЦИИ

Глава 12. Тестирование, мониторинг и развертывание	384
Приложение А. Настройка Kafka Streams.....	417
Приложение Б. Настройка ksqlDB	424
Об авторе	429
Иллюстрация на обложке.....	430

Оглавление

Предисловие	15
Введение	17
Кому адресована книга.....	18
Структура издания.....	19
Исходный код.....	20
Версия Kafka Streams	20
Версия ksqldb.....	20
Условные обозначения.....	21
Использование исходного кода примеров.....	22
Благодарности.....	22
От издательства.....	24

ЧАСТЬ I. КАФКА

Глава 1. Краткое введение в Kafka.....	26
Модель взаимодействия.....	27
Как хранятся потоки.....	31
Темы и разделы	34
События.....	36
Кластер и брокеры.....	37
Группы потребителей.....	39
Установка Kafka	41
Hello, Kafka	43
Заключение.....	45

ЧАСТЬ II. БИБЛИОТЕКА KAFKA STREAMS

Глава 2. Начало работы с Kafka Streams.....	48
Экосистема Kafka	48
До появления Kafka Streams.....	49
Рождение Kafka Streams	51

Обзор функционала	52
Эксплуатационные характеристики	53
Масштабируемость	54
Надежность	55
Удобство сопровождения	55
Сравнение с другими системами	56
Модель развертывания	56
Модель обработки	56
Каппа-архитектура	57
Сценарии использования	59
Топология обработчиков	61
Субтопологии	63
Обработка вглубь	64
Преимущества программирования потоков данных	66
Задачи и потоки выполнения	67
Высокоуровневый DSL и низкоуровневый API	
узлов-обработчиков	70
Начало практической работы: Hello, Streams	71
Настройка проекта	72
Создание проекта	72
Добавление зависимости Kafka Streams	74
Вариант на базе DSL	75
API узлов-обработчиков	77
Потоки данных и таблицы	80
Потоково-табличный дуализм	83
KStream, KTable, GlobalKTable	84
Заключение	85
Глава 3. Обработка без сохранения состояния	86
Обработка с сохранением и без сохранения состояния	87
Обработка потока твитов	88
Настройка проекта	90
Добавление узла-источника KStream	90
Сериализация/десериализация	94
Пользовательская версия класса Serdes	95
Определение классов данных	96

Пользовательский десериализатор.....	97
Пользовательский сериализатор.....	98
Класс Serdes для твитов.....	99
Фильтрация данных.....	100
Ветвление данных.....	102
Перевод твитов.....	104
Слияние потоков.....	106
Обогащение твитов.....	107
Класс данных Avro.....	107
Анализ тональности.....	109
Сериализация данных Avro.....	111
Сериализация формата Avro без сохранения схемы в реестре.....	112
Сериализация формата Avro с сохранением схемы в реестре.....	113
Добавление узла-приемника.....	114
Запуск кода.....	115
Эмпирическая проверка.....	115
Заключение.....	118
Глава 4. Обработка с сохранением состояния.....	119
Преимущества операций с сохранением состояния.....	120
Обзор методов.....	121
Хранилища состояний.....	122
Общие характеристики.....	123
Постоянные хранилища и хранилища в оперативной памяти.....	125
Список лидеров видеоигр.....	126
Настройка проекта.....	128
Модели данных.....	129
Добавление узлов-источников.....	130
KStream.....	131
KTable.....	132
GlobalKTable.....	133
Абстракции потока и таблиц.....	135
Соединение.....	135
Операторы соединения.....	137
Типы соединений.....	137

Совместное секционирование.....	138
Интерфейс ValueJoiner	141
Соединение KStream и KTable	143
Соединение KStream и GlobalKTable	145
Группировка записей.....	145
Группировка потоков.....	146
Группировка таблиц	147
Агрегирование	147
Агрегирование потоков	148
Агрегирование таблиц	151
Объединение фрагментов кода.....	152
Интерактивные запросы.....	154
Материализованные хранилища	154
Доступ на чтение из хранилища состояний	155
Запросы к неоконным хранилищам «ключ — значение»	156
Локальные запросы.....	158
Удаленные запросы	159
Заключение	166
Глава 5. Окна и время.....	167
Приложение для контроля состояния пациентов	168
Настройка проекта	171
Модели данных	171
Семантики времени	172
Экстракторы отметок времени	174
Встроенные экстракторы отметок времени	175
Свои собственные экстракторы отметок времени	177
Регистрация потоков со своими экстракторами отметок времени.....	178
Оконная обработка потоков	179
Типы окон	179
Выбор типа окна.....	184
Оконное агрегирование.....	184
Вывод результатов оконной обработки	186
Период отсрочки	188
Подавление	189

Фильтрация и изменение ключей оконных таблиц KTable	192
Оконные соединения	193
Потоки данных, управляемые временем	194
Приемник предупреждений	196
Запрос оконных хранилищ «ключ — значение»	197
Заключение	199
Глава 6. Расширенное управление состоянием	201
Организация хранилища состояний на диске	202
Отказоустойчивость	204
Темы журналов изменений	204
Резервные реплики	207
Перебалансировка: враг состояния (хранилища)	207
Предотвращение миграции состояния	209
Закрепленное назначение	209
Статическое членство	212
Уменьшение влияния перебалансировки	213
Пошаговая кооперативная перебалансировка	214
Управление размером состояния	216
Исключение повторных операций записи с помощью кэширования	222
Мониторинг хранилища состояний	224
Обработка событий изменения состояния	224
Обработка событий восстановления хранимого состояния	226
Встроенные метрики	227
Интерактивные запросы	228
Нестандартные хранилища состояний	229
Заключение	230
Глава 7. Processor API	231
Когда использовать Processor API	232
Служба цифровых двойников IoT	233
Настройка проекта	236
Модели данных	237
Добавление узлов-источников	239
Добавление узлов-обработчиков без состояния	241

Создание узлов без состояния.....	242
Создание узлов с состоянием.....	245
Периодическое выполнение функций с Punctuate	249
Доступ к метаданным записей	251
Добавление узла-приемника	253
Интерактивные запросы.....	254
Все вместе.....	255
Объединение Processor API и DSL	258
Обработчики и преобразователи.....	259
Все вместе: реорганизация.....	264
Заключение	266

ЧАСТЬ III. KSQLDB

Глава 8. Знакомство с ksqlDB.....	268
Что такое ksqlDB.....	269
Когда следует использовать ksqlDB.....	270
Эволюция базы данных нового типа	272
Интеграция с Kafka Streams.....	273
Интеграция с Kafka Connect.....	276
Сравнение ksqlDB с традиционной базой данных SQL.....	278
Сходства.....	278
Отличия	280
Архитектура	282
Сервер ksqlDB	282
Клиенты ksqlDB.....	285
Режимы развертывания	286
Интерактивный режим.....	287
Автономный режим.....	288
Учебный проект	289
Установка ksqlDB.....	289
Запуск сервера ksqlDB.....	290
Предварительное создание тем	291
Использование интерфейса командной строки ksqlDB CLI.....	291
Заключение	294

Глава 9. Интеграция данных в ksqlDB.....	295
Обзор Kafka Connect	296
Внешняя и встроенная интеграция с Connect	297
Внешняя интеграция.....	298
Встроенная интеграция.....	299
Настройка рабочих процессов Connect	300
Конвертеры и форматы сериализации	302
Учебный проект	304
Установка коннекторов	305
Создание экземпляров коннекторов в ksqlDB	306
Вывод списка коннекторов.....	308
Получение описаний коннекторов.....	309
Удаление коннекторов.....	310
Проверка коннектора-источника	310
Взаимодействие с кластером Kafka Connect напрямую.....	311
Анализ управляемых схем	312
Заключение	313
Глава 10. Основы потоковой обработки с ksqlDB	314
Учебный проект: мониторинг изменений в Netflix.....	315
Настройка проекта	317
Исходные темы	318
Типы данных	319
Пользовательские типы.....	320
Коллекции.....	322
Создание исходных коллекций.....	323
Оператор WITH.....	325
Работа с потоками и таблицами.....	326
Вывод списка потоков и таблиц.....	326
Получение описаний потоков и таблиц.....	327
Изменение потоков и таблиц.....	328
Удаление потоков и таблиц.....	329
Простые запросы.....	329
Вставка значений.....	330
Простая выборка (временные push-запросы)	331

Проекция.....	333
Фильтрация	333
Подстановочные знаки	334
Развертывание/упрощение сложных вложенных структур	335
Условные выражения	336
COALESCE.....	337
IFNULL	337
Оператор CASE.....	337
Запись результатов обратно в Kafka (постоянные запросы).....	338
Создание производных коллекций.....	338
Все вместе.....	342
Заключение	344
Глава 11. Продвинутая обработка потоков с ksqlDB	345
Настройка проекта	346
Инициализация окружения из файла SQL	346
Обогащение данных.....	348
Соединения	349
Оконные соединения	354
Агрегирование	357
Основы агрегирования.....	358
Оконное агрегирование.....	360
Материализованные представления	366
Клиенты.....	368
Pull-запросы	368
curl	370
Push-запросы.....	372
Push-запросы из curl	372
Функции и операторы	373
Операторы.....	373
Вывод списка доступных функций	373
Получение описаний функций	374
Создание своих функций	376
Дополнительная информация о пользовательских функциях ksqlDB.....	381
Заключение	382

ЧАСТЬ IV. ПУТЬ К ПРОМЫШЛЕННОЙ ЭКСПЛУАТАЦИИ

Глава 12. Тестирование, мониторинг и развертывание	384
Тестирование.....	385
Тестирование запросов ksqldb.....	385
Тестирование приложений Kafka Streams.....	388
Поведенческие тесты	395
Оценка производительности.....	398
Оценка производительности кластера Kafka.....	400
Заключительные замечания о тестировании.....	402
Мониторинг	402
Виды мониторинга	403
Извлечение метрик JMX.....	403
Развертывание.....	406
Контейнеры ksqldb.....	407
Контейнеры Kafka Streams.....	408
Оркестрация контейнеров.....	410
Операции	411
Повторная обработка данных в приложении Kafka Streams.....	411
Ограничение скорости вывода приложением	413
Обновление Kafka Streams	414
Обновление ksqldb.....	415
Заключение	416
Приложение А. Настройка Kafka Streams	417
Управление конфигурацией	417
Конфигурационные свойства	418
Конфигурационные свойства потребителей.....	423
Приложение Б. Настройка ksqldb	424
Параметры запросов	425
Параметры сервера.....	426
Настройки безопасности	428
Об авторе	429
Иллюстрация на обложке.....	430

Предисловие

Бизнес все чаще строится вокруг событий — информации о происходящем в компании, поступающей в режиме реального времени. Но какую именно инфраструктуру можно считать подходящей для полного использования информации о событиях? Это вопрос, над которым я задумался в 2009 году, запустив проект Apache Kafka в LinkedIn. В 2014 году я стал соучредителем Confluent, чтобы дать окончательный ответ на него. Платформе потоковой передачи событий требуется не только возможность хранения и доступа к дискретным событиям, но также механизм подключения к множеству внешних систем и поддержка глобального управления схемами, метриками и мониторингом. А самой важной, пожалуй, является поддержка потоковой обработки — возможность выполнения непрерывных вычислений с бесконечными потоками данных. Без этого платформа потоковой передачи событий просто неполноценна.

Сейчас как никогда потоковая обработка играет ключевую роль во взаимодействиях компаний с внешним миром. В 2011 году Марк Андриссен (Marc Andreessen) написал статью под названием *Why Software Is Eating the World*. Основная ее идея: любой процесс, который можно воплотить в программе, в конечном счете будет воплощен. Марк оказался провидцем: программное обеспечение проникло во все мыслимые сферы.

Но менее очевидный и более важный результат — успех бизнеса стал все больше зависеть от программного обеспечения. Другими словами, ключевые процессы, образующие основу бизнеса — от создания продукта до взаимодействия с клиентами и предоставления услуг, — все чаще определяются, отслеживаются и выполняются с помощью программного обеспечения. Что изменилось благодаря внедрению этой всеобъемлющей динамики? То, что программное обеспечение в таком новом мире вряд ли будет напрямую взаимодействовать с человеком. Более вероятно, что оно будет программно запускать действия или реагировать на действия других частей программного обеспечения, которые непосредственно поддерживают бизнес.

Возникает вопрос: насколько хорошо подходят для этого развивающегося мира традиционные архитектуры приложений, основанные на существующих базах данных? Практически все базы данных, от самых известных реляционных баз данных до новейших хранилищ пар «ключ — значение», следуют парадигме

пассивного хранения информации, когда база данных ожидает команд, чтобы что-то извлечь или изменить. Эта парадигма ориентирована на приложения, взаимодействующие с человеком: пользователь осуществляет мониторинг интерфейса и инициирует действия, которые преобразуются в запросы к базе данных. Думается, что это только половина проблемы хранения данных. Пассивная составляющая хранения должна быть дополнена способностью реагировать на события и обрабатывать их.

События и потоковая обработка — ключ к успеху в этом новом мире. События образуют непрерывный поток данных в бизнесе, а потоковая обработка автоматически выполняет код в ответ на изменения на любом уровне детализации, делая это в соответствии с накопленной информацией обо всех изменениях, которые произошли до этого. Современные системы потоковой обработки, такие как *Kafka Streams* и *ksqlDB*, упрощают создание приложений для мира, говорящего на языке программного обеспечения.

В своей книге Митч Сеймур (*Mitch Seymour*) доходчиво описывает эти передовые системы, исходя из фундаментальных принципов. Он рассматривает понятия, лежащие в их основе, подробно объясняет нюансы работы каждой системы и приводит практические примеры использования потоковой обработки в реальном мире. Важность этой парадигмы программирования трудно переоценить, и освоение *Kafka Streams* и *ksqlDB* помогут добиться успеха в ее применении на практике.

*Джей Крепс (Jay Kreps),
один из создателей Apache Kafka,
сооснователь и генеральный директор
Confluent*

Введение

Инженеры и специалисты в области обработки данных, к которым я отношу и себя, никогда не испытывали недостатка в технологиях, конкурирующих за наше внимание. Что бы мы ни делали: просматривали любимые форумы или презентации на технических конференциях, выискивали новости информационных технологий или читали технические блоги — всегда найдется много достойного нашего внимания, порой голова идет кругом.

Но, найдя тихий уголок, отодвинув весь шум на задний план и поразмыслив не спеша, мы начинаем отличать закономерности от белого шума. Мы живем в эпоху взрывного роста объемов данных, и в помощь нам для их обработки в больших масштабах создано множество технологий. Нам говорят, что это современные решения для современных проблем, мы сидим и обсуждаем «большие данные», как будто это авангардная идея, хотя на самом деле сосредоточенность на их объеме — это только половина дела.

Технологии, решающие одну лишь проблему большого объема данных, как правило, основаны на методах пакетной обработки данных. Они предполагают применение задания к некоторому массиву данных, накопившихся за определенный период времени. В каком-то смысле это похоже на попытку выпить океан одним глотком. С существующими вычислительными мощностями и парадигмами некоторым технологиям действительно удастся добиться желаемого, хотя и за счет длительной задержки.

Однако современные данные обладают еще одним свойством, на котором мы сосредоточимся в книге: они перемещаются по сетям устойчивыми и бесконечными потоками. Технологии, которые мы рассмотрим далее, — Kafka Streams и ksqlDB — специально созданы для обработки таких непрерывных потоков данных в режиме реального времени. Они обеспечивают огромные конкурентные преимущества по сравнению с разнообразными технологиями, пытающимися «выпить океан». В конце концов, многие бизнес-задачи зависят от времени, и если требуется обрабатывать и преобразовывать данные по мере их поступления, то Kafka Streams и ksqlDB помогут организовать это легко и эффективно.

Изучение Kafka Streams и ksqlDB — еще и отличный способ познакомиться с более широкими понятиями, связанными с обработкой потоков данных, включая моделирование данных разными способами (в форме потоков и таблиц), применение преобразований без сохранения состояния, использование локального состояния для сложных операций (соединения, агрегирования), представление различных семантик времени, группировка данных во временные сегменты/окна и многое другое. Другими словами, знание Kafka Streams и ksqlDB поможет вам различать и оценивать решения обработки потоков, которые существуют в настоящее время и могут появиться в будущем.

Я рад поделиться с вами своими знаниями этих технологий, повлиявших на мою карьеру и помогавших мне в свое время решать технологические задачи, которые, как тогда казалось, были мне не по зубам. Фактически к тому моменту, когда вы закончите читать это предложение, одно из моих приложений Kafka Streams успеет обработать девять миллионов событий. Чувство, которое вы испытаете, оснатив бизнес реальной ценностью и не потратив много времени на поиск решения, вдохновит вас на длительную и основательную работу с этими технологиями. А краткие и выразительные языковые конструкции сделают процесс создания этих ценностей похожим больше на искусство, чем на рутинный труд.

И так же, как в любом другом виде искусства, человеческая природа требует поделиться своими достижениями. Поэтому можете считать эту книгу сборником моих любимых произведений из области потоковой обработки: Kafka Streams и ksqlDB, том 1.

Кому адресована книга

Эта книга адресована специалистам по обработке данных, желающим научиться создавать масштабируемые приложения потоковой обработки для перемещения и преобразования больших объемов данных в режиме реального времени. Подобные умения часто необходимы для поддержки интеллектуальной обработки данных, аналитических конвейеров, обнаружения угроз, обработки событий и многого другого. Специалисты по данным и аналитики, занимающиеся анализом потоков данных в реальном режиме времени и желающие усовершенствовать свои навыки, тоже смогут почерпнуть немало полезного из этой книги. В ней автору удалось отойти от привычной пакетной обработки, которая обычно доминировала в этих областях. Предварительный опыт работы с Apache Kafka не требуется, хотя некоторое знакомство с языком программирования Java облегчит знакомство с Kafka Streams.

Структура издания

Эта книга состоит из 12 глав.

- Глава 1 содержит введение в Kafka и инструкции по запуску кластера Kafka с одним узлом.
- Глава 2 содержит введение в Kafka Streams, начиная с общих сведений и обзора архитектуры и до описания шагов по запуску простого приложения Kafka Streams.
- Главы 3 и 4 посвящены обсуждению операторов высокоуровневого предметно-ориентированного языка (Domain-Specific Language, DSL) Kafka Streams с сохранением и без сохранения состояния. Обе главы наглядно демонстрируют использование этих операторов для решения интересной бизнес-задачи.
- Глава 5 содержит обсуждение роли времени в приложениях потоковой обработки данных, она демонстрирует, как использовать окна для выполнения сложных операций с сохранением состояния, включая оконные операции соединения и агрегирования данных. Здесь показаны ключевые концепции на примере прогностической медицины.
- Глава 6 описывает внутренние особенности механизмов обработки с сохранением состояния, а также дает некоторые советы по работе с приложениями Kafka Streams с состоянием.
- Глава 7 посвящена низкоуровневому программному интерфейсу Kafka Streams Processor API, который можно использовать для планирования периодических функций, а также для доступа к состоянию приложения и метаданным записей. Эта глава основана на примерах использования Интернета вещей (Internet of Things, IoT).
- Глава 8 содержит введение в `ksqlDB` и обсуждает историю и архитектуру этой технологии. Здесь вы увидите, как установить и запустить экземпляр сервера `ksqlDB`, и познакомитесь с некоторыми приемами работы с интерфейсом командной строки `ksqlDB`.
- Глава 9 описывает функции интеграции данных `ksqlDB`, поддерживаемые в Kafka Connect.
- Главы 10 и 11 подробно рассматривают диалект `ksqlDB SQL`, демонстрируют приемы работы с различными типами коллекций, выполнения запросов на передачу и извлечение данных и многое другое. Представление идей в этой

главе будет основано на сценарии использования Netflix: отслеживании изменений в различных шоу и фильмах и предоставлении доступа к этим изменениям другим приложениям.

- Глава 12 представляет сведения, необходимые для развертывания приложений Kafka Streams и ksqlDB в реальных условиях. Здесь же вы узнаете о мониторинге, тестировании и контейнеризации приложений.

Исходный код

Исходный код примеров для этой книги доступен на GitHub: <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb>.

Инструкции по сборке и запуску каждого примера вы найдете в том же репозитории.

Версия Kafka Streams

На момент написания книги последней была версия Kafka Streams 2.7.0. Именно она используется здесь, хотя многие примеры будут работать и с более старыми или новыми версиями библиотеки Kafka Streams. Будут приложены все силы, чтобы своевременно обновлять исходный код при включении критических изменений в новые версии библиотеки и размещать эти обновления в отдельной ветке (например, `kafka-streams-2.8`).

Версия ksqlDB

Когда я писал эту книгу, последней была версия ksqlDB 0.14.0. Совместимость со старыми и новыми версиями ksqlDB менее вероятна, чем в случае Kafka Streams, из-за постоянного и быстрого развития этой технологии и отсутствия основной версии (например, 1.0) на момент публикации книги. Будут приложены все силы, чтобы своевременно обновлять исходный код при включении критических изменений и размещать эти обновления в отдельной ветке (например, `ksqldb-0.15`). Однако для опробования примеров из этой книги рекомендуется не использовать версии старше 0.14.0.

Условные обозначения

В этой книге используются следующие условные обозначения.

Курсив

Курсивом выделены новые термины или важные понятия.

Моноширинный шрифт

Используется для листингов программ, а также внутри абзацев для обозначения таких элементов, как переменные и функции, базы данных, типы данных, переменные среды, операторы и ключевые слова, имена файлов и их расширения.

Моноширинный полужирный шрифт

Служит для обозначения команды или другого текста, который пользователь должен ввести самостоятельно.

Моноширинный курсив

Обозначает текст, который должен быть заменен значениями, введенными пользователем, или значениями, определяемыми контекстом.

Шрифт без засечек

Используется для обозначения URL, адресов электронной почты, названий кнопок и других элементов интерфейса, каталогов.



Эта пиктограмма указывает на совет или предложение.



Эта пиктограмма указывает на общее примечание.



Эта пиктограмма указывает на предупреждение.

Использование исходного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb>.

Если у вас возникнут вопросы технического характера по использованию примеров кода, направляйте их по электронной почте на адрес bookquestions@oreilly.com.

В общем случае все примеры кода из книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Если вы разрабатываете программу и используете в ней несколько фрагментов кода из книги, вам не нужно обращаться за разрешением. Но для продажи или распространения примеров из книги вам потребуется разрешение от издательства O'Reilly. Вы можете отвечать на вопросы, цитируя данную книгу или примеры из нее, но для включения существенных объемов программного кода из книги в документацию вашего продукта потребуется разрешение.

Мы рекомендуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

За получением разрешения на использование значительных объемов программного кода из книги обращайтесь по адресу permissions@oreilly.com.

Благодарности

Прежде всего хочу поблагодарить свою жену Элиз и дочь Изабель. Написание такого объемного труда потребовало огромных временных затрат, и ваше терпение и поддержка на протяжении всего процесса мне очень помогли. Как бы мне ни нравилось писать эту книгу, я очень скучал по вам обоим и с нетерпением жду новых встреч с дочкой.

Хочу также поблагодарить своих родителей, Энджи и Гая, за то, что научили меня ценить труд, каким бы тяжелым он ни был, и за то, что являлись бесконечным источником вдохновения. Ваша поддержка помогала мне преодолевать трудности на протяжении многих лет, и я бесконечно благодарен вам обоим.

Успешный результат моих усилий не был бы достигнут без людей, нашедших время на анализ содержания текста и давших бесценные отзывы и советы:

это Матиас Дж. Сакс (Matthias J. Sax), Роберт Йокота (Robert Yokota), Нитин Шарма (Nitin Sharma), Рохан Десаи (Rohan Desai), Джефф Блейел (Jeff Bleiel) и Дэнни Эльфанбаум (Danny Elfanbaum). Спасибо вам всем за помощь в создании этой книги, она такая же ваша, как и моя.

Многие учебные примеры основаны на сценариях реального использования, и я в долгу перед всеми членами сообщества, которые без утайки поделились своим опытом работы с Kafka Streams и ksqlDB, будь то конференции, подкасты, блоги или личное общение. Ваш опыт помог написать эту книгу, в которой особое внимание уделяется практическому применению потоковой обработки. В частности, Нитин Шарма предложил к использованию для примеров по ksqlDB идеи, основанные на опыте Netflix, а Рамеш Срингери (Ramesh Sringeri) поделился своим опытом обработки потоков в детской больнице в городе Атланта (США) и вдохновил на создание учебного примера по прогнозной медицине. Спасибо вам обоим.

Особая благодарность Майклу Дрогалису (Michael Drogalis) за то, что поддержал идею написания этой книги. Спасибо за то, что познакомил меня со многими рецензентами моего текста, а также с Джеем Крепсом, любезно согласившимся написать предисловие. Спасибо Еве Бызек (Yeva Byzek) и Биллу Беджеку (Bill Bejeck), установившим высокую планку в реализации подобных книг. Спасибо вам обоим за ваш вклад в этой области.

В своей карьере я встретил много людей, которые помогли мне достичь нынешних высот. Спасибо Марку Конде (Mark Conde), Тому Стэнли (Tom Stanley) и Барри Боудену (Barry Bowden) — они были отличными наставниками и помогли мне стать хорошим инженером. Эрин Фусаро (Erin Fusaro) просто был надежной опорой и точно знал, что сказать, когда я чувствовал себя разбитым. Джастин Исаси (Justin Isasi) постоянно поддерживал меня и отмечал все мои усилия. Шон Сойер (Sean Sawyer) предложил мне несколько лет тому назад попробовать новую штуку под названием Kafka Streams, когда у меня не получалось решить задачу с использованием традиционных технологий. Томас Холмс (Thomas Holmes) и Мэтт Фармер (Matt Farmer) неоднократно делились со мной своим опытом и помогли мне стать хорошим инженером. Спасибо также команде Data Services в Mailchimp за помощь в решении некоторых действительно сложных проблем и за то, что вдохновляли меня своей работой.

Наконец, спасибо моим друзьям и семье, которые не бросают меня, даже когда я пропадаю на месяцы, погружаясь в работу над новым проектом. Спасибо, что остаетесь со мной.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Часть I

Kafka

ГЛАВА 1

Краткое введение в Kafka

https://t.me/it_boooks

Объем данных в мире растет в геометрической прогрессии. По информации Всемирного экономического форума, количество хранимых байтов уже намного превысило количество звезд в наблюдаемой Вселенной.

Говоря о данных, мы, как правило, подразумеваем множество байтов в хранилищах, в реляционных базах или в распределенных файловых системах. Во всех этих случаях данные находятся в *состоянии покоя*. Другими словами, они где-то неподвижно отдыхают, а если возникает необходимость их обработки, запускается какой-то запрос или задача.

Это традиционный способ восприятия данных. И они действительно могут накапливаться в различных местах, но значительно чаще они находятся в движении. Датчики Интернета вещей, медицинские приборы, финансовые системы, программное обеспечение для анализа пользователей и клиентов, журналы приложений и серверов и многое другое генерируют непрерывные потоки данных. Даже где-то осевшие данные перед этим некоторое время путешествовали по сети.

В ситуации, когда речь заходит об обработке в режиме реального времени, мы не можем просто дождаться, пока данные где-то накопятся, и только после этого приступить к созданию запросов и задач. Разумеется, и такие сценарии тоже практикуются, но часто обрабатывать, обогащать и преобразовывать данные требуется в момент их появления. Это значит, что нужен совсем другой подход. Необходима технология, предоставляющая доступ к данным в *состоянии потока*, технология, которая позволяет быстро и эффективно работать с непрерывными и бесконечными потоками. И здесь на помощь приходит Apache Kafka.

Apache Kafka (или просто Kafka) — платформа для приема, хранения и обработки потоков данных. Это крайне интересная технология, и я хотел рассказать вам

про самую захватывающую, на мой взгляд, ее особенность: механизм потоковой обработки. Но, не зная, как функционирует платформа Kafka, невозможно понять, как работают библиотека Kafka Streams и база данных ksqlDB.

Именно поэтому первую главу я посвятил важным концепциям и терминологии, необходимым для понимания дальнейшего материала. Читатели, уже имеющие представление о платформе Kafka, смело могут ее пропустить.

Ниже найдете список тем, которые будут рассмотрены в этой главе.

- Как Kafka упрощает обмен данными между системами.
- Основные компоненты архитектуры Kafka.
- Какая абстракция хранилища наиболее точно моделирует потоки.
- Что обеспечивает надежность хранения данных в Kafka.
- Способы обеспечения высокой доступности и отказоустойчивости на уровне обработки данных.

Завершит главу инструкция по установке и запуску брокера Kafka. Для начала посмотрим на модель взаимодействия между его компонентами.

Модель взаимодействия

Наверное, один из самых распространенных видов взаимодействия между системами — синхронное взаимодействие в рамках модели «клиент — сервер». Под системами в данном контексте подразумеваются приложения, микросервисы, базы данных и все прочие компоненты, отвечающие за чтение и запись данных в сети. Изначально модель «клиент — сервер» предполагает прямую связь между системами, как показано на рис. 1.1.

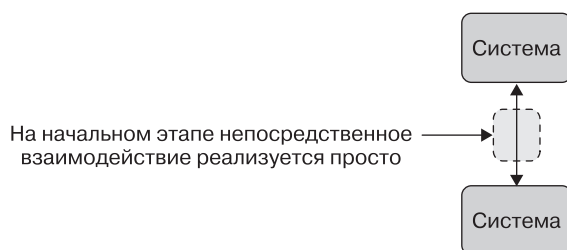


Рис. 1.1. Простое в обслуживании соединение из точки в точку удобно при небольшом количестве систем

В качестве примера вы можете представить приложение, в синхронном режиме запрашивающее данные из базы, или набор микросервисов, напрямую обменивающихся данными.

Но такая модель с трудом масштабируется. Увеличение количества систем порождает запутанную сеть каналов связи, которую сложно проектировать и обслуживать. Рисунок 1.2 демонстрирует, насколько усложняется ситуация даже при относительно небольшом числе систем.

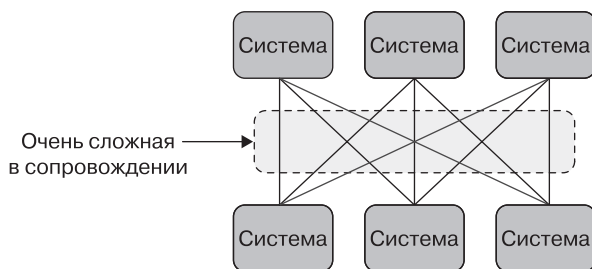


Рис. 1.2. Добавление систем порождает запутанную сеть каналов связи, которую трудно обслуживать

Вот некоторые недостатки модели «клиент — сервер».

- *Сильная связь между системами*, потому что для взаимодействия они должны знать о существовании друг друга. Такая связь затрудняет их поддержку и обновление.
- Синхронное взаимодействие не оставляет права на ошибку, ведь при выходе из строя одной из систем *доставка данных не гарантирована*.
- Каждая система может использовать собственные протоколы связи, варианты масштабирования при повышении нагрузки, стратегии обработки сбоев и другие параметры. В результате иногда приходится обслуживать сразу несколько видов систем, что *осложняет поддержку*, ведь в подобном случае набор приложений уже невозможно рассматривать как «стадо», с ним приходится работать как с «домашними питомцами».
- Принимающую систему легко перегрузить сверх меры, так как скорость поступления новых запросов или данных не контролируется. Без *буферизации* система попадает в зависимость от поведения приложений, которые отправляют к ней запросы.
- Нет четкого представления о том, что передается между системами. В модели «клиент — сервер» основное внимание уделяется *запросам и ответам на них*, а не передаваемым данным, как это должно быть в системах, управляемых данными.

- Взаимодействие *невоспроизводимо*, что затрудняет восстановление состояния системы.

Брокер сообщений Kafka упрощает взаимодействие систем, выступая в качестве централизованного коммуникационного узла (его часто сравнивают с центральной нервной системой). Благодаря этому узлу системы могут отправлять и получать данные, ничего не зная друг о друге. Реализуемый им шаблон передачи сообщений «*производитель — потребитель*» (или pub/sub) дает более простую модель взаимодействия, показанную на рис. 1.3.

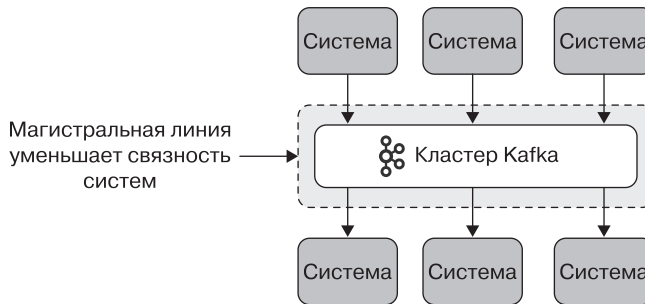


Рис. 1.3. Брокер сообщений Kafka устраняет сложности, возникающие при взаимодействии из точки в точку, выступая в роли хаба

Более подробный рис. 1.4 демонстрирует основные компоненты модели взаимодействия Kafka.

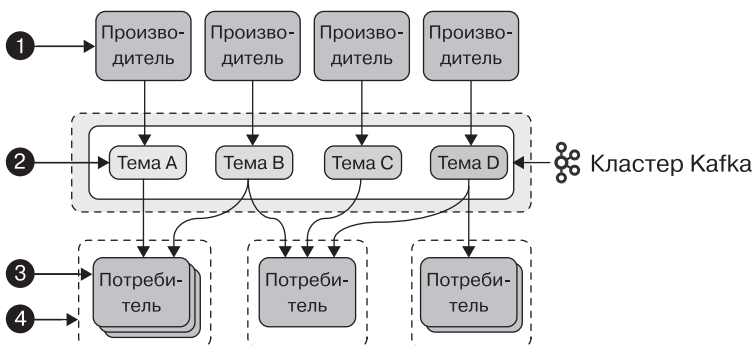


Рис. 1.4. Основные компоненты платформы Kafka

❶ Вместо того чтобы использовать взаимодействие систем напрямую, *производители* (producers) просто публикуют данные в одной или нескольких темах, не заботясь о том, кто будет их читать.

❷ *Темами*, или *топиками* (topics), называются именованные потоки (или каналы) связанных данных, хранящиеся в кластере Kafka. Они играют ту же роль, что и таблицы в базе данных (группируют связанные данные). Однако они не навязывают конкретную схему, а хранят необработанные байты, что делает обращение с ними очень гибким¹.

❸ *Потребители* (consumers) представляют собой процессы, которые читают данные из одной или нескольких тем. Они не общаются напрямую с производителями, а скорее «слушают» данные из любого интересного им потока.

❹ Потребители могут работать в *группе*, распределяя задачи по нескольким процессам.

В модели взаимодействий Kafka основное внимание уделяется потокам данных и легкости чтения и записи этих данных различными процессами. Такая модель имеет следующие преимущества.

- Системы становятся *несвязанными* и более простыми в обслуживании, поскольку для генерации и использования данных им не требуется информация о других системах.
- Асинхронная связь дает *более надежные гарантии доставки*. Вышедший из строя потребитель после возвращения в рабочее состояние просто продолжает работу с прерванного места. В группе задачи вышедшего из строя потребителя передаются другому члену группы.
- Появляется возможность стандартизировать протокол связи (кластеры Kafka используют высокопроизводительный двоичный протокол на базе TCP), а также стратегии масштабирования и механизмы отказоустойчивости, которые определяются группами потребителей. Это позволяет создавать более или менее единообразное программное обеспечение, и в целом оно проще для понимания.
- Потребители обрабатывают данные с доступной им скоростью. Необработанные данные хранятся надежным и отказоустойчивым способом, пока не появится возможность ими заняться. Другими словами, если поток, из которого читает потребитель, превращается в своего рода пожарный шланг, кластер Kafka срабатывает как буфер, предотвращая перегрузку приемника.
- *События* (events) позволяют классифицировать передаваемые данные. События представляют собой фрагменты данных с определенной струк-

¹ Хранящиеся в темах необработанные массивы байтов и их десериализация в структуры более высокого уровня, такие как объекты JSON/записи Avro, будут рассматриваться в главе 3.

турой. Подробно мы поговорим о них чуть ниже в одноименном разделе, а пока важно запомнить: именно они дают возможность сосредоточиться на данных внутри потоков, избавляя нас от необходимости расшифровывать, что именно происходит на коммуникационном уровне, как это приходится делать в модели клиент-сервер.

- Восстановить состояние системы можно в любой момент, просто повторив события внутри темы.

Важное отличие используемой в Kafka модели «производитель — потребитель» от модели «клиент — сервер» заключается в том, что связь в Kafka не двунаправленная, то есть потоки текут только в одну сторону. Если система генерирует данные в одной теме, а обрабатывает (например, обогащает или преобразует) их другая система, обработанные данные требуется записать в новую тему, откуда ими сможет воспользоваться исходный процесс. Такой подход упрощает координацию, но меняет способы взаимодействия.

Исходя из того, что потоки текут в одном направлении и могут иметь несколько производителей, а ниже по течению несколько потребителей, легко спроектировать систему, слушающую все интересные для нее потоки байтов и генерирующую данные, которыми она хочет поделиться с одной или несколькими системами. С темами вы будете много работать в следующих главах (все создаваемые приложения Kafka Streams и `ksqlDB` будут читать и, как правило, делать запись в одну или несколько тем), так что, когда вы завершите чтение книги, все эти вещи станут для вас привычными и обыденными.

Теперь, когда вы знаете, что коммуникационная модель Kafka упрощает взаимодействие систем, а именованные потоки (темы) действуют как каналы передачи данных, подробно рассмотрим, что происходит с потоками на уровне хранения.

Как хранятся потоки

Когда группа инженеров LinkedIn¹ открыла потенциал управляемой потоками данных платформы, встал вопрос: как смоделировать неограниченные и непрерывные потоки данных на уровне хранения?

В конечном счете была выбрана концепция хранилища (<https://oreil.ly/Y2Fe5>), которая уже использовалась такими системами, как традиционные базы данных,

¹ Изначально разработку Kafka вели Джей Крепс (Jay Kreps), Неха Нархеде (Neha Narkhede) и Джун Пао (Jun Rao).

хранилища пар «ключ — значение» и системы контроля версий. Это простой, но мощный *журнал фиксации* (commit log), который дальше я буду называть просто журналом.



Журналами в этой книге называются не *регистрационные журналы приложений*, содержащие информацию о запущенных процессах, а специальная структура данных, подробное описание которой я дам чуть ниже.

Итак, наш журнал представляет собой структуру данных, фиксирующую *упорядоченную последовательность* событий, причем обновление этой структуры возможно *только путем присоединения новых записей* (append-only). В качестве упражнения создадим простой журнал `user_purchases` и заполним его фиктивными данными. Это делается с помощью следующей команды:

```
# создаем файл журнала
touch users.log

# генерируем в журнале четыре фиктивных записи
echo "timestamp=1597373669,user_id=1,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=2,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=3,purchases=1" >> users.log
echo "timestamp=1597373669,user_id=4,purchases=1" >> users.log
```

В журнале мы обнаружим данные четырех пользователей, совершивших по одной покупке:

```
# выводим на экран содержимое журнала
cat users.log

# вывод
timestamp=1597373669,user_id=1,purchases=1
timestamp=1597373669,user_id=2,purchases=1
timestamp=1597373669,user_id=3,purchases=1
timestamp=1597373669,user_id=4,purchases=1
```

Еще раз повторяю, что обновление журнала осуществляется только добавлением записей. Если пользователь с `user_id=1` совершит вторую покупку, в конец журнала будет добавлена новая запись, потому что исходная информация об этом пользователе *не подлежит изменению*:

```
# добавляем в журнал новую запись
echo "timestamp=1597374265,user_id=1,purchases=2" >> users.log

# выводим на экран содержимое журнала
cat users.log
```

```
# вывод
timestamp=1597373669,user_id=1,purchases=1 ❶
timestamp=1597373669,user_id=2,purchases=1
timestamp=1597373669,user_id=3,purchases=1
timestamp=1597373669,user_id=4,purchases=1
timestamp=1597374265,user_id=1,purchases=2 ❷
```

❶ Внесенная в журнал запись не подлежит изменениям. Это означает, что мы не можем редактировать ее, например, чтобы изменить значение счетчика покупок `purchases`.

❷ Для обновления значения счетчика добавим в конец журнала еще одну запись. В результате журнал будет содержать как старую, так и новую записи.

Системе, которая хочет узнать значение счетчика покупок для первого пользователя, достаточно по очереди прочитать записи в журнале. Интересующую ее информацию будет содержать последняя запись для `user_id=1`. Это подводит нас к следующему свойству журналов — упорядоченности.

В приведенном выше выводе журнала можно заметить, что записи упорядочены по временным меткам (первый столбец `timestamp`), но это простое совпадение. Никакой сортировки по временным меткам в Kafka не происходит. На самом деле каждая запись в журнале имеет *фиксированную позицию*. Если вывести содержимое журнала с номерами строк, вы увидите номер этой позиции в первом столбце:

```
# выводим содержимое журнала с номерами строк
cat -n users.log

# вывод
1 timestamp=1597373669,user_id=1,purchases=1
2 timestamp=1597373669,user_id=2,purchases=1
3 timestamp=1597373669,user_id=3,purchases=1
4 timestamp=1597373669,user_id=4,purchases=1
5 timestamp=1597374265,user_id=1,purchases=2
```

Без фиксированного порядка записей каждый процесс мог бы по-своему считывать обновления строки с `user_id=1`, в результате получая разные показатели для количества покупок этого пользователя. Именно упорядоченность журналов обеспечивает детерминированность¹ обработки разными процессами².

¹ Детерминированность означает, что одни и те же входные данные всегда будут порождать одинаковый вывод.

² Вот почему традиционные базы данных используют для репликации журналы. Ведь именно там содержатся сведения обо всех операциях записи в базу. Достаточно *по порядку* обработать их в реплике, чтобы снова гарантированно получить тот же набор данных.

В приведенном выше примере для обозначения позиции каждой записи журнала использовались номера строк, в то время как Kafka рассматривает позицию каждой записи в распределенном журнале как *смещение* (offset). Смещение отсчитывается с 0, как показано на рис. 1.5. Такой подход позволяет группам потребителей при чтении из одного журнала фиксировать свою позицию.

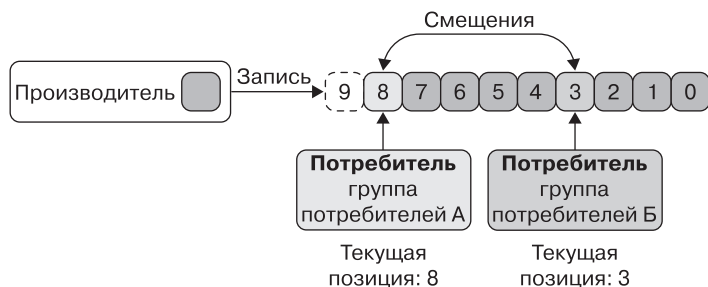


Рис. 1.5. Смещение, которое потребитель прочитал/обработал, позволяет каждой группе потребителей запомнить свою позицию

Надеюсь, рассмотренный выше пример дал вам некоторое представление о способе хранения потоков в Kafka. Теперь можно перейти к изучению конструкций более высокого уровня, таких как темы и разделы.

Темы и разделы

При обсуждении коммуникативной модели Kafka я уже упоминал про такую концепцию, как именованные потоки или темы. Темы дают гибкий подход к хранящимся в них данным и могут быть как *однородными* (homogeneous topics), то есть содержащими данные только одного типа, так и *неоднородными* (heterogeneous topics), содержащими сразу несколько типов данных¹ (рис. 1.6).

¹ На эту тему Мартин Клеппманн (Martin Kleppmann) написал интересную статью, доступную по адресу <https://oreil.ly/tDZMm>. Он рассматривает плюсы и минусы каждой из стратегий, а также причины, по которым можно предпочесть одну стратегию другой. Кроме того, в статье Роберта Ёкоты (Robert Yokota), расположенной по адресу <https://oreil.ly/hpScS>, подробно объясняется, как поддерживать несколько типов событий при управлении схемами данных через реестр.

Как вы помните, для моделирования потоков на уровне хранения Kafka используются журналы фиксации, обновляющиеся только путем присоединения новых записей. Означает ли это, что каждой теме соответствует свой журнал? Не совсем. Дело в том, что Kafka — это распределенный журнал, а единственный объект сложно сделать распределенным. Для достижения некоторого уровня параллелизма в способах распространения и обработки журналов требуется много объектов. Вот почему темы Kafka разбиты на более мелкие блоки, называемые *разделами* (partitions).

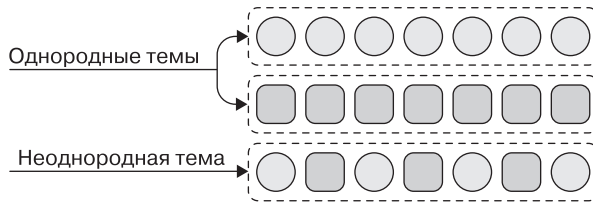


Рис. 1.6. Существуют разные стратегии хранения событий в темах; однородные темы обычно содержат один тип событий (например, `clicks`), в то время как неоднородные темы содержат несколько типов событий (например, `clicks` и `page_views`)

Разделы представляют собой отдельные журналы (то есть структуры данных, которые мы обсуждали в предыдущем разделе этой главы). Именно на уровне разделов темы Kafka реализована абстракция журнала фиксации, и именно тут обеспечивается упорядочение записей, причем каждый такой раздел имеет свой собственный набор смещений. Глобального упорядочения на уровне тем нет, поэтому связанные записи часто направляются производителями в один раздел¹.

В идеале данные равномерно распределяются по всем разделам темы. Но можно получить и разделы разного размера, как показано на рис. 1.7.

Тему можно разбить на произвольное количество разделов, и чем их больше, тем выше будут параллелизм и пропускная способность. Но слишком большое их число имеет и свои минусы². Я еще не раз коснусь этого момента, а пока важно

¹ Существуют разные стратегии разделения. В Kafka Streams и `ksqlDB` реализована популярная стратегия, в которой разделение выполняется на основе ключа записи (его можно извлечь из записи или установить вручную). Подробно мы поговорим об этом в следующих главах.

² Это и более длительные периоды восстановления после некоторых сбоев, и повышенное потребление ресурсов (дескрипторов файлов, памяти), и увеличение сквозной задержки.

запомнить, что читать из раздела может только один потребитель из группы (при этом чтение из одного раздела доступно потребителям из разных групп, как показано на рис. 1.5).

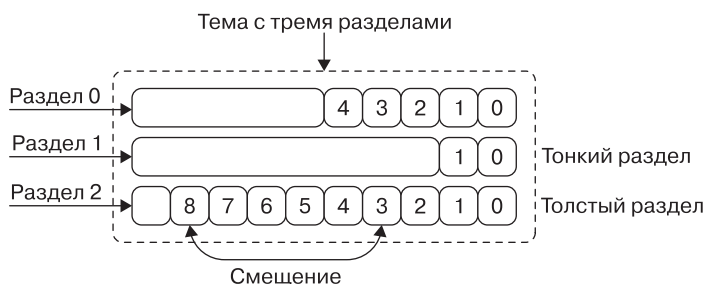


Рис. 1.7. Тема Kafka с тремя разделами

Следовательно, для распределения нагрузки между N потребителями в одной группе потребуется N разделов. Идеальной является ситуация, когда потребителей в группе меньше, чем разделов в теме, из которой выполняется чтение. В этом случае каждый потребитель может обрабатывать несколько разделов. Если же потребителей в группе больше, чем разделов в теме, некоторые из потребителей будут простаивать.

Теперь, вооружившись новыми сведениями, мы можем улучшить определение темы. Тема — это состоящий из нескольких разделов именованный поток, в котором каждый раздел представляет собой журнал фиксации, хранящий упорядоченную последовательность записей и обновляемый только путем добавления новых записей.

Но что именно хранится в разделах темы? Об этом я расскажу вам ниже.

События

Обсуждение обработки данных в темах получится неполным без информации о том, какие именно данные там хранятся.

В литературе по Kafka, в том числе в официальной документации, их называют различными терминами: сообщения, записи, события. Я предпочитаю последний вариант и далее буду использовать именно его. *Событие* (event) представляет собой снабженную временной меткой пару «ключ — значение», описывающую, *что именно произошло*. Элементы, из которых оно состоит, показаны на рис. 1.8.

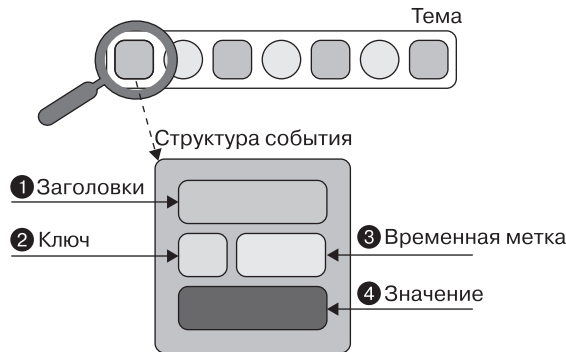


Рис. 1.8. Структура событий, которые хранятся в разделах тем

- ❶ В заголовки уровня приложения добавляют метаданные о событии, необязательный параметр. В этой книге он будет упоминаться достаточно редко.
- ❷ Ключи также необязательны, но они играют важную роль в распределении данных по разделам. В следующих главах вы увидите, что они используются для идентификации связанных записей.
- ❸ Каждому событию сопоставляется временная метка. Подробно мы поговорим о метках в главе 5.
- ❹ Значения содержат фактическую информацию о событиях в виде массива байтов. За превращение этих байтов в более информативную структуру (например, в объект JSON или в запись Avro) отвечает клиент. Процесс десериализации будет подробно рассмотрен в главе 3.

Теперь, когда вы лучше представляете структуру хранящихся в темах данных, рассмотрим кластерную модель развертывания Kafka.

Кластер и брокеры

При наличии централизованной точки связи особую важность приобретают вопросы надежности и отказоустойчивости. Кроме того, чтобы магистраль связи была способна выдерживать повышение нагрузки, она должна быть масштабируемой. Вот почему Kafka представляет собой кластер, а в хранении и извлечении данных принимают участие несколько машин, называемых *брокерами* (brokers).

Кластеры Kafka могут быть очень большими, даже охватывающими несколько центров обработки данных и географических регионов. Но я в основном буду

рассказывать про кластер Kafka, состоящий из одного узла, поскольку для начала работы с Kafka Streams и ksqlDB этого вполне достаточно. В реальных условиях обычно требуется как минимум три брокера. Кроме того, как правило, желательно настроить репликацию тем, чтобы копии данных оказывались у нескольких брокеров. Вы увидите это в упражнении к текущей главе. Такой подход позволяет достичь высокой доступности и избежать потери данных в случае выхода из строя какой-либо из машин.

То есть, когда речь заходит о данных, хранящихся и реплицирующихся между брокерами, на самом деле имеются в виду индивидуальные разделы внутри темы. Например, на рис. 1.9 показана тема с тремя разделами, распределенными между тремя брокерами.

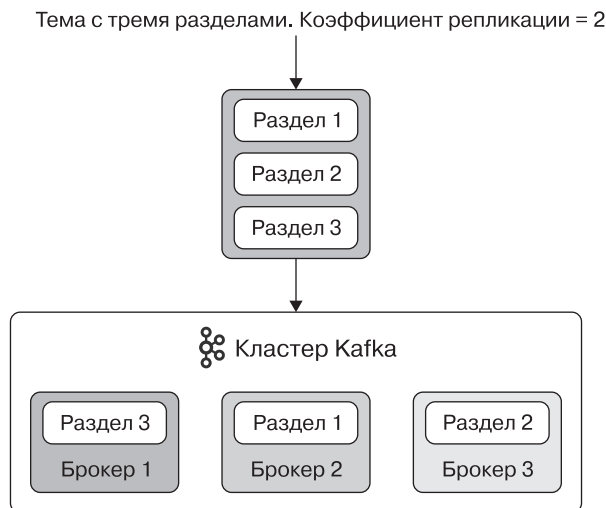


Рис. 1.9. Разделы распределены по доступным брокерам. Это означает, что тема может распределиться на несколько компьютеров в кластере Kafka

Такой подход позволяет иметь достаточно большие темы, обработка которых выходит за пределы возможностей отдельной машины. Для повышения отказоустойчивости и доступности при настройке темы задается коэффициент репликации. Например, при коэффициенте репликации 2 разделы будут храниться на двух брокерах, как показано на рис. 1.10.

При каждой репликации раздела между несколькими брокерами один из них назначается *ведущим* (leader). Именно он будет обрабатывать все запросы производителей и потребителей на чтение/запись из данного раздела. Другие брокеры, содержащие реплицированные разделы, называются *ведомыми* (followers) и про-

сто копируют данные из ведущего брокера. Если ведущий перестает работать, его роль передается одному из ведомых.

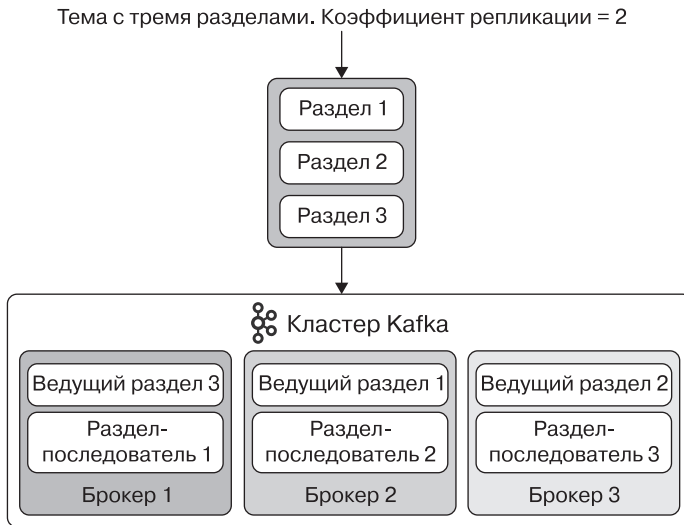


Рис. 1.10. Увеличение коэффициента репликации до 2 приведет к тому, что разделы будут храниться на двух брокерах

Наконец, брокеры играют важную роль в поддержании членства в группах потребителей. Посмотрим, как это происходит.

Кроме того, поскольку нагрузка на кластер со временем увеличивается, его можно расширить, добавив брокеры и переназначив разделы. Это позволит перенести данные со старых машин на новую.

Группы потребителей

Брокер сообщений Kafka всегда стараются настроить на обеспечение высокой пропускной способности и малой задержки. Чтобы воспользоваться этим преимуществом на стороне потребителя, следует распараллелить работу нескольких процессов. Это делается с помощью групп потребителей.

Состав группы потребителей со временем может меняться. При увеличении нагрузки в нее могут добавляться новые потребители, а существующие периодически отключаются для планового обслуживания либо из-за неожиданного сбоя. Соответственно, нужен способ управления членством в группах и перераспределения выполняемой работы.

Для каждой группы потребителей выделяется специальный брокер, называемый *координатором группы*. Он отвечает за получение *контрольных сигналов* (heartbeats) от потребителей и запуск *перевыравнировки*, если какой-то из потребителей перестает посылать такой сигнал. Схематично это представлено на рис. 1.11.

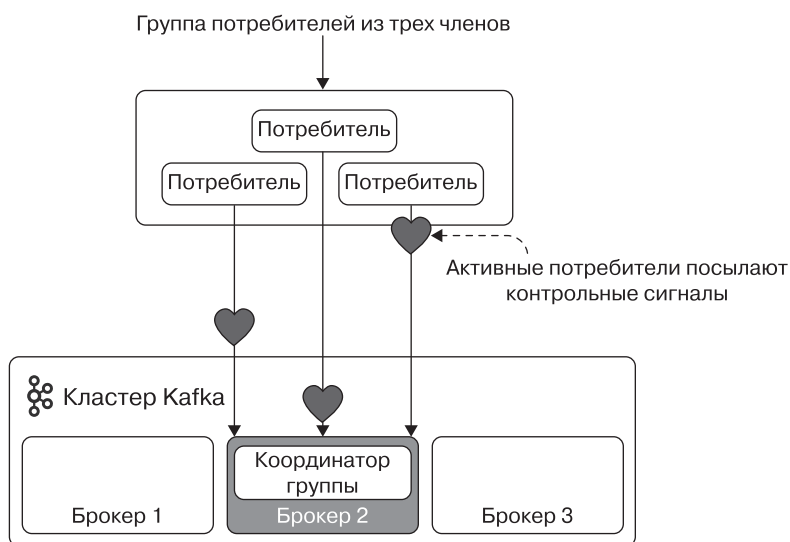


Рис. 1.11. Три потребителя, составляющие группу, посылают контрольные сигналы координатору

Каждому активному члену группы потребителей может быть выделен раздел. Рисунок 1.12 демонстрирует пример распределения работы между тремя активными потребителями.



Рис. 1.12. Три активных потребителя делят между собой нагрузку чтения/обработки в трех разделах темы Kafka

Когда какой-то потребитель выходит из строя и перестает посылать контрольные сигналы в кластер, его работа автоматически передается активным потребителям. Схематично это показано на рис. 1.13.



Рис. 1.13. Прекращение работы одного из потребителей ведет к переконфигурированию нагрузки

Как видите, именно группы потребителей обеспечивают высокую доступность и отказоустойчивость на уровне обработки данных. Давайте на практике посмотрим, как все это работает. Для начала познакомимся с процессом установки Kafka.

Установка Kafka

Официальная документация (https://oreil.ly/rU-j_) содержит подробную инструкцию по установке брокера сообщений Kafka вручную. Но для простоты в большинстве представленных в книге примеров Kafka и приложения потоковой обработки будут развертываться в контейнере Docker (<https://www.docker.com/>).

Для установки Kafka воспользуемся инструментом Docker Compose и далее будем работать с образами Docker от Confluent¹.

Первым делом скачайте Docker со страницы <https://oreil.ly/1kS0h> и выполните его установку.

¹ Существует множество образов Docker, подходящих для запуска Kafka. Но я рекомендую использовать образ от Confluent, в котором можно запускать еще и базу данных ksqlDB и реестр схем Confluent Schema Registry.

Сохраните следующую конфигурацию в файл `docker-compose.yml`:

```
---
version: '2'

services:
  zookeeper: ❶
    image: confluentinc/cp-zookeeper:6.0.0
    hostname: zookeeper
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000

  kafka: ❷
    image : confluentinc/cp-enterprise-kafka :6.0.0
    hostname: kafka
    container_name: kafka
    depends_on:
      - zookeeper
    ports:
      - "29092:29092"
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: |
        PLAINTEXT:PLAINTEXT,PLAINTEXT_HOST:PLAINTEXT
      KAFKA_ADVERTISED_LISTENERS: |
        PLAINTEXT://kafka:9092,PLAINTEXT_HOST://localhost:29092
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
```

❶ Первый контейнер `zookeeper` содержит хранилище ZooKeeper. Я не упомянул ZooKeeper во введении, так как на момент написания книги стоял вопрос о его удалении из Kafka. Но пока эта централизованная служба хранения метаданных, таких как конфигурационные настройки темы, все еще продолжает использоваться.

❷ В контейнере `kafka` находится результат установки Kafka. Именно внутри него предстоит работать нашему брокеру, состоящему из одноузлового кластера. Здесь же будут выполняться и сценарии взаимодействия с кластером.

Локальный кластер Kafka запускается командой:

```
docker-compose up
```

Все, кластер Kafka приведен в рабочее состояние, и мы можем перейти к рассмотрению первого примера.

Hello, Kafka

Давайте посмотрим, как происходит создание темы, запись в нее данных *производителем* и, наконец, чтение данных темы *потребителем*. Первым делом нужно войти в контейнер с брокером Kafka. Это делается следующей командой:

```
docker-compose exec kafka bash
```

Создадим тему `users`. В этом нам поможет встроенный в Kafka консольный сценарий `kafka-topics`. Вот как выглядит эта процедура:

```
kafka-topics \ ❶  
  --bootstrap-server localhost:9092 \ ❷  
  --create \ ❸  
  --topic users \ ❹  
  --partitions 4 \ ❺  
  --replication-factor 1 ❻
```

```
# вывод
```

```
Created topic users.
```

- ❶ `kafka-topics` — встроенный в Kafka консольный сценарий.
- ❷ `bootstrap server` — это разделенный запятыми список пар «хост/порт», которые представляют собой адреса брокеров Kafka в кластере.
- ❸ Для взаимодействия с темами Kafka применяются различные флаги, в частности `--list`, `--describe` и `--delete`. В примере использован флаг `--create`, создающий новую тему.
- ❹ Созданная тема получила имя `users`.
- ❺ Тему разбили на четыре раздела.
- ❻ Мы работаем с одноузловым кластером, поэтому коэффициент репликации оставим равным 1. Но в рабочей среде для повышения отказоустойчивости этому параметру обычно присваивают более высокое значение (например, 3).



Все сценарии, перечисленные в этом разделе, включены в комплект поставки файлов исходного кода Kafka. В стандартной версии Kafka они имеют расширение `.sh` (например, `kafka-topics.sh`, `kafka-console-producer.sh` и т. п.). Но на платформе Confluent расширения не используются. Именно поэтому в приведенном выше фрагменте кода написано `kafka-topics`, а не `kafka-topics.sh`.

Теперь выведем описание темы и ее конфигурацию:

```
kafka-topics \
  --bootstrap-server localhost:9092 \
  --describe \ ❶
  --topic users

# вывод
Topic: users PartitionCount: 4 ReplicationFactor: 1 Configs:
  Topic: users Partition: 0 Leader: 1 Replicas: 1 Isr: 1
  Topic: users Partition: 1 Leader: 1 Replicas: 1 Isr: 1
  Topic: users Partition: 2 Leader: 1 Replicas: 1 Isr: 1
  Topic: users Partition: 3 Leader: 1 Replicas: 1 Isr: 1
```

❶ Флаг `--describe` позволяет посмотреть конфигурационные настройки темы.

Добавим в тему данные с помощью встроенного сценария `kafka-console-producer`:

```
kafka-console-producer \ ❶
  --bootstrap-server localhost:9092 \
  --property key.separator=, \ ❷
  --property parse.key=true \
  --topic users
```

❶ Когда мы начнем работать с Kafka Streams и `ksqlDB`, будут использоваться процессы-производители, встроенные в базовую библиотеку Java, и сценарий `kafka-console-producer` понадобится разве что для тестирования и разработки.

❷ Для темы `users` создается набор пар «ключ — значение». Это свойство указывает, что в качестве разделительного символа будет использоваться запятая (,).

На экране появится приглашение на ввод. Введите перечисленные ниже пары «ключ — значение», и они будут добавлены в тему `users`. Для выхода из командной строки нажмите сочетание клавиш `Control+C`:

```
>1,mitch
>2,elyse
>3,isabelle
>4,sammy
```

Прочитаем содержимое нашей темы с помощью сценария `kafka-console-consumer`:

```
kafka-console-consumer \ ❶
  --bootstrap-server localhost:9092 \
  --topic users \
  --from-beginning ❷
```

```
# вывод
mitch
```



```
elyse
isabelle
sammy
```

❶ Сценарий `kafka-console-consumer` тоже входит в дистрибутив Kafka. Как и в случае со сценарием `kafka-console-producer`, в дальнейшем мы в основном будем пользоваться встроенными в Kafka Streams и `ksqlDB` процессами-потребителями, а не сценарием `kafka-console-consumer`, который останется для тестирования.

❷ Флаг `--from-begin` указывает, что потребление сообщений из заданной темы должно начинаться с первого смещения, то есть с начала темы.

По умолчанию сценарий `kafka-console-consumer` выводит только значение, в то время как события содержат и другую информацию, например ключ, временную метку и заголовки. Чтобы их увидеть, добавим в сценарий дополнительные свойства¹:

```
kafka-console-consumer \
  --bootstrap-server localhost:9092 \
  --topic users \
  --property print.timestamp=true \
  --property print.key=true \
  --property print.value=true \
  --from-beginning
```

```
# вывод
CreateTime:1598226962606 1 mitch
CreateTime:1598226964342 2 elyse
CreateTime:1598226966732 3 isabelle
CreateTime:1598226968731 4 sammy
```

Вот и все! Теперь вы умеете выполнять элементарные действия с кластером Kafka. Для остановки запущенных контейнеров после завершения работы используйте команду:

```
docker-compose down
```

Заключение

Коммуникационная модель Kafka упрощает взаимодействие систем, а ее быстрый и надежный уровень хранения дает возможность легко работать с потоками данных. Благодаря кластерному развертыванию Kafka обеспечивает доступность и высокую отказоустойчивость на уровне хранения, ведь данные

¹ Начиная с версии 2.7, для вывода заголовков сообщений можно использовать флаг `--property print.headers=true`.

копируются на другие машины, называемые брокерами. На уровнях обработки и потребления потоков высокая доступность, отказоустойчивость и масштабируемость рабочей нагрузки обеспечиваются способностью кластера получать контрольные сигналы и на их основе обновлять состав групп потребителей. Все эти особенности сделали брокер сообщений Kafka одной из самых популярных платформ для обработки потоков.

Этой информации вам хватит, чтобы приступить к работе с Kafka Streams и ksqlDB. В следующей главе вы узнаете, как библиотека Kafka Streams вписывается в более широкую экосистему Kafka, и на примерах увидите, как ей пользоваться.

Часть II

Библиотека Kafka Streams

ГЛАВА 2

Начало работы с Kafka Streams

https://t.me/it_boooks

Kafka Streams — это легкая, но мощная библиотека Java для обогащения, преобразования и обработки потоков данных в реальном времени. В этой главе я расскажу о ней в общих чертах. Мой рассказ можно сравнить с первым свиданием, на котором вы немного узнаете об истории библиотеки Kafka Streams и познакомитесь с ее функционалом.

К концу этого свидания, ну-у... то есть этой *главы*, вы будете владеть следующей информацией.

- Место библиотеки Kafka Streams в экосистеме Kafka.
- Зачем была создана библиотека Kafka Streams.
- Каким функционалом и рабочими характеристиками она обладает.
- Кому она нужна.
- Как она выглядит относительно других решений для обработки потоковых данных.
- Как создать и запустить базовое приложение Kafka Streams.

Впрочем, довольно предисловий. Начнем наше метафорическое свидание с простого вопроса: *где вы живете* (в экосистеме Kafka)?

Экосистема Kafka

Библиотека Kafka Streams «живет» в группе технологий, совокупность которых называется *экосистемой Kafka*. В главе 1 вы узнали, что в основе Apache Kafka лежит распределенный журнал, обновляемый только путем добавления в него новых записей. Мы можем добавлять в него сообщения и читать их оттуда. В кодовую базу Kafka входят также API-интерфейсы для взаимодействия с этим

журналом (который разделен на категории, называемые *темами*). Три API экосистемы Kafka, связанные с перемещением данных, представлены в табл. 2.1.

Таблица 2.1. API для перемещения данных в брокер Kafka и из него

API	Взаимодействие с темой	Примеры
API производителей	<i>Запись</i> сообщений в темы Kafka	<ul style="list-style-type: none"> • Filebeat. • Rsyslog. • Пользовательские варианты производителей
API потребителей	<i>Чтение</i> сообщение из тем Kafka	<ul style="list-style-type: none"> • Logstash. • Kafkacat. • Пользовательские варианты потребителей
API коннекторов	<i>Соединение</i> внешних хранилищ данных, API и файловых систем с темами Kafka. Отвечает как за <i>чтение</i> из тем (коннекторы-приемники), так и за <i>запись</i> в темы (коннекторы-источники)	<ul style="list-style-type: none"> • Коннектор JDBC-источника. • Коннектор Elasticsearch-приемника. • Пользовательские варианты коннекторов

Перемещение данных через брокер сообщений Kafka, безусловно, важно для создания конвейеров, но немало задач требует мгновенной *реакции* на поступающую информацию. Речь идет о так называемой *поточковой обработке* (stream processing). Создавать приложения потоковой обработки с помощью Kafka можно разными способами. Но для начала я хочу вам рассказать, как такие приложения создавались до появления Kafka Streams и как в экосистеме Kafka, наряду с другими API, появилась выделенная библиотека потоковой обработки.

До появления Kafka Streams

Пока не появилась Kafka Streams, в экосистеме Kafka была пустота¹. Это была совсем не та пустота, которую можно почувствовать во время утренней медитации и потом ощутить себя освеженными и просветленными. Нет, я имею в виду пробел, сильно затруднявший создание приложений для обработки потоковой информации. На уровне библиотек обработка данных в темах Kafka просто не поддерживалась.

¹ Здесь я имею в виду официальную экосистему, включающую в себя все компоненты, поддерживаемые в рамках проекта Apache Kafka.

В ранних версиях экосистемы Kafka существовало два основных способа создания приложений для обработки потоков. Вы могли:

- напрямую использовать API потребителей и API производителей;
- обратиться к другим фреймворкам, реализующим обработку потоков (например, Apache Spark Streaming или Apache Flink).

Имея API потребителей и производителей, можно напрямую читать из потока и записывать в поток, а также реализовывать любые логические схемы обработки данных. Главное — знать какой-то язык программирования (Python, Java, Go, C/C++, Node.js и т. п.) и быть готовыми с нуля писать много кода. Это очень простые API. Они лишены множества базисных элементов, которые позволили бы им считаться API для обработки потоков. В частности, в них отсутствуют:

- локальное и отказоустойчивое состояние¹;
- многие операторы для преобразования потоков данных;
- более совершенные варианты представления потоков²;
- продуманное управление временем³.

Отсутствие перечисленного затрудняет выполнение любых нестандартных действий, таких как агрегирование записей, объединение событий, произошедших в один период времени, или узкоспециализированные запросы к потоку. Абстракций, которые помогли бы в решении таких задач, в API потребителей и производителей нет, так что весь код приходится писать самостоятельно.

Второй вариант, то есть привлечение сторонних фреймворков, например Apache Spark или Apache Flink, привносит в систему ненужную сложность. Подробно о недостатках этого подхода мы поговорим в разделе «Сравнение с другими системами», а пока я замечу только, что решение, которое предоставляет функционал потоковой обработки, одновременно создавая дополнительную нагрузку на обрабатывающий кластер, нельзя считать простым и оптимальным. Кроме того, нам требуется лучшая интеграция с Kafka, особенно при работе с промежуточными представлениями данных за пределами тем, играющих роли источника и приемника.

¹ Один из первых разработчиков Apache Kafka Джей Крепс (Jay Kreps) подробно обсуждал это в блоге O'Reilly еще в 2014 году (<https://oreil.ly/vzRH->).

² Речь идет об агрегированных потоках/таблицах, о которых я расскажу чуть попозже.

³ Вопросам, связанным со временем, я посвятил целую главу, но все равно рекомендую посмотреть отличную презентацию Маттиаса Джей Сакса (Matthias J. Sax) с конференции Kafka Summit в 2019 году (<https://oreil.ly/wr123>).

К счастью, сообщество Kafka осознало, насколько в экосистеме Kafka не хватает API потоковой обработки, и решило создать его¹.

Рождение Kafka Streams

В 2016 году после появления первой версии Kafka Streams (она же *Streams API*) экосистема Kafka навсегда изменилась. Многочисленные приложения обработки потоков, в которых многое предлагалось делать вручную, уступили место более продвинутым приложениям, использующим разработанные сообществом шаблоны и абстракции для обработки потоков данных в реальном времени.

В отличие от API производителей, потребителей и коннекторов, библиотека Kafka Streams призвана помочь не только в *перемещении* данных через брокер Kafka, но и в *обработке* их потоков в режиме реального времени². Благодаря богатому набору операторов и примитивов обработки потоков упрощается преобразование потоковых данных и при необходимости запись их новых представлений обратно в Kafka (когда преобразованные или обогащенные события нужно сделать доступными для следующих систем в конвейере).

Рисунок 2.1 демонстрирует, какое место в экосистеме Kafka занимают упоминавшиеся выше API. Библиотека Kafka Streams функционирует в ней на уровне обработки потоков.

Из диаграммы видно, что библиотека Kafka Streams работает на сильно нагруженном уровне экосистемы Kafka, в месте, куда сходятся данные из многих источников. Именно здесь реализуются сложные варианты *обогащения*, *преобразования* и *обработки* данных. Именно на этом уровне до появления Kafka Streams (когда приходилось пользоваться API потребителей/производителей) кропотливо создавались собственные абстракции обработки потоков или преодолевались сложности, возникшие из-за применения стороннего фреймворка. Поэтому давайте знакомиться с функционалом Kafka Streams, позволяющим легко и эффективно работать на этом уровне.

¹ Большое спасибо Гочжен Ванну (Guozhang Wang), который отправил в Kafka Improvement Proposal предложение, послужившее основанием для создания Kafka Streams. См. <https://oreil.ly/l2wbc>.

² После того как к компоненту Kafka Connect добавили функцию преобразования отдельных сообщений, он тоже стал пригоден для обработки событий, но все равно в плане функционала он не выдерживает никакого сравнения с библиотекой Kafka Streams.

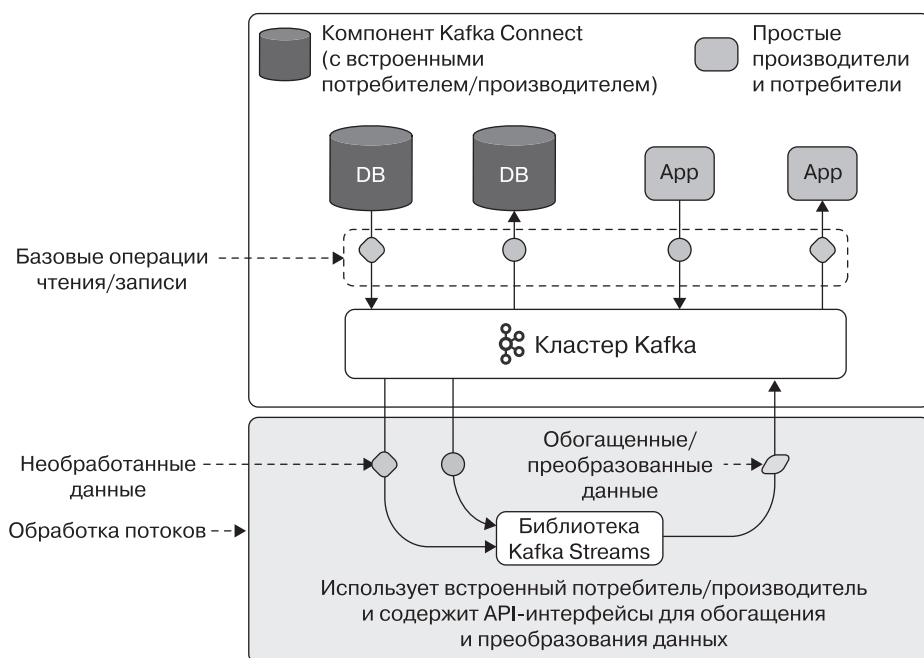


Рис. 2.1. Библиотека Kafka Streams — «мозг» экосистемы Kafka, потребляющий записи из потока событий, обрабатывающий данные и при необходимости записывающий расширенные или преобразованные записи обратно в Kafka

Обзор функционала

Благодаря своему огромному функционалу библиотека Kafka Streams — отличный выбор для современных приложений потоковой обработки. Вот основные элементы этого функционала.

- Высокоуровневый DSL, напоминающий API для работы с потоками на языке Java. Он обеспечивает плавный и функциональный подход к обработке потоков данных и отличается легкостью изучения и применения.
- Низкоуровневый Processor API, предоставляющий разработчикам детальный контроль над происходящим.
- Удобные абстракции для моделирования данных в виде потоков или таблиц.
- Возможность объединения потоков и таблиц, предназначенная для преобразования и обогащения данных.

- Операторы и утилиты для создания приложений потоковой обработки как без сохранения, так и с сохранением состояния.
- Поддержка операций, привязанных ко времени, в том числе оконных и периодических функций.
- Простая установка. Так как Kafka Streams — это просто библиотека, ее можно добавить в любое приложение Java¹.
- Масштабируемость, надежность, удобство сопровождения.

В процессе знакомства с этим функционалом вы быстро поймете, почему эта библиотека так широко используется и так любима. Как высокоуровневый DSL, так и низкоуровневый Processor API не только легки в освоении, но и чрезвычайно эффективны. Даже сложные задачи потоковой обработки (например, соединение движущихся потоков данных) решаются с помощью небольшого кода, что делает процесс разработки по-настоящему легким.

Последний пункт приведенного выше списка касается долговременной стабильности приложений для обработки потоковых данных. В конце концов, на этапе изучения многие технологии вызывают искренний интерес, но на самом деле важно, подходят ли они для решения реальных, как правило, куда более сложных, чем учебные, задач. Поэтому прежде чем погрузиться в освоение Kafka Streams, имеет смысл оценить, насколько эта технология жизнеспособна. Для этого рассмотрим ее эксплуатационные характеристики.

Эксплуатационные характеристики

В прекрасной книге Мартина Клеппмана «Высоконагруженные приложения»² для систем обработки данных выделены три важные метрики.

- Масштабируемость.
- Надежность.
- Удобство сопровождения.

Именно на их основе я собираюсь оценивать библиотеку Kafka Streams. Определим каждую из метрик и узнаем, насколько рассматриваемое средство отвечает заданным ими требованиям.

¹ Библиотека Kafka Streams работает и с другими языками, созданными для JVM, в том числе с языками Scala и Kotlin. Однако все примеры в книге написаны только на Java.

² Клеппман М. Высоконагруженные приложения. Программирование, масштабирование, поддержка. — Питер, 2018.

Масштабируемость

Система считается *масштабируемой* (scalable), если она остается работоспособной при увеличении нагрузки. В главе 1 вы узнали, что масштабирование тем Kafka сводится к добавлению дополнительных разделов и при необходимости дополнительных брокеров (последнее необходимо при выходе темы за пределы емкости имеющегося кластера Kafka).

В Kafka Streams единицей работы тоже считается одна тема-раздел, и Kafka автоматически распределяет работу между группами потребителей¹.

Это имеет два важных последствия.

- Поскольку расширение тем происходит путем добавления разделов, масштабировать объем работы, которую выполняет приложение Kafka Streams, можно, увеличивая количество разделов в исходных темах².
- Благодаря группам потребителей общий объем работы, выполняемой приложением Kafka Streams, может быть распределен между несколькими взаимодействующими экземплярами приложения.

Немного проиллюстрирую второй пункт списка. Развертывание приложения Kafka Streams почти всегда означает развертывание нескольких экземпляров, каждый из которых будет отвечать за свой фрагмент работы. Например, для темы из 32 разделов можно развернуть четыре экземпляра приложения. В этом случае на каждый из них придется восемь разделов ($4 \times 8 = 32$). Если же развернуть 16 экземпляров приложения, каждый из них будет обрабатывать два раздела ($16 \times 2 = 32$).

Итак, способность Kafka Streams справляться с повышенной нагрузкой (то есть *масштабируемость*) достигается добавлением дополнительных разделов (единиц работы) и экземпляров приложений (работников).

Кроме того, Kafka Streams *адаптивна*, что позволяет без особых усилий (хотя и вручную) уменьшать и увеличивать число экземпляров приложения. Впрочем, горизонтальное масштабирование равно количеству задач, созданных для конкретной топологии. Более подробно этот вопрос будет рассматриваться в подразделе «Задачи и потоки выполнения» этой главы.

¹ С одной темой может работать одновременно несколько групп потребителей, при этом каждая группа обрабатывает сообщения независимо от других.

² Хотя разделы можно добавлять в существующие темы, рекомендуется создать тему с желаемым количеством разделов и перенести туда все существующие задачи.

Надежность

Такая характеристика систем данных, как надежность, важна не только для обслуживающего персонала (никто не хочет, чтобы его будили в три часа ночи из-за неисправности), но и для клиентов (которых не устраивает выход системы из строя и уж тем более потеря или повреждение данных). В Kafka Streams есть различный устойчивый к сбоям функционал¹, наиболее очевидный вариант которого был описан в разделе «Группы потребителей» предыдущей главы.

Если один экземпляр приложения Kafka Streams выходит из строя (например, по причине аппаратного сбоя), брокер Kafka автоматически перераспределяет его задачи между остальными экземплярами. После устранения сбоя (или в более современных архитектурах, использующих систему оркестрации, таких как Kubernetes, — после перемещения приложения на работоспособный узел) Kafka возобновляет работу. Именно эта способность корректно обрабатывать ошибки обеспечивает *надежность* Kafka Streams.

Удобство сопровождения

Общеизвестно, что большая часть стоимости программного обеспечения связана не с его разработкой, а с его текущим обслуживанием — исправлением ошибок, поддержанием работоспособности систем, анализом сбоев...

Мартин Клеппман

Поскольку Kafka Streams — это библиотека Java, выявление неисправностей и исправление ошибок не должны вызывать затруднения. Мы работаем с автономными приложениями, а шаблоны как для устранения неполадок, так и для мониторинга приложений Java хорошо известны. Скорее всего, вам уже приходилось ими пользоваться. Это шаблоны ведения и анализа журнала приложения, сбора метрик приложения и JVM, профилирование и трассировка и т. п.

Кроме того, благодаря лаконичности и наглядности API Kafka Streams, обслуживание на уровне кода занимает меньше времени, чем в случае более сложных библиотек. Это достаточно простая процедура, доступная даже новичкам. Даже если приложение Kafka Streams никто не трогал много месяцев, скорее всего, вам не потребуется много времени, чтобы понять его код. По тем же причинам специалисты по сопровождению новых проектов обычно быстро осваивают приложения Kafka Streams, что делает сам процесс сопровождения еще более удобным.

¹ В эту категорию попадает и функционал, характерный для приложений с отслеживанием состояния, о которых пойдет речь в главе 4.

Сравнение с другими системами

К этому моменту свидания с Kafka Streams вы, скорее всего, уже ответили себе на вопрос, стоит ли начинать с этой библиотекой долгосрочные отношения. Тем не менее посмотреть на существующие альтернативы все-таки стоит.

И в самом деле, как оценить, насколько хороша технология, без ее сравнения с конкурентами? Поэтому сопоставим Kafka Streams с некоторыми популярными технологиями в области обработки потоковых данных¹. И начнем рассмотрение с модели развертывания.

Модель развертывания

Фреймворк Apache Flink и расширение Spark Streaming для фреймворка Apache Spark требуют для программы потоковой обработки выделенного кластера. Это увеличивает как сложность системы, так и затраты вычислительных ресурсов. Даже опытные инженеры из солидных компаний признают, что вычислительный кластер обходится недешево. Например, Нитин Шарма из компании Netflix в интервью рассказывал, что, когда они создавали на базе фреймворка Apache Flink приложение и кластер, адаптация к его нюансам заняла около шести месяцев.

С другой стороны, Kafka Streams реализована как *библиотека* Java. Диспетчер кластера в этом случае не нужен, достаточно добавить в приложение Java зависимость от этой библиотеки. Создаваемые таким способом приложения для обработки потоковой информации представляют собой отдельные программы, что дает большую свободу при мониторинге, упаковке и развертывании кода. Например, на платформе Mailchimp приложения Kafka Streams развертываются с помощью тех же шаблонов и инструментов, что и другие внутренние Java-приложения. Фактически все это дает нам такое огромное преимущество, как возможность немедленной интеграции приложений Kafka Streams в любую систему.

Теперь сравним модель обработки данных у Kafka Streams и ее конкурентов.

Модель обработки

Еще одним ключевым отличием Kafka Streams от конкурентов стала *модель обработки по отдельным событиям* (event-at-a-time processing). Каждое событие обрабатывается в момент его поступления. Это считается настоящей

¹ Провести полное сравнение невозможно из-за постоянного появления новых решений для обработки потоковых данных, поэтому я рассмотрю только самые популярные и давно существующие варианты систем-аналогов, доступные на момент написания этого текста.

поточковой передачей и обеспечивает меньшую задержку, чем альтернативный подход, в котором поток интерпретируется как непрерывная последовательность микропакетов данных. Речь идет о так называемом *микропакетировании* (micro-batching). Пакеты создаются через регулярные интервалы времени (например, каждые 500 миллисекунд) и отправляются на обработку. Разницу между двумя подходами иллюстрирует рис. 2.2.

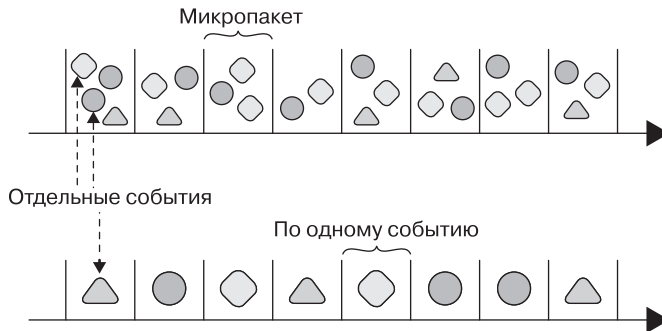


Рис. 2.2. При микропакетировании записи через регулярные интервалы времени группируются в небольшие партии и передаются обработчикам; обработка каждого события в момент его наступления позволяет не ждать, пока сформируется пакет



В фреймворках на базе микропакетной архитектуры для увеличения *пропускной способности* часто начинают увеличивать *задержку*. В Kafka Streams за счет распределения данных по разделам можно добиться чрезвычайно низкой задержки при сохранении высокой пропускной способности.

Наконец, разберем архитектуру обработки данных в Kafka Streams и посмотрим, чем ее ориентация на потоковую передачу отличается от других систем.

Каппа-архитектура

Выбирая между Kafka Streams и альтернативными решениями, важно учитывать, требует ли система, которую вы строите, поддержки как пакетной, так и потоковой обработки. На момент написания этого текста Kafka Streams фокусировалась исключительно на сценариях использования потоковой передачи¹ (так называемая каппа-архитектура), в то время как в фреймворках Apache Flink и Apache Spark поддерживается как пакетная, так и потоковая

¹ Имеется устаревшее, но еще открытое предложение добавить в Kafka Streams поддержку пакетной обработки (<https://oreil.ly/v3DbO>).

обработка (это *лямбда-архитектура*). К сожалению, архитектуры, в которых поддерживаются оба варианта, не лишены недостатков. О слабых местах гибридной системы почти за два года до появления Kafka Streams высказывался Джей Крепис (<https://oreil.ly/RwkNi>):

Операционная нагрузка по запуску и отладке двух систем очень высока. И любая новая абстракция может предоставить только функционал, который одновременно поддерживается обеими системами.

Эти проблемы не помешали росту популярности проекта Apache Beam, который представляет собой унифицированную модель программирования для пакетной и потоковой обработки. Впрочем, Apache Beam, в отличие от Apache Flink, напрямую сравнить с Kafka Streams не получится, так как он находится на другом уровне абстракции. Большая часть работы отдается на откуп механизму выполнения, в качестве которого может выступать, например, Apache Flink или Apache Spark. Поэтому при сравнении с Kafka Streams нужно рассматривать не только Beam API, но и механизм выполнения, которым он пользуется.

Кроме того, конвейерам на базе Apache Beam не хватает важного функционала, который присутствует в Kafka Streams. Вот что пишет об этом Роберт Йокота (<https://oreil.ly/24zG9>), который создал экспериментальную программу Kafka Streams Beam Runner (<https://oreil.ly/24zG9>) и поддерживает несколько инновационных проектов в экосистеме Kafka¹.

Приведем возможный перечень различий между двумя системами.

- Платформа Kafka Streams позволяет обрабатывать как *потоки*, так и *отношения*.
- Платформа Apache Beam предназначена только для работы с *потоками*.

При этом платформа потоково-реляционной обработки имеет перечисленные ниже дополнительные возможности.

- Отношения (или таблицы) — полноправные сущности, так как каждое из них уникально.
- Отношения можно преобразовать в другие отношения.
- Отношения допускают произвольные запросы.

¹ К этим проектам относятся в числе прочих поддерживаемая брокером Kafka реляционная база данных KarelDB, библиотека для анализа графов на базе Kafka Streams и многое другое. См. <https://yokota.blog>.

Все эти особенности будут демонстрироваться в следующих главах, пока же я только замечу, что многое из наиболее мощного функционала Kafka Streams (включая запросы состояния потока) недоступно в Apache Beam и в других универсальных фреймворках¹. Кроме того, капша-архитектура предлагает более простой и более специализированный подход к работе с потоками данных, позволяющий усовершенствовать процесс разработки и упростить эксплуатацию и обслуживание программного обеспечения. Так что, если сценарии использования не требуют пакетной обработки, гибридная система только внесет ненужную сложность.

Надеюсь, этот небольшой обзор дал вам представление о том, чем Kafka Streams отличается от конкурентов. Теперь давайте посмотрим, как применяется эта библиотека.

Сценарии использования

Библиотека Kafka Streams оптимизирована для быстрой и эффективной обработки бесконечных наборов данных. Следовательно, она отлично подходит для областей, в которых важна немедленная обработка информации и низкая задержка. Например:

- обработка финансовых данных (компания Flipkart <https://oreil.ly/dAcby>), мониторинг купли-продажи, обнаружение мошенничества;
- алгоритмическая торговля;
- мониторинг фондового рынка/криптовбиржи;
- отслеживание и пополнение ресурсов в реальном времени (компания Walmart <https://oreil.ly/Vof76>);
- бронирование мероприятий, выбор мест (компания Ticketmaster <https://oreil.ly/V4t1h>);
- отслеживание доставки электронной почты (платформа Mailchimp);

¹ На момент написания этого текста в Apache Flink появилась бета-версия запросов состояния. При этом про соответствующий API в официальной документации сообщалось следующее: «Клиентский интерфейс для запросов состояния в настоящее время дорабатывается, и поэтому его стабильность не гарантирована. Вероятно, в следующих версиях Flink в него будут внесены критические изменения». Так что тщательно продуманный и работоспособный API для запросов состояния пока имеется только в Kafka Streams.

- обработка телеметрических данных для видеоигр (компания Activision, разработавшая серию игр Call of Duty <https://oreil.ly/Skan3>);
- индексация поиска (сайт поиска услуг Yelp <https://oreil.ly/IhCnC>);
- отслеживание/вычисление геопространственных данных (например, сравнение расстояний, определение времени прибытия);
- обработка информации с датчиков умного дома/Интернета вещей (иногда называемых аббревиатурой AIOT, что расшифровывается как Artificial Intelligence of Things — «искусственный интеллект вещей»);
- захват изменения данных (компания Redhat <https://oreil.ly/INs3z>);
- спортивные трансляции/интерактивные виджеты (Gracenote <https://oreil.ly/YeX33>);
- платформы для размещения рекламы в реальном времени (Pinterest <https://oreil.ly/cBgSG>);
- прогнозирование в сфере здравоохранения, мониторинг жизненных показателей (Детская больница Атланты <https://oreil.ly/4MYLc>);
- инфраструктура для обмена сообщениями (Slack https://oreil.ly/_n7sZ), чат-боты, виртуальные ассистенты;
- конвейеры (Twitter <https://oreil.ly/RuPPV>) и платформы (Kafka Graphs <https://oreil.ly/8IHKТ>) машинного обучения.

Список можно продолжить, но во всех приведенных примерах есть общий момент: принимать решения или обрабатывать данные требуется *в реальном времени*. Диапазон возможных сценариев использования поражает воображение: от любительской обработки выходных сигналов с датчиков умного дома до отслеживания состояния больных в медицинских учреждениях, как это сделано в детской больнице Атланты.

Библиотека Kafka Streams также отлично подходит для создания микросервисов поверх потоков событий. Она не только упрощает типичные операции обработки потоковых данных (фильтрацию, соединение, создание окон и преобразование данных), но и позволяет отображать состояние потока с помощью *интерактивных запросов* (interactive queries), о которых я расскажу в соответствующем разделе 4 главы. Состояние потока может иметь вид набора (например, общее количество просмотров каждого видео на потоковой платформе) или последнего представления быстро меняющейся сущности в потоке событий (например, последняя цена акций рассматриваемой компании).

Итак, вы получили некоторое представление о том, кто пользуется Kafka Streams и для каких сценариев она лучше всего подходит, теперь, перед тем как мы начнем наконец писать код, кратко разберем архитектуру этой платформы.

Топология обработчиков

Kafka Streams использует парадигму программирования, называемую *программированием потоков данных* (dataflow programming, DFP). Это метод представления программ в виде последовательности входных и выходных данных, а также этапов их обработки. Благодаря этому программы пишутся естественным и интуитивно понятным способом. Я считаю, что это одна из причин легкости освоения Kafka Streams.



Я собираюсь немного углубиться в архитектуру Kafka Streams. Те, кто предпочитает начать с практики, могут сразу перейти к разделу «Начало практической работы: Hello, Streams», а потом вернуться сюда.

Вместо классического построения программы в виде последовательности шагов логическая схема обработки потоков в приложении Kafka Streams структурирована как ориентированный ациклический граф (directed acyclic graph, DAG). Пример такого графа показан на рис. 2.3. Данные проходят через набор узлов-обработчиков (они обозначены прямоугольниками), а соединяющие их линии представляют собой потоки входных и выходных данных (которые передаются от одного обработчика к другому).

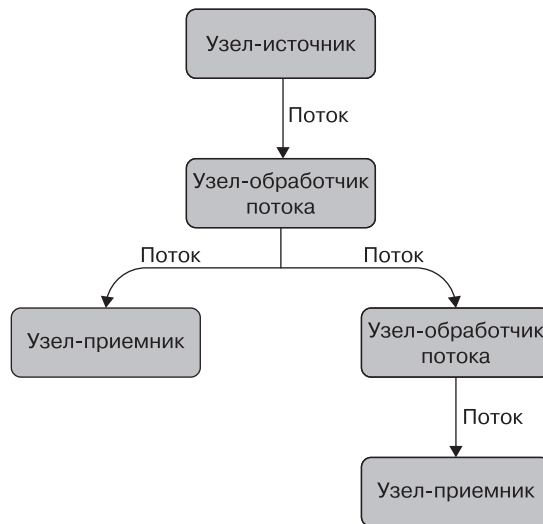


Рис. 2.3. Свою архитектуру Kafka Streams частично заимствует из программирования потоков данных, поэтому программы структурированы в виде графа узлов-обработчиков, через которые проходят данные

В Kafka Streams есть три основных типа обработчиков потоков.

Узел-источник (source processor)

Это место, откуда в приложение Kafka Streams поступает информация. Данные считываются из темы Kafka и отправляются в один или несколько *узлов-обработчиков*.

Обычный обработчик потока (stream processor)

Отвечает за применение логической схемы преобразования к данным из входящего потока. В высокоуровневом DSL такие обработчики определяются с помощью набора встроенных *операторов*, предоставляемых библиотекой Kafka Streams. Подробно они будут рассмотрены в следующих главах. Примеры операторов: `filter`, `map`, `flatMap` и `join`.

Узел-приемник (sink processor)

Место, где обогащенные, преобразованные, отфильтрованные или иным образом обработанные данные записываются обратно в Kafka, передаются другому приложению потоковой обработки или через компонент Kafka Connect отправляются в хранилище данных ниже по потоку. Приемники, как и источники, связаны с темами Kafka.

Набор операций и преобразований, через который проходят данные, образует *топологию узлов-обработчиков*, которую я далее буду называть просто *топологией*. Во всех упражнениях первой части книги мы первым делом будем проектировать топологию, то есть ориентированный ациклический граф (DAG), соединяющий источник, обработчики потока и приемник. Затем она будет реализовываться в виде кода на языке Java. В качестве примера рассмотрим следующее упражнение.

СЦЕНАРИЙ

Предположим, мы хотим создать чат-бот на основе Kafka Streams. Тема `slack-mentions` содержит все сообщения Slack, в которых упоминается наш бот `@StreamsBot`. Спроектируем его таким образом, чтобы после каждого упоминания он ожидал какой-то команды, например `@StreamsBot restart myservice`.

Нам нужен механизм проверки корректности сообщений Slack с последующими командами. При положительном результате проверки сообщение из исходной темы записывается в тему `valid-mentions`. Если же команда некорректна (например, содержит орфографическую ошибку, скажем, `@StreamsBot restart serverrr`), она будет записана в тему `invalid-mentions`.

Топология, реализующая указанные действия, показана на рис. 2.4.

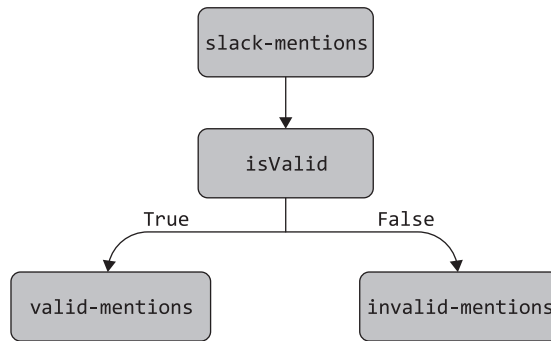


Рис. 2.4. Пример топологии с одним узлом-источником, читающим сообщения Slack (`slack-mentions`), с одним узлом-обработчиком, проверяющим корректность каждого сообщения (`isValid`), и двумя узлами-приемниками, направляющими сообщения в две темы (`valid-mentions` и `invalid-mentions`) в зависимости от результатов проверки

Уже в следующей главе мы начнем проектировать топологии и реализовывать их с использованием Kafka Streams API. Но сначала рассмотрим еще одну концепцию: субтопологии.

Субтопологии

В Kafka Streams также существует понятие субтопологий. Выше мы разработали топологию, которая принимает события из темы-источника (`slack-mentions`) и выполняет предварительную обработку потока сообщений. Но если мы имеем дело с набором тем-источников, Kafka Streams в большинстве случаев¹ выполнит разбиение на более мелкие субтопологии, чтобы еще больше распараллелить работу. Это можно сделать, поскольку операции с разными входными потоками могут выполняться независимо друг от друга.

Например, добавим к нашему чат-боту два обработчика потока. Пусть один потребляет данные из темы `valid-mentions` и выполняет адресованные боту команды (например, `restart server`), а другой — из темы `invalid-mentions` и отправляет в Slack сообщение об ошибке².

¹ Исключение составляют соединенные темы. В этом случае чтение из каждой темы-источника выполняет единая топология без дальнейшего разделения на субтопологии, иначе соединение работать не будет. Дополнительную информацию вы найдете в подразделе «Совместное секционирование» главы 4.

² В этом примере запись ведется в две промежуточные темы (`valid-mentions` и `invalid-mentions`), после чего мы немедленно потребляем из этих тем данные. Обычно такой подход применяется для определенных операций (например, для перераспределения данных). В данном случае это сделано исключительно с демонстрационной целью.

Как показано на рис. 2.5, теперь чтение в нашей топологии происходит из трех тем: `slack-mentions`, `valid-mentions` и `invalid-mentions`. При каждом чтении из нового источника Kafka Streams делит топологию на более мелкие части, допускающие независимое исполнение. В нашем примере для приложения чат-бота получилось три субтопологии. На рисунке они обозначены звездочкой.

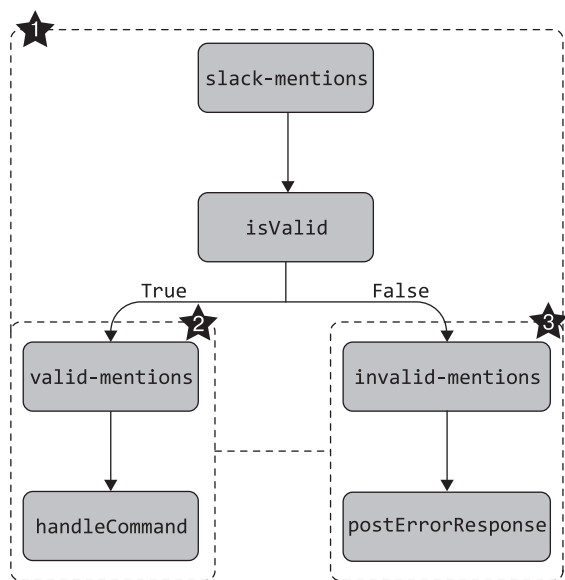


Рис. 2.5. Топология, разделенная на три субтопологии (отмечены пунктирными линиями)

Обратите внимание, что темы `valid-mentions` и `invalid-mentions` в первой субтопологии служат приемником, а во второй и третьей — источником. В подобных случаях прямой обмен данными между субтопологиями отсутствует. Записи в Kafka генерируются приемником, а читаются источниками.

Теперь, когда вы получили представление о том, как представить программу обработки потока в виде топологии, посмотрим, как этот поток проходит через узлы-обработчики в приложении Kafka Streams.

Обработка вглубь

При обработке данных в Kafka Streams происходит движение вглубь графа. Каждая запись проходит через все узлы топологии, и только потом на вход поступает новая запись. Примерная схема прохождения потока данных через Kafka Streams показана на рис. 2.6.

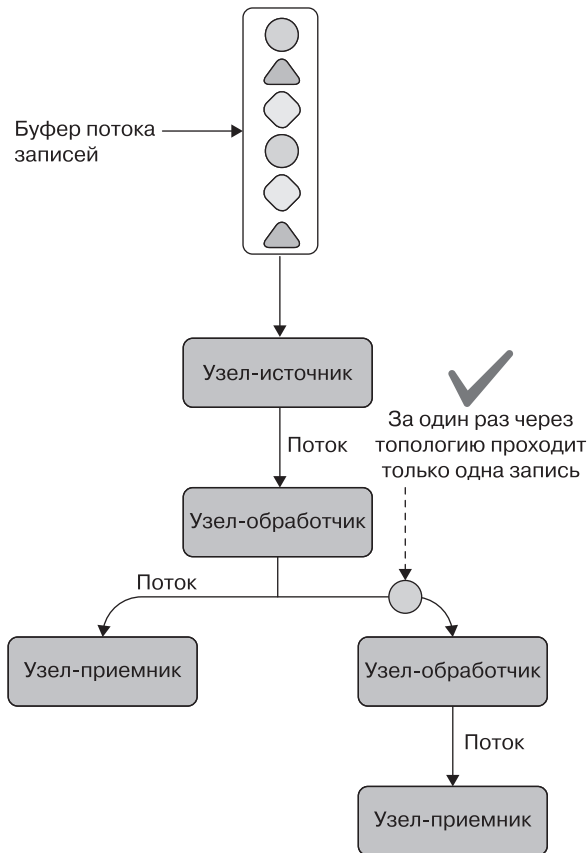


Рис. 2.6. Следующая запись начинает обрабатываться только после прохождения предыдущей через все узлы топологии

Стратегия обхода узлов топологии в глубину значительно упрощает отслеживание потока данных, но, к сожалению, при этом медленные операции обработки блокируют обработку следующих в очереди записей. Рисунок 2.7 демонстрирует ситуацию, которая в принципе невозможна в Kafka Streams: одновременное прохождение через топологию набора записей.



Если у нас есть набор субтопологий, правило одного события применяется не к топологии в целом, а к каждой из субтопологий.

Теперь посмотрим, что собой представляет программирование потоков данных и какие преимущества оно дает при создании приложений потоковой обработки.

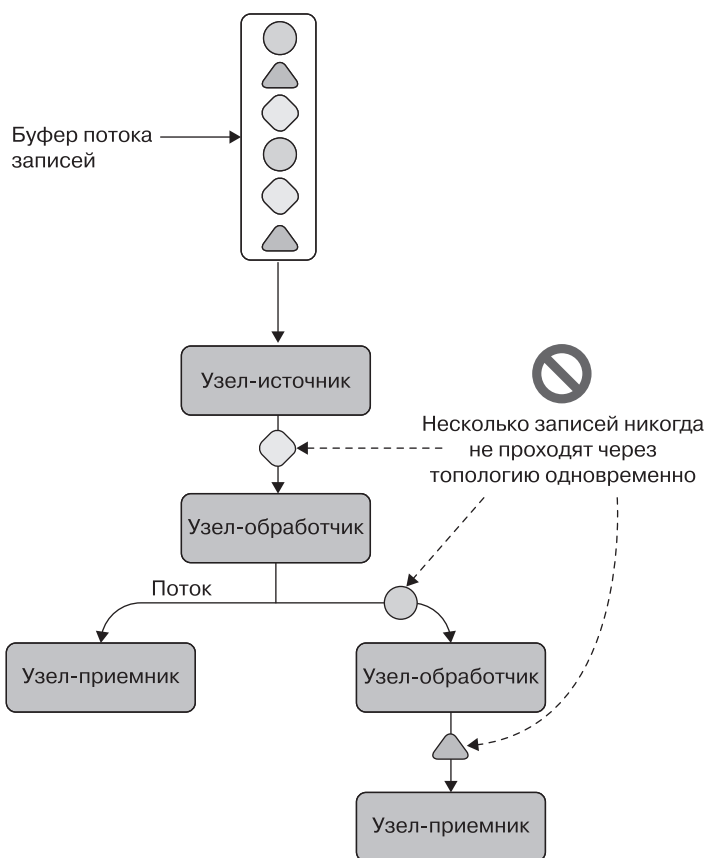


Рис. 2.7. Ситуация, которая не может возникнуть в Kafka Streams

Преимущества программирования потоков данных

Используя для создания приложений потоковой обработки библиотеку Kafka Streams и модель программирования потока данных, вы получаете ряд преимуществ. Во-первых, программа, представленная в виде ориентированного графа, проще для понимания. Для выяснения пути прохождения данных через приложение не нужно учитывать множество условных выражений и других управляющих конструкций. Узлы источника и приемника показывают точки входа данных в программу и выхода из нее, а расположенные между ними узлы-обработчики показывают, что происходит с данными внутри программы.

Кроме того, представление программы обработки потоковой информации в виде ориентированного графа позволяет стандартизировать формулировку задач по обработке данных в реальном времени и, соответственно, способ построения

решений. Приложение Kafka Streams, написанное одним человеком, будет до некоторой степени знакомо всем, кто раньше работал с этой библиотекой, не только из-за предоставляемых библиотекой абстракций, но и благодаря универсальному подходу к решению задач. Потоки данных всегда определяются с помощью операторов (узлов) и потоков (то есть связывающих эти узлы ребер). Все это упрощает разработку и поддержку приложений Kafka Streams.

Кроме того, ориентированные графы делают визуализацию потока данных интуитивно понятной даже людям, далеким от программирования. Часто команда разработчиков имеет отличный от других команд взгляд на то, как работает программа. Люди, не являющиеся разработчиками, обычно воспринимают программное обеспечение как своего рода черный ящик. Но сейчас, в эпоху законов о конфиденциальности и регламентов о защите персональных данных, требуется плотная совместная работа инженеров, юридических групп и остальных заинтересованных сторон. Соответственно, предоставленная возможность простым и наглядным способом описать, как именно приложение обрабатывает данные, позволяет коллегам, сосредоточенным на других аспектах бизнес-задачи, понять, что происходит, и даже внести в разработку свой вклад.

Наконец, топология, состоящая из источника, приемника и набора обработчиков, действует как своего рода *шаблон*, который легко создать и распараллелить между несколькими потоками и экземплярами приложений. Возможность реплицировать программу обработки потока при возрастании объема данных обеспечивает нужную производительность.

Осталось понять, как работает процесс репликации топологий, а для этого первым делом следует уяснить взаимосвязь между задачами, потоками выполнения и разделами.

Задачи и потоки выполнения

На момент определения топологии речи о выполнении программы еще нет. Мы всего лишь создаем шаблон прохождения данных через приложение. Этот шаблон может быть многократно реализован в одном экземпляре приложения и распараллелен для множества *задач* (tasks) и *потоков выполнения* (stream threads)¹. В приложении, обрабатывающем потоки данных, существует тесная взаимосвязь между количеством задач/потоков выполнения и максимальным объемом работы, который это приложение может выполнить. Именно ее понимание дает ключ к достижению хорошей производительности.

¹ Из множества типов потоков (threads), которые могут выполняться приложением Java, мы будем говорить только о тех, которые создаются и управляются библиотекой Kafka Streams, то есть отвечают за обработку потоков данных (streams).

Первым делом определим, что такое задачи.

Задача — минимальная единица параллельно выполняемой работы в приложении Kafka Streams.

При этом максимальный параллелизм приложения ограничен максимальным количеством потоковых задач. В свою очередь, число потоковых задач определяется максимальным количеством разделов тем-источников, из которых приложение считывает данные.

Энди Брайант (Andy Bryant)

Запишем эту цитату в виде математической формулы. По ней можно рассчитать количество задач, которые могут быть созданы для каждой конкретной субтопологии¹:

```
max(source_topic_1_partitions, ... source_topic_n_partitions)
```

Например, если входные данные берутся из одной темы, содержащей 16 разделов (partitions), то Kafka Streams создаст 16 задач, каждая из которых породит собственную копию базовой топологии обработчиков. После этого Kafka Streams укажет каждой задаче, из какого раздела она должна брать данные.

Как видите, задачи — это просто логические блоки, которые создают собственные экземпляры топологии узлов-обработчиков на основе назначенных им разделов. При этом реально решают каждую задачу *потоки выполнения* (threads). В Kafka Streams они спроектированы изолированными и потокобезопасными². В то время как количество создаваемых Kafka Streams задач определяется по приведенной выше формуле, количество потоков выполнения вы задаете самостоятельно через свойство `num.stream.threads`. Верхняя граница значения этого свойства равна количеству задач. Есть разные стратегии выбора этого параметра³.

¹ Еще раз напомним, что топология обработчиков в Kafka Streams может состоять из набора субтопологий, поэтому количество задач для всей программы рассчитывается как сумма задач по всем субтопологиям.

² Это не означает, что плохо реализованный узел-обработчик застрахован от проблем параллелизма. Однако по умолчанию потоки выполнения, обрабатывающие потоки данных, не имеют общего состояния.

³ Например, можно ориентироваться на количество ядер процессора, к которым у приложения есть доступ. Если экземпляр приложения использует четыре ядра, а топология поддерживает 16 задач, счетчику потоков выполнения можно присвоить значение 4: по одному потоку на ядро. Но если у этого же приложения есть доступ к 48 ядрам, вы сможете запустить максимум 16 потоков выполнения, так как верхняя граница значения счетчика определяется количеством задач.

Для наглядности давайте визуализируем два варианта конфигурации, отличающиеся количеством потоков выполнения. Предположим, приложение Kafka Streams считывает данные из темы, содержащей четыре раздела (обозначенные как P1–P4).

В первом случае присвоим параметру `num.stream.threads` значение 2, то есть наши задачи будут обрабатываться двумя потоками. Поскольку тема-источник имеет четыре раздела, будут созданы четыре задачи, которые распределятся по двум потокам, как показано на рис. 2.8.

Ситуация, когда поток выполняет более одной задачи, совершенно нормальна, но иногда количество потоков желательно увеличить, чтобы в полной мере использовать доступные ресурсы центрального процессора. Количество задач при этом остается прежним, меняется только их распределение между потоками. Например, поменяв значение параметра `num.stream.threads` на 4, мы получим ситуацию, показанную на рис. 2.9.

Теперь, когда вы получили представление об архитектуре Kafka Streams, взглянем, какие API эта библиотека предоставляет для создания приложений потоковой обработки.

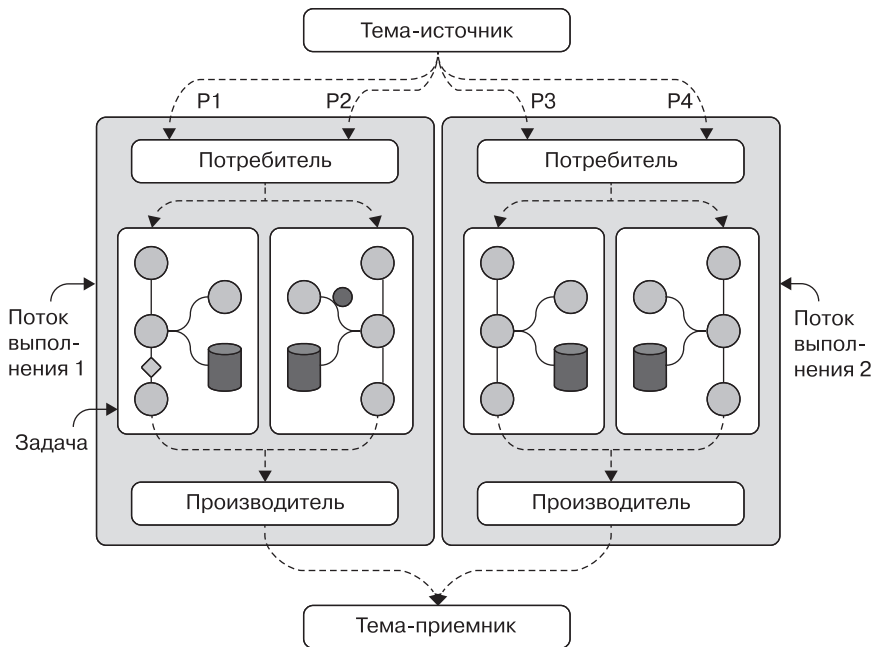


Рис. 2.8. Четыре задачи Kafka Streams, выполняемые двумя потоками

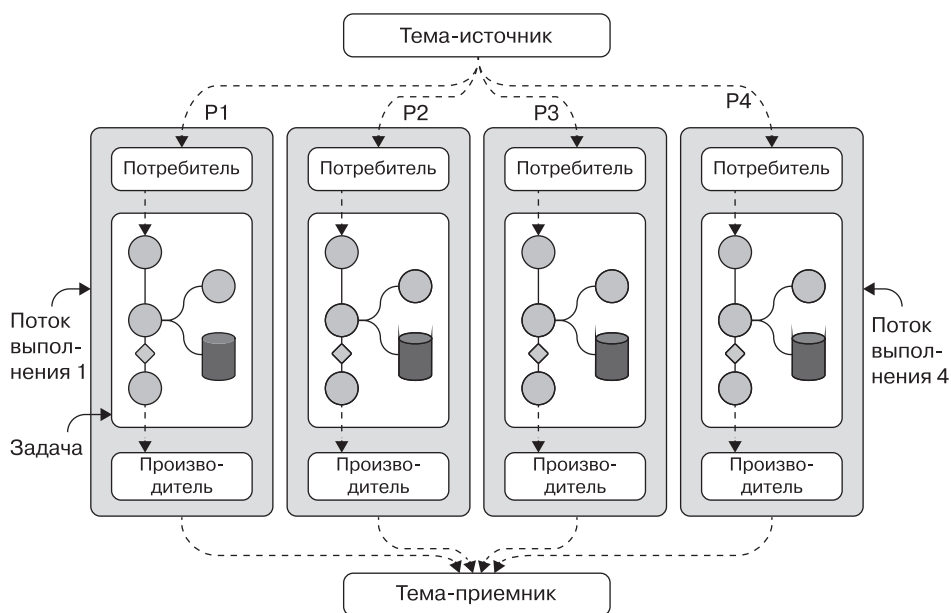


Рис. 2.9. Четыре задачи Kafka Streams, выполняемые четырьмя потоками

Высокоуровневый DSL и низкоуровневый API узлов-обработчиков

Различные решения будут проявлять себя на разных уровнях абстракции.

Джеймс Клар (James Clear)¹

В сфере разработки программного обеспечения принято считать, что за повышение уровня абстракции, как правило, приходится платить. Чем сильнее вы абстрагируетесь от деталей, тем больше утрачивается контроль над программой. Принимая решения об использовании Kafka Streams, имеет смысл задать себе вопрос, от какого типа контроля вы готовы отказаться, решив реализовать приложение для обработки потоковой информации с помощью высокоуровневой библиотеки вместо того, чтобы разрабатывать его напрямую с применением низкоуровневых API узлов-обработчиков.

¹ Из статьи First Principles: Elon Musk on the Power of Thinking for Yourself (<https://oreil.ly/Ry4nI>).

Впрочем, к счастью, Kafka Streams позволяет разработчикам, опираясь на имеющийся опыт и предпочтения, выбирать уровень абстракции, лучше всего подходящий для реализации конкретного проекта.

Можно выбрать один из двух API, схематично представленных на рис. 2.10:

- высокоуровневый язык DSL;
- низкоуровневый API узлов-обработчиков.



Рис. 2.10. Уровни абстракции API Kafka Streams

Эти API предоставляют различные возможности. Если вы хотите создать приложение для обработки потоков, используя функциональный стиль программирования, а также применять для работы с данными (потоками и таблицами) абстракции более высокого уровня, лучше выбрать DSL.

Если же вам требуется низкоуровневый доступ к данным (например, запись метаданных), возможность планировать периодические функции, более детально управлять состоянием приложения или контролировать время определенных операций, оптимально выбрать API узлов-обработчиков.

Разницу между этими двумя уровнями абстракции лучше всего показать на примере. Попробуем написать нашу первую программу.

Начало практической работы: Hello, Streams

Пришло время применить полученные знания на практике. Начнем мы со стандартной программы Hello, world, с которой обычно начинают изучение новых языков программирования и библиотек. Я покажу два варианта ее написания: с помощью высокоуровневого DSL и с помощью низкоуровневого API

узлов-обработчиков. Функционально эти варианты эквивалентны. При каждом получении сообщения из темы `users` они будут выводить на экран приветствие. Например, получив сообщение `Mitch`, каждый вариант программы отобразит надпись `Hello, Mitch`.

Но прежде чем начнется собственно программирование, требуется настроить проект.

Настройка проекта

Для всех упражнений из этой книги требуется работающий кластер Kafka. Поэтому в код каждой главы включен файл `docker-compose.yml`, позволяющий запускать этот кластер в контейнере Docker. Поскольку приложения Kafka Streams предназначены для работы за пределами кластера Kafka (например, на других машинах), этот кластер лучше рассматривать как отдельную часть инфраструктуры, не входящую в состав создаваемого приложения Kafka Streams.

Для запуска кластера Kafka клонируйте репозиторий и перейдите в каталог, содержащий инструкцию к текущей главе. Это делается следующими командами:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-02/hello-streams
```

А это команда запуска кластера:

```
docker-compose up
```

Брокер будет слушать порт 29092¹. Кроме того, предыдущая команда запустит контейнер, который создаст тему `users`. Только после этого мы сможем приступить к работе над первым приложением.

Создание проекта

Для компиляции и запуска наших приложений мы будем пользоваться инструментом сборки Gradle². Существуют и альтернативные варианты, например Maven, но я предпочел остановиться на Gradle из-за большей легкости чтения файлов сборки.

¹ При наличии утилиты `telnet` этот факт можно проверить командой `echo 'exit' | telnet localhost 29092`. Если порт открыт, в выходных данных появится `Connected to localhost`.

² Инструкции по установке Gradle вы найдете по адресу <https://gradle.org>. Для упражнений, представленных в книге, использовалась версия 6.6.1.

Еще Gradle можно использовать для быстрой начальной загрузки любых новых приложений Kafka Streams. Для этого нужно создать папку проекта, перейти в нее и воспользоваться командой `gradle init`. Вот как это выглядит:

```
$ mkdir my-project && cd my-project

$ gradle init \
  --type java-application \
  --dsl groovy \
  --test-framework junit-jupiter \
  --project-name my-project \
  --package com.example
```

Исходный код представленных в книге программ уже содержит инициализированную структуру проекта, поэтому команда `gradle init` не требуется. Я упоминаю о ней, исходя из предположения, что в какой-то момент вы начнете писать собственные приложения Kafka Streams и потребуется быстрый способ их начальной загрузки.

Вот, как выглядит базовая структура проекта для приложения Kafka Streams:

```
.
├── build.gradle ❶
├── src
│   ├── main
│   │   ├── java ❷
│   │   └── resources ❸
│   └── test
│       └── java ❹
```

❶ Это файл сборки проекта, содержащий все зависимости будущего приложения, включая библиотеку Kafka Streams.

❷ Исходный код и определения топологии сохраняются в папку `src/main/java`.

❸ Папка `src/main/resources` обычно используется для хранения конфигурационных файлов.

❹ Тесты для модулей и топологии, которые мы обсудим в подразделе «Тестирование приложений Kafka Streams» главы 12, находятся в папке `src/test/java`.

Итак, теперь вы знаете, как запустить новый проект Kafka Streams, и даже знакомы с его структурой. Пришло время добавить в проект зависимость Kafka Streams.

Добавление зависимости Kafka Streams

Для начала работы с Kafka Streams нужно добавить эту библиотеку в файл сборки в качестве зависимости. (В проектах Gradle файл сборки называется `build.gradle`.) Вот пример такого файла:

```
plugins {  
    id 'java'  
    id 'application'  
}  
  
repositories {  
    jcenter()  
}  
  
dependencies {  
    implementation 'org.apache.kafka:kafka-streams:2.7.0' ❶  
}  
  
task runDSL(type: JavaExec) { ❷  
    main = 'com.example.DslExample'  
    classpath sourceSets.main.runtimeClasspath  
}  
  
task runProcessorAPI(type: JavaExec) { ❸  
    main = 'com.example.ProcessorApiExample'  
    classpath sourceSets.main.runtimeClasspath  
}
```

❶ В проект добавлена зависимость Kafka Streams.

❷ Здесь, в отличие от остальных приведенных в книге примеров, создаются две версии топологии. В этой строке задача Gradle добавляется для выполнения DSL-версии приложения.

❸ А эта задача Gradle добавлена для выполнения версии на базе API узлов-обработчиков.

Осталось собрать проект, то есть фактически перенести в него из удаленных репозитория указанные зависимости. Это делается следующей командой:

```
./gradlew build
```

Вот и все! Библиотека Kafka Streams добавлена и готова к использованию. Теперь можно переходить к написанию кода приложения.

Вариант на базе DSL

Процесс создания приложения средствами DSL исключительно прост. Первым делом построим топологию узлов-обработчиков, в чем нам поможет класс `StreamsBuilder`:

```
StreamsBuilder builder = new StreamsBuilder();
```

Теперь, как описывалось в разделе «Топология обработчиков» выше, нужно добавить узел-источник для чтения данных из темы Kafka (в нашем случае это тема `users`). Выбор метода для решения этой задачи зависит от того, как именно мы собираемся моделировать данные. Подробно это будет обсуждаться ниже, в разделе «Потоки данных и таблицы», а пока я предлагаю представить данные в виде потока. Вот как выглядит добавление узла-источника в этом случае:

```
KStream<Void, String> stream = builder.stream("users"); ❶
```

❶ В записи `KStream<Void, String>` описывается пара «ключ — значение». В нашем случае ключ пуст (`Void`), а значение принадлежит к типу `String`. Подробно смысл этой строки будет разбираться в следующей главе.

Пришло время добавить обработчик потока. Нам нужно вывести на экран приветствие для каждого сообщения, поэтому воспользуемся оператором `foreach` с простым лямбда-выражением:

```
stream.foreach(  
    (key, value) -> {  
        System.out.println("(DSL) Hello, " + value);  
    });
```

Осталось построить топологию и запустить приложение:

```
KafkaStreams streams = new KafkaStreams(builder.build(), config);  
streams.start();
```

Полный код, включающий в себя шаблон запуска программы, приведен в примере 2.1.

Пример 2.1. Программа Hello, world на базе языка DSL

```
class DslExample {  
  
    public static void main(String[] args) {  
        StreamsBuilder builder = new StreamsBuilder(); ❶
```

```

KStream<Void, String> stream = builder.stream("users"); ❷

stream.foreach( ❸
    (key, value) -> {
        System.out.println("(DSL) Hello, " + value);
    });

// опущено для краткости
Properties config = ...; ❹

KafkaStreams streams = new KafkaStreams(builder.build(), config); ❺
streams.start();

// закрытие Kafka Streams при остановке JVM (например, в случае SIGTERM)
Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❻
}
}

```

❶ Для построения топологии используется метод `StreamsBuilder`.

❷ Добавление узла-источника, который будет осуществлять чтение из темы `users`.

❸ Оператор языка DSL `foreach` используется для вывода простого сообщения. С многочисленными операторами языка DSL я познакомлю вас в следующих главах.

❹ Для краткости мы опустили процесс выбора конфигурации Kafka Streams. Он будет рассмотрен в следующих главах. Именно в этих настройках указывается, из какого кластера Kafka должно читать наше приложение и к какой группе потребителей оно принадлежит.

❺ Построение топологии и запуск библиотеки Kafka Streams.

❻ Закрываем библиотеку Kafka Streams после завершения работы JVM.

Осталось запустить приложение командой:

```
./gradlew runDSL --info
```

Теперь приложение перехватывает входящие данные. Если помните, в разделе «Hello, Kafka» главы 1 упоминалось, что произвести данные для кластера Kafka можно с помощью консольного сценария `kafka-console-producer`. Здесь требуется вот такая последовательность команд:

```
docker-compose exec kafka bash ❶
```

```

kafka-console-producer \ ❷
  --bootstrap-server localhost:9092 \
  --topic users

```


❶ Консольные сценарии доступны внутри контейнера `kafka`, где работает брокер для нашего кластера разработки. Эти сценарии можно скачать как часть официального дистрибутива Kafka.

❷ Запуск локального производителя, который будет записывать данные в тему `users`.

Увидев приглашение на ввод данных, создайте одну или несколько записей, печатая имя пользователя и нажимая клавишу `Enter`. Например, сформируйте вот такой список имен:

```
>angie
>guy
>kate
>mark
```

Затем нажмите `Control+C` для выхода из командной строки. Приложение Kafka Streams должно вывести на экран следующие приветствия:

```
(DSL) Hello, angie
(DSL) Hello, guy
(DSL) Hello, kate
(DSL) Hello, mark
```

Как видите, приложение работает корректно. В следующих главах я покажу более интересные примеры, здесь же основное внимание следует обратить на процессы определения топологии и запуска приложения. Именно они составляют основу для всего остального. А теперь посмотрим, как эта же топология создается с помощью низкоуровневого API узлов-обработчиков.

API узлов-обработчиков

В API узлов-обработчиков отсутствуют абстракции, доступные в высокоуровневом DSL. Поэтому для построения топологии применяются такие методы, как `Topology.addSource`, `Topology.addProcessor` и `Topology.addSink` (последний мы сейчас использовать не будем). Первым делом создадим новый экземпляр `Topology`:

```
Topology topology = new Topology();
```

Теперь нам нужны узел-источник для чтения данных из темы `users` и узел-обработчик для вывода приветствия на экран. Последний ссылается на класс `SayHelloProcessor`, который мы скоро реализуем:

```
topology.addSource("UserSource", "users"); ❶
topology.addProcessor("SayHello", SayHelloProcessor::new, "UserSource"); ❷
```

❶ Первый аргумент метода `addSource` — имя узла-обработчика. Мы назвали его `UserSource`. Ссылка на него появится в следующей строке при подключении дочернего обработчика, определяющего, как данные должны проходить через нашу топологию. Вторым аргументом — название темы, из которой будет осуществляться чтение. В нашем случае это тема `users`.

❷ Здесь создается следующий узел-обработчик `SayHello`. Его действия задает класс `SayHelloProcessor`, который мы создадим в следующем разделе. В API узлов-обработчиков для соединения узлов друг с другом указывается имя родительского обработчика. В рассматриваемом случае обработчик `UserSource` является предком по отношению к обработчику `SayHello`, то есть данные передаются из первого во второй.

При создании приложения средствами DSL на этом этапе строилась топология, и затем она запускалась методом `streams.start()`. Здесь мы поступаем аналогично:

```
KafkaStreams streams = new KafkaStreams(topology, config);
streams.start();
```

Для запуска кода практически все готово, осталось только реализовать класс `SayHelloProcessor`. При создании узлов с помощью API узлов-обработчиков необходимо реализовать интерфейс `Processor`, который определяет методы инициализации обработчиков потока (`init`), логическую схему обработки отдельных записей (`process`) и функцию очистки состояния (`close`). В нашем примере процедуры инициализации и очистки не требуются.

Мы будем использовать вот такую простую реализацию класса `SayHelloProcessor`. Более сложные примеры и все методы интерфейса `Processor` (`init`, `process` и `close`) будут подробно рассматриваться в главе 7.

```
public class SayHelloProcessor implements Processor<Void, String, Void, Void> {
❶
    @Override
    public void init(ProcessorContext<Void, Void> context) {} ❷

    @Override
    public void process(Record<Void, String> record) { ❸
        System.out.println("(Processor API) Hello, " + record.value());
    }

    @Override
    public void close() {} ❹
}
```

❶ В описании интерфейса `Processor` (`Processor <Void, String, ..., ...>`) первые два параметра задают тип ключа и тип значения *входных* данных. Ключи

в нашем случае отсутствуют, а значениями выступают имена пользователей (то есть текстовые строки), поэтому используются типы `Void` и `String`. Последнее два параметра (`Processor <..., ..., Void, Void>`) задают тип ключа и тип значения *выходных* данных. Наш класс `SayHelloProcessor` просто выводит на экран приветствие, то есть мы не пересылаем ниже по графу никаких выходных ключей или значений. Поэтому здесь подойдет тип `Void`¹.

❷ Инициализация в этом примере не требуется, поэтому тело метода оставлено пустым. В описании интерфейса `ProcessorContext (ProcessorContext <Void, Void>)` указаны тип ключа и тип значения выходных данных. Но мы не пересылаем дальше никаких сообщений, поэтому в обоих случаях это тип `Void`.

❸ Логическая схема обработки помещена в метод интерфейса `Processor`, имеющий исчерпывающее название `process`. В рассматриваемом случае он должен всего лишь выводить на экран приветствие. Обратите внимание, что в обобщенном шаблоне для интерфейса `Record` фигурируют типы ключа и значения *входных* записей.

❹ В этом примере специальной очистки не требуется.

Теперь можно запустить код, используя уже знакомую по предыдущему разделу команду:

```
./gradlew runProcessorAPI --info
```

На экране должны появиться следующие строки, свидетельствующие о том, что приложение Kafka Streams работает должным образом:

```
(Processor API) Hello, angie  
(Processor API) Hello, guy  
(Processor API) Hello, kate  
(Processor API) Hello, mark
```

Как видите, API узлов-обработчиков позволяет написать такую же программу. Но, несмотря на его широкие возможности, с которыми я познакомлю вас в главе 7, DSL часто оказывается более предпочтительным, так как он дает доступ к двум мощным абстракциям: потокам и таблицам. Давайте разберем их более подробно.

¹ Эта версия интерфейса `Processor` появилась в Kafka Streams 2.7, переведя в разряд устаревших версию, которая использовалась в Kafka Streams 2.6 и ранее. В более ранней версии указывались только типы вводимых данных, что вызывало некоторые проблемы при проверке безопасности типов, поэтому рекомендуется пользоваться последней версией интерфейса `Processor`.

Потоки данных и таблицы

Если вы внимательно посмотрите на код примера 2.1, то заметите, что для чтения из темы Kafka в *поток данных* использовался оператор DSL `stream`:

```
KStream<Void, String> stream = builder.stream("users");
```

Однако потоки Kafka поддерживают еще одно представление данных: *таблицы*. Посмотрим, как потоки данных связаны с таблицами, и поговорим о том, в каких случаях лучше использовать каждый из вариантов.

Еще раз напомним, что при проектировании топологии нам следует определить набор узлов-источников и узлов-приемников, связанных с темами, из которых приложение будет осуществлять чтение и запись. Вместо непосредственной работы с темами Kafka предметно-ориентированный язык Kafka Streams позволяет работать с двумя их *представлениями*, у каждого из которых есть свои варианты применения. То есть данные из темы Kafka можно смоделировать в виде *потока* (stream) или *таблицы* (которую также называют *потокм журналов изменений*). Разницу между этими представлениями проще всего пояснить на примере.

Предположим, тема содержит журнал подключений по ssh и каждая запись в этом журнале связана с идентификатором пользователя, как показано в табл. 2.2.

Таблица 2.2. Снабженные ключами записи из одного раздела темы

Ключ	Значение	Смещение
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

Прежде чем начинать какие-либо манипуляции с этими данными, нужно решить, какую абстракцию мы будем использовать: поток или таблицу. Решение следует принимать, исходя из того, что именно мы хотим отслеживать: только последнее состояние/представление рассматриваемого ключа или всю историю сообщений.

Сравним оба варианта.

Потоки

Поток можно рассматривать как вставки в базу данных. В таком представлении журнала можно проследить за каждой записью. Представление темы в виде потока демонстрируется в табл. 2.3.

Таблица 2.3. Записи в журнале подключений по ssh представлены в виде потока данных

Ключ	Значение	Смещение
mitch	{ "action": "login" }	0
mitch	{ "action": "logout" }	1
elyse	{ "action": "login" }	2
isabelle	{ "action": "login" }	3

Таблицы

Таблицу можно рассматривать как обновление базы данных. В этом представлении журналов для каждого ключа сохраняется только текущее состояние (либо последняя запись, либо какой-то результат агрегирования). Таблицы обычно строятся из *сжатых тем*. Это темы, у которых параметр `cleanup.policy` имеет значение `compact`, сигнализируя Kafka, что сохранять нужно только последнее состояние каждого ключа. Табличное представление нашей темы показано в табл. 2.4.

Таблица 2.4. Табличное представление записей в журнале подключений по ssh

Ключ	Значение	Смещение
Mitch	{ "action": "logout" }	1
Elyse	{ "action": "login" }	2
Isabelle	{ "action": "login" }	3

Таблицы по своей природе сохраняют состояние и часто используются в Kafka Streams для агрегирования¹. Впрочем, данные в табл. 2.4 вовсе не результат агрегирования, а просто последние сохраненные ssh-события для каждого идентификатора пользователя. Хотя операцию агрегирования таблицы тоже поддерживают. Например, вместо отслеживания последней записи для каждого ключа мы можем легко подсчитать количество связанных с ним записей, получив в итоге табл. 2.5.

Таблица 2.5. Агрегированное табличное представление записей в журнале подключений по ssh

Ключ	Количество значений	Смещение
mitch	2	1
elyse	1	2
isabelle	1	3

Внимательные читатели, скорее всего, обратили внимание на разницу в устройстве уровня хранения Kafka (который представляет собой распределенный журнал, обновляющийся только путем добавления записей) и таблицы. Записи в Kafka не подлежат редактированию. Соответственно, возникает вопрос: каким образом можно смоделировать данные как обновления, используя *табличное* представление темы Kafka?

Дело в том, что таблица реализуется на стороне Kafka Streams с использованием хранилища данных типа «ключ — значение» на базе RocksDB². Потребляя упорядоченный поток событий и фиксируя в хранилище на стороне клиента (в Kafka Streams его называют *хранилищем состояний*) последнюю запись для каждого ключа, мы и получаем представление в виде таблицы. Другими словами, таблица появляется не в Kafka, а на стороне клиента.

Реализует эту идею небольшой фрагмент кода на языке Java. Поток в нем представлен списком `List`, так как именно список содержит упорядоченный набор записей³, а таблица создается итеративным обходом этого списка

¹ Фактически таблицы иногда называют агрегированными потоками. См. статью *Of Streams and Tables in Kafka and Stream Processing, Part 1* Майкла Нолла (<https://oreil.ly/dgSCn>), в которой подробно раскрывается эта тема.

² Это высокопроизводительная встроенная база данных для быстрых накопителей, изначально разработанная в Facebook. Более подробно RocksDB и другие хранилища данных типа «ключ — значение» будут рассматриваться в главах 4–6.

³ При этом индекс каждого элемента списка будет представлять смещение записи в теме Kafka.

(`stream.forEach`) и сохранением с помощью структуры данных `Map` последней записи для каждого ключа:

```
import java.util.Map.Entry;

var stream = List.of(
    Map.entry("a", 1),
    Map.entry("b", 1),
    Map.entry("a", 2));

var table = new HashMap<>();

stream.forEach((record) -> table.put(record.getKey(), record.getValue()));
```

Выведем на экран значения для потока и таблицы:

```
stream ==> [a=1, b=1, a=2]

table ==> {a=2, b=1}
```

В Kafka Streams этот функционал реализуется более продвинутыми средствами, например, вместо хранящей данные в оперативной памяти структуры `Map` можно использовать отказоустойчивые структуры данных. Но связь между потоками и таблицами не ограничивается возможностью представить бесконечный поток в виде таблицы. Посмотрим, как она работает в другую сторону.

Потоково-табличный дуализм

Дуализм таблиц и потоков иллюстрирует тот факт, что таблицы могут быть представлены в виде потоков, а потоки могут использоваться для восстановления таблиц. Преобразование потока в таблицу я показал в предыдущем разделе, как объяснение несоответствия между не подлежащими редактированию записями в журнале Kafka и гибкой структурой таблицы, допускающей обновление данных.

Способность реконструировать таблицы из потоков не уникальна для Kafka Streams. По такому же принципу происходит процесс репликации в MySQL: на базе потока событий (то есть изменений строк) в копии базы реконструируется исходная таблица. В системе управления базами данных Redis есть файл журнала AOF (append-only file), в котором фиксируются все команды на запись, приводящие к изменению реальных данных в памяти. В случае сбоя сервера Redis воспроизведение этого файла позволяет восстановить весь набор данных.

Теперь посмотрим на вторую сторону этой медали. Таблица, по сути, дает представление потока в конкретный момент времени. Как вы уже видели,

обновляются они при поступлении новых записей. При преобразовании таблицы в поток обновление реализуется как вставка. В конец журнала просто добавляется новая запись. Это наглядно иллюстрируют следующие строки кода:

```
var stream = table.entrySet().stream().collect(Collectors.toList());  
  
stream.add(Map.entry("a", 3));
```

Выведя на экран содержимое потока, вы увидите, что на этот раз вместо обновления произошла вставка:

```
stream ==> [a=2, b=1, a=3]
```

До сих пор в примерах использовались стандартные библиотеки Java, так как требовалось выработать у вас интуитивное понимание концепций потока и таблицы. Но в реальной работе с потоками и таблицами в Kafka Streams применяются специализированные абстракции, с которыми я вас сейчас познакомлю.

KStream, KTable, GlobalKTable

Одно из преимуществ высокоуровневого DSL — наличие набора абстракций, сильно упрощающего работу с потоками и таблицами. Вот их список.

KStream

Абстракция секционированного *потока записей*, в котором для представления данных используется семантика вставки (то есть каждое событие считается независимым от остальных).

KTable

Абстракция секционированной таблицы (то есть *потока изменений*), в которую данные добавляются путем обновления (приложение отслеживает последнее представление каждого ключа). Из-за секционирования каждая задача Kafka Streams содержит только часть полной таблицы¹.

GlobalKTable

Эта абстракция, в отличие от KTable, содержит полную (то есть несекционированную) копию базовых данных. О том, в каких случаях следует использовать KTable, а в каких — GlobalKTable, я расскажу в главе 4.

Приложения Kafka Streams могут использовать как несколько абстракций потоков/таблиц, так и только какую-то одну. Это зависит от реализуемого сценария и более подробно будет рассматриваться в следующих главах.

¹ Здесь предполагается, что в исходной теме более одного раздела.

Заключение

Поздравляю с завершением первого свидания с Kafka Streams! Вот что вы на нем узнали.

- В экосистеме Kafka библиотека Kafka Streams находится на уровне обработки потоков. Именно здесь происходят преобразование и обогащение данных.
- Библиотека Kafka Streams создана для упрощения процесса создания приложений потоковой обработки. Для этого она оснащена простым функциональным API и набором примитивов потоковой обработки. В случаях, когда необходим более детальный контроль генерируемой топологии, можно прибегнуть к низкоуровневому API узлов-обработчиков.
- Освоить и развернуть Kafka Streams проще, чем фреймворк Apache Flink и расширение Spark Streaming. Кроме того, в Kafka Streams поддерживается поочередная обработка событий, что, собственно, и считается настоящей потоковой обработкой. Эта библиотека отлично подходит для ситуаций, когда принимать решения и обрабатывать данные требуется в реальном времени. Кроме того, она надежна, удобна в обслуживании, масштабируема и эластична.
- Установка и запуск Kafka Streams очень просты. Примеры кода из этой главы находятся в хранилище <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb>.

В следующей главе я расскажу, как в Kafka Streams осуществляется обработка без сохранения состояния. Вы получите практический опыт работы с новыми операторами языка DSL, которые помогут построить более продвинутые и мощные приложения для обработки потоковых данных.

ГЛАВА 3

Обработка без сохранения состояния

https://t.me/it_boooks

Простейшая форма потоковой обработки не требует запоминания ранее просмотренных событий. Каждое событие обрабатывается¹ и сразу забывается. Эта парадигма называется *обработкой без сохранения состояния* (stateless processing), и в Kafka Streams множество операций выполняются именно таким способом.

Некоторые из этих операций будут рассмотрены в этой главе, и вы увидите, как легко они помогают решить некоторые часто встречающиеся задачи. В частности, вы узнаете, как осуществляются:

- фильтрация записей;
- добавление и удаление полей;
- изменение ключей записей;
- ветвление потоков;
- слияние потоков;
- преобразование записей в один или несколько выходов;
- поочередное обогащение записей.

Со всеми этими операциями я познакомлю вас на практическом примере. Из сервиса микроблогов Twitter мы будем читать поток данных, связанных с криптовалютами, и с помощью операций без сохранения состояния преобразуем этот поток в торговые сигналы. К концу главы вы получите первые навыки обогащения и преобразования неструктурированных данных, заложив таким образом базу для восприятия более продвинутых концепций.

¹ Обработка — многозначное слово. Я использую его в самом широком смысле, подразумевая процессы обогащения и преобразования данных, реагирования на них, а при необходимости и записи обработанных данных в тему.

Но для начала рассмотрим, что же это такое — обработка без сохранения состояния и чем она отличается от *обработки с сохранением состояния* (stateful processing).

Обработка с сохранением и без сохранения состояния

При создании любого приложения Kafka Streams важно учитывать, требуется ли отслеживать состояние обрабатываемых данных. От ответа на этот вопрос зависит тип будущего приложения. Вот чем отличаются ситуации с отслеживанием истории или без него.

- В *приложениях без сохранения состояния* каждое событие обрабатывается независимо от всех остальных. Данные в этом случае представлены в виде потока (см. раздел «Потоки и таблицы» главы 2). Другими словами, приложение обрабатывает каждое событие как автономную вставку и не требует информации о более ранних событиях.
- *Приложения с сохранением состояния* должны помнить о событиях, случившихся на одном или нескольких предыдущих узлах-обработчиках. Обычно это необходимо для агрегирования, оконной обработки данных или соединения потоков событий. Это более сложный вариант приложения из-за необходимости отслеживать дополнительные данные или *состояние*.

Приложение для потоковой обработки, которое мы в итоге создадим средствами высокоуровневого языка DSL, сводится к набору операций, выполняемых в разных местах топологии¹. Операции преобразования данных представляют собой различные методы (например, `filter`, `map`, `flatMap`, `join` и т. п.), которые будут применяться к событиям по мере их прохождения через топологию. Некоторые из них, такие как `filter`, считаются операциями *без сохранения состояния*, ведь для работы этому методу достаточно просмотреть текущую запись. В нашем примере он будет определять, следует ли пересылать записи следующим узлам-обработчикам. Другие операции, такие как метод `count`, *сохраняют состояние*, поскольку им требуются сведения о предыдущих событиях. Без информации о том, сколько событий он уже обработал к текущему моменту, метод `count` не сможет отслеживать количество сообщений.

Если приложению Kafka Streams *достаточно* операций без сохранения состояния (то есть у него нет необходимости запоминать ранее случившиеся события), оно считается *не сохраняющим состояние*. Но добавление в него хотя бы одной

¹ Сейчас мы сосредоточимся на языке DSL, а про обработку без сохранения состояния и с сохранением состояния с помощью API узлов-обработчиков я расскажу в главе 7.

сохраняющей состояние операции превращает его в приложение *с сохранением состояния*. Обеспечивать удобство обслуживания, масштабируемость и отказоустойчивость при этом становится гораздо сложнее, поэтому такой вариант потоковой обработки мы рассмотрим отдельно в следующей главе.

Не волнуйтесь, если пока все выглядит не очень понятно. Я наглядно продемонстрирую все эти концепции в процессе работы над приложением Kafka Streams. После практики использования операций без сохранения состояния многое станет на свои места. Поэтому просто приступим к делу.

Обработка потока ТВИТОВ

В качестве примера рассмотрим вариант алгоритмической торговли. Для так называемой *высокочастотной торговли* (high-frequency trading, HFT) необходимо программное обеспечение, которое будет автоматически классифицировать ценные бумаги и в зависимости от результата осуществлять их покупку или продажу. Оно должно с минимальной задержкой обрабатывать различные типы торговых сигналов и реагировать на них.

Соответственно, нашему вымышленному программному обеспечению требуется приложение потоковой обработки, которое поможет в оценке рыночных перспектив различных типов криптовалют (Bitcoin, Ethereum, Ripple и пр.). На базе этой оценки написанный нами алгоритм для биржевой торговли будет принимать решения о покупке/продаже¹. В микроблогах Twitter своими соображениями по поводу различных криптовалют делятся миллионы пользователей, поэтому мы используем Twitter в качестве источника данных для нашего приложения.

Первым делом составим пошаговую схему разработки топологии, которая затем ляжет в основу нашего приложения Kafka Streams. Ключевые концепции выделены курсивом.

1. Твиты, в которых упоминаются определенные цифровые валюты (`#bitcoin`, `#ethereum`), потребляются из темы-источника `tweets`.
 - Все записи сделаны в формате JSON, и нужно понять, как правильно *десериализовать* их в классы данных более высокого уровня.
 - Для упрощения кода при десериализации удаляются лишние поля. Отбор подмножества полей называется *проекцией*. Это одна из наиболее распространенных задач потоковой обработки.

¹ Разумеется, такой алгоритм будет незавершенным, поскольку в идеале для биржевой торговли следует учитывать множество типов сигналов, а не только эмоциональные оценки пользователей сервиса микроблогов.

2. Ретвиты из обработки исключаются с помощью *фильтрации* данных.
3. Твиты, написанные не на английском языке, *ответвляются* в отдельный поток для последующего перевода.
4. Неанглоязычные твиты переводятся на английский. Это происходит путем *отображения* входного значения (твит на иностранном языке) в выходное значение (англоязычный твит).
5. Переведенные твиты *сливаются* в поток англоязычных.
6. Оценивается эмоциональная окраска (тональность) каждого твита, чтобы понять отношение пользователей сервиса микроблогов Twitter к обсуждаемым криптовалютам. В одном твите может упоминаться несколько криптовалют, поэтому каждое входное значение (твит) мы преобразуем в переменное количество выходных значений методом `flatMap`.
7. Обогащенные твиты сериализуются с помощью фреймворка Avro и записываются в тему-приемник `crypto-sentiment`. Именно отсюда будет потреблять данные наш вымышленный алгоритм биржевой торговли, принимающий инвестиционные решения на основе обнаруженных сигналов.

Составленная по этой схеме топология узлов-обработчиков показана на рис. 3.1. Это маршрут прохождения данных через наше будущее приложение Kafka Streams.

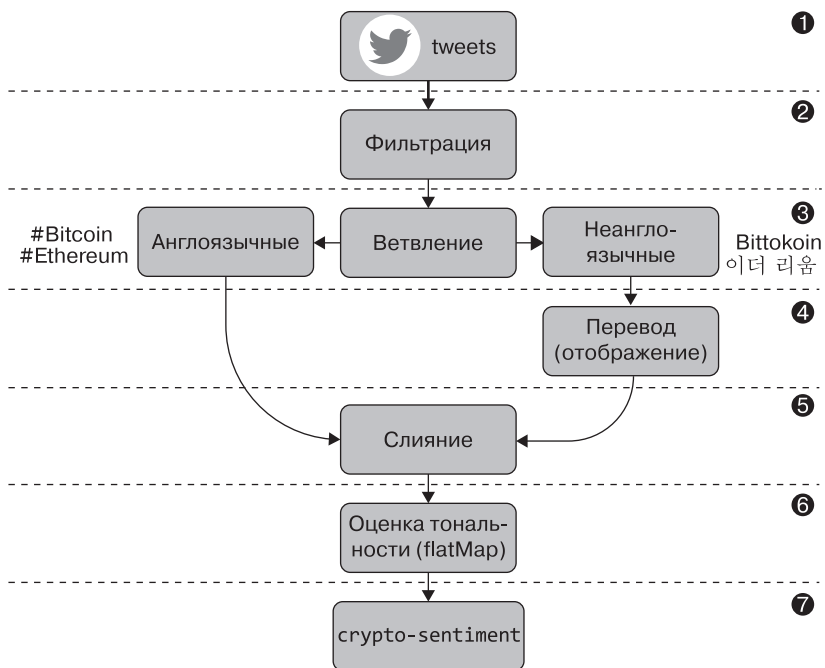


Рис. 3.1. Топология будущего приложения для обогащения твитов

Теперь с готовой топологией в руках и пошаговым планом ее реализации можно приступать к созданию приложения Kafka Streams. Первым делом выполним настройку проекта, а затем организуем поток твитов из темы-источника.

Настройка проекта

Примеры кода к этой главе вы найдете по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>.

Чтобы использовать репозиторий на каждом этапе работы над нашей топологией, клонируйте его и перейдите в каталог, содержащий инструкцию к текущей главе. Это делается следующими командами:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-03/crypto-sentiment
```

Сборка проекта осуществляется командой:

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



Детали реализации перевода твитов на английский язык и анализа их тональности (шаги 4 и 6) я показывать не буду, так как для демонстрации операций без сохранения состояния в Kafka Streams они не требуются. Но в репозитории GitHub представлен код полностью работающего примера, и те, кому интересны эти детали, могут найти информацию в файле README.md проекта.

Теперь для работы над приложением Kafka Streams все готово.

Добавление узла-источника KStream

Все приложения Kafka Streams потребляют данные из одной или нескольких исходных тем. В нашем примере фигурирует всего одна тема-источник `tweets`. Ее заполнение осуществляется с помощью специального коннектора (<https://oreil.ly/yvEoX>), который собирает твиты через Streaming API и передает записи в кодировке JSON в Kafka. Своеобразный анализ значения твита¹ демонстрирует пример 3.1.

¹ Ключам записей мы в этой главе уделять внимания не будем, поскольку наше приложение не выполняет с ними никаких операций.

Пример 3.1. Значение записи в теме-источнике tweets

```
{
  "CreatedAt": 1602545767000,
  "Id": 1206079394583924736,
  "Text": "Anyone else buying the Bitcoin dip?",
  "Source": "",
  "User": {
    "Id": "123",
    "Name": "Mitch",
    "Description": "",
    "ScreenName": "timeflown",
    "URL": "https://twitter.com/timeflown",
    "FollowersCount": "1128",
    "FriendsCount": "1128"
  }
}
```

Мы знаем, какие данные нам требуются, теперь приложению Kafka Streams нужно извлечь их из темы-источника. В предыдущей главе мы говорили о том, что для представления потока записей без сохранения состояния применяется абстракция `KStream`. Вот код добавления узла-источника `KStream` в приложение Kafka Streams:

```
StreamsBuilder builder = new StreamsBuilder(); ❶
```

```
KStream<byte[], byte[]> stream = builder.stream("tweets"); ❷
```

❶ В приложениях на базе высокоуровневого DSL топологии узлов-обработчиков строятся с использованием экземпляра `StreamsBuilder`.

❷ Экземпляры `KStream` создаются путем передачи имени темы в метод `StreamsBuilder.stream`. Метод `stream` принимает дополнительные параметры, о которых я расскажу чуть позже.

Обратите внимание, что только что созданный экземпляр `KStream` имеет параметры типа `byte[]`:

```
KStream<byte[], byte[]>
```

Я уже кратко касался этого в предыдущей главе, а сейчас напомним, что в описании интерфейса `KStream` фигурируют два параметра: один задает в теме Kafka тип ключей (K), а второй — тип значений (V). В библиотеке Kafka Streams интерфейс описывается вот так:

```
public interface KStream<K, V> {
    // опущено для краткости
}
```

Из нашего экземпляра `KStream`, описанного как `KStream <byte[], byte[]>`, следует, что записи, потребляемые из темы `tweets`, представлены в виде байтовых

массивов ключей и значений. Однако в начале раздела я упоминал, что коннектор-источник преобразует записи твитов в объекты JSON (см. пример 3.1). Что это нам дает?

По умолчанию Kafka Streams представляет проходящие через приложение данные в виде байтовых массивов. Это связано с тем, что брокер Kafka хранит и передает данные в виде необработанных последовательностей байтов, то есть представление данных в виде байтового массива будет рабочим во всех случаях (поэтому именно его имеет смысл использовать по умолчанию). Хранение и передача необработанных последовательностей байтов обеспечивает, во-первых, гибкость работы брокера Kafka, поскольку клиентам не навязывается определенный формат данных, а во-вторых, скорость этой работы, так как для передачи таких данных по сети брокер использует меньше памяти и циклов центрального процессора¹. Но это означает, что для работы с более высокоуровневыми объектами и форматами, такими как строки (с разделителями и без), JSON, Avro, Protobuf и т. п. клиентам Kafka, в том числе и приложениям Kafka Streams, приходится выполнять сериализацию и десериализацию этих байтовых потоков².

Прежде чем мы перейдем к десериализации записей твитов в объекты более высокого уровня, напомним код запуска нашего будущего приложения Kafka Streams. Для упрощения тестирования полезно отделить код построения топологии Kafka Streams от кода, запускающего приложение. Шаблон нашего кода будет содержать два класса. Первый, показанный в примере 3.2, отвечает за построение топологии.

Пример 3.2. Класс Java, определяющий нашу топологию Kafka Streams

```
class CryptoTopology {

    public static Topology build() {
        StreamsBuilder builder = new StreamsBuilder();

        KStream<byte[], byte[]> stream = builder.stream("tweets");
        stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream")); ❶

        return builder.build();
    }
}
```

¹ Хранение и передача данных в виде байтового массива позволяет Kafka использовать так называемое нулевое копирование (zero-copy). Это означает, что при сериализации/десериализации данных не приходится переключаться между режимом пользователя и режимом ядра, так как обработка происходит на стороне клиента. Такой подход дает выигрыш в производительности.

² Когда я пишу, что коннектор предоставляет записи из Twitter в формате JSON, это не означает, что записи твитов хранятся в Kafka в виде необработанного текста JSON. Просто байты, представляющие твиты в теме Kafka, после десериализации должны оказаться в этом формате.

❶ Оператор `print` позволяет легко просматривать данные по мере их прохождения через приложение. Обычно его рекомендуется применять только на стадии разработки.

Второй класс, который мы назовем `App`, создаст экземпляр нашей топологии и запустит его, как показано в примере 3.3.

Пример 3.3. Отдельный класс Java для запуска приложения Kafka Streams

```
class App {
    public static void main(String[] args) {
        Topology topology = CryptoTopology.build();

        Properties config = new Properties(); ❶
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
        config.put(StreamsConfig.BootstrapServersConfig, "localhost:29092");

        KafkaStreams streams = new KafkaStreams(topology, config); ❷

        Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❸

        System.out.println("Starting Twitter streams");
        streams.start(); ❹
    }
}
```

❶ Kafka Streams требует установки базовой конфигурации, включающей идентификатор приложения, который соответствует группе потребителей, и серверы начальной загрузки Kafka. Конфигурационные настройки осуществляются через объект `Properties`.

❷ Создаем экземпляр объекта `KafkaStreams` с топологией узлов-обработчиков и конфигурацией потоков.

❸ Добавляем хук отключения для корректного прекращения работы приложения Kafka Streams при получении глобального сигнала остановки.

❹ Запускаем приложение Kafka Streams. Обратите внимание, что метод `streams.start()` не блокируется, так как за создание топологии отвечают фоновые обрабатывающие потоки. Именно поэтому требуется хук отключения.

Все готово. После запуска приложения и предоставления ему данных из темы `tweets` мы увидим на экране необработанные байтовые массивы (загадочные наборы символов, которые появляются после запятой в каждой строке вывода):

```
[tweets-stream]: null, [B@c52d992
[tweets-stream]: null, [B@a4ec036
[tweets-stream]: null, [B@3812c614
```

Очевидно, что работать с низкоуровневыми байтовыми массивами несколько затруднительно. Для реализации следующих шагов обработки потока нужен другой метод представления данных в теме-источнике. В этом нам помогут такие процессы, как сериализация и десериализация данных.

Сериализация/десериализация

Схема обработки потока входящих байтов платформой Kafka показана на рис. 3.2. За преобразование этого потока в объекты более высокого уровня отвечают клиенты, в нашем случае Kafka Streams. Такое преобразование называется *десериализацией*. Для возвращения данных в Kafka клиент должен преобразовать их обратно в байтовый массив.

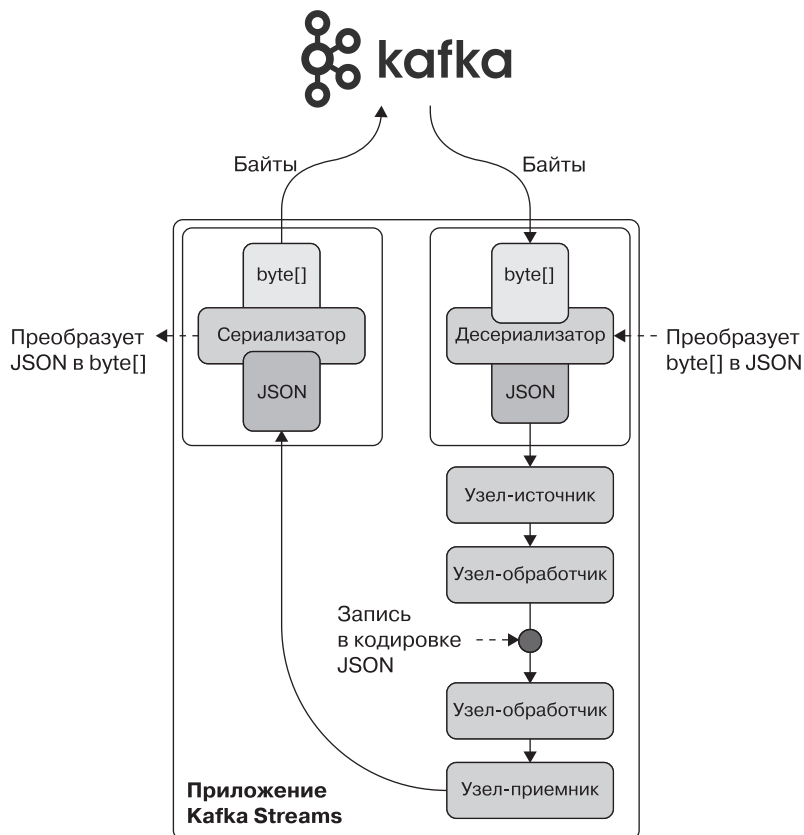


Рис. 3.2. Схема, дающая представление о том, где в приложении Kafka Streams происходят процессы десериализации и сериализации

В библиотеке Kafka Streams классы сериализатора и десериализатора часто объединяются в один класс *Serdes*. Существующие реализации для разных типов данных перечислены в табл. 3.1¹. Например, реализация для типа `String` (доступная через метод `Serdes.String()`) включает в себя классы сериализатора и десериализатора строк.

Таблица 3.1. Доступные в Kafka Streams реализации класса *Serdes*

Тип данных	Класс <i>Serdes</i>
<code>byte[]</code>	<code>Serdes.ByteArray()</code> , <code>Serdes.Bytes()</code>
<code>ByteBuffer</code>	<code>Serdes.ByteBuffer()</code>
<code>Double</code>	<code>Serdes.Double()</code>
<code>Integer</code>	<code>Serdes.Integer()</code>
<code>Long</code>	<code>Serdes.Long()</code>
<code>String</code>	<code>Serdes.String()</code>
<code>UUID</code>	<code>Serdes.UUID()</code>
<code>Void</code>	<code>Serdes.Void()</code>

Каждый раз, когда возникает необходимость десериализовать или сериализовать данные в Kafka Streams, первым делом проверяйте, можно ли решить поставленную задачу с помощью одного из встроенных классов *Serdes*. Несложно заметить, что в Kafka Streams отсутствуют классы *Serdes* для некоторых распространенных форматов, таких как JSON², Avro и Protobuf. В таких случаях приходится писать собственные версии. Именно этим мы сейчас и займемся, ведь твиты, с которыми предстоит работать нашему приложению, представлены как объекты JSON.

Пользовательская версия класса *Serdes*

Как я уже упоминал, в теме-источнике твиты должны фигурировать в виде объектов JSON, в то время как брокер Kafka хранит необработанные байты. Итак, первым делом нам нужен код, десериализующий твиты в высокоуровневые объекты JSON, чтобы упростить обработку данных в нашем приложении. Теоретически вместо написания собственного класса можно воспользоваться

¹ В следующих версиях, скорее всего, появятся дополнительные классы *Serdes*. Полный список доступных классов можно посмотреть в официальной документации (<https://kafka.apache.org/26/documentation/streams/developer-guide/datatypes#available-serdes>).

² Пример класса *Serdes* для формата JSON включен в исходный код Kafka, но на момент написания книги он еще не был представлен официально.

встроенным методом `Serdes.String()`, но это затруднит работу с потоком твитов, так как у нас не будет простого доступа ко всем полям объекта `tweet`¹.

Для распространенных форматов данных изобретать велосипед, реализуя собственную низкоуровневую логику сериализации/десериализации, не приходится. Для сериализации и десериализации формата JSON в языке Java существует множество библиотек, и мы просто воспользуемся одной из них. Это разработанная программистами Google библиотека Gson (https://oreil.ly/C2_5K) с интуитивно понятным API для преобразования байтов JSON в объекты Java. Десериализация байтовых массивов с помощью Gson, которой можно пользоваться всякий раз, когда требуется прочитать записи JSON из Kafka, выглядит так:

```
Gson gson = new Gson();
byte[] bytes = ...; ❶
Type type = ...; ❷
gson.fromJson(new String(bytes), type); ❸
```

❶ Необработанные байты, которые нужно десериализовать.

❷ `type` — класс, который будет использоваться для представления десериализованной записи.

❸ Метод `fromJson` преобразует необработанные байты в класс Java.

Библиотека Gson поддерживает и обратную процедуру, то есть преобразование объектов Java в байтовые массивы. Вот как выглядит сериализация средствами Gson:

```
Gson gson = new Gson();
gson.toJson(instance).getBytes(StandardCharsets.UTF_8);
```

Как видите, сложную задачу сериализации/десериализации JSON мы переложили на библиотеку Gson, и теперь можно просто воспользоваться ее возможностями для создания нашей собственной версии класса `Serdes`. Первым делом определим класс данных, в который будет происходить десериализация необработанных байтовых массивов.

Определение классов данных

Функционал библиотеки Gson (и некоторых других библиотек, например Jackson) позволяет преобразовывать байтовые массивы JSON в объекты Java. Чтобы сделать это, нужно определить класс данных или POJO (старый до-

¹ Потребуется сложная работа с регулярными выражениями, и даже в этом случае хороший результат не гарантирован.

брый Java-объект), содержащий поля, которые мы хотим десериализовать из исходного объекта.

Показанный в примере 3.4 класс данных для представления необработанных записей твитов в приложении Kafka Streams довольно прост. Для каждого поля задается свойство, которое нужно взять из необработанного твита (например, `createdAt`), и метод чтения/записи для доступа к этому свойству (например, `getCreatedAt`).

Пример 3.4. Класс данных для десериализации твитов

```
public class Tweet {  
    private Long createdAt;  
    private Long id;  
    private String lang;  
    private Boolean retweet;  
    private String text;  
  
    // методы чтения и записи опущены для краткости  
}
```

Ненужные нам поля из исходной записи можно просто исключить из класса данных, и библиотека Gson автоматически их удалит. Каждый раз при создании десериализатора думайте о том, какие поля можно отбросить, как не требующиеся в будущем приложении. Процесс выбора из доступных полей необходимого подмножества называется *проекцией*. Это аналог выбора столбцов оператором SELECT в языке SQL.

Наш класс данных готов, пришло время реализовать десериализатор Kafka Streams, который будет преобразовывать необработанные байтовые массивы из темы `tweets` в высокоуровневые объекты `Tweet`, работать с которыми намного проще.

Пользовательский десериализатор

Для создания пользовательской версии десериализатора требуется минимум кода, особенно если воспользоваться библиотекой, которая возьмет на себя большую часть процедуры преобразования байтовых массивов. Я для этой цели взял библиотеку Gson. Осталось реализовать интерфейс `Deserializer` в клиентской библиотеке Kafka. Вот как это делается:

```
public class TweetDeserializer implements Deserializer<Tweet> {  
    private Gson gson = new GsonBuilder()  
        .setFieldNamingPolicy(FieldNamingPolicy.UPPER_CAMEL_CASE) ❶  
        .create();
```

```

@Override
public Tweet deserialize(String topic, byte[] bytes) { ❷
    if (bytes == null) return null; ❸
    return gson.fromJson(
        new String(bytes, StandardCharsets.UTF_8), Tweet.class); ❹
}

```

❶ Библиотека Gson поддерживает различные форматы имен полей JSON. Но коннектор, отвечающий за интеграцию Kafka с сервисом микроблогов Twitter, использует для имен полей стиль CamelCase, поэтому мы устанавливаем аналогичную политику, чтобы гарантировать корректную десериализацию объектов JSON.

❷ Метод `deserialize` заменен нашей собственной схемой десериализации записей в теме `tweets`, которая возвращает экземпляр нашего класса данных (`Tweet`).

❸ Условие, запрещающее десериализовать пустые массивы.

❹ С помощью библиотеки Gson байтовый массив десериализуется в объект `Tweet`.

Теперь осталось написать код сериализатора.

Пользовательский сериализатор

Код нашей версии сериализатора также очень прост. Нужно реализовать метод `serialize` интерфейса `Serializer`, входящего в клиентскую библиотеку Kafka. Большую часть тяжелой работы мы снова переложим на библиотеку Gson. Вот код сериализатора объектов `Tweet`, которым мы будем пользоваться в этой главе:

```

class TweetSerializer implements Serializer<Tweet> {
    private Gson gson = new Gson();

    @Override
    public byte[] serialize(String topic, Tweet tweet) {
        if (tweet == null) return null; ❶
        return gson.toJson(tweet).getBytes(StandardCharsets.UTF_8); ❷
    }
}

```

❶ Условие, запрещающее сериализацию пустого объекта `Tweet`.

❷ Непустые объекты `Tweet` метод `toJson()` из библиотеки Gson преобразует в массив байтов.

Десериализатор и сериализатор твитов готовы, и их можно превратить в единый класс `Serdes`.

Класс Serdes для твитов

До сих пор код, реализующий десериализатор и сериализатор, был несложным. Несложно реализовать и нашу версию класса Serdes, единственным предназначением которой будет объединение десериализатора и сериализатора в удобный оберточный класс, используемый в Kafka Streams. Вот как он выглядит:

```
public class TweetSerdes implements Serde<Tweet> {

    @Override
    public Serializer<Tweet> serializer() {
        return new TweetSerializer();
    }

    @Override
    public Deserializer<Tweet> deserializer() {
        return new TweetDeserializer();
    }
}
```

Теперь можно немного отредактировать код из примера 3.2, сделав его из такого:

```
KStream<byte[], byte[]> stream = builder.stream("tweets");
stream.print(Printed.<byte[], byte[]>toSysOut().withLabel("tweets-stream"));
```

таким:

```
KStream<byte[], Tweet> stream = ❶
    builder.stream(
        "tweets",
        Consumed.with(Serdes.ByteArray(), new TweetSerdes())); ❷

stream.print(Printed.<byte[], Tweet>toSysOut().withLabel("tweets-stream"));
```

❶ Обратите внимание, что тип значений изменился с `byte[]` на `Tweet`. Скоро вы увидите, как работа с экземплярами класса `Tweet` упростит нам жизнь.

❷ Через вспомогательный объект `Consumed` в явном виде задаем ключ и значение `Serdes`, которые будут использоваться для этого потока `KStream`. В частности, значение `new TweetSerdes()` приводит к тому, что поток заполняется уже не байтовыми массивами, а объектами `Tweet`.

Обратите внимание, что ключи по-прежнему сериализуются как байтовые массивы. Это связано с тем, что наша топология не требует каких-либо действий с ключами записей, поэтому нет смысла их десериализовать¹. Если запустить

¹ Лишние процедуры сериализации/десериализации отрицательно сказываются на производительности приложения.

наше приложение Kafka Streams и передать в исходную тему какие-нибудь данные, мы увидим, что сейчас оно работает с объектами `Tweet`. С ними связаны удобные методы чтения из исходных полей JSON. Эти методы пригодятся на следующих шагах обработки потока:

```
[tweets-stream]: null, Tweet@182040a6  
[tweets-stream]: null, Tweet@46efe0cb  
[tweets-stream]: null, Tweet@176ef3db
```

Итак, мы создали поток `KStream`, который использует класс данных `Tweet`, можно приступить к реализации остальной топологии. Теперь требуется убрать из потока твитов все повторные публикации.



Устранение ошибок десериализации в Kafka Streams

Всегда нужно четко указывать, как приложение обрабатывает ошибки десериализации. Иначе, если в теме-источнике окажется некорректная запись, можно столкнуться с неприятным сюрпризом. В Kafka Streams существует специальный параметр `DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG`, в котором указывается обработчик возникающих при десериализации исключений. Можно как реализовать собственную версию обработчика, так и воспользоваться одним из встроенных, например `LogAndFailExceptionHandler` (который регистрирует ошибку и останавливает приложение Kafka Streams) или `LogAndContinueExceptionHandler` (который только заносит в журнал данные об ошибке, но приложение при этом продолжает работу).

Фильтрация данных

Одна из наиболее распространенных задач без сохранения состояния в приложениях потоковой обработки — фильтрация данных. Это выборка для обработки только части записей. Остальные записи при этом игнорируются. Основную идею фильтрации демонстрирует рис. 3.3.

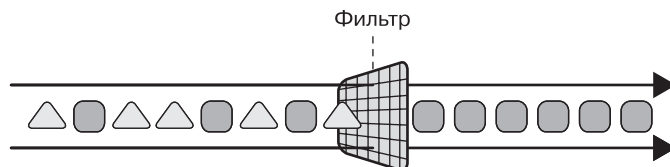


Рис. 3.3. Фильтрация позволяет обрабатывать только часть данных из потока событий

В Kafka Streams для фильтрации в основном¹ применяются два метода:

- `filter`;
- `filterNot`.

Среди параметров метода `filter` фигурирует логическое выражение, называемое *предикатом*, которое позволяет определить, следует ли сохранять сообщение. Если предикат возвращает значение `true`, запись передается для дальнейшей обработки. В случае значения `false` запись из обработки исключается. Нам нужно отфильтровать все повторные публикации твитов, поэтому используем в качестве предиката метод `Tweet.isRetweet()`.

Поскольку `Predicate` — это функциональный интерфейс², с методом `filter` можно использовать лямбда-выражение. Пример 3.5 демонстрирует, как с помощью этого метода убрать все повторные публикации твитов.

Пример 3.5. Использование метода `filter` в языке DSL

```
KStream<byte[], Tweet> filtered = stream.filter(
    (key, tweet) -> {
        return !tweet.isRetweet();
    });
```

Метод `filterNot` очень похож на метод `filter`, но пользуется инвертированной булевой логикой (то есть удаляются записи, для которых он возвращает значение `true`). В примере 3.5 фигурировало отрицательное условие фильтрации: `!tweet.isRetweet()`. Такой же результат можно получить, перенеся отрицание на уровень метода, как показано в примере 3.6.

Пример 3.6. Использование метода `filterNot` в языке DSL

```
KStream<byte[], Tweet> filtered = stream.filterNot(
    (key, tweet) -> {
        return tweet.isRetweet();
    });
```

Функционально эти два фрагмента кода эквивалентны. Лично я в случаях, когда логическая схема фильтрации содержит отрицание, для большей читабельности предпочитаю метод `filterNot`. Поэтому для исключения повторных публикаций твитов далее будет использоваться код из примера 3.6.

¹ Позже я покажу примеры фильтрации методами `flatMap` и `flatMapValues`. Но если при фильтрации не требуется создавать более одной записи, я рекомендую пользоваться методами `filter` или `filterNot`.

² Функциональный интерфейс в Java — это интерфейс со всего одним абстрактным методом. В интерфейсе `Predicate` описан единственный абстрактный метод `test()`.



Проводить фильтрацию следует как можно раньше. Нет смысла преобразовывать или дополнять данные, которые на следующих этапах могут оказаться выброшенными, особенно если их обработка требует больших вычислительных ресурсов.

Итак, шаги 1 и 2 топологии с рис. 3.1 реализованы. Можно переходить к третьему шагу: разделению отфильтрованного потока на базе языка твита.

Ветвление данных

В предыдущем разделе я показал, как с помощью логического условия, называемого предикатом, выполняется фильтрация потока. Но в Kafka Streams предикаты можно также использовать для разделения (или ветвления) потоков. *Ветвление* (branching) обычно требуется, когда события в зависимости от какого-то атрибута обрабатываются разными способами или направляются в разные темы-приемники.

В нашем списке требований было указано, что тема-источник может содержать твиты на разных языках. Это означает, что для части записей из нашего потока (твиты не на английском языке) нужна дополнительная обработка: перевод на английский язык. Именно здесь нам на помощь придет ветвление, которое схематично представлено на рис. 3.4.

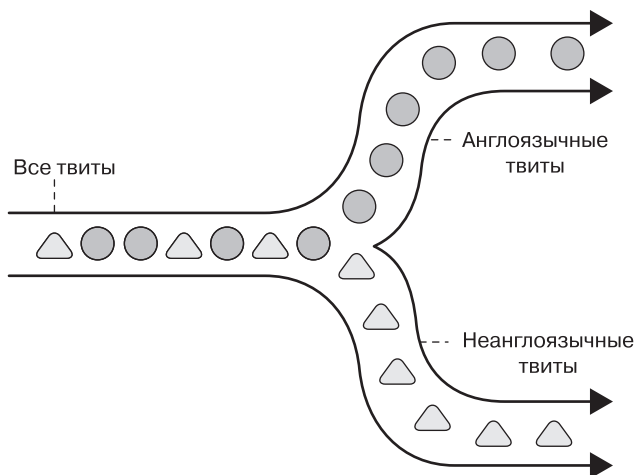


Рис. 3.4. Один поток делится на несколько выходных потоков

Создадим два лямбда-выражения: одно для записи твитов на английском языке, а другое — для всех остальных. Написанный ранее код десериализации

позволяет воспользоваться другим методом чтения в нашем классе данных: `Tweet.getLang()`. Это нужно для реализации логической схемы ветвления. Вот предикаты, применяемые для ветвления потока:

```
Predicate<byte[], Tweet> englishTweets =  
    (key, tweet) -> tweet.getLang().equals("en"); ❶
```

```
Predicate<byte[], Tweet> nonEnglishTweets =  
    (key, tweet) -> !tweet.getLang().equals("en"); ❷
```

❶ Предикат, соответствующий англоязычным твитам.

❷ Предикат, соответствующий всем остальным твитам.

Условия ветвления определены, теперь можно использовать метод `branch`, принимающий один или несколько предикатов и возвращающий список выходных потоков для каждого из них. Записи по порядку проверяются на соответствие каждому предикату и по результатам проверки добавляются в одну из ветвей. Если ни одного соответствия не обнаружено, запись удаляется:

```
KStream<byte[], Tweet>[] branches =  
    filtered.branch(englishTweets, nonEnglishTweets); ❸
```

```
KStream<byte[], Tweet> englishStream = branches[0]; ❹
```

```
KStream<byte[], Tweet> nonEnglishStream = branches[1]; ❺
```

❸ Создаем две ветви потока, оценивая язык твита.

❹ Поскольку первым идет предикат `englishTweets`, в первый поток `KStream` из нашего списка (с индексом 0) пойдут англоязычные твиты.

❺ Второе условие ветвления (предикат `nonEnglishTweets`) отправит твиты, которым требуется перевод, во второй поток нашего списка `KStream` (с индексом 1).



Возможно, в следующих версиях Kafka Streams метод ветвления изменится. Если вы пользуетесь версией более новой, чем 2.7.0, проверьте, не были ли внесены изменения в API. Это указывается в статье *KIP-418: A method-chaining way to branch KStream* (https://oreil.ly/h_GvU). Возможно, будет добавлена обратная совместимость. Текущая реализация работает с последней на момент написания этого текста версией Kafka Streams (2.7.0).

Итак, у нас есть две ветки потока (`englishStream` и `nonEnglishStream`), к каждой из которых можно применить свою логическую схему обработки. Мы завершили третий шаг топологии с рис. 3.1 и переходим к переводу неанглоязычных твитов на английский.

Перевод твитов

На данный момент у нас два потока записей: твиты на английском языке (`englishStream`) и твиты на других языках (`nonEnglishStream`). Требуется провести анализ тональности каждого твита, но API, которым мы для этого воспользуемся, поддерживает ограниченное количество языков (английский входит в их число). Следовательно, каждый твит из потока `nonEnglishStream` нужно перевести на английский язык.

С точки зрения потоковой обработки требуется сделать так, чтобы каждая входная запись преобразовывалась в одну выходную запись (ключ или значение которой могут быть как того же, так и другого типа). К счастью, в Kafka Streams есть два метода, обеспечивающих именно такой результат:

- `map`;
- `mapValues`.

Схема операции отображения показана на рис. 3.5.

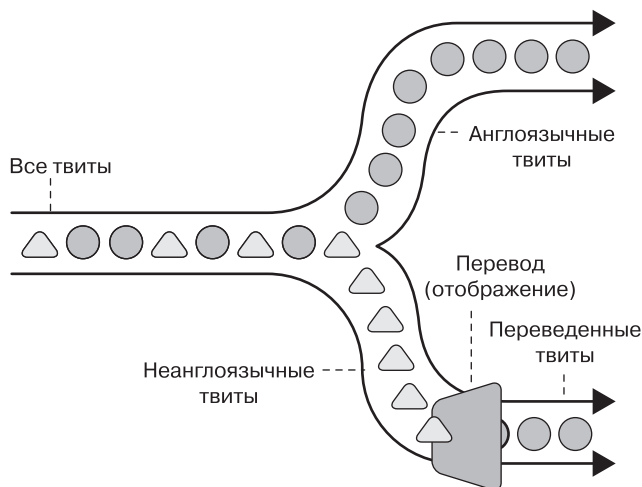


Рис. 3.5. Операции отображения позволяют выполнять преобразование записей 1:1

Методы `map` и `mapValues` очень похожи (оба выполняют отображение 1:1), и оба подходят для решения нашей задачи. Единственное отличие состоит в том, что для метода `map` требуется указывать не только новое значение, но и новый ключ записи, тогда как методу `mapValues` достаточно нового значения.

Для начала рассмотрим реализацию с помощью метода `map`. Представим, что мы решили не только перевести тексты твитов, но и сгенерировать для них новые

ключи. Новым ключом послужит имя автора каждого твита. Вот как будет выглядеть такой код:

```
KStream<byte[], Tweet> translatedStream = nonEnglishStream.map(
    (key, tweet) -> { ❶
        byte[] newKey = tweet.getUsername().getBytes();
        Tweet translatedTweet = languageClient.translate(tweet, "en");
        return KeyValue.pair(newKey, translatedTweet); ❷
    });
```

❶ Функция отображения вызывается с ключом (*key*) и значением (*tweet*) записи.

❷ Функция отображения должна возвращать новый ключ *и* новое значение записи, представленное объектом класса `KeyValue`. В качестве нового ключа в коде указан `username` в Twitter, а новым значением станет переведенный твит. Детальное рассмотрение схемы перевода текста выходит за рамки этого упражнения, но узнать подробности реализации можно, просмотрев исходный код.

Впрочем, в спецификации приложения, которое мы создаем, не было требования переназначать ключи записей. В следующей главе вы увидите, что необходимость переназначения ключей часто возникает, когда следующие этапы обработки предусматривают операции с сохранением состояния¹. Но в данном случае таких операций не будет, поэтому, хотя у нас уже есть вполне рабочий фрагмент кода, напишем более простую реализацию. Для этого мы воспользуемся методом `mapValues`, который меняет только значения записей:

```
KStream<byte[], Tweet> translatedStream =
    nonEnglishStream.mapValues(
        (tweet) -> { ❶
            return languageClient.translate(tweet, "en"); ❷
        });
```

❶ В метод `mapValues` передается *только* значение записи.

❷ Метод должен вернуть новое значение, в данном случае это переведенный на английский язык твит.

Именно с этим фрагментом кода мы будем дальше работать, так как у нас нет необходимости менять ключи каких-либо записей.



По возможности рекомендуется пользоваться методом `mapValues`, а не `map`, поскольку это потенциально увеличивает эффективность выполнения программы Kafka Streams.

¹ Переназначение ключей позволяет гарантировать размещение связанных данных в одной потоковой задаче, что важно при операциях агрегирования и соединения. Эти вещи будут подробно рассматриваться в следующей главе.

После перевода у нас образовалось два потока `KStream` с англоязычными твитами:

- `englishStream` — исходный поток англоязычных твитов;
- `translatedStream` — поток твитов, переведенных с других языков.

Это завершает шаг 4 построения топологии обработчиков с рис. 3.1. Для анализа тональности твитов фактически все готово. Но отдельно выполнять эту операцию для каждого потока не имеет смысла. Чтобы не плодить одинаковый код, *сольем* два потока в один.

Слияние потоков

Для объединения потоков в Kafka Streams существует простой способ. Слияние можно рассматривать как противоположность ветвления. Обычно к нему прибегают, когда в приложении к разным потокам применяется одна схема обработки.



В SQL аналогом метода `merge` является запрос на объединение. Например:

```
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;
```

Сейчас перед нами стоит задача провести анализ тональности двух потоков: `englishStream` и `TranslatedStream`. При этом все прошедшие анализ твиты необходимо записать в один поток `crypto-sentiment`. Это идеальная ситуация для проведения операции, схематично представленной на рис. 3.6.

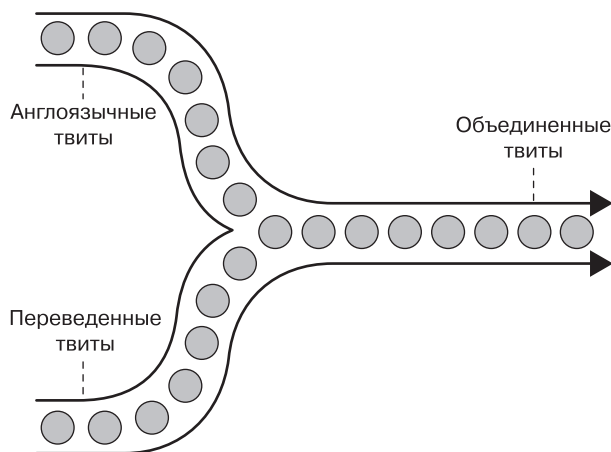


Рис. 3.6. Операции слияния превращают несколько потоков в один

Код, реализующий слияние потоков, очень прост. Достаточно передать в метод `merge` имена потоков, которые мы хотим объединить:

```
KStream<byte[], Tweet> merged = englishStream.merge(translatedStream);
```

Итак, мы превратили два потока в один и можем перейти к следующему шагу.

Обогащение твитов

Мы приближаемся к цели: обогатить каждый твит, проанализировав его тональность. Но, если помните, текущий класс данных `Tweet` представляет собой структуру, хранящую необработанные твиты из *темы-источника*. Для представления обогащенных записей, которые мы будем записывать в *тему-приемник* (`crypto-sentiment`), потребуется новый класс данных. На этот раз в качестве формата сериализации мы используем не JSON, а Avro (<https://oreil.ly/eFV8s>). Посмотрим на процесс создания класса данных Avro для представления обогащенных записей в нашем приложении Kafka Streams.

Класс данных Avro

В сообществе Kafka формат Avro очень популярен во многом благодаря компактному байтовому представлению (это выгодно для приложений с высокой пропускной способностью), встроенной поддержке схем¹ и инструменту управления схемами, который называется реестром (Schema Registry). Реестр схем хорошо работает с Kafka Streams, с момента его появления в нем была сильная поддержка формата Avro². Есть у него и другие преимущества. Например, некоторые коннекторы Kafka могут использовать схемы Avro для автоматического определения структуры таблиц в хранилищах данных ниже по потоку, что облегчает интеграцию данных.

При работе со схемами данных Avro можно использовать как общий, так и специфичный для Avro формат записей. Общий формат подходит для случаев, когда во время выполнения схема записи неизвестна. Доступ к именам полей при этом осуществляется через универсальные методы чтения и записи, например `GenericRecord.get(String key)` и `GenericRecord.put(String key, Object value)`.

¹ Схемы определяют имена и типы полей для записи. Они позволяют различным приложениям и службам заключать строгий контракт по поводу используемого формата данных.

² Форматы Protobuf и JSON поддерживаются, начиная с реестра схем Confluent версии 5.5. См. <https://oreil.ly/4hsQh>.

Записи, сгенерированные Avro, представляют собой классы Java с гораздо более удобным интерфейсом доступа к данным. Скажем, сгенерировав для записей класс `EntitySentiment`, мы получим методы чтения/записи для каждого поля. Например: `entitySentiment.getSentimentScore()`¹.

В приложении, которое мы создаем, формат выходных записей известен заранее (следовательно, известна и схема), поэтому мы будем использовать запись, сгенерированную Avro (ниже я называю ее классом данных). Создадим в папке `src/main/avro` нашего проекта Kafka Streams файл с именем `entity_sentiment.avsc` и добавим в него пример 3.7. Это определение схемы, которая будет применяться для выходных записей, то есть твитов, обогащенных оценкой тональности.

Пример 3.7. Схема Avro для обогащенных твитов

```
{
  "namespace": "com.magicalpipelines.model", ❶
  "name": "EntitySentiment", ❷
  "type": "record",
  "fields": [
    {
      "name": "created_at",
      "type": "long"
    },
    {
      "name": "id",
      "type": "long"
    },
    {
      "name": "entity",
      "type": "string"
    },
    {
      "name": "text",
      "type": "string"
    },
    {
      "name": "sentiment_score",
      "type": "double"
    },
    {
      "name": "sentiment_magnitude",
      "type": "double"
    },
    {
      "name": "salience",
      "type": "double"
    }
  ]
}
```

¹ В данном случае `entitySentiment` — это сгенерированный Avro экземпляр класса `EntitySentiment`. Этого класса пока не существует, мы создадим его чуть позже.

❶ Желаемое имя пакета для будущего класса данных.

❷ Имя класса Java, содержащего модель данных на основе Avro. Этот класс будет использоваться на последующих этапах обработки потока.

Итак, мы определили схему, и теперь из этого определения нужно сгенерировать класс данных, добавив в проект некоторые зависимости. Для этого нужно внести в файл `build.gradle` следующие строки:

```
plugins {  
    id 'com.commercehub.gradle.plugin.avro' version '0.9.1' ❶  
}  
  
dependencies {  
    implementation 'org.apache.avro:avro:1.8.2' ❷  
}
```

❶ Подключаемый модуль Gradle используется для автоматического создания классов Java из определений схемы Avro. Такое определение было показано в примере 3.7.

❷ Зависимость, которая содержит основные классы для работы с Avro.

При сборке проекта (см. раздел «Настройка проекта» в начале главы) будет автоматически сгенерирован новый класс данных¹ `EntitySentiment`, содержащий набор полей для хранения результатов эмоционального анализа твита (`sentiment_score`, `sentiment_magnitude` и `salience`) и соответствующих методов чтения/записи. Теперь можно приступить к оценкам настроения твитов и знакомству с новым набором операторов DSL.

Анализ тональности

При рассмотрении процедуры перевода твитов на английский язык вы научились выполнять преобразование записей методами `map` и `mapValues`. Но эти методы создают для каждой входной записи ровно одну выходную запись. Но иногда случается так, что требуется получить ноль или, наоборот, несколько выходных записей.

В качестве примера рассмотрим твит, в котором упоминается несколько криптовалют:

```
#bitcoin is looking super strong. #ethereum has me worried though
```

В этом вымышленном твите упоминаются две криптовалюты, или *сущности* (Bitcoin и Ethereum), при этом первая оценивается в позитивном ключе,

¹ Подключаемый модуль Gradle от Avro автоматически просканирует папку `src/main/` авро и передаст компилятору Avro все обнаруженные файлы схем.

а вторая скорее в негативном. Современные библиотеки и службы обработки естественного языка (Natural Language Processing, NLP) зачастую позволяют проанализировать тональность каждой содержащейся в тексте сущности, в результате чего одна входная строка дает на выходе несколько записей. Например, в рассматриваемом случае мы получим:

```
{"entity": "bitcoin", "sentiment_score": 0.80}
{"entity": "ethereum", "sentiment_score": -0.20}
```

В Kafka Streams такое преобразование реализуют два метода:

- `flatMap`;
- `flatMapValues`.

Оба метода позволяют получить на выходе ноль, одну или несколько записей, как показано на рис. 3.7.

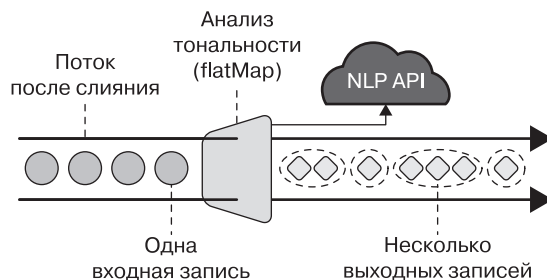


Рис. 3.7. Метод `flatMap` позволяет преобразовать одну входную запись в ноль, одну или более выходных записей

Подобно уже знакомому вам методу `map`, метод `flatMap` модифицирует не только значение записи, но и ее ключ. При этом метод `flatMapValues` меняет только значение, так что в нашем случае он подходит больше и для анализа тональности твитов мы будем использовать именно его. В блоке кода, отвечающем за этот анализ, обратите внимание на новый класс данных `EntitySentiment`, сгенерированный Avro:

```
KStream<byte[], EntitySentiment> enriched = merged.flatMapValues(
    (tweet) -> {
        List<EntitySentiment> results =
            languageClient.getEntitySentiment(tweet); ❶

        results.removeIf(
            entitySentiment -> !currencies.contains(
                entitySentiment.getEntity()); ❷

        return results; ❸
    });
```

- ❶ Получение списка оценок для каждой сущности в твите.
- ❷ Удаление всех сущностей, не совпадающих ни с одной из отслеживаемых криптовалют. Окончательный размер списка после операции удаления — величина переменная (это ключевая характеристика значений, возвращаемых методами `flatMap` и `flatMapValues`). Список может содержать ноль или более элементов.
- ❸ Возвращение полученного списка в выходной поток. «Распаковку» этого списка, то есть превращение каждого его элемента в отдельную запись потока выполнит `Kafka Streams`.



В ситуациях, не требующих модификации ключей, я рекомендую использовать метод `flatMapValues` вместо метода `flatMap`, так как это позволяет `Kafka Streams` более эффективно выполнять программу.

Осталось сделать последний шаг: записать обогащенные оценками тональности данные в новую тему. Для этого нам потребуются объекты `Serde`, которыми мы воспользуемся, чтобы превратить записи `EntitySentiment` в формате Avro в байтовые массивы.

Сериализация данных Avro

Как я уже упоминал, брокер `Kafka` представляет собой платформу обработки потока байтов. Следовательно, для передачи записей `EntitySentiment` в тему-приемник необходимо сериализовать их из формата Avro в байтовый массив.

При сериализации с помощью фреймворка Avro возможны два варианта.

- Схема Avro добавляется к каждой записи.
- Схема Avro сохраняется в реестре `Confluent Schema Registry`, и к каждой записи добавляется только ее идентификатор.

Как показано на рис. 3.8, преимущество первого подхода состоит в том, что вместе с приложением `Kafka Streams` не приходится настраивать и запускать отдельную службу `Confluent Schema Registry`. Это REST-служба для создания и извлечения схем Avro, Protobuf и JSON, которая требует отдельного развертывания и отдельного обслуживания. Кроме того, это дополнительная точка отказа. Но за возможность обойтись без нее приходится платить увеличением размера записей за счет добавленной к ним схемы.

При этом, чтобы выжать из приложения `Kafka Streams` максимум, желательно по возможности уменьшать размер полезной нагрузки. Кроме того, если

в будущем запланировано постоянное изменение схем записей и модели данных, проверки совместимости схем помогают обеспечить безопасность и надежность этих изменений¹.

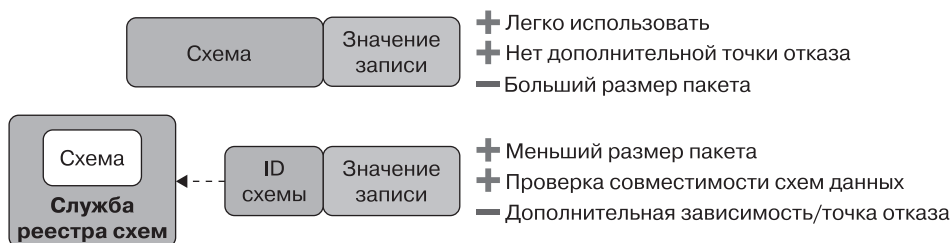


Рис. 3.8. Преимущества и недостатки добавления схемы Avro к каждой записи

Классы `Serdes` для Avro доступны для обоих подходов, так что добавить к нашему приложению возможность сериализовать данные Avro очень просто. Посмотрим на процедуру настройки класса `AvroSerdes` в обоих случаях.

Сериализация формата Avro без сохранения схемы в реестре

Сериализатор и десериализатор для формата Avro можно реализовать самостоятельно, но я предпочел воспользоваться готовым пакетом с открытым исходным кодом `com.mitchseymour:kafka-registryless-avro-serdes`². Для доступа к нему добавьте в файл `build.gradle` следующие строки:

```
dependencies {
    implementation 'com.mitchseymour:kafka-registryless-avro-serdes:1.0.0'
}
```

Всякий раз, когда в приложении Kafka Streams возникает необходимость использовать класс `Serdes`, достаточно передать в метод `AvroSerdes.get` сгенерированный Avro класс, как показано ниже:

```
AvroSerdes.get(EntitySentiment.class)
```

Полученный класс `Serdes` можно использовать везде, где обычно применялись встроенные в Kafka версии этого класса.

¹ Дополнительную информацию о совместимости схем можно получить в статье (<https://oreil.ly/kVwHQ>), которую Гвен Шапира (Gwen Shapira) написала в 2019 году.

² Репозиторий с кодом этого сериализатора Avro находится по адресу <https://oreil.ly/m1kk7>.

Сериализация формата Avro с сохранением схемы в реестре

Компания Confluent опубликовала пакет сериализатора/десериализатора формата Avro с сохранением схемы в реестре. Чтобы им воспользоваться, добавьте в файл `build.gradle` следующие строки:

```
repositories {
    mavenCentral()

    maven {
        url "https://packages.confluent.io/maven/" ❶
    }
}

dependencies {
    implementation ('io.confluent:kafka-streams-avro-serde:6.0.1') { ❷
        exclude group: 'org.apache.kafka', module: 'kafka-clients' ❸
    }
}
```

❶ Добавление репозитория Maven от Confluent, поскольку именно здесь находится компонент реестра схем для Avro Serdes.

❷ Добавление зависимости для использования этого реестра схем.

❸ Исключение несовместимой промежуточной зависимости, которая на момент написания этого текста входила в библиотеку `kafka-streams-avro-serde`¹.

Поддерживающий Avro Serdes реестр схем требует дополнительной настройки, поэтому, чтобы улучшить читаемость кода, добавим фабричный метод, создающий для каждого класса данных в проекте свой экземпляр Serdes. Вот, к примеру, как выглядит этот экземпляр для класса `TweetSentiment`:

```
public class AvroSerdes {
    public static Serde<EntitySentiment> EntitySentiment(
        String url, boolean isKey) {

        Map<String, String> serdeConfig =
            Collections.singletonMap("schema.registry.url", url); ❶
        Serde<EntitySentiment> serde = new SpecificAvroSerde<>();
        serde.configure(serdeConfig, isKey);
        return serde;
    }
}
```

¹ При работе с будущими версиями библиотеки `kafka-streams-avro-serde` этот шаг может оказаться ненужным, но рабочая на момент написания книги версия (6.0.1) конфликтовала с зависимостью в `Kafka Streams`. 2.7.0.

❶ Для использования сериализатора/десериализатора Avro с сохранением схем в реестре нужно настроить конечную точку Schema Registry.

Вот код, позволяющий создать экземпляр Serdes в любом месте приложения Kafka:

```
AvroSerdes.EntitySentiment("http://localhost:8081", false) ❶
```

❶ Актуализуем информацию для конечной точки Schema Registry.



Регистрировать, удалять и редактировать схемы, а также смотреть их список (<https://oreil.ly/GXBgI>) можно непосредственно в реестре схем. Но использующий реестр схем сериализатор/десериализатор Avro в Kafka Streams регистрирует схемы автоматически. Кроме того, для повышения производительности этот Serdes прибегает к локальному кэшированию схем и их идентификаторов, избавляя от необходимости каждый раз искать их заново.

Теперь, когда у нас есть средство сериализации и десериализации формата Avro, пришло время добавить узел-приемник.

Добавление узла-приемника

Осталось записать обогащенные данные в тему-приемник `crypto-sentiment`. Это можно реализовать тремя методами:

- `to`;
- `through`;
- `repartition`.

Для добавления дополнительных узлов-обработчиков нужны новые экземпляры `KStream`. Их можно получить методами `repartition` и `through` (последний перед публикацией этой книги был объявлен устаревшим, но все еще широко используется, и ожидается, что он будет иметь обратную совместимость). По сути дела, эти методы снова вызывают оператор `builder.stream`, генерируя с помощью Kafka Streams дополнительные субтопологии (см. подраздел «Субтопологии» в главе 2). Но в случаях, когда обработка потока подходит к завершению, следует использовать возвращающий значение `void` оператор `to`, так как в основной поток `KStream` уже не нужно добавлять узлы-обработчики.

Мы сейчас находимся на последнем шаге построения топологии, поэтому будем использовать оператор `to`. Из двух вариантов сериализатора/десериализатора

Автоматически возьмем тот, который задействует реестр схем, чтобы уменьшить размеры сообщений, а заодно оставить пространство для редактирования схемы. Вот код добавления узла-приемника:

```
enriched.to(
    "crypto-sentiment",
    Produced.with(
        Serdes.ByteArray(),
        AvroSerdes.EntitySentiment("http://localhost:8081", false)));
```

Все. Мы полностью реализовали топологию. Осталось запустить код и убедиться, что он корректно работает.

Запуск кода

Для работы приложения потребуется кластер Kafka и экземпляр реестра схем. Здесь поможет контейнер Docker Compose, входящий в исходный код этого упражнения¹. После запуска кластера Kafka и службы реестра схем запустите приложение Kafka Streams с помощью следующей команды:

```
./gradlew run --info
```

Все готово к тестированию. Убедимся, что наше приложение работает должным образом.

Эмпирическая проверка

В главе 2 мы уже говорили о том, что эмпирическая проверка — один из самых простых способов убедиться в корректности работы приложения. Она включает в себя генерацию данных в локальном кластере Kafka и наблюдение за данными, записываемыми в тему-приемник. Проще всего это сделать, сохранив несколько твитов в текстовый файл и с помощью сценария `kafka-console-producer` записав их в тему-источник `tweets`.

Исходный код содержит файл `test.json` с двумя записями. Им мы и воспользуемся для тестирования. Примечание: в файле `test.json` записи находятся в упрощенной форме, но в примере 3.8 они приведены в «красивом» виде для улучшения читаемости.

¹ Инструкции по запуску кластера Kafka и реестра схем находятся по адресу <https://oreil.ly/DEoaJ>.

Пример 3.8. Два вида твитов для тестирования приложения Kafka Streams

```

{
  "CreatedAt": 1577933872630,
  "Id": 10005,
  "Text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "Lang": "en",
  "Retweet": false, ❶
  "Source": "",
  "User": {
    "Id": "14377871",
    "Name": "MagicalPipelines",
    "Description": "Learn something magical today.",
    "ScreenName": "MagicalPipelines",
    "URL": "http://www.magicalpipelines.com",
    "FollowersCount": "248247",
    "FriendsCount": "16417"
  }
}
{
  "CreatedAt": 1577933871912,
  "Id": 10006,
  "Text": "RT Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "Lang": "en",
  "Retweet": true, ❷
  "Source": "",
  "User": {
    "Id": "14377870",
    "Name": "Mitch",
    "Description": "",
    "ScreenName": "Mitch",
    "URL": "http://mitchseymour.com",
    "FollowersCount": "120",
    "FriendsCount": "120"
  }
}

```

❶ Это оригинальный твит (ID 10005), для которого следует выполнить оценку тональности.

❷ Это (ID 10006) повторная публикация, которую следует проигнорировать.

Следующей командой добавим эти записи в наш локальный кластер Kafka:

```

kafka-console-producer \
  --bootstrap-server kafka:9092 \
  --topic tweets < test.json

```

На другой вкладке активируем потребление обогащенных записей с помощью сценария `kafka-console-consumer`:


```
kafka-console-consumer \
  --bootstrap-server kafka:9092 \
  --topic crypto-sentiment \
  --from-beginning
```

Вывод будет содержать странные символы:

```
◆◆◆◆[◆◆|Bitcoin has a lot of promise.
I'm not too sure about #ethereumbitcoin`ff◆?`ff◆? -◆◆?
◆◆◆◆[◆◆|Bitcoin has a lot of promise.
I'm not too sure about #ethereumethereum◆◆◆1◆◆◆1◆◆◆?
```

Дело в том, что формат Авро двоичный. Так как в данном случае мы пользуемся реестром схем, нам доступен разработанный Confluent консольный сценарий для улучшения читабельности данных Авро. Просто используйте `kafka-avro-console-consumer` вместо `kafka-console-consumer`:

```
kafka-avro-console-consumer \
  --bootstrap-server kafka:9092 \
  --topic crypto-sentiment \
  --from-beginning
```

После этого вывод приобретет вот такой вид:

```
{
  "created_at": 1577933872630,
  "id": 10005,
  "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "entity": "bitcoin",
  "sentiment_score": 0.699999988079071,
  "sentiment_magnitude": 0.699999988079071,
  "salience": 0.47968605160713196
}
{
  "created_at": 1577933872630,
  "id": 10005,
  "text": "Bitcoin has a lot of promise. I'm not too sure about #ethereum",
  "entity": "ethereum",
  "sentiment_score": -0.20000000298023224,
  "sentiment_magnitude": -0.20000000298023224,
  "salience": 0.030233483761548996
}
```

Как видите, в выходных данных отсутствует идентификатор твита 10006. Повторная публикация была отфильтрована на втором шаге топологии. В то же время твит с идентификатором 10005 породил две выходные записи. Это ожидаемое поведение, потому что в нем упоминаются две криптовалюты (и каждой дана своя оценка). Как видите, метод `flatMapValues` работает корректно (шестой

шаг топологии). Проверку работы узла, отвечающего за перевод на английский язык, я оставляю вам для самостоятельного упражнения. Для этого в файл `test.json` нужно добавить твит на иностранном языке¹.

Заключение

В этой главе вы познакомились со способами создания приложений потоковой обработки без сохранения состояния. В частности, вы научились:

- фильтровать данные методами `filter` и `filterNot`;
- создавать новые ветки потоков методом `branch`;
- объединять потоки методом `merge`;
- выполнять преобразования записи 1:1 методами `map` и `mapValues`;
- выполнять преобразования записи 1:N методами `flatMap` и `flatMapValues`;
- выводить записи в тему-приемник методами `to`, `through` и `repartition`;
- сериализовать и десериализовать данные с помощью пользовательских сериализаторов, десериализаторов и реализаций класса `Serdes`.

На очереди более сложные операции обработки потоков. В следующей главе мы рассмотрим задачи с отслеживанием состояния, такие как соединение и агрегирование потоков, а также оконная обработка данных.

¹ Может потребоваться дополнительная настройка. См. файл README приведенного в этой главе упражнения (<https://oreil.ly/o6Ofk>).

Обработка с сохранением состояния

https://t.me/it_boooks

В предыдущей главе вы научились выполнять преобразования без сохранения состояния потоков записей, используя абстракцию `KStream` и набор методов `Kafka Streams`. Эти преобразования не требуют информации о ранее произошедших событиях, поэтому легко понять, как они устроены, и реализовать их. Каждое такое преобразование рассматривается как свершившийся *факт* и обрабатывается независимо от остальных.

Но `Kafka Streams` умеет фиксировать информацию о событиях. Собранная информация, или *состояние*, позволяет реализовывать более сложные варианты обработки потока, включая соединение и агрегирование данных. Обо всем этом я расскажу ниже. В частности, будут рассмотрены следующие темы.

- Преимущества потоковой обработки с сохранением состояния.
- Отличие фактов от поведений.
- Методы с сохранением состояния, доступные в `Kafka Streams`.
- Фиксация и запрос состояния в `Kafka Streams`.
- Представление локального секционированного состояния через абстракцию `KTable`.
- Представление глобального реплицированного состояния через абстракцию `GlobalKTable`.
- Выполнение операций с сохранением состояния, в том числе соединения и агрегирования данных.
- Получение информации о состоянии с помощью интерактивных запросов.

Как и в предыдущей главе, концепции будут объясняться на примере. Вам предстоит создать обновляющуюся в реальном времени таблицу лидеров видеоигр. Выполняя это задание, вы познакомитесь с множеством методов, умеющих

сохранять состояние. Кроме того, вас ждет подробный рассказ про одну из наиболее распространенных форм обогащения данных в приложениях с сохранением состояния — про соединения. Но начнем мы с разговора о преимуществах обработки с сохранением состояния.

Преимущества операций с сохранением состояния

Отслеживание состояний помогает понять, *как события связаны друг с другом*, и затем использовать эту информацию для реализации более сложных сценариев потоковой обработки. Сведения о взаимосвязи событий позволяют:

- распознавать закономерности и поведения в потоках событий;
- выполнять агрегирование;
- обогащать данные путем их соединения.

Кроме того, потоковая обработка с сохранением состояния обеспечивает нас еще одной абстракцией для представления данных. Поочередно воспроизводя события с сохранением последнего состояния каждого ключа во встроенном хранилище, мы можем сформировать представление непрерывного и неограниченного потока записей на определенный момент времени. Такой моментальный снимок называют *таблицей*. В Kafka Streams есть различные типы табличных абстракций, с которыми я вас постепенно познакомлю.

Таблицы не только составляют основу потоковой обработки с сохранением состояния. К ним еще можно делать запросы. Именно возможность запрашивать моментальный снимок быстро движущегося потока событий делает Kafka Streams *платформой реляционной обработки потоков*¹ и позволяет создавать не только приложения потоковой обработки, но и управляемые событиями микросервисы с малой задержкой.

Наконец, потоковая обработка с сохранением состояния дает возможность строить более продуманные ментальные модели данных. Интересную точку зрения в своей статье, посвященной событийно-ориентированному программированию (<https://oreil.ly/Q-hop>), высказал Нил Эйвери (Neil Avery), рассматривая разницу между фактами и поведением:

«Событие представляет собой какой-то факт, что-то совершившееся; то, что невозможно изменить...»

¹ Дополнительную информацию о платформах потоковой реляционной обработки вы найдете в блоге Роберта Йокоты (<https://oreil.ly/7u71d>).

Приложения без сохранения состояния, которые мы рассматривали в предыдущей главе, ориентированы на факты. Каждое событие рассматривается как независимый, атомарный факт, который можно обработать заранее заданным способом и забыть о нем.

Но, научившись с помощью операторов с сохранением состояния моделировать *поведения* (behaviors), вы сможете создавать более сложные запросы к данным. Что такое поведение? По словам Нила:

«Поведение полностью определяется собранными фактами».

В реальном мире события (или факты) редко бывают изолированными. Как правило, они связаны друг с другом, и, соответственно, путем фиксации фактов можно подойти к пониманию их смысла. Достаточно рассмотреть сохраненные приложением сопутствующие события.

Распространенный пример поведения как совокупности фактов — проблема брошенной корзины. Пользователь добавляет в корзину один или несколько товаров, а затем сеанс завершается либо вручную (если пользователь выходит из системы), либо автоматически (в случае длительного отсутствия активности). Обработка любого отдельного факта мало скажет о том, в какой стадии находилось оформление заказа. Но фиксация и анализ всех фактов (а именно это обеспечивает обработка с сохранением состояния) позволяет распознавать *поведение* и реагировать на него, что имеет большую коммерческую ценность, чем взгляд на ситуацию как на набор отдельных событий.

Теперь, когда я показал вам преимущества потоковой обработки с сохранением состояния, познакомимся с методами Kafka Streams, которые ее реализуют.

Обзор методов

Перечень методов, которыми мы будем пользоваться для операций с сохранением состояния, приводится в табл. 4.1.

Все эти методы можно *комбинировать* друг с другом, моделируя еще более сложные соотношения между событиями (поведения). Например, *оконное соединение* (windowed join) позволяет понять связь между дискретными потоками событий в течение определенного периода времени. В следующей главе вы познакомитесь с *оконными агрегированиями* (windowed aggregations), еще одним полезным способом комбинирования методов с сохранением состояния.

Таблица 4.1. Методы, сохраняющие состояние, и их назначение

Сценарий использования	Назначение	Методы
Соединение данных	Обогащение события информацией или контекстом, полученным из отдельного потока или таблицы	<code>join</code> (внутреннее соединение) <code>leftJoin</code> <code>outerJoin</code>
Агрегирование данных	Вычисление постоянно обновляющегося математического или комбинаторного преобразования связанных событий	<code>aggregate</code> <code>count</code> <code>reduce</code>
Оконная обработка данных	Группировка событий, близко расположенных во времени	<code>windowedBy</code>

В отличие от знакомых по предыдущей главе методов без сохранения состояния новые методы сложнее устроены и имеют дополнительные требования к вычислениям и хранению¹. Поэтому сначала мы рассмотрим подробности реализации обработки с сохранением состояния в Kafka Streams и только потом перейдем к практике.

Первым делом важно понять, каким образом в Kafka Streams сохраняется и запрашивается состояние.

Хранилища состояний

Как несложно догадаться, для применения методов с сохранением состояния требуется, чтобы приложение хранило сведения о ранее замеченных событиях. Например, приложение, подсчитывающее количество обнаруженных журналов регистрации ошибок, должно отслеживать для каждого ключа одно значение — счетчик, обновляемый при каждом потреблении информации из нового журнала ошибок. Этот счетчик дает нам исторический контекст записи. Вместе с ключом он становится частью состояния приложения.

Для поддержки методов с сохранением состояния (таких как `count`, `aggregate`, `join` и т. п.) нужен способ хранения и извлечения сохраненных данных, то есть состояния. Эти потребности в Kafka Streams удовлетворяются абстракцией хранилища, называемой *хранилищем состояний* (state store). Так как одно приложение Kafka Streams может использовать множество методов с сохранением состояния, оно может иметь несколько хранилищ состояния одновременно.

¹ В памяти, на диске или в их комбинации.



Этот раздел содержит информацию о способах фиксации и хранения данных в Kafka Streams. Если вы предпочитаете сразу начать с практики, переходите к разделу «Список лидеров видеонигр» далее.

В Kafka Streams доступно множество реализаций хранилища состояний и возможностей выбора их конфигурации. Каждая из них имеет свои достоинства и недостатки, а также области применения. Каждый раз, когда в приложении Kafka Streams вы прибегаете к методу с сохранением состояния, нужно учитывать, какой тип хранилища требуется этому методу. Кроме того, настройка хранилища состояний зависит от критериев оптимизации (например, оптимизацию можно нацелить на достижение высокой пропускной способности, простоты эксплуатации, быстрого восстановления в случае сбоя и т. п.). В большинстве случаев, если вы не указываете в явном виде тип хранилища или не переопределяете его конфигурацию, Kafka Streams достаточно рационально выбирает параметры по умолчанию.

Типы и конфигурации хранилищ отличаются довольно сильно, поэтому начнем мы с рассмотрения общих характеристик их реализаций по умолчанию, а затем отдельно поговорим о двух обширных категориях, таких как постоянные хранилища и хранилища в памяти. Более подробно эта тема будет обсуждаться в главе 6, а также в процессе выполнения практического задания.

Общие характеристики

Входящие в Kafka Streams реализации хранилищ состояний имеют ряд общих свойств. Посмотрим на них более подробно, чтобы лучше понять, как функционируют эти хранилища.

Встроенность

Входящие в состав Kafka Streams реализации хранилища состояний *встроены* в приложения Kafka Streams на уровне задач (про этот уровень я рассказывал в подразделе «Задачи и потоки выполнения» главы 2). Их преимущество перед внешними хранилищами заключается в том, что для доступа к состоянию не требуется обращения по сети, приводящего к ненужной задержке и к снижению скорости обработки. Кроме того, встраивание на уровне задач устраняет целый класс проблем параллелизма, возникающих при совместном доступе к состоянию.

В случае удаленного хранилища состояний также приходится отдельно беспокоиться о доступности удаленной системы. Разрешив Kafka Streams управлять локальным хранилищем состояний, мы гарантируем его постоянную доступность и существенно уменьшаем количество ошибок. Еще хуже вариант

с *централизованным* удаленным хранилищем, поскольку оно становится единой точкой отказа для всех экземпляров приложения. Таким образом, стратегия Kafka Streams по размещению состояния приложения рядом с приложением повышает не только производительность, но и доступность.

Все по умолчанию входящие в состав Kafka Streams хранилища состояний используют RocksDB. Это быстрое встроенное хранилище ключей и значений, изначально разработанное компанией Facebook. Хранение пар «ключ — значение» в нем происходит в виде произвольных потоков байтов, поэтому оно хорошо работает с брокером Kafka, разделяющим сериализацию и хранение. Более того, чтение и запись выполняются чрезвычайно быстро благодаря множественным оптимизациям системы хранения LevelDB¹, которая послужила основой для RocksDB.

Множественные режимы доступа

Хранилища состояний поддерживают различные режимы доступа и шаблоны запросов. Топологии обработчиков требуется доступ на чтение и запись к хранилищам состояний. Однако при создании микросервисов на базе *интерактивных запросов* клиентам требуется только доступ *на чтение* базового состояния. Это гарантирует неизменность состояния за пределами топологии и реализуется с помощью специальной оболочки, через которую клиенты могут безопасно запрашивать состояние приложения Kafka Streams.

Отказоустойчивость

По умолчанию в Kafka выполняется резервное копирование хранилища состояний в тему журнала изменений². В случае сбоя восстановить хранилище, а значит, и состояние приложения можно воспроизведением событий из этой темы. Кроме того, для ускорения восстановления состояния приложения Kafka Streams позволяет пользователям создавать *резервные реплики* (standby replicas), которые иногда называют *теньвыми копиями* (shadow copies). Благодаря этому хранилища состояний получают такое важное качество, присущее системам высокой доступности, как *избыточность*. Кроме того, приложения, допускающие запрос своего состояния, могут полагаться на резервные реплики для обслуживания трафика запросов в случае выхода из строя других экземпляров, что также способствует высокой доступности.

¹ Система хранения LevelDB была написана в Google, но когда ей начали пользоваться инженеры из Facebook, оказалось, что она работает слишком медленно. Благодаря изменению однопоточного уплотнения (compaction) в LevelDB на многопоточное и применению для чтения фильтра Блума удалось значительно повысить производительность как чтения, так и записи.

² Журналирование можно отключить. При этом хранилище состояний потеряет отказоустойчивость.

Использование ключей

Операции, использующие хранилища состояния, базируются на ключах. Ключ записи определяет, как именно текущее событие связано со всеми остальными. Базовая структура зависит от типа выбранного хранилища состояний¹, но каждую реализацию можно смоделировать как некую форму хранилища ключей, причем ключи могут быть как простыми, так и составными (то есть многомерными)².



Несколько осложняет понимание изложенного тот факт, что в Kafka Streams определенные типы хранилищ состояния называют хранилищами пар «ключ — значение», хотя по этому же принципу функционируют все встроенные хранилища. В этой главе и далее, говоря про хранилища «ключ — значение», я имею в виду неоконные хранилища состояний (про оконные хранилища речь пойдет в следующей главе).

Теперь, когда у вас появилось представление об общих характеристиках базовых хранилищ состояний в Kafka Streams, на примере двух обширных категорий хранилищ посмотрим, чем различаются отдельные реализации.

Постоянные хранилища и хранилища в оперативной памяти

Наиболее существенная разница между различными реализациями хранилищ заключается в их местоположении. Можно асинхронно сбрасывать состояния на диск. Это так называемое *постоянное* (persistent) хранилище. А можно расположить хранилище состояний в *оперативной памяти* (RAM). Первый вариант имеет два основных преимущества:

- можно хранить состояния, которые не помещаются в памяти;
- в случае сбоя такие хранилища восстанавливаются быстрее.

В случае постоянного хранилища часть состояний может быть сохранена в памяти, но когда там перестает хватать места или буфер записи превышает указанное в настройках значение, начинается запись на диск. Этот процесс называется *переливом на диск* (spilling to disk). Что касается второго пункта, хранение состояния приложения на диске избавляет Kafka Streams от необходимости при потере состояния (например, из-за сбоя в системе, миграции экземпляра и т. п.)

¹ Например, inMemoryKeyValueStore использует класс Java TreeMap, реализованный на основе красно-черных деревьев, в то время как все постоянные хранилища ключей и значений пользуются RocksDB.

² Например, в оконных хранилищах состояний содержатся ключи и значения, но в состав первых, кроме ключа записи, входит еще и оконное время.

воспроизводить всю тему. Достаточно воспроизвести данные с момента остановки приложения до момента возвращения к работе.



Папку для постоянного хранилища состояний можно указать с помощью свойства `StreamsConfig.STATE_DIR_CONFIG`. По умолчанию это папка `/tmp/kafka-streams`, но настоятельно рекомендуется выбрать другой вариант расположения, за пределами папки `/tmp`.

К сожалению, постоянные хранилища состояний устроены сложнее и могут работать медленнее, чем хранилища в памяти, которые извлекают данные непосредственно из RAM. Дополнительная сложность появляется из-за требований к внешнему хранилищу (то есть к жесткому диску). Кроме того, для настройки такого хранилища состояний нужно уметь работать с RocksDB (впрочем, в большинстве приложений это не вызывает сложностей).

К сожалению, прирост производительности, который дает размещение хранилища состояний в памяти, зачастую оказывается недостаточным для оправдания его использования, поскольку восстановление после сбоя в случае его использования занимает больше времени. Всегда можно увеличить производительность приложения, создав дополнительные разделы для распараллеливания работы. Поэтому я рекомендую начинать с постоянных хранилищ и переключаться на хранилища в памяти только в ситуациях, когда это даст заметное увеличение производительности. Для сокращения времени восстановления в этом случае имеет смысл пользоваться резервными репликами.

Теперь, когда у вас появилось некоторое представление о том, что собой представляют хранилища состояний и как они обеспечивают обработку с сохранением состояния/поведения, перейдем к упражнению и на практике посмотрим, как работают все эти концепции.

Список лидеров видеоигр

Индустрия видеоигр — пример области, в которой *потокковая обработка с сохранением состояния* дает отличные результаты, ведь как игрокам, так и игровым системам требуется немедленная обратная связь, а значит, как можно более низкая задержка. Именно поэтому компания Activision (которая создала серию игр *Call of Duty* и выполнила переиздание игр *Crash Bandicoot* и *Spyro*) использует Kafka Streams для телеметрии¹.

¹ О том, как это происходит в Activision, рассказывает Ярослав Ткаченко в подкасте Streaming Audio (<https://oreil.ly/gNYZZ>).

Таблица лидеров, созданием которой вы займетесь в этой главе, потребует моделирования данных новыми для вас способами. В частности, вы научитесь использовать абстракции таблиц Kafka Streams для моделирования данных в виде последовательности обновлений. Кроме того, мы подробно рассмотрим процессы соединения и агрегирования данных, позволяющие понять или вычислить взаимосвязь между событиями.

Затем я покажу, как с помощью *интерактивных запросов* получить из таблицы лидеров самую актуальную информацию. Вы научитесь создавать в Kafka Streams управляемые события микросервисы, тем самым расширив типы клиентов, с которыми могут обмениваться данными приложения потоковой обработки¹.

Посмотрим на архитектуру будущей таблицы лидеров видеоигр. Рисунок 4.1 демонстрирует топологию, которую вам предстоит реализовать в этой главе.

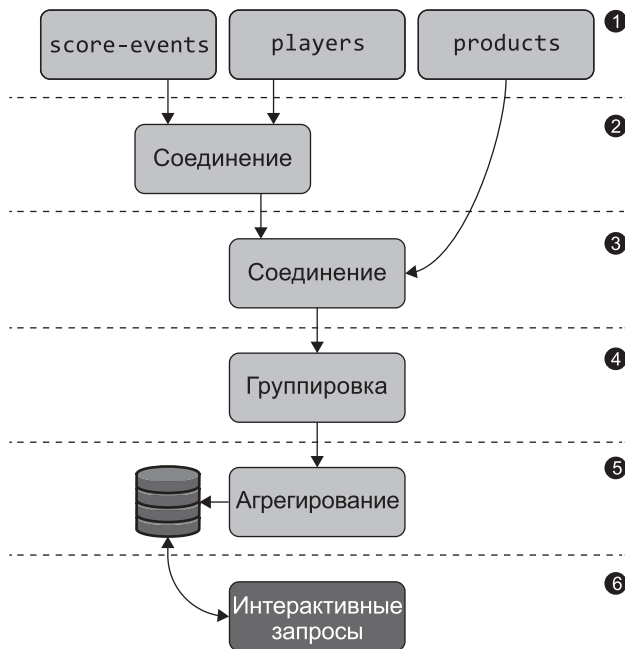


Рис. 4.1. Топология будущего приложения, генерирующего таблицу лидеров видеоигр

¹ Вы уже видели, как Kafka Streams ведет запись напрямую в темы-приемники, отправляя обработанные/обогащенные данные в целевые приложения. А вот интерактивные запросы могут использоваться клиентами, у которых в произвольный момент возникает необходимость обратиться к приложению Kafka Streams.

- ❶ Наш кластер Kafka состоит из трех тем, они перечислены ниже.
 - Тема `score-events` содержит результаты игр. Записи не имеют ключей и, как следствие, циклически распределяются по разделам темы.
 - Тема `players` содержит профили игроков. Ключом каждой записи является идентификатор игрока.
 - Тема `products` содержит информацию о различных видеоиграх. Ключом каждой записи является идентификатор продукта.
- ❷ Нужно обогатить результаты игр подробной информацией об игроках. Это достигается путем соединения (`join`).
- ❸ Обогатив данные из темы `score-events` данными из темы `player`, нужно добавить в результирующий поток подробную информацию о продукте. Это также реализуется путем соединения.
- ❹ Поскольку перед агрегированием требуется сгруппировать данные, выполним группировку обогащенного потока.
- ❺ Для определения трех лучших результатов каждой игры используем процедуру агрегирования.
- ❻ Наконец, нужно транслировать наружу рекорды для каждой игры. Этого можно достичь, создав RESTful-микросервис с помощью функции интерактивных запросов.

С топологией мы определились, теперь можно переходить к настройке проекта.

Настройка проекта

Весь приведенный в этой главе код можно найти по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>.

Если вы хотите использовать его в процессе проработки всех шагов нашей топологии, клонируйте репозиторий и перейдите в каталог, содержащий инструкцию к текущей главе. Это делается следующими командами:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-04/video-game-leaderboard
```

Сборка проекта осуществляется командой:

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

Настройка завершена, и можно приступить к работе над таблицей лидеров.

Модели данных

Как всегда, первым делом следует определиться с моделями данных. Наши исходные темы содержат данные в формате JSON, поэтому модели данных определим с помощью классов данных POJO. Именно их мы будем сериализовать и десериализовать, используя любую библиотеку сериализации JSON (я выбрал Gson, но вы, если хотите, можете взять Jackson или любую другую)¹.

В своих проектах я предпочитаю группировать модели данных в виде пакетов, например, `com.magicalpipelines.model`. В виде дерева файлов схема классов данных для текущего упражнения выглядит так:

```
src/
├── main
│   ├── java
│   │   └── com
│   │       ├── magicalpipelines
│   │       │   ├── model
│   │       │   │   ├── ScoreEvent.java ❶
│   │       │   │   ├── Player.java    ❷
│   │       │   │   └── Product.java    ❸
```

❶ Класс данных `ScoreEvent.java` используется для представления записей в теме `score-events`.

❷ Класс данных `Player.java` применяется для представления записей в теме `players`.

❸ Класс данных `Product.java` используется для представления записей в теме `products`.

Теперь создадим класс данных для каждой темы, как показано в табл. 4.2.

¹ Если бы наши темы содержали данные Avro, вместо указанных действий можно было бы определить модель данных в файле схемы Avro, как это делалось в главе 3.

Таблица 4.2. Примеры записей и классов данных для каждой темы

Тема Kafka	Пример записи	Класс данных
score-events	<pre>{ "score": 422, "product_id": 6, "player_id": 1 }</pre>	<pre>public class ScoreEvent { private Long playerId; private Long productId; private Double score; }</pre>
players	<pre>{ "id": 2, "name": "Mitch" }</pre>	<pre>public class Player { private Long id; private String name; }</pre>
products	<pre>"id": 1, "name": "Super Smash Bros" }</pre>	<pre>public class Product { private Long id; private String name; }</pre>



Сериализация и десериализация подробно обсуждались в одноименном разделе главы 3. В упражнении по созданию таблицы лидеров используются написанные мной сериализатор, десериализатор и класс `Serdes`. Как реализован `Serdes` для каждого из перечисленных в табл. 4.2 классов данных, можно посмотреть в исходном коде упражнения.

Добавление узлов-источников

Итак, мы определились с классами данных, и можно переходить к созданию узлов-источников. В нашей топологии их три, так как чтение будет осуществляться из трех тем. Первым делом следует выбрать абстракцию `Kafka Streams` для представления данных в базовой теме.

До сих пор мы имели дело только с абстракцией `KStream`, которая применялась для представления потоков записей без сохранения состояния. Но согласно нашей топологии темы `products` и `players` будут использоваться для поиска. Соответственно, здесь хорошо подойдет абстракция в виде таблицы¹. Но перед тем, как мы начнем сопоставлять темы с абстракциями `Kafka Streams`, важно понять разницу между `KStream`-, `KTable`- и `GlobalKTable`-представлениями. По мере рассмотрения каждого варианта мы будем заполнять табл. 4.3.

¹ Абстракцию `KStreams` также можно использовать для поиска/соединения, но это уже будут оконные операции, речь о которых пойдет только в следующей главе.

Таблица 4.3. Темы и соответствующие им абстракции

Тема Kafka	Абстракция
score-events	???
players	???
products	???

KStream

Для выбора наиболее подходящей абстракции имеет смысл определить характер темы и ее конфигурацию, а также пространство ключей записей темы-источника. Как правило, приложения Kafka Streams с сохранением состояния используют одну или несколько абстракций таблиц, но, если для одного или нескольких источников данных семантика модифицируемой таблицы не требуется, потоки **KStream** можно применять в комбинации с **KTable** или **GlobalKTable**.

Тема **score-events** содержит необработанные события в виде набранных игроками очков. Они не имеют ключей и, следовательно, циклически распределяются по несжатой теме. Поскольку таблицы базируются на ключах, понятно, что для темы **score-events** нужно использовать абстракцию **KStream**. Чтобы взять другую абстракцию, следует поменять стратегию снабжения ключами (в любом приложении, которое генерирует данные для нашей темы-источника), но это не всегда реализуемо. Кроме того, нас интересует *рекорд* каждого игрока, а не *последние набранные им очки*, поэтому семантика таблицы (то есть сохранение последней записи для рассматриваемого ключа) не соответствует способам, которыми мы собираемся использовать тему **score-events**.

Итак, данные в этой теме будут представлены абстракцией **KStream**. Укажем это в табл. 4.3.

Тема Kafka	Абстракция
score-events	KStream
players	???
products	???

Темы **players** и **products** содержат ключи, кроме того, нас интересует только последняя для каждого уникального ключа запись. Следовательно, для этих тем абстракция **KStream** уже не подойдет. Посмотрим, нельзя ли в этих случаях воспользоваться абстракцией **KTable**.

KTable

Сжатая тема `players` содержит профили игроков, при этом каждая запись снабжена ключом в виде идентификатора игрока. Так как нас интересует только последнее состояние игрока, эту тему имеет смысл представить с помощью абстракции на основе таблиц (`KTable` или `GlobalKTable`).

Выбор между `KTable` и `GlobalKTable` базируется на состоянии пространства ключей. Если оно очень велико (то есть содержит множество уникальных ключей) или ожидается его сильный рост, имеет смысл использовать абстракцию `KTable`. Это позволит распределять фрагменты общего состояния по всем запущенным экземплярам приложения. Подобное разделение снижает издержки на локальное хранение каждого экземпляра Kafka Streams.

Но еще важнее при этом выборе учитывать такой фактор, как необходимость синхронизированной по времени обработки. В `KTable` синхронизация по времени есть, поэтому при чтении из нескольких источников (например, в случае соединения) Kafka Streams смотрит на временную метку, чтобы определить очередность обработки записей. В результате мы получаем комбинацию записей, сделанных в одно время, отсюда операция соединения становится более предсказуемой. А вот таблицы `GlobalKTable` *не* синхронизируются по времени и «полностью заполняются до начала обработки»¹. Соответственно, операция соединения выполняется с самой последней версией `GlobalKTable`, что меняет семантику программы.

Впрочем, подробно на этих вещах мы сейчас останавливаться не будем, так как время и роль, которую оно играет в Kafka Streams, — это материал следующей главы. Посмотрим только на пространство ключей. Тема `players` содержит записи для всех уникальных игроков в системе. Количество записей зависит от того, на каком этапе жизненного цикла компании или продукта мы находимся. Сначала оно может быть небольшим, но, как правило, ожидается, что со временем это число значительно вырастет, поэтому для темы `players` мы воспользуемся абстракцией `KTable`.

Рисунок 4.2 демонстрирует, как с помощью `KTable` базовое состояние распределяется между несколькими экземплярами приложения.

¹ Более подробно эту тему рассматривают Флориан Троссбах (Florian Trossbach) и Маттиас Джей Сакс (Matthias J. Sax) в статье *Crossing the Streams: Joins in Apache Kafka* (<https://oreil.ly/dUo3a>).

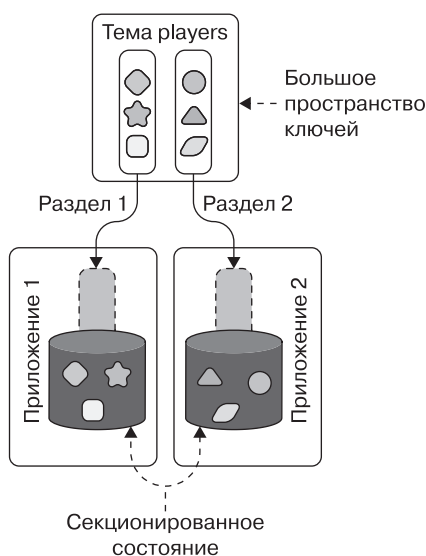


Рис. 4.2. Абстракцию KTable имеет смысл использовать, когда состояние требуется разделить между несколькими экземплярами приложения, синхронизировав процесс обработки

Вот обновленная версия табл. 4.3.

Тема Kafka	Абстракция
score-events	KStream
players	KTable
products	???

Осталась всего одна, относительно небольшая тема — **products**. В этом случае нам нужна возможность полностью воспроизводить состояние для всех экземпляров приложения. Рассмотрим абстракцию, которая позволяет это делать.

GlobalKTable

Тема **products** похожа на тему **players** по своей конфигурации (она сжата) и по ограниченному пространству ключей. Мы следим за последней записью для каждого уникального идентификатора продукта, причем количество продуктов фиксировано. Но количество уникальных ключей в теме **products** меньше, чем в теме **players**. И даже если в таблице лидеров будут отслеживаться рекорды

нескольких сотен игр, пространство состояний все равно получится достаточно небольшим, чтобы полностью поместиться в памяти.

Тема `products` не только имеет меньший размер, но и наполнена относительно статичными данными. Создание видеоигр занимает много времени, поэтому большого количества обновлений в этой теме мы не ждем.

К таким характеристикам (небольшие и статические данные) лучше всего подходит абстракция `GlobalKTable`. Именно ею мы и воспользуемся для темы `products`. В результате каждый из экземпляров Kafka Streams будет хранить полную копию информации о продукте, как показано на рис. 4.3. Позже вы увидите, что это значительно упрощает процедуру соединения.

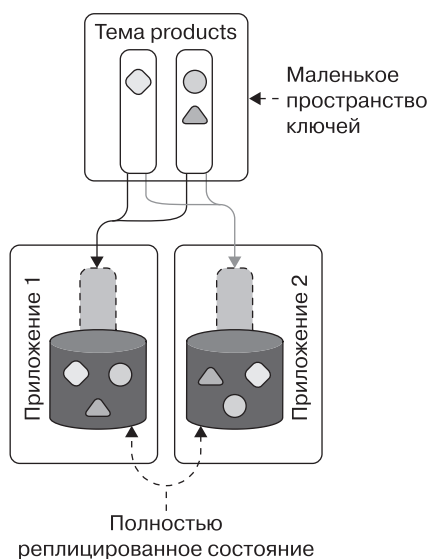


Рис. 4.3. Абстракция `GlobalKTable` применяется, когда пространство ключей невелико и не требуется синхронизация по времени

Внесем последнюю правку в таблицу, сопоставляющую темы и абстракции.

Тема Kafka	Абстракция
<code>score-events</code>	<code>KStream</code>
<code>players</code>	<code>KTable</code>
<code>products</code>	<code>GlobalKTable</code>

Итак, абстракции, которые будут представлять каждую из тем-источников, выбраны, давайте их создадим.

Абстракции потока и таблиц

Получить абстракции для потока и таблиц очень просто. Вот блок кода на языке DSL, создающий объекты `KStream`, `KTable` и `GlobalKTable`:

```
StreamsBuilder builder = new StreamsBuilder();

KStream<byte[], ScoreEvent> scoreEvents =
    builder.stream(
        "score-events",
        Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent())); ❶

KTable<String, Player> players =
    builder.table(
        "players",
        Consumed.with(Serdes.String(), JsonSerdes.Player())); ❷

GlobalKTable<String, Product> products =
    builder.globalTable(
        "products",
        Consumed.with(Serdes.String(), JsonSerdes.Product())); ❸
```

❶ Объект `KStream` представляет данные в теме `score-events`, не имеющей ключей.

❷ Создаем секционированную (или шардированную) таблицу для темы `players`, используя объект `KTable`.

❸ Создаем объект `GlobalKTable` для темы `products`, которая будет целиком воспроизводиться во всех экземплярах приложения.

Первый этап нашей топологии реализован. Пришло время заняться соединением потоков и таблиц.

Соединение

В реляционных базах наборы данных чаще всего комбинируются с помощью операций соединения¹. В реляционных системах данные часто многомерны и разбросаны по множеству таблиц. С аналогичной ситуацией нередко можно

¹ Запросы `UNION` — еще один метод объединения наборов данных в реляционных базах. Поведение метода `merge` больше всего напоминает именно этот запрос.

столкнуться в Kafka. Это имеет место либо потому, что события поступают из разных мест, либо так удобно разработчикам, так как они привыкли к реляционным моделям данных, или же потому, что некоторые интеграции Kafka (например, JDBC Kafka Connector, Debezium, Maxwell) приносят как необработанные данные, так и модели данных из систем-источников.

Но при любых вариантах распределения данных в Kafka возможность комбинировать данные из разных потоков и таблиц на основе их *взаимосвязи* расширяет потенциал обогащения данных в Kafka Streams. Более того, операция соединения сильно отличается от слияния потоков, которое вы видели на рис. 3.6. Метод **merge** в Kafka Streams просто сливает все записи в один поток. Это метод без сохранения состояния, так как его применяют в случаях, когда неинтересен контекст объединяемых событий.

Соединения можно рассматривать как особый вид *условного слияния*, при котором учитывается взаимосвязь между событиями, причем записи не в точности копируются в поток-приемник, а, скорее, комбинируются. Более того, взаимоотношения между записями должны быть зафиксированы и сохранены, чтобы опираться на них в процессе соединения. Рисунок 4.4 демонстрирует упрощенную схему одного из вариантов соединения.

Как и в реляционных системах, в Kafka Streams поддерживается нескольких типов соединения. Поэтому прежде чем объединять поток **score-events** с таблицей **players**, посмотрим на перечень доступных операторов, чтобы выбрать наиболее подходящий для нашего конкретного случая.

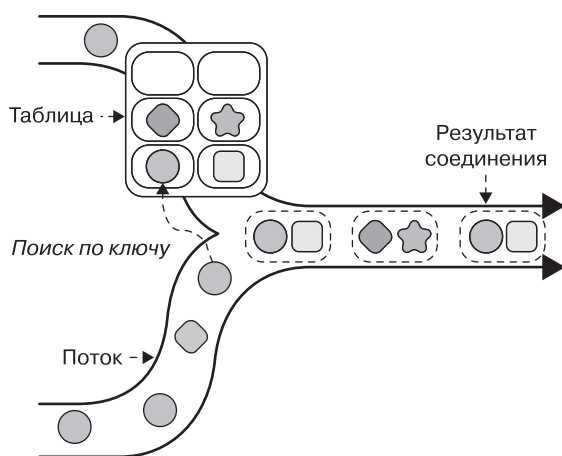


Рис. 4.4. Соединение сообщений

Операторы соединения

В Kafka Streams существует три варианта операторов, осуществляющих соединение потоков и таблиц. Они перечислены в табл. 4.4.

Таблица 4.4. Операторы соединения

Оператор	Описание
<code>join</code>	Внутреннее соединение. Применяется, когда входные записи с обеих сторон используют один и тот же ключ
<code>leftJoin</code>	Соединение слева. Семантика зависит от типа: <ul style="list-style-type: none"> соединение «поток — таблица»: возникает при получении записи слева. При отсутствии справа записи с таким же ключом правое значение приравнивается к нулю; соединение «поток — поток» или «таблица — таблица»: в отличие от предыдущего варианта поиск может инициировать и появление записи справа. Если при этом слева нет записи с совпадающим ключом, соединение вообще не дает результата
<code>outerJoin</code>	Внешнее соединение. Возникает при получении записи с любой из сторон. Если с другой стороны запись с таким же ключом отсутствует, ее значение считается равным нулю



При обсуждении различий между операторами соединения я упоминал *различные стороны*. Нужно запомнить, что *правая часть* всегда передается соответствующему оператору соединения в качестве параметра. Например:

```
KStream<String, ScoreEvent> scoreEvents = ...;
KTable<String, Player> players = ...;
```

```
scoreEvents.join(players, ...); ❶
```

❶ Поток `scoreEvents` находится слева, а таблица `players` справа.

Давайте посмотрим, какие типы соединений можно создать с помощью этих операторов.

Типы соединений

В табл. 4.5 перечислены типы соединений, поддерживаемые в Kafka Streams. Про последний столбец мы поговорим в следующем разделе.

Таблица 4.5. Типы соединений

Тип	Оконный	Операторы	Требуется ли совместное секционирование
KStream-KStream	Да*	join leftJoin outerJoin	Да
KTable-KTable	Нет	join leftJoin outerJoin	Да
KStream-KTable	Нет	join leftJoin	Да
KStream-GlobalKTable	Нет	join leftJoin	Нет

* Обратите внимание, что соединения типа KStream-KStream являются оконными. Они будут подробно обсуждаться в следующей главе.

В упражнении этой главы нам понадобятся соединения двух типов:

- KStream-KTable, чтобы соединить поток `score-events` и таблицу `players`;
- KStream-GlobalKTable, чтобы соединить результат предыдущей операции с глобальной таблицей `products`.

Мы проведем внутреннее соединение с помощью оператора `join`, потому что операция должна происходить только при совпадении с обеих сторон. В таблице указано, что для соединения такого типа (KStream-KTable) требуется совместное секционирование. Посмотрим, что это такое.

Совместное секционирование

Слышен ли звук падающего дерева в лесу, если рядом никого нет?

Философская загадка

Эта знаменитая загадка ставит вопрос о роли наблюдателя в возникновении события. В Kafka Streams тоже всегда следует помнить о влиянии наблюдателя на *обработку события*.

В подразделе «Задачи и потоки выполнения» главы 2 мы говорили о том, что каждый раздел сопоставляется одной задаче Kafka Streams. В нашей аналогии эти задачи будут играть роль наблюдателей, поскольку именно они отвечают за

фактическое потребление событий и их обработку. Нет гарантии, что события из разных разделов будут обрабатываться одной и той же задачей Kafka Streams. Соответственно, мы имеем потенциальную *проблему наблюдаемости*.

Она схематически представлена на рис. 4.5. Когда связанные события обрабатываются разными задачами, связь между ними невозможно точно определить из-за наличия двух наблюдателей. Но операция соединения призвана комбинировать именно связанные события, соответственно, из-за проблемы наблюдаемости это становится невозможным.

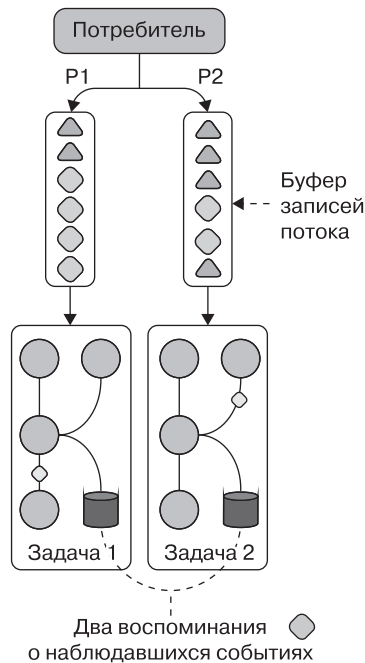


Рис. 4.5. Попытка соединить связанные записи, которые обрабатываются разными задачами, порождает проблему наблюдаемости

Чтобы понять взаимосвязь событий при соединении или агрегировать последовательность событий, нужно гарантировать нахождение связанных событий в одном разделе. Только в этом случае они будут обрабатываться одной задачей.

Направить связанные события в один раздел позволяет следующий порядок совместного секционирования.

- Записи с обеих сторон должны иметь один и тот же ключ и должны быть секционированы по этому ключу с использованием одинаковой стратегии.

- Входные темы с обеих сторон должны содержать одинаковое количество разделов. Это единственное требование, которое проверяется при запуске. Если оно не выполняется, появляется исключение `TopologyBuilderException`.

Первое требование у нас не выполнено. Напомню, что записи в теме `score-events` не имеют ключей, а мы хотим соединить их с таблицей `players`, в которой ключами служат идентификаторы игроков. Следовательно, *перед соединением* необходимо добавить такие же ключи в тему `score-events`. Это можно сделать с помощью оператора выбора ключа `selectKey`, как показано в примере 4.1.

Пример 4.1. Применение оператора `selectKey`

```
KStream<String, ScoreEvent> scoreEvents = builder
    .stream(
        "score-events",
        Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
    .selectKey((k, v) -> v.getPlayerId().toString()); ❶
```

❶ Оператор `selectKey` используется для изменения ключей записей. Мы это делаем, чтобы соблюсти первое требование совместного секционирования: наличие одинаковых ключей на каждой стороне будущего соединения.

Процедура изменения ключей схематично показана на рис. 4.6.

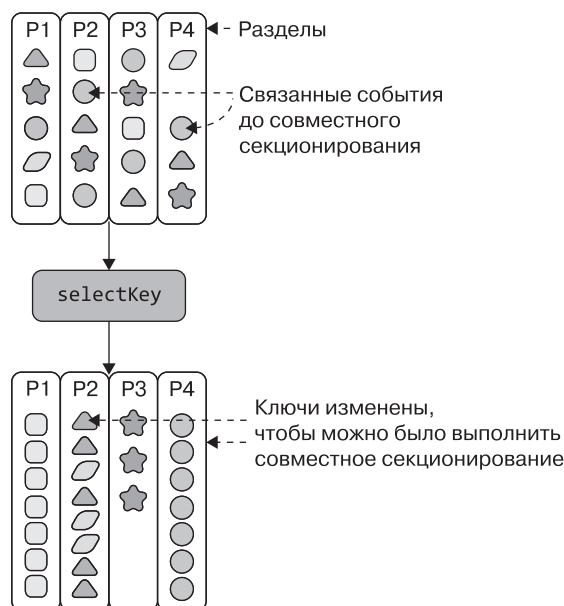


Рис. 4.6. Изменение ключей гарантирует попадание связанных записей в один раздел



Когда мы добавляем в топологию оператор, переназначающий ключи, базовые данные *помечаются для повторного секционирования*. Это означает, что при появлении ниже по потоку оператора, который считывает новый ключ, Kafka Streams:

- отправляет данные с новыми ключами во внутреннюю тему, где происходит секционирование;
- заново считывает данные с новыми ключами.

Такая процедура гарантирует, что связанные записи (то есть записи с одинаковым ключом) на последующих этапах топологии будут обрабатываться одной и той же задачей. Но передача данных по сети в специальную тему делает процесс смены ключей весьма ресурсоемким.

А как насчет соединения типа `KStream-GlobalkTable`, которое потребуется для добавления темы `products`? В табл. 4.5 указано, что для соединений с `GlobalkTable` совместное секционирование не требуется, поскольку состояние полностью реплицируется в каждом экземпляре приложения Kafka Streams. Следовательно, проблема наблюдаемости при соединении с `GlobalkTable` не возникает.

Для соединения наших потоков и таблиц все практически готово. Осталось понять, как на самом деле комбинируются записи во время операции `join`.

Интерфейс `ValueJoiner`

В SQL при выполнении соединения нужно использовать оператор `JOIN` в сочетании с оператором `SELECT`, чтобы задать форму (или *проекцию*) получающейся в результате записи. Например:

```
SELECT a.customer_name, b.purchase_amount ❶  
FROM customers a  
LEFT JOIN purchases b  
ON a.customer_id = b.customer_id
```

❶ Проекция полученной в результате соединения записи включает в себя два столбца.

Однако в Kafka Streams для указания способа соединения записей используется интерфейс `ValueJoiner`, который на базе участвующих в операции записей создает новую, комбинированную запись. Процедуру соединения потока `score-events` с таблицей `players` можно описать следующим псевдокодом:

```
(scoreEvent, player) -> combine(scoreEvent, player);
```

Но это еще не все. Как правило, существует специальный класс данных, выполняющий одно из следующих действий:

- создание контейнера для каждого из значений, участвующих в соединении;
- извлечение соответствующего поля с каждой стороны соединения с сохранением извлеченных значений в свойствах класса.

Рассмотрим оба подхода. Начнем с класса-оболочки для соединения `score-events -> players`. Простая реализация такого класса данных показана в примере 4.2.

Пример 4.2. Класс данных для построения соединенных записей `score-events -> players`

```
public class ScoreWithPlayer {
    private ScoreEvent scoreEvent;
    private Player player;

    public ScoreWithPlayer(ScoreEvent scoreEvent, Player player) { ❶
        this.scoreEvent = scoreEvent; ❷
        this.player = player;
    }

    // методы доступа опущены для краткости
}
```

❶ Конструктор содержит параметры обеих сторон соединения. Слева это `ScoreEvent`, а справа — `Player`.

❷ Ссылка на каждую участвующую в соединении запись сохраняется внутри класса-оболочки.

Укажем новый класс-оболочку в качестве возвращаемого типа в интерфейсе `ValueJoiner`. Реализация этого интерфейса показана в примере 4.3. Он объединяет `ScoreEvent` (из потока `KStream` событий темы `score-events`) и `Player` (из таблицы `KTable` темы `players`) в экземпляр `ScoreWithPlayer`.

Пример 4.3. Интерфейс, объединяющий темы `score-events` и `players`

```
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
    (score, player) -> new ScoreWithPlayer(score, player); ❶
```

❶ Здесь также можно было просто использовать ссылку на статический метод, например `ScoreWithPlayer :: new`.

Теперь осталось соединить экземпляр `ScoreWithPlayer` с объектом `Product` (из таблицы `GlobalKTable` темы `products`). Для этого можно снова использовать

шаблон-оболочку, а можно просто извлечь нужные свойства с каждой стороны соединения.

Вот реализация класса данных, который извлекает нужные значения и сохраняет их в соответствующих свойствах класса:

```
public class Enriched {
    private Long playerId;
    private Long productId;
    private String playerName;
    private String gameName;
    private Double score;

    public Enriched(ScoreWithPlayer scoreEventWithPlayer, Product product) {
        this.playerId = scoreEventWithPlayer.getPlayer().getId();
        this.productId = product.getId();
        this.playerName = scoreEventWithPlayer.getPlayer().getName();
        this.gameName = product.getName();
        this.score = scoreEventWithPlayer.getScoreEvent().getScore();
    }

    // методы доступа опущены для краткости
}
```

Для построения интерфейса `ValueJoiner`, осуществляющего соединение `KStream-GlobalKTable`, все готово. Его код приведен в примере 4.4.

Пример 4.4. Интерфейс `ValueJoiner`, представленный в виде лямбда-выражения

```
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
    (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
```

Итак, мы объяснили Kafka Streams, как именно нужно выполнять соединение наших записей, теперь можно перейти к фактической реализации этих соединений.

Соединение `KStream` и `KTable`

Пришло время соединить поток `score-events` с таблицей `players`. Так как соединение должно осуществляться только в случае, когда запись `ScoreEvent` может быть сопоставлена с записью `Player` (через ключи), проведем внутреннее соединение:

```
Joined<String, ScoreEvent, Player> playerJoinParams =
    Joined.with(❶
        Serdes.String(),
```

```

        JsonSerdes.ScoreEvent(),
        JsonSerdes.Player()
    );

    KStream<String, ScoreWithPlayer> withPlayers =
        scoreEvents.join( ❷
            players,
            scorePlayerJoiner, ❸
            playerJoinParams
        );

```

❶ Параметры соединения определяют способ сериализации ключей и значений для участвующих в процедуре записей.

❷ Оператор `join` выполняет внутреннее соединение.

❸ Это интерфейс `ValueJoiner`, созданный в примере 4.3. Новое значение `ScoreWithPlayer` создается из двух соединенных записей. В коде класса данных `ScoreWithPlayer` в примере 4.2 можно увидеть, как значения слева и справа передаются в конструктор.

Как видите, все очень просто. Более того, если сейчас запустить код, а затем вывести список доступных в кластере Kafka тем, вы увидите, что Kafka Streams породила две дополнительные темы.

- Тему для операции смены ключей, которая выполнялась в примере 4.1.
- Тему для резервной копии хранилища состояний, которое используется оператором `join`. Это часть отказоустойчивого поведения, которое обсуждалось выше в пункте «Отказоустойчивость» подраздела «Общие характеристики» в этой главе.

Список тем выводится с помощью консольного сценария¹:

```

$ kafka-topics --bootstrap-server kafka:9092 --list

players
products
score-events
dev-KSTREAM-KEY-SELECT-0000000001-repartition ❶
dev-players-STATE-STORE-0000000002-changelog ❷

```

❶ Внутренняя тема повторного секционирования (`repartition`), созданная Kafka Streams. Она имеет префикс с идентификатором нашего приложения Kafka Streams (`dev`).

¹ Если вы не пользуетесь платформой Confluent, сценарий будет называться `kafka-topics.sh`.

❷ Внутренняя тема журнала изменений (changelog), созданная Kafka Streams. Как и в предыдущем случае, в ее названии фигурирует префикс с идентификатором нашего приложения Kafka Streams.

Теперь можно переходить ко второму соединению.

Соединение KStream и GlobalKTable

При обсуждении требований к совместному секционированию я упоминал, что в случае соединения с `GlobalKTable` наличие одинаковых ключей не обязательно. У локальной задачи есть полная копия таблицы, так что соединение можно выполнить, используя какой-то атрибут значения записи на стороне `KStream`¹. Это более эффективный подход, чем переназначение ключей через буферную тему, призванное гарантировать обработку связанных записей одной задачей.

Для реализации соединения типа `KStream-GlobalKTable` нужно создать интерфейс `KeyValueMapper`, задающий способ сопоставления записей `KStream` и `GlobalKTable`. В рассматриваемом случае для этой цели из значения `ScoreWithPlayer` можно извлечь идентификатор продукта, как показано в этом фрагменте кода:

```
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper =  
    (leftKey, scoreWithPlayer) -> {  
        return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());  
    };
```

Итак, у нас есть интерфейс `KeyValueMapper` и созданный в примере 4.4 интерфейс `ValueJoiner`, так что для соединения все готово:

```
KStream<String, Enriched> withProducts =  
    withPlayers.join(products, keyMapper, productJoiner);
```

Мы реализовали второй и третий этапы топологии нашей таблицы лидеров. Теперь нужно сгруппировать обогащенные записи, подготовив их к агрегированию.

Группировка записей

В Kafka Streams перед агрегированием потоков или таблиц необходимо сгруппировать предназначенные для этой процедуры `KStream` или `KTable`. Как и в случае с переназначением ключей перед соединением, группировка нужна, чтобы гарантировать обработку связанных записей одной задачей Kafka Streams.

¹ Со стороны `GlobalKTable` для поиска по-прежнему будет использоваться ключ записи.

Процедуры группировки потоков и таблиц немного отличаются, поэтому мы рассмотрим их по отдельности.

Группировка потоков

Для группировки потоков `KStream` применяются два метода:

- `groupBy`;
- `groupByKey`.

Работа метода `groupBy` похожа на процесс изменения ключей потока методом `selectKey`, который переназначает ключи и заставляет Kafka Streams установить флаг, запускающий повторное секционирование. Если ниже по течению присутствует оператор, считывающий новые ключи, Kafka Streams автоматически создает тему для повторного секционирования и возвращает данные в Kafka для завершения процесса смены ключей.

Применение метода `groupBy` демонстрирует пример 4.5.

Пример 4.5. Изменение ключей и группировка потока `KStream` методом `groupBy`

```
KGroupedStream<String, Enriched> grouped = withProducts.groupBy(  
    (key, value) -> value.getProductId().toString(), ❶  
    Grouped.with(Serdes.String(), JsonSerdes.Enriched())); ❷
```

❶ Для выбора нового ключа можно воспользоваться лямбда-выражением, поскольку метод `groupBy` ожидает в качестве параметра экземпляр функционального интерфейса `KeyValueMapper`.

❷ Параметр `Grouped` позволяет передавать дополнительные параметры группировки, в том числе ключ и значение `Serdes`, которые будут использоваться при сериализации записей.

Но если записи не требуют переназначения ключей, лучше прибегнуть к более производительному методу `groupByKey`. В этом случае флаг, запускающий повторное секционирование, не устанавливается, что позволяет избежать дополнительных сетевых вызовов, связанных с отправкой данных обратно в Kafka. Вот пример применения метода `groupByKey`:

```
KGroupedStream<String, Enriched> grouped =  
    withProducts.groupByKey(  
        Grouped.with(Serdes.String(),  
            JsonSerdes.Enriched()));
```

Нас интересуют рекорды для каждого идентификатора продукта, при этом ключами в обогащенном потоке служат идентификаторы игроков, поэтому в топологии нашей таблицы лидеров будет использоваться метод `groupBy`.

Независимо от выбранного метода группировки потока Kafka Streams возвращает экземпляр нового типа `KGroupedStream`. Это промежуточное представление потока событий после группировки по ключам, позволяющее осуществлять агрегирование.

Группировка таблиц

В отличие от группировки потоков группировка таблиц осуществляется только методом `groupBy`. Но в этом случае он возвращает не экземпляр `KGroupedStream`, а другое промежуточное представление, перегруппированное по ключу. Оно называется `KGroupedTable`. В остальном процессы группировки `KTables` и `KStream` идентичны. Например, сгруппировать таблицу `players` для ее последующей агрегации (скажем, подсчета количества игроков) позволяет следующий код:

```
KGroupedTable<String, Player> groupedPlayers =  
    players.groupBy(  
        (key, value) -> KeyValue.pair(key, value),  
        Grouped.with(Serdes.String(), JsonSerdes.Player()));
```

Этот код в нашем упражнении не потребуется. Я привел его исключительно для демонстрации. Теперь, когда вы научились группировать как потоки, так и таблицы и завершили шаг 4 топологии узлов-обработчиков, можно переходить к следующему этапу.

Агрегирование

Остался последний шаг — реализовать подсчет рекордов для каждой игры, и таблица лидеров будет готова. В Kafka Streams такое агрегирование выполняют следующие методы:

- `aggregate`;
- `reduce`;
- `count`.

В общих чертах агрегированием называется объединение нескольких входных значений в одно выходное. Обычно агрегирование принято воспринимать как математическую операцию, но это не всегда так. Метод `count` — это действительно математическая операция, которая вычисляет количество событий для каждого ключа, а вот методы `aggregate` и `reduce` более универсальны и могут объединять значения с использованием любой комбинационной логической схемы.



Методы `reduce` и `aggregate` очень похожи и различаются только возвращаемым типом. Метод `reduce` требует совпадения типа выходных и входных данных, в то время как метод `aggregate` допускает другой тип выходной записи.

Агрегирование можно применять как к потокам, так и к таблицам, хотя и с небольшими отличиями в семантике, поскольку потоки неизменяемы, в то время как таблицы допускают редактирование. В результате используются немного различающиеся версии методов `aggregate` и `reduce`. Версия для потоков принимает два параметра: инициализатор и сумматор, а версия для таблиц — три: инициализатор, сумматор и вычитатель¹.

Рассмотрим процесс агрегирования потоков на примере объединения сведений о рекордно высоких очках.

Агрегирование потоков

Процедуру агрегирования потоков записей можно разбить на две части. Во-первых, это создание функции для присвоения нового агрегированного значения. Такая функция называется *инициализатором* (`initializer`). Во-вторых, создается функция для агрегирования новых поступающих записей с указанным ключом. Эта функция называется *сумматором* (`adder`).

Инициализатор

В ситуации, когда топология Kafka Streams обнаруживает новый ключ, требуется способ присвоения начального значения. В этом нам поможет интерфейс `Initializer`, который, как и многие классы в Kafka Streams API, относится к функциональным интерфейсам (то есть содержащим всего один абстрактный метод). Это позволяет представить его в виде лямбда-выражения.

Например, при рассмотрении внутренней структуры агрегата `count` обнаруживается инициализатор, который устанавливает начальное значение равным 0:

```
Initializer<Long> countInitializer = () -> 0L; ❶
```

❶ Инициализатор задается как лямбда-выражение, так как `Initializer` — это функциональный интерфейс.

¹ Потому что потоки обновляются только путем добавления в них новых записей.

В более сложных случаях можно создать собственный настраиваемый инициализатор. Например, при построении таблицы лидеров видеоигр требуется вычислить три самых высоких результата для каждой игры. В такой ситуации имеет смысл создать отдельный класс с логической схемой отслеживания этих трех результатов и предоставлять его экземпляр всякий раз, когда возникает необходимость инициализировать новый результат агрегирования.

В нашем упражнении это будет класс `HighScores`. Ему потребуется базовая структура данных для хранения трех лучших результатов каждой видеоигры. Для этого можно воспользоваться, например, классом `TreeSet` из стандартной библиотеки Java. Он предоставляет коллекцию для хранения отсортированных элементов и хорошо подходит для наших целей.

Вот начальная реализация класса данных, который мы будем использовать для агрегирования рекордов:

```
public class HighScores {  
    private final TreeSet<Enriched> highScores = new TreeSet<>();  
}
```

Теперь нужно указать Kafka Streams способ инициализации нового класса. Для этого достаточно создать экземпляр этого класса:

```
Initializer<HighScores> highScoresInitializer = HighScores::new;
```

Осталось реализовать логическую схему агрегирования. Перед нами стоит задача отследить три лучших результата для каждой видеоигры.

Сумматор

Следующий шаг в решении задачи агрегирования потока — определить логическую схему объединения двух агрегатов. Это достигается с помощью интерфейса `Aggregator`, который, как и `Initializer`, является функциональным интерфейсом и может быть представлен в виде лямбда-выражения. Реализующая его функция должна принимать три параметра:

- ключ записи;
- значение записи;
- текущее значение агрегирования.

Вот код создания агрегатора рекордов:

```
Aggregator<String, Enriched, HighScores> highScoresAdder =  
    (key, value, aggregate) -> aggregate.add(value);
```

Обратите внимание, что параметр `aggregate` — это экземпляр класса `HighScores`. Поскольку наш агрегатор вызывает метод `HighScores.add`, его просто нужно реализовать в нашем классе `HighScores`. Код выглядит очень просто: метод `add` всего лишь добавляет новые высокие очки во внутреннюю коллекцию `TreeSet` и, если три верхние позиции уже заполнены, удаляет данные из нижней позиции:

```
public class HighScores {
    private final TreeSet<Enriched> highScores = new TreeSet<>();

    public HighScores add(final Enriched enriched) {
        highScores.add(enriched); ❶

        if (highScores.size() > 3) { ❷
            highScores.remove(highScores.last());
        }

        return this;
    }
}
```

❶ При каждом вызове сумматора (метода `HighScores.add`) добавляется новая запись в структуру `TreeSet`, которая выполняет автоматическую сортировку.

❷ При наличии более трех записей о рекордах нижняя запись удаляется.

Чтобы структура `TreeSet` знала, как сортировать объекты `Enriched` (и, следовательно, имела возможность идентифицировать подлежащую удалению запись `Enriched` с наименьшим количеством очков), реализуем интерфейс `Comparable`, как показано в примере 4.6.

Пример 4.6. Обновленный класс `Enriched`, реализующий интерфейс `Comparable`

```
public class Enriched implements Comparable<Enriched> { ❶

    @Override
    public int compareTo(Enriched o) { ❷
        return Double.compare(o.score, score);
    }

    // опущено для краткости
}
```

❶ Добавление в класс `Enriched` реализации интерфейса `Comparable`, поскольку для определения трех самых высоких результатов необходимо сравнение одного объекта `Enriched` с другим.

❷ В нашей реализации метода `compareTo` свойство `score` используется для сравнения двух объектов `Enriched`.

Сейчас у нас есть как инициализатор, так и сумматор, и можно выполнить агрегирование, как показано в примере 4.7.

Пример 4.7. Агрегирование рекордов методом `aggregate`

```
KTable<String, HighScores> highScores =  
    grouped.aggregate(highScoresInitializer, highScoresAdder);
```

Агрегирование таблиц

Процессы агрегирования таблиц и потоков очень похожи. В случае с таблицами также требуются *инициализатор* и *сумматор*. Но так как таблицы допускают редактирование, им необходима возможность обновлять агрегированное значение при удалении ключа¹. Поэтому для них определен третий параметр, называемый *вычитателем* (subtractor).

Вычитатель

В примере с созданием таблицы лидеров вычитатель не требуется, поэтому предположим, что мы хотим подсчитать количество игроков в таблице `players`. В принципе, эта задача решается с помощью метода `count`, но, чтобы продемонстрировать процесс создания вычитателя, напомним собственную функцию агрегирования, по сути эквивалентную этому методу. Вот ее базовая реализация:

```
KGroupedTable<String, Player> groupedPlayers = players.groupBy(  
    (key, value) -> KeyValue.pair(key, value),  
    Grouped.with(Serdes.String(), JsonSerdes.Player()));  
  
groupedPlayers.aggregate(  
    () -> 0L, ❶  
    (key, value, aggregate) -> aggregate + 1L, ❷  
    (key, value, aggregate) -> aggregate - 1L); ❸
```

❶ Инициализатор присваивает результату агрегирования начальное значение 0.

❷ При обнаружении нового ключа сумматор увеличивает значение текущего счетчика.

❸ При удалении ключа вычитатель уменьшает значение текущего счетчика.

К этому моменту мы уже создали приличный объем кода. Посмотрим, как отдельные написанные нами фрагменты сочетаются друг с другом.

¹ Тема удаления ключей пока не рассматривалась. Мы коснемся ее в главе 6, когда будем обсуждать очистку хранилищ состояния.

Объединение фрагментов кода

До сих пор мы писали код, отвечающий за обработку отдельных фрагментов топологии будущей таблицы лидеров. Пришла пора превратить их в единое целое, как это сделано в примере 4.8.

Пример 4.8. Топология узлов-обработчиков нашего приложения, создающего таблицу лидеров видеоигр

```
// сборщик для конструирования топологии
StreamsBuilder builder = new StreamsBuilder();

// регистрация потока событий с очками игроков
KStream<String, ScoreEvent> scoreEvents = ❶
    builder
        .stream(
            "score-events",
            Consumed.with(Serdes.ByteArray(), JsonSerdes.ScoreEvent()))
        .selectKey((k, v) -> v.getPlayerId().toString()); ❷

// создание секционированной таблицы с идентификаторами игроков
KTable<String, Player> players = ❸
    builder.table("players", Consumed.with(Serdes.String(),
        JsonSerdes.Player()));

// создание глобальной таблицы продуктов
GlobalKTable<String, Product> products = ❹
    builder.globalTable(
        "products",
        Consumed.with(Serdes.String(), JsonSerdes.Product()));

// объединение параметров для соединения scoreEvents – players
Joined<String, ScoreEvent, Player> playerJoinParams =
    Joined.with(Serdes.String(), JsonSerdes.ScoreEvent(), JsonSerdes.Player());

// соединение scoreEvents – players
ValueJoiner<ScoreEvent, Player, ScoreWithPlayer> scorePlayerJoiner =
    (score, player) -> new ScoreWithPlayer(score, player);
KStream<String, ScoreWithPlayer> withPlayers =
    scoreEvents.join(players, scorePlayerJoiner, playerJoinParams); ❺

// сопоставление записей score-with-player с записями products
KeyValueMapper<String, ScoreWithPlayer, String> keyMapper = (leftKey,
    scoreWithPlayer) -> {
    return String.valueOf(scoreWithPlayer.getScoreEvent().getProductId());
};

// соединение потока withPlayers с глобальной таблицей product
ValueJoiner<ScoreWithPlayer, Product, Enriched> productJoiner =
    (scoreWithPlayer, product) -> new Enriched(scoreWithPlayer, product);
KStream<String, Enriched> withProducts = withPlayers.join(products, keyMapper,
    productJoiner); ❻
```

```
// группировка обогащенного потока product
KGroupedStream<String, Enriched> grouped = withProducts.groupBy( ❷
    (key, value) -> value.getProductId().toString(),
    Grouped.with(Serdes.String(), JsonSerdes.Enriched()));

// начальное значение нашего агрегата – новый экземпляр HighScores
Initializer<HighScores> highScoresInitializer = HighScores::new;

// метод HighScores.add содержит схему агрегирования высших баллов
Aggregator<String, Enriched, HighScores> highScoresAdder = (key, value,
    aggregate) -> aggregate.add(value);

// агрегирование и материализация хранилища состояний
KTable<String, HighScores> highScores =
    grouped.aggregate( ❸
        highScoresInitializer,
        highScoresAdder);
```

❶ Чтение темы `score-events` в поток `KStream`.

❷ Изменение ключей сообщений, необходимое для совместного секционирования.

❸ Чтение темы `players` в таблицу `KTable`. Эта тема имеет большое пространство ключей, что позволяет шардировать состояние по нескольким экземплярам приложения. Кроме того, нам требуется синхронизированная по времени обработка для соединения `score-events -> players`. Именно этим обусловлен выбор варианта `KTable`.

❹ Чтение темы `products` в таблицу `GlobalKTable`. В данном случае пространство ключей невелико, а синхронизированная по времени обработка не требуется.

❺ Соединение потока `score-events` и таблицы `players`.

❻ Соединение обогащенного потока `score-events` с таблицей `products`.

❼ Группировка обогащенного потока, необходимая для последующего агрегирования.

❽ Агрегирование сгруппированного потока. Логическая схема агрегирования находится в классе `HighScores`.

Настроим конфигурацию нашего приложения и начнем потоковую передачу:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev");
props.put(StreamsConfig.BootstrapServersConfig, "localhost:9092");

KafkaStreams streams = new KafkaStreams(builder.build(), props);
streams.start();
```

Приложение готово к приему записей и подсчету рекордов, которые будут отображаться в таблице лидеров. Осталось сделать так, чтобы эти результаты могли увидеть внешние клиенты. Это последний шаг нашей топологии. Посмотрим, как раскрыть состояние приложения с помощью интерактивных запросов.

Интерактивные запросы

Одна из отличительных черт Kafka Streams — способность показывать состояние приложения как локально, так и куда-то вовне. Последнее дает возможность легко создавать управляемые событиями микросервисы с чрезвычайно низкой задержкой. Соответственно, отобразить результаты агрегирования мы сможем с помощью интерактивных запросов.

Для этого первым делом следует *материализовать* хранилище состояний. Сейчас я покажу вам, как это делается.

Материализованные хранилища

Как вы уже знаете, такие методы, как `aggregate`, `count`, `reduce`, используют хранилища состояний для управления внутренним состоянием. Но при внимательном рассмотрении процедуры агрегирования рекордов в примере 4.7 вы не обнаружите упоминаний хранилища. Дело в том, что фигурирующий там вариант метода `aggregate` использует *внутреннее хранилище состояний*, доступ к которому есть только у топологии узлов-обработчиков.

Дать произвольным запросам доступ на чтение из базового хранилища состояний позволяет один из перегружаемых методов, локально материализующих хранилище состояний. *Материализованные хранилища состояний* имеют имя и доступны для запросов не только изнутри топологии. Здесь нам пригодится класс `Materialized`. Его применение показано в примере 4.9.

Пример 4.9. Материализованное хранилище состояний с минимальной конфигурацией

```
KTable<String, HighScores> highScores = grouped.aggregate(
    highScoresInitializer,
    highScoresAdder,
    Materialized.<String, HighScores, KeyValueStore<Bytes, byte[]>> ❶
        as("leader-boards") ❷
        .withKeySerde(Serdes.String()) ❸
        .withValueSerde(JsonSerdes.HighScores()));
```

❶ В этот вариант метода `Materialized.as` входят три обобщенных типа:

- тип ключей хранилища (в данном случае `String`);
- тип значений хранилища (в данном случае `HighScores`);
- тип хранилища состояний (в данном случае мы будем использовать простое хранилище ключей и значений, представленное интерфейсом `KeyValueStore<Bytes, byte[]>`).

❷ В явном виде указываем имя хранилища, чтобы сделать его доступным для запросов извне топологии.

❸ Для настройки материализованного хранилища состояний можно использовать различные параметры, в том числе ключ и значение `Serdes`, о них мы поговорим в главе 6.

Материализованное хранилище состояний `leader-boards` готово предоставлять данные в ответ на запросы. Но сначала нужно получить ссылку на это хранилище в `Kafka Streams`.

Доступ на чтение из хранилища состояний

Для получения доступа на чтение из хранилища состояний требуются:

- имя хранилища состояний;
- тип хранилища состояний.

В примере 4.9 вы видели, что наше хранилище состояний называется `leader-boards`. С помощью фабричного класса `QueryableStoreTypes` сформируем доступную только для чтения оболочку этого хранилища. Класс содержит целый набор методов:

- `QueryableStoreTypes.keyValueStore();`
- `QueryableStoreTypes.timestampedKeyValueStore();`
- `QueryableStoreTypes.windowStore();`
- `QueryableStoreTypes.timestampedWindowStore();`
- `QueryableStoreTypes.sessionStore();`

Нам достаточно простого хранилища ключей и значений, поэтому воспользуемся методом `QueryableStoreType.keyValueStore()`. Зная имя и тип хранилища состояний, методом `KafkaStreams.store()` мы легко создадим его экземпляр, допускающий интерактивные запросы. Эта процедура показана в примере 4.10.

Пример 4.10. Создание экземпляра хранилища ключей и значений для работы с интерактивными запросами

```
ReadOnlyKeyValueStore<String, HighScores> stateStore = streams.store(
    StoreQueryParameters.fromNameAndType(
        "leader-boards",
        QueryableStoreTypes.keyValueStore()));
```

Теперь мы можем делать запросы к экземпляру нашего хранилища состояний. Но сначала рассмотрим типы доступных нам запросов.

Запросы к неоконным хранилищам «ключ — значение»

Каждый тип хранилища состояний поддерживает свои типы запросов. Например, оконные хранилища (такие как `ReadOnlyWindowStore`) позволяют искать ключи с использованием временных диапазонов, в то время как простые хранилища «ключ — значение» (такие как `ReadOnlyKeyValueStore`) поддерживают точечный поиск, сканирование диапазонов и подсчет запросов.

Оконные хранилища состояний будут обсуждаться в следующей главе, а пока посмотрим, какие виды запросов можно отправлять к нашему хранилищу `leader-boards`.

Определить типы запросов, которые доступны для конкретного хранилища состояний, можно, посмотрев на лежащий в его основе интерфейс. Из определения интерфейса видно, что простые хранилища «ключ — значение» поддерживают несколько типов запросов:

```
public interface ReadOnlyKeyValueStore<K, V> {

    V get(K key);

    KeyValueIterator<K, V> range(K from, K to);

    KeyValueIterator<K, V> all();

    long approximateNumEntries();
}
```

Рассмотрим каждый из них подробнее и начнем с точечного поиска (`get()`).

Точечный поиск

Точечный поиск, который является, пожалуй, наиболее распространенным типом запросов, — это запрос на поиск отдельного ключа. Для его выполнения применяется метод `get`. Например:

```
HighScores highScores = stateStore.get(key);
```


Имейте в виду, что в ответ на такой запрос возвращается или десериализованный экземпляр значения (в данном случае объект `HighScores`), или значение `null`, если ключ не обнаружен.

Сканирование диапазона

Простые хранилища «ключ — значение» также поддерживают запросы на сканирование диапазона. В ответ на запрос возвращается итератор с ключами из указанного диапазона. После завершения работы важно закрыть итератор, чтобы избежать утечки памяти.

Вот пример кода с запросом диапазона, перебором всех результатов и закрытием итератора:

```
KeyValueIterator<String, HighScores> range = stateStore.range(1, 7); ❶

while (range.hasNext()) {
    KeyValue<String, HighScores> next = range.next(); ❷

    String key = next.key;
    HighScores highScores = next.value; ❸

    // какие-то действия с объектом HighScores
}

range.close(); ❹
```

❶ Возвращение итератора, осуществляющего перебор всех ключей из выбранного диапазона.

❷ Получение следующего элемента итерации.

❸ Значение `HighScores` находится в свойстве `next.value`.

❹ Закрытие итератора. Альтернативный способ: использовать при получении итератора конструкцию `try-with-resources`.

Все записи

Запрос `all()`, как и запрос на сканирование диапазона, возвращает итератор с парами «ключ — значение». Он напоминает запрос `SELECT *` без фильтров и возвращает итератор со всеми записями хранилища состояний, а не только с записями из заданного диапазона ключей. Этот итератор после завершения работы тоже крайне важно закрыть, чтобы избежать утечек памяти. Вот пример запроса `all()`. Код перебора результатов и закрытия итератора опущен для краткости, так как он полностью повторяет код из предыдущего примера:

```
KeyValueIterator<String, HighScores> range = stateStore.all();
```

Количество записей

Последний тип запросов похож на запрос `COUNT(*)`. Он возвращает приблизительное количество записей в хранилище состояний.



Когда используется постоянное хранилище RocksDB, приблизительное значение возвращается, потому что вычисление точного количества — крайне ресурсоемкая операция, а в случае с кэшированным хранилищем RocksDB еще и очень сложная. Вот выдержка из FAQ по RocksDB (<https://oreil.ly/1r9GD>): «Получение точного количества ключей [в] базах данных на основах LSM-деревьев, таких как RocksDB, — нетривиальная задача из-за наличия дубликатов ключей и предназначенных к удалению записей (то есть записей, помеченных маркером удаления). Для получения точного количества ключей в такой базе требуется ее полное сжатие. Кроме того, наличие в базе данных RocksDB операторов слияния также уменьшает точность подсчета ключей».

В то же время для хранилищ, расположенных в памяти, возвращается точный результат.

Вот пример запроса такого типа к простому хранилищу «ключ — значение»:

```
long approxNumEntries = stateStore.approximateNumEntries();
```

Итак, вы получили представление о том, как делаются запросы, и пришло время применить эти знания на практике.

Локальные запросы

Каждый экземпляр приложения Kafka Streams может запрашивать собственное локальное состояние. Однако важно помнить, что, если не материализовать `GlobalKTable` или не запустить экземпляр приложения Kafka Streams¹, локальное состояние будет всего лишь частичным представлением состояния приложения в целом. Это обусловлено природой таблиц `KTable`.

К счастью для нас, в Kafka Streams есть дополнительные методы, упрощающие подключение распределенных хранилищ состояний и *выполнение удаленных запросов*. Посмотрим, как с помощью удаленных запросов узнать состояние нашего приложения.

¹ Последнее делать не рекомендуется. Запуск одного приложения Kafka Streams консолидирует все состояние приложения в один экземпляр, но эти приложения предназначены для распределенного запуска с целью обеспечения максимальной производительности и отказоустойчивости.

Удаленные запросы

Чтобы запросить полное состояние приложения, необходимо:

- узнать, какие экземпляры содержат фрагменты состояния приложения;
- добавить вызов удаленных процедур (RPC) или REST-*службу*, чтобы сделать локальное состояние видимым другим запущенным экземплярам приложения¹;
- добавить RPC или REST-*клиент* для запросов удаленных хранилищ состояний из запущенного экземпляра приложения.

При этом вы можете свободно выбирать серверные и клиентские компоненты для связи между экземплярами. Лично я для реализации REST-службы воспользовался фреймворком Javalin из-за его простого API. А для REST-клиента я взял простой в использовании OkHttp, разработанный компанией Square. Чтобы добавить в приложение эти зависимости, внесите в файл `build.gradle` следующие обновления:

```
dependencies {  
  
    // требуется для интерактивных запросов (сервер)  
    implementation 'io.javalin:javalin:3.12.0'  
  
    // требуется для интерактивных запросов (клиент)  
    implementation 'com.squareup.okhttp3:okhttp:4.9.0'  
  
    // остальные зависимости  
}
```

Теперь нужно найти способ получать информацию о том, какие экземпляры запущены в текущий момент и где они выполняются. Последнее достигается с помощью параметра `APPLICATION_SERVER_CONFIG`, в котором указывается имя хоста приложения и порт, на котором прослушивается сервис запросов:

```
Properties props = new Properties();  
  
props.put(StreamsConfig.APPLICATION_SERVER_CONFIG, "myapp:8080"); ❶  
  
// остальные свойства Kafka Streams опущены для краткости  
  
KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

❶ Настройка конечной точки. Эта конфигурация передается остальным экземплярам приложения через протокол группы потребителей Kafka. Важно указать пару «IP-адрес/порт», которой остальные экземпляры будут пользоваться для

¹ А при желании и другим клиентам, например людям.

связи с вашим приложением (`localhost` работать не будет, поскольку в зависимости от экземпляра он будет принадлежать разным IP-адресам).

Имейте в виду, что задание конфигурационного параметра `APPLICATION_SERVER_CONFIG` не сообщает Kafka Streams, что требуется начать прослушивание всех настроенных портов. Если быть точным, в Kafka Streams нет встроенной службы RPC. Сведения про хост/порт передаются в остальные экземпляры приложения Kafka Streams с помощью специальных методов API, о которых я расскажу чуть позже. А сначала настроим REST-службу на прослушивание соответствующего порта (в нашем случае 8080).

С точки зрения легкости поддержки кода REST-службу для нашей таблицы лидеров имеет смысл поместить в отдельный файл. Вот как она реализована:

```
class LeaderboardService {
    private final HostInfo hostInfo; ❶
    private final KafkaStreams streams; ❷

    LeaderboardService(HostInfo hostInfo, KafkaStreams streams) {
        this.hostInfo = hostInfo;
        this.streams = streams;
    }

    ReadOnlyKeyValueStore<String, HighScores> getStore() { ❸
        return streams.store(
            StoreQueryParameters.fromNameAndType(
                "leader-boards",
                QueryableStoreTypes.keyValueStore());
        )
    }

    void start() {
        Javalin app = Javalin.create().start(hostInfo.port()); ❹

        app.get("/leaderboard/:key", this::getKey); ❺
    }
}
```

❶ `HostInfo` — класс-оболочка в Kafka Streams, содержащий имя хоста и порт. Скоро я покажу, как он создается.

❷ Нам нужно отслеживать локальный экземпляр Kafka Streams. В следующем блоке кода к нему будут применяться методы API.

❸ Добавление метода, который извлекает хранилище состояний с результатами агрегирования. Аналогичный метод, предназначенный только для чтения контейнер хранилища состояний, вы видели в примере 4.10.

❹ Запуск на настроенном порте веб-службы на базе Javalin.

❺ Фреймворк Javalin позволяет легко добавлять конечные точки. Адресу URL просто сопоставляется метод, который мы вскоре реализуем. Параметры адре-

са, которые указываются двоеточием в начале (например, `:key`), позволяют создавать динамические конечные точки, что идеально подходит для точечных поисковых запросов.

Теперь нужно реализовать конечную точку `/leaderboard/:key`, которая будет показывать самые высокие баллы для заданного ключа (в данном случае в роли ключа выступает идентификатор продукта). Еще раз напомним, что для получения из хранилища состояний дискретных значений применяется точечный поиск. Вот пример очень простой реализации:

```
void getKey(Context ctx) {  
    String productId = ctx.pathParam("key");  
    HighScores highScores = getStore().get(productId); ❶  
    ctx.json(highScores.toList()); ❷  
}
```

❶ Точечный поиск извлекает из локального хранилища состояний дискретное значение.

❷ Примечание: метод `toList()` есть в исходном коде.

К сожалению, этого мало. Представим, что у нас запущено два экземпляра приложения Kafka Streams. Возможность получения ответа на запрос зависит от того, к *какому* экземпляру и *когда* он был отправлен (перевыравнивание потребителей может вызывать смещение состояния). Причины такого поведения демонстрирует рис. 4.7.

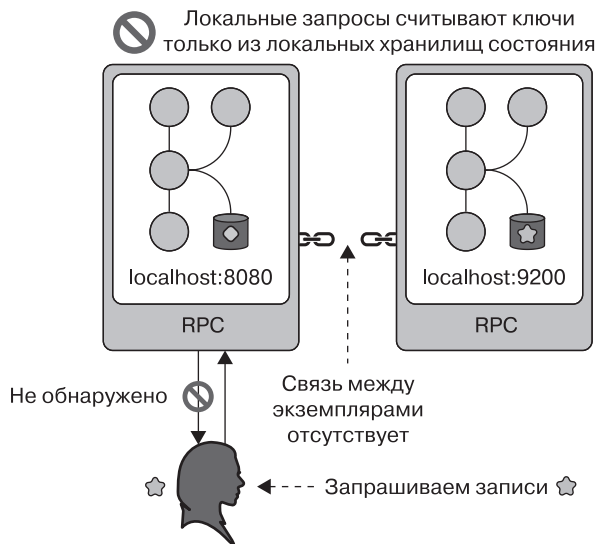


Рис. 4.7. Когда состояние разделено между несколькими экземплярами приложения, локальных запросов недостаточно

К счастью, в Kafka Streams есть метод `queryMetadataForKey`¹, позволяющий найти локальный или удаленный экземпляр приложения с конкретным ключом. Пример 4.11 демонстрирует усовершенствованную реализацию нашего метода `getKey`.

Пример 4.11. Обновленная реализация метода `getKey`, использующая удаленные запросы для извлечения данных из разных экземпляров приложения

```
void getKey(Context ctx) {

    String productId = ctx.pathParam("key");

    KeyQueryMetadata metadata =
        streams.queryMetadataForKey(
            "leader-boards", productId, Serdes.String().serializer()); ❶

    if (hostInfo.equals(metadata.activeHost())) {
        HighScores highScores = getStore().get(productId); ❷

        if (highScores == null) { ❸
            // игра не обнаружена
            ctx.status(404);
            return;
        }

        // игра обнаружена, поэтому возвращаем рекордный счет
        ctx.json(highScores.toList()); ❹
        return;
    }

    // удаленный экземпляр обладает запрошенным ключом
    String remoteHost = metadata.activeHost().host();
    int remotePort = metadata.activeHost().port();
    String url = String.format(
        "http://%s:%d/leaderboard/%s",
        remoteHost, remotePort, productId); ❺

    OkHttpClient client = new OkHttpClient();
    Request request = new Request.Builder().url(url).build();

    try (Response response = client.newCall(request).execute()) { ❻
        ctx.result(response.body().string());
    } catch (Exception e) {
        ctx.status(500);
    }
}
```

❶ Метод `queryMetadataForKey` позволяет найти хост, на котором присутствует запрошенный ключ.

¹ Он пришел на замену методу `metadataForKey`, который широко использовался в версиях < 2.5, но уже официально объявлен устаревшим.

❷ Если ключ есть у локального экземпляра, запрос отправляется к локальному хранилищу состояний.

❸ Метод `queryMetadataForKey` не проверяет, существует ли ключ. Он использует заданный по умолчанию разделитель потока¹, чтобы определить, где *мог бы находиться ключ, если бы он существовал*. Поэтому мы проводим проверку на значение `null` (именно оно возвращается, если ключ не найден) и в случае ее положительного результата возвращаем ответ `404`.

❹ Возвращение отформатированного ответа с рекордными очками.

❺ На этом этапе понятно, что если ключ вообще существует, то он находится на удаленном хосте. Поэтому создается URL-адрес на базе метаданных, которые включают в себя хост и порт экземпляра Kafka Streams, потенциально содержащего указанный ключ.

❻ Отправка запроса и возвращение результата в случае успеха.

Схема соединения распределенных хранилищ состояния с помощью комбинации процедуры обнаружения экземпляров и RPC/REST-служб показана на рис. 4.8.

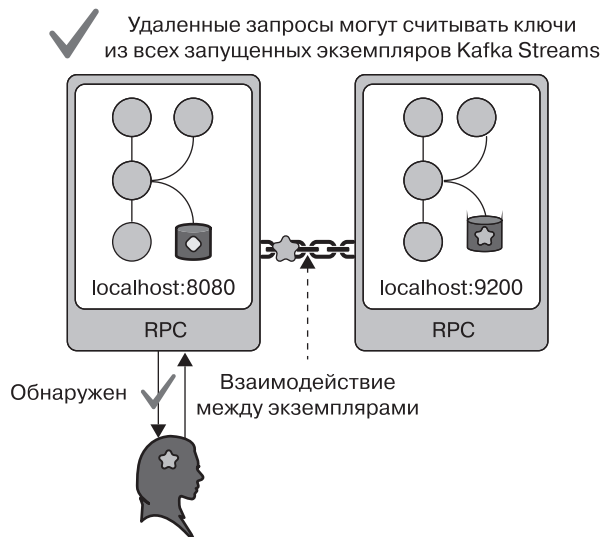


Рис. 4.8. Удаленные запросы позволяют интересоваться состоянием других запущенных экземпляров приложения

¹ Существует перегруженная версия метода `queryMetadataForKey`, которая также принимает пользовательскую версию разделителя `StreamPartitioner`.

А как быть с запросами других типов? Например, что делать, если требуется подсчитать количество записей во всех распределенных хранилищах состояний? Метод `QueryMetadataForKey` в этом случае не сработает, так как он ищет только экземпляры, хранилище которых использует заданный ключ. Тут на помощь приходит другой метод Kafka Streams — `allMetadataForStore`. Он возвращает конечную точку для каждого запущенного приложения Kafka Streams с одинаковым идентификатором *и* хотя бы одним активным разделом для указанного имени хранилища.

Добавим в службу, отображающую количество рекордов во всех запущенных экземплярах приложения, новую конечную точку:

```
app.get("/leaderboard/count", this::getCount);
```

Теперь для получения общего количества записей в удаленных хранилищах состояний применим упоминавшийся выше метод `getCount`, работающий на базе метода `allMetadataForStore`:

```
void getCount(Context ctx) {
    long count = getStore().approximateNumEntries(); ❶

    for (StreamsMetadata metadata : streams.allMetadataForStore("leader-
boards")) {
        if (!hostInfo.equals(metadata.hostInfo())) {
            continue; ❸
        }
        count += fetchCountFromRemoteInstance( ❹
            metadata.hostInfo().host(),
            metadata.hostInfo().port());
    }

    ctx.json(count);
}
```

❶ В качестве начального значения счетчику присваивается количество записей в локальном хранилище состояний.

❷ Метод `allMetadataForStore` возвращает пары «хост/порт» для каждого экземпляра Kafka Streams, содержащего фрагмент запрашиваемого состояния.

❸ Если метаданные предназначены для текущего хоста, цикл продолжается, поскольку информацию о количестве записей в локальном хранилище состояний мы уже получили.

❹ Если метаданные не относятся к локальному экземпляру, получаем счетчик из удаленного экземпляра. Детали получения счетчика `CountFromRemoteInstance` здесь опущены, так как в данном случае процесс похож на то, что вы уже видели в примере 4.11, где создавался экземпляр REST-клиента и отправлялся запрос

на удаление экземпляра приложения. Если вас интересуют подробности реализации, посмотрите исходный код к этой главе.

Последний шаг создания топологии таблицы лидеров завершен. Можно запустить наше приложение, сгенерировать для него какие-то данные и посмотреть, как работают запросы.

Данные-заполнители для каждой из тем-источников показаны в примере 4.12.



Для тем, снабженных ключами (players и products), ключ записи имеет формат <ключ>|<значение>. Записи для заполнения темы score-events имеют формат <значение>.

Пример 4.12. Фиктивные записи для заполнения тем-источников

```
# players
1|{"id": 1, "name": "Elyse"}
2|{"id": 2, "name": "Mitch"}
3|{"id": 3, "name": "Isabelle"}
4|{"id": 4, "name": "Sammy"}

# products
1|{"id": 1, "name": "Super Smash Bros"}
6|{"id": 6, "name": "Mario Kart"}

# score-events
{"score": 1000, "product_id": 1, "player_id": 1}
{"score": 2000, "product_id": 1, "player_id": 2}
{"score": 4000, "product_id": 1, "player_id": 3}
{"score": 500, "product_id": 1, "player_id": 4}
{"score": 800, "product_id": 6, "player_id": 1}
{"score": 2500, "product_id": 6, "player_id": 2}
{"score": 9000.0, "product_id": 6, "player_id": 3}
{"score": 1200.0, "product_id": 6, "player_id": 4}
```

Если добавить эти данные в соответствующие темы, а затем начать отправку запросов к службе `leaderboard`, вы убедитесь, что наше приложение Kafka Streams не только обработало информацию о рекордах, но и предоставляет результаты наших операций с сохранением состояния. Вот пример ответа на интерактивный запрос:

```
$ curl -s localhost:7000/leaderboard/1 | jq '.'
```

```
[
  {
    "playerId": 3,
    "productId": 1,
    "playerName": "Isabelle",
    "gameName": "Super Smash Bros",
```

```
        "score": 4000
    },
    {
        "playerId": 2,
        "productId": 1,
        "playerName": "Mitch",
        "gameName": "Super Smash Bros",
        "score": 2000
    },
    {
        "playerId": 1,
        "productId": 1,
        "playerName": "Elyse",
        "gameName": "Super Smash Bros",
        "score": 1000
    }
]
```

Заключение

В этой главе вы узнали, как Kafka Streams фиксирует информацию о потребляемых событиях и как использовать эту информацию (состояние) для более сложных задач обработки потоков.

- Как выполнить соединение `KStream-Ktable`.
- Как переназначить ключи сообщений, чтобы выполнить процедуру совместного секционирования, без которой невозможны операции соединения определенных типов.
- Как выполнить соединение `KStream-GlobalKTable`.
- Как сгруппировать записи в промежуточные представления (`KGroupedStream`, `KGroupedTable`) для подготовки данных к агрегированию.
- Как осуществить агрегирование потоков и таблиц.
- Как с помощью интерактивных запросов, как локальных, так и удаленных, узнать состояние приложения.

Пришло время обсудить следующий аспект программирования с сохранением состояния, который касается не только произошедших в приложении событий, но и времени, когда они произошли. В обработке с сохранением состояния время играет ключевую роль, поэтому так важно познакомиться с различными представлениями времени, а также с основанными на нем абстракциями из библиотеки Kafka Streams.

Окна и время

Время — настолько важное понятие, что мы измеряем нашу жизнь его течением. Каждый год в определенный день полдюжины людей окружают меня и поют поздравление с днем рождения. И когда последняя нота затихнет в воздухе, к ногам этой таинственной силы, которую мы и называем временем, преподносится торт. Мне нравится думать, что торт для меня, но в действительности он преподносится времени.

Время не только сложным узором вплетено в физический мир, но и пронизывает наши потоки событий. Чтобы раскрыть всю мощь Kafka Streams, нужно понимать взаимосвязь между событиями и временем. В этой главе мы подробно исследуем эту взаимосвязь и познакомимся с тем, что называется *окнами*. Окна позволяют группировать события в привязке к явным интервалам времени и могут использоваться для создания более сложных соединений и агрегатов (которые мы впервые рассмотрели в предыдущей главе).

К концу этой главы вы будете знать и понимать:

- как различать время события, время загрузки и время обработки;
- как создать свой экстрактор отметок времени для связывания событий с определенной отметкой и семантикой времени;
- как время управляет потоком данных через Kafka Streams;
- какие типы окон поддерживаются в Kafka Streams;
- как выполняются оконные соединения;
- как производится оконное агрегирование;
- какие существуют стратегии для работы с поздними и внеочередными событиями;
- как использовать оператор `suppress` для обработки результатов оконных операций;
- как запрашивать оконные хранилища ключей-значений.

Так же как в предыдущих главах, вы будете знакомиться с этими понятиями на учебном примере. А теперь кратко рассмотрим приложение, которое мы создадим в этой главе.

Приложение для контроля состояния пациентов

Некоторые из наиболее важных приложений потоковой обработки, учитывающих течение времени, относятся к области медицины. Системы контроля состояния пациентов способны производить сотни измерений в секунду, и быстрая обработка/реагирование на эти данные важны для лечения некоторых видов заболеваний. Вот почему в Детской больнице города Атланта используется Kafka Streams и `ksqlDB`. Они помогают в масштабе реального времени прогнозировать необходимость в ближайшем будущем хирургического вмешательства для детей с черепно-мозговой травмой¹.

Вдохновившись этим вариантом применения, мы рассмотрим несколько концепций потоковой обработки данных, связанных с фактором времени, и создадим приложение мониторинга жизненно важных функций пациента. Вместо мониторинга черепно-мозговой травмы мы попытаемся выявить наличие заболевания, называемого *синдромом системной воспалительной реакции* (Systemic Inflammatory Response Syndrome, SIRS). По словам Бриджит Кадри (Bridgette Kadri), ассистента врача в Медицинском университете Южной Каролины, есть несколько жизненно важных показателей, включая температуру тела, артериальное давление и частоту сердечных сокращений, которые могут служить индикаторами SIRS. В этом примере мы рассмотрим два из этих параметров: температуру тела и частоту сердечных сокращений. Когда оба этих жизненно важных показателя достигают определенных пороговых значений (частота сердечных сокращений ≥ 100 ударов в минуту и температура тела $\geq 38^\circ\text{C}$), приложение будет посылать запись в тему *предупреждений*, чтобы своевременно уведомить медицинский персонал².

¹ Для этого измеряется внутричерепное давление, а результаты измерений агрегируются и передаются прогностической модели. Когда модель определяет, что давление достигнет опасного уровня в течение следующих 30 минут, медицинские работники уведомляются о необходимости принятия соответствующих мер. Все это стало возможным благодаря потоковой обработке с учетом фактора времени. Более подробно об этом варианте прикладного применения можно узнать из интервью Тима Берглунда (Tim Berglund) с Рамешем Шрингери (Ramesh Sringeri), доступного по адресу <https://oreil.ly/GHbVd>.

² На самом деле мы не будем реализовывать приложение для контроля за состоянием пациента, потому что для демонстрации функций Kafka Streams, ориентированных на время, это не требуется.

Рассмотрим архитектуру нашего приложения контроля состояния пациентов. На рис. 5.1 показана структура, которую мы реализуем в этой главе, а за ним приводится дополнительная информация о каждом шаге.

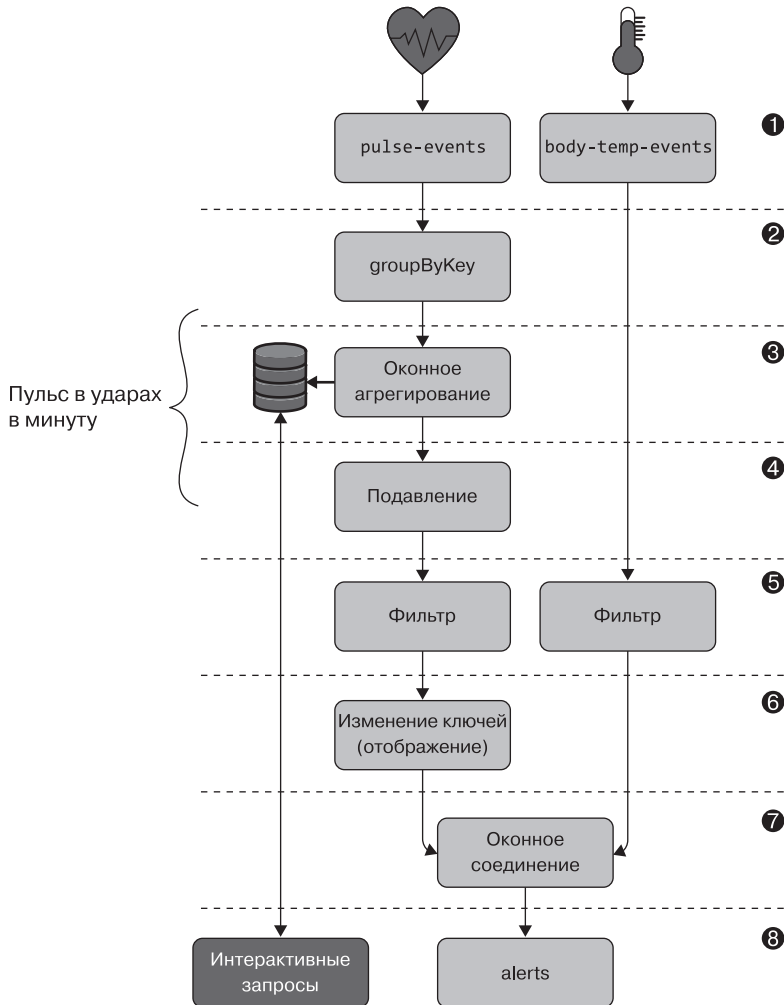


Рис. 5.1. Структура приложения контроля за состоянием пациента

- ❶ Наш кластер Kafka содержит две темы для хранения жизненно важных показателей состояния пациента.
- Тема `pulse-events` заполняется измерениями, поступающими с датчика сердцебиения. Каждый раз, когда датчик улавливает сердцебиение пациента,

он добавляет запись в эту тему. Записи снабжаются ключом с идентификатором пациента.

- Тема `body-temp-events` заполняется измерениями, поступающими с беспроводного датчика температуры тела. После каждого измерения температуры тела пациента в эту тему добавляется запись. Эти записи тоже снабжаются ключом с идентификатором пациента.
- ❷ Чтобы обнаружить повышенную частоту сердцебиения, необходимо преобразовать исходные события пульса в частоту сердечных сокращений, измеряемую в *ударах в минуту*. Как было показано в предыдущей главе, для этого следует сначала сгруппировать записи, чтобы выполнить предварительное условие Kafka Streams, необходимое для агрегирования.
- ❸ Для преобразования событий сердцебиения в частоту сердечных сокращений предполагается использовать оконное агрегирование. Поскольку за единицу измерения принято количество ударов в минуту, размер окна будет равен 60 секундам.
- ❹ Оператор подавления `suppress` будет использоваться только для окончательного вычисления количества ударов в минуту в окне. Мы увидим, зачем так делать, когда будем обсуждать упомянутый оператор далее в этой главе.
- ❺ Для определения симптомов инфекции предполагается фильтровать все показатели жизненно важных функций, выходящие за пределы нормальных значений (частота сердечных сокращений ≥ 100 ударов в минуту, температура тела $\geq 38^\circ\text{C}$).
- ❻ Как вскоре будет показано, оконное агрегирование меняет ключ записи. Поэтому необходимо повторно связать записи сердечного ритма с идентификатором пациента, чтобы выполнить требования к совместному секционированию для соединения записей.
- ❼ К двум потокам жизненно важных показателей применяется оконное соединение. Поскольку соединение производится *после* фильтрации повышенной частоты пульса и температуры тела, каждая запись, полученная в результате соединения, может служить признаком для отправки предупреждения о SIRS.
- ❽ Наконец, результаты оконного агрегирования сердечного ритма будут доступны для интерактивных запросов. Кроме того, данные из потока, полученного в результате соединения, будут помещаться в тему `alerts`.

Кратко пробежимся по настройке проекта, чтобы вы могли затем спокойно заниматься изучением примеров.

Настройка проекта

Код примеров для этой главы можно найти по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>.

Если вы решите опробовать каждый этап создания проекта, то клонируйте репозиторий и перейдите в каталог, содержащий инструкции к текущей главе. Сделать это можно следующими командами:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-05/patient-monitoring
```

Собрать проект можно командой:

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

Теперь, завершив настройку, приступим к реализации нашего приложения мониторинга пациентов.

Модели данных

Как всегда, сначала определимся с моделями данных. Поскольку каждое измерение показателей жизнедеятельности связано с отметкой времени, сначала создадим простой интерфейс для реализации классов данных. Этот интерфейс позволит единообразно извлекать отметки времени из записей и пригодится позже, когда мы реализуем экстрактор отметок времени. Вот определение этого интерфейса:

```
public interface Vital {
    public String getTimestamp();
}
```

Далее представлены классы данных. Имейте в виду, что методы доступа (включая метод интерфейса `getTimestamp`) были опущены для краткости.

Тема Kafka	Пример записи	Класс данных
pulse-events	<pre>{ "timestamp": "2020-11-05T09:02:00.000Z" }</pre>	<pre>public class Pulse implements Vital { private String timestamp; }</pre>
body-temp-events	<pre>{ "timestamp": "2020-11-04T09:02:06.500Z", "temperature": 101.2, "unit": "F" }</pre>	<pre>public class BodyTemp implements Vital { private String timestamp; private Double temperature; private String unit; }</pre>

Теперь, получив представление о том, как выглядят исходные данные, мы почти готовы зарегистрировать входные потоки. Однако это приложение предъявляет особые требования к времени, а до сих пор мы почти ничего не говорили о связи записей с его отметками. Поэтому прежде чем регистрировать входные потоки, рассмотрим различные семантики времени в Kafka Streams.

Семантики времени

В Kafka Streams существует несколько понятий времени, и выбор правильной семантики имеет большое значение для операций, основанных на времени, включая оконные соединения и агрегирование. В этом разделе мы поговорим о различных понятиях времени в Kafka Streams, начав с простых определений.

Время события

Время создания события в источнике. Это время может встраиваться в данные события или устанавливаться непосредственно с помощью клиента-производителя Kafka, начиная с версии 0.10.0.

Время приема

Время добавления события в тему брокером Kafka. Это время всегда больше времени события.

Время обработки

Время обработки события приложением Kafka Streams. Это время всегда больше времени события и времени приема. Время обработки не такое постоянное, как время события, потому что повторная обработка тех же данных (например, для исправления ошибок) сгенерирует новую отметку времени обработки и, следовательно, может привести к недетерминированному разделению потока на окна.

Схема на рис. 5.2 иллюстрирует физическое проявление этих понятий времени в потоке событий.

Время события, вероятно, является наиболее интуитивным понятием времени, потому что описывает, когда событие фактически произошло. Например, если датчик сердцебиения регистрирует пульс в 9:02, то время события — 9:02.

Время события обычно встраивается в данные, описывающие событие, как показано ниже:

```
{
  "timestamp": "2020-11-12T09:02:00.000Z", ❶
  "sensor": "smart-pulse"
}
```

❶ Встроенная отметка времени, которую придется извлекать при необходимости.



Рис. 5.2. Разные семантики времени в Kafka Streams на примере измерения частоты сердцебиения

Как вариант, производители Kafka позволяют переопределять отметку времени по умолчанию в каждой записи, что также можно использовать для реализации семантики времени события. Однако разработчикам систем, использующих этот метод привязки отметок времени, важно знать о двух параметрах конфигурации Kafka (одна конфигурация на уровне брокера, и одна — на уровне темы), чтобы случайно не получить семантику времени приема. Вот эти параметры:

- `log.message.timestamp.type` (на уровне брокера);
- `message.timestamp.type` (на уровне темы).

Они могут принимать два возможных значения: `CreateTime` и `LogAppendTime`. Кроме того, параметр конфигурации на уровне темы имеет приоритет над параметром уровня брокера. Если параметр `message.timestamp.type` на уровне темы настроен значением `LogAppendTime`¹, то отметка времени, добавленная

¹ Напрямую через конфигурацию уровня темы или косвенно через конфигурацию уровня брокера без переопределения в конфигурации уровня темы.

в сообщение производителем, будет затираться локальным системным временем брокера при добавлении записи в тему (поэтому отметки времени фактически будут иметь семантику времени приема, независимо от ваших намерений). Чтобы получить семантику времени события и сохранить в записях отметки времени производителя, используйте значение `CreateTime`.

Преимущество семантики времени события заключается в том, что это время более значимо для самого события и, следовательно, понятнее для пользователей. Время события также позволяет сделать детерминированными операции, зависящие от времени (например, при повторной обработке данных). Для нашего случая время обработки не подходит. Обычно время обработки используется, когда операции, применяемые к данным, не учитывают время, или время обработки события более значимо для семантики приложения, чем время возникновения события, или нет возможности связать событие с определенным моментом времени по какой-либо причине. Интересно отметить, что последняя проблема невозможности связать запись с временем события иногда решается привязкой времени приема. В системах, где задержка между созданием события и его добавлением в тему невелика, время приема вполне можно использовать для аппроксимации времени события, поэтому такой прием может быть жизнеспособной альтернативой в случаях, когда нет возможности использовать время события¹.

Теперь, познакоившись с разными понятиями времени в Kafka Streams, нам нужно выбрать наиболее подходящую семантику. А как это сделать, узнаем в следующем разделе.

Экстракторы отметок времени

Экстракторы отметок времени (timestamp extractors) в Kafka Streams отвечают за связывание записей с отметками времени, которые затем можно использовать в операциях, зависящих от времени, таких как оконное соединение и оконное агрегирование. Каждая реализация экстрактора отметок времени должна соответствовать следующему интерфейсу:

```
public interface TimestampExtractor {  
    long extract(  
        ConsumerRecord<Object, Object> record, ❶  
        long partitionTime ❷  
    );  
}
```

¹ Маттиас Джей Сакс (Matthias J. Sax) опубликовал отличную презентацию (<https://oreil.ly/MRiCu>), в которой обсуждаются этот и другие вопросы, затронутые в данной главе.

- ❶ Текущая запись, обработанная потребителем.
- ❷ Kafka Streams запоминает самую последнюю отметку времени, наблюдавшуюся в каждом разделе, откуда потребляются записи, и передает эту отметку времени методу `extract` в параметре `partitionTime`.

Второй параметр, `partitionTime`, представляет наибольший интерес, потому что его можно использовать как запасной вариант, если нет возможности извлечь отметку времени. Мы поговорим об этом чуть ниже, а пока посмотрим, какие экстракторы отметок времени уже имеются в Kafka Streams.

Встроенные экстракторы отметок времени

`FailOnInvalidTimestamp` — экстрактор отметок времени по умолчанию. Извлекает из записи потребителя отметку времени, которая представляет время события (если в `message.timestamp.type` установлено значение `CreateTime`) или время приема (если в `message.timestamp.type` установлено значение `LogAppendTime`). Этот экстрактор генерирует исключение `StreamsException`, если отметка времени недействительна. Отметка времени считается недействительной, если она имеет отрицательное значение (это возможно, если запись была создана с использованием формата сообщения старше 0.10.0). На момент написания этой книги прошло более четырех лет от даты выпуска версии 0.10.0, поэтому отрицательные/недействительные отметки времени встречаются все реже.

Для достижения семантики времени события можно также использовать экстрактор `LogAndSkipOnInvalidTimestamp`. Но, в отличие от `FailOnInvalidTimestamp`, обнаружив недопустимую отметку времени, этот экстрактор просто выводит предупреждение. Это позволяет Kafka Streams продолжить обработку, просто пропустив запись с недопустимой отметкой времени.

На случай, когда может понадобиться семантика времени обработки, существует еще один встроенный экстрактор. Как можно видеть в следующем фрагменте, `WallclockTimestampExtractor` просто возвращает локальное системное время:

```
public class WallclockTimestampExtractor implements TimestampExtractor {  
  
    @Override  
    public long extract(  
        final ConsumerRecord<Object, Object> record,  
        final long partitionTime  
    ) {  
        return System.currentTimeMillis(); ❶  
    }  
}
```

❶ `WallclockTimestampExtractor` — один из экстракторов отметок времени, встроенных в Kafka Streams; он просто возвращает текущее системное время.

Независимо от используемого экстрактора отметок времени, вы можете переопределить экстрактор по умолчанию, установив свойство `DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG`, как показано в примере 5.1.

Пример 5.1. Переопределение экстрактора отметок времени по умолчанию в Kafka Streams

```
Properties props = new Properties();
props.put(❶
    StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
    WallclockTimestampExtractor.class
);

// ... другие параметры конфигурации

KafkaStreams streams = new KafkaStreams(builder.build(), props);
```

❶ Переопределение экстрактора по умолчанию.

Когда в приложении Kafka Streams (таком как наше приложение контроля состояния пациента) используются окна, применение семантики времени обработки может непреднамеренно давать побочные эффекты. Например, мы в нашем приложении предполагаем фиксировать количество сердцебиений в течение одной минуты. Если для оконного агрегирования использовать семантику времени обработки (например, через `WallclockTimestampExtractor`), то границы окна вообще будут представлять не период измерения пульса, а время, в течение которого наше приложение наблюдало события пульса. Если приложение запоздает хотя бы на несколько секунд, событие выйдет за пределы предполагаемого окна и тем самым повлияет на наши ожидания (например, на способность обнаруживать повышенную частоту пульса).



Когда отметка времени извлекается и затем связывается с записью, запись считается *проштампованной*.

После знакомства со встроенными экстракторами отметок времени становится ясно, что нам нужен другой, свой экстрактор, потому что время события включено в данные наших событий (пульса и замеров температуры тела). В следующем разделе мы посмотрим, как можно создать свой экстрактор отметок времени.

Свои собственные экстракторы отметок времени

Очень часто, когда прикладной логике нужна семантика времени события, а отметка времени встроена в данные записи, создается свой экстрактор отметок времени. В следующем фрагменте кода показано определение такого экстрактора отметок времени, который мы будем использовать для оценки показателей жизнедеятельности пациента. Как упоминалось выше, наш собственный экстрактор отметок времени реализует интерфейс `TimestampExtractor`, включенный в `Kafka Streams`:

```
public class VitalTimestampExtractor implements TimestampExtractor {

    @Override
    public long extract(ConsumerRecord<Object, Object> record, long partitionTime)
    {
        Vital measurement = (Vital) record.value(); ❶
        if (measurement != null && measurement.getTimestamp() != null) { ❷
            String timestamp = measurement.getTimestamp(); ❸
            return Instant.parse(timestamp).toEpochMilli(); ❹
        }
        return partitionTime; ❺
    }
}
```

❶ Объект записи приводится к типу `Vital`. Как раз здесь и пригодится наш интерфейс: он позволяет единообразно извлекать отметки времени из записей `Pulse` и `BodyTemp`.

❷ Проверяется наличие действительной записи и отметки времени в ней прежде, чем попытаться извлечь ее.

❸ Извлечение отметки времени из записи.

❹ Метод `TimestampExtractor.extract` должен возвращать отметку времени в миллисекундах. Поэтому здесь извлеченное значение преобразуется в миллисекунды.

❺ Если получить отметку времени по какой-либо причине не удалось, можно вернуть время раздела как примерную оценку момента, когда произошло событие.

Один из аспектов, который необходимо учитывать при использовании экстракторов отметок времени, — это необходимость принимать решение, как обрабатывать записи, не имеющие действительных отметок времени. Наиболее распространены три варианта:

- сгенерировать исключение и остановить обработку, дав разработчикам возможность исправить ошибку;

- вернуть время раздела;
- вернуть отрицательную отметку времени, что позволит Kafka Streams пропустить запись и продолжить обработку.

В нашей реализации `VitalTimestampExtractor` мы решили вернуть время раздела, то есть самое большое значение отметки времени, наблюдавшееся в текущем разделе.

Теперь, создав свой экстрактор отметок времени, зарегистрируем наши входные потоки.

Регистрация потоков со своими экстракторами отметок времени

Регистрация набора входных потоков уже знакома вам, но на этот раз мы передадим дополнительный параметр `Consumed` с нашей реализацией экстрактора отметок времени (`VitalTimestampExtractor`). В примере 5.2 показано, как зарегистрировать два входных потока со своими экстракторами отметок времени и тем самым сделать первый шаг в реализации топологии нашего процессора (см. рис. 5.1).

Пример 5.2. Переопределение экстрактора отметок времени для входных потоков

```
StreamsBuilder builder = new StreamsBuilder(); ❶

Consumed<String, Pulse> pulseConsumerOptions =
    Consumed.with(Serdes.String(), JsonSerdes.Pulse())
        .withTimestampExtractor(new VitalTimestampExtractor()); ❷

KStream<String, Pulse> pulseEvents =
    builder.stream("pulse-events", pulseConsumerOptions); ❸

Consumed<String, BodyTemp> bodyTempConsumerOptions =
    Consumed.with(Serdes.String(), JsonSerdes.BodyTemp())
        .withTimestampExtractor(new VitalTimestampExtractor()); ❹

KStream<String, BodyTemp> tempEvents =
    builder.stream("body-temp-events", bodyTempConsumerOptions); ❺
```

❶ Как обычно, для построения топологии процессора используется `StreamsBuilder`.

- ❷ Вызовом `Consumed.withTimestampExtractor` мы сообщаем Kafka Streams, что для извлечения отметок времени из показателей жизнедеятельности должен использоваться наш экстрактор (`VitalTimestampExtractor`).
- ❸ Регистрация потока для захвата событий сердцебиения.
- ❹ Назначается наш собственный экстрактор отметок времени из показателей температуры тела.
- ❺ Регистрация потока для захвата событий измерения температуры тела.

Переопределить экстрактор отметок времени по умолчанию можно также другим способом, показанным в примере 5.1. Оба способа хороши, но пока мы будем использовать способ назначения экстрактора для каждого входного потока в отдельности. Теперь, зарегистрировав входные потоки, перейдем ко второму и третьему шагам реализации топологии нашего процессора (см. рис. 5.1): к группировке и оконной обработке потока `pulse-events`.

Оконная обработка потоков

Тема `pulse-events` получает новую запись всякий раз, когда фиксируется сердцебиение пациента. Однако нас интересует частота сердечных сокращений пациента, которая измеряется количеством ударов в минуту. Мы знаем, что для подсчета количества ударов сердца можно использовать оператор `count`, но подсчитываться должны только записи, попадающие в каждое 60-секундное окно. Для этого используется прием *оконного агрегирования*. Оконная обработка — это метод группировки записей в разные временные подгруппы с целью агрегирования и соединения. Kafka Streams поддерживает несколько видов оконной обработки, поэтому далее мы рассмотрим их по очереди, чтобы определить, какая реализация лучше всего подходит для системы мониторинга пациентов.

Типы окон

Окна используются для группировки записей с *временной близостью*. Понятие «временная близость» может иметь разные смысловые значения, в зависимости от используемой семантики времени. Например, для семантики времени события это понятие может означать «события, произошедшие примерно в одно и то же время», а для семантики времени обработки: «события, обработанные примерно в одно и то же время». В большинстве случаев «примерно в одно и то же время» определяется

размером окна (например, пять минут, один час и т. д.), с другой стороны, «окна сеансов», которые мы обсудим ниже, определяются периодами активности.

В Kafka Streams поддерживаются четыре типа окон. Далее мы обсудим характеристики каждого типа, а затем используем эти свойства для создания дерева решений, чтобы выбрать тип окон для нашего проекта. Полученное дерево решений вы сможете также использовать в своих приложениях.

Переворачивающиеся окна

Переворачивающиеся окна (tumbling windows) — это окна фиксированного размера, которые никогда не перекрываются. Они определяются единственным свойством — *размером окна* (в миллисекундах) и имеют предсказуемые временные диапазоны, потому что всегда выравниваются по началу эпохи¹. Ниже показано, как создать переворачивающееся окно в Kafka Streams:

```
TimeWindows tumblingWindow =
    TimeWindows.of(Duration.ofSeconds(5)); ❶
```

❶ Размер окна — пять секунд.

Как можно видеть на рис. 5.3, переворачивающееся окно легко представить: не нужно беспокоиться о перекрывающихся границах окон или о том, что одна и та же запись может оказаться в нескольких окнах.

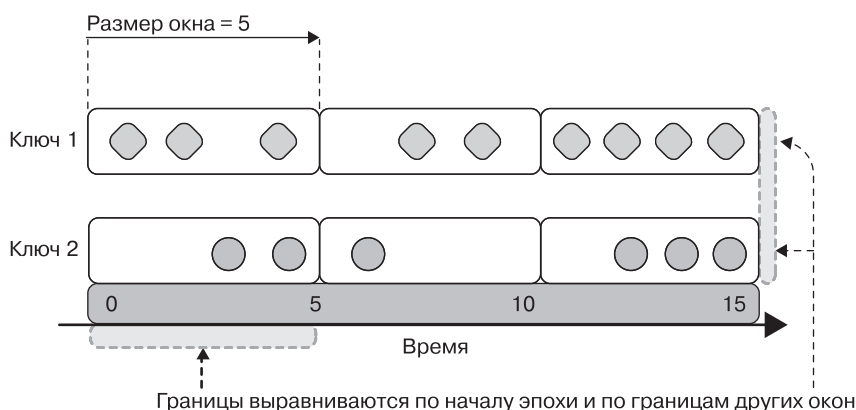


Рис. 5.3. Переворачивающееся окно

¹ Выдержка из документации по Java: «Выравнивание по началу эпохи означает, что первое окно начинается с нулевой отметки времени». Другими словами, размер окна 5000 будет иметь границы 0–5000, 5000–10 000 и т. д. Обратите внимание, что время начала включено в окно, а время окончания — нет.

Шагающие окна

Шагающие окна (hopping windows) — это окна фиксированного размера, которые могут перекрываться¹. При настройке шагающего окна необходимо указать *размер окна* и *шаг* (интервал перемещения окна вперед). Когда шаг меньше размера окна, как показано на рис. 5.4, окна будут перекрываться и некоторые записи будут включаться в несколько окон. Кроме того, шагающие окна имеют предсказуемые временные диапазоны, поскольку выравниваются по началу эпохи, включают время начала и исключают время окончания. В следующем фрагменте показано, как создать простое шагающее окно с помощью Kafka Streams:

```
TimeWindows hoppingWindow = TimeWindows
    .of(Duration.ofSeconds(5)) ❶
    .advanceBy(Duration.ofSeconds(4)); ❷
```

❶ Размер окна — пять секунд.

❷ Шаг (интервал) смещения окна — четыре секунды.

На рис. 5.4 показано, как можно визуальное представить шагающее окно.

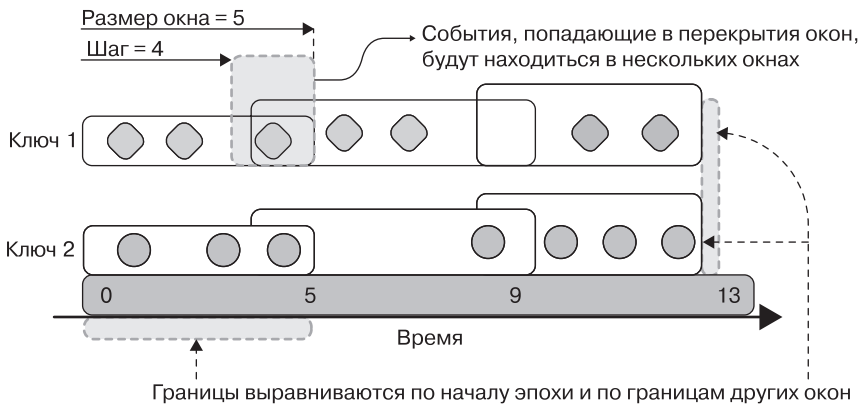


Рис. 5.4. Шагающее окно

Окна сеансов

Окна сеансов (session windows) — это окна переменных размеров, определяемых периодами активности, за которыми следуют промежутки бездействия. Окна сеансов определяются одним параметром — *интервалом бездействия*. Если

¹ Существуют системы, в которых для обозначения шагающих окон используется термин «скользящее окно». Однако в Kafka Streams шагающие окна имеют некоторые отличия от скользящих, которые мы вскоре обсудим.

интервал бездействия составляет пять секунд, то все записи с отметкой времени, попадающей в пятисекундный интервал от предыдущей записи с тем же ключом, будут включены в одно и то же окно. Иначе, если отметка времени в новой записи превышает интервал бездействия (в данном случае пять секунд), то, начиная с нее, будет создано новое окно. В отличие от переворачивающихся и шагающих окон, нижняя и верхняя границы включаются в окно. В следующем фрагменте показано, как определяется окно сеанса:

```
SessionWindows sessionWindow = SessionWindows
    .with(Duration.ofSeconds(5)); ❶
```

❶ Окно сеанса с интервалом бездействия пять секунд.

Как показано на рис. 5.5, окна сеансов не выравниваются по началу чего-либо (диапазон зависит от конкретного ключа) и имеют переменную длину. Протяженность окна целиком и полностью зависит от отметок времени в записях: горячие (активные) ключи дают более протяженные окна, а менее активные ключи — менее протяженные.

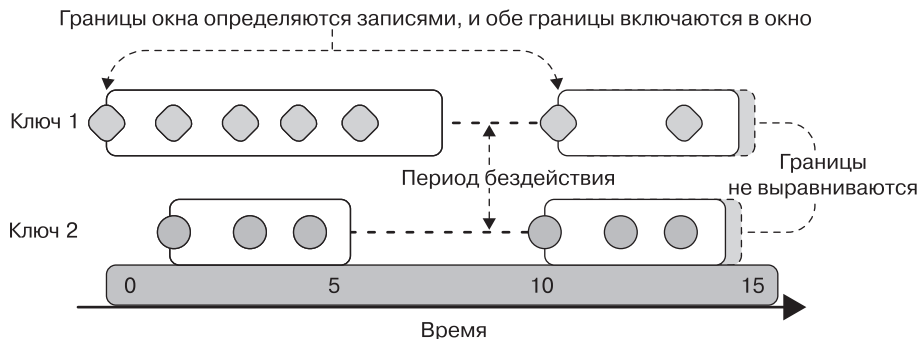


Рис. 5.5. Окна сеансов

Скользящие окна соединений

Скользящие окна соединений (sliding join windows) — это окна фиксированного размера. Они используются для вычисления соединений и создаются с помощью класса `JoinWindows`. Две записи попадают в одно окно, если разность их отметок времени меньше или равна размеру окна. Соответственно, так же, как в окнах сеансов, окно включает обе границы, и нижнюю, и верхнюю. Вот пример создания окна соединения протяженностью пять секунд:

```
JoinWindows joinWindow = JoinWindows
    .of(Duration.ofSeconds(5)); ❶
```

❶ Разность отметок времени не должна превышать пяти секунд, чтобы записи попали в одно окно.

Визуально окна соединений выглядят немного не так, как окна других типов, потому что используются для вычисления соединений и охватывают несколько входных потоков. На рис. 5.6 показано, как пятисекундное окно определяет записи, объединяющиеся в оконном соединении.

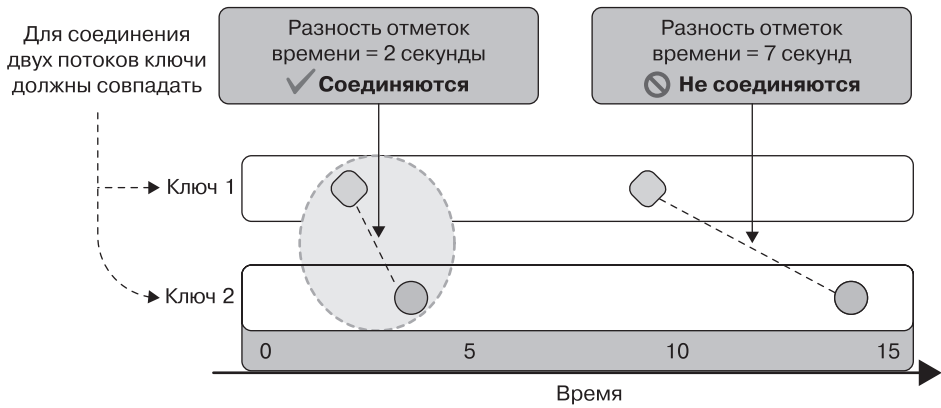


Рис. 5.6. Окно соединения

Скольльзящие окна агрегирования

В предыдущем разделе был представлен особый тип скользящих окон, который используется для вычисления соединений. Начиная с версии Kafka Streams 2.7.0, скользящие окна также можно использовать для агрегирования. Так же как скользящие окна соединений, границы скользящего окна агрегирования выравниваются по отметкам времени в записях (не от начала эпохи), а нижняя и верхняя границы включаются в окно. Кроме того, записи попадают в одно окно, если разность между их отметками времени находится в пределах указанного размера окна. Вот пример создания скользящего окна протяженностью пять секунд и периодом отсрочки ноль секунд¹:

```
SlidingWindows slidingWindow =
    SlidingWindows.withTimeDifferenceAndGrace(
        Duration.ofSeconds(5),
        Duration.ofSeconds(0));
```

¹ Скользящие окна агрегирования — единственный тип окон, для которого можно явно установить период отсрочки. Подробнее о периодах отсрочки мы поговорим далее в этой главе.

Выбор типа окна

Теперь, познакомившись с типами окон, которые поддерживаются в Kafka Streams, нужно решить, какой из них подходит для преобразования исходных событий сердцебиения в частоту сердечных сокращений с использованием оконного агрегирования.

Окна сеансов не подходят, потому что размер окна продолжает увеличиваться, пока в потоке наблюдаются события. Это не соответствует нашему требованию к фиксированному размеру окон 60 секунд. Скользящее окно соединения используется только для вычисления соединения, поэтому такой тип тоже можно исключить (хотя далее в этом проекте мы будем применять скользящее окно соединения).

Для предполагаемого агрегирования можно было бы использовать оставшиеся типы окон, но для простоты выровняем их границы по началу эпохи (что исключает скользящие окна агрегирования) и исключим возможность перекрытия (что исключает шагающие окна). В результате у нас остается только один тип — переворачивающиеся окна. Выбрав тип окна, можно приступить к реализации оконного агрегирования.

Оконное агрегирование

Для подсчета частоты сердечных сокращений мы выбрали переворачивающееся окно и теперь можем реализовать соответствующее оконное агрегирование. Сначала создадим окно вызовом метода `Timewindows.of`, а затем разобьем поток на окна с помощью оператора `windowedBy`. В следующем фрагменте показано, как выполнить оба этих шага:

```
Timewindows tumblingWindow =
    Timewindows.of(Duration.ofSeconds(60));

KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey() ❶
        .windowedBy(tumblingWindow) ❷
        .count(Materialized.as("pulse-counts")); ❸

pulseCounts
    .toStream() ❹
    .print(Printed.<Windowed<String>, Long>toSysOut().withLabel("pulse-counts")); ❺
```

❶ Группировка записей — необходимое условие для агрегирования. Однако записи в теме `pulse-events` уже снабжены ключами с использованием нужной схемы (то есть по идентификатору пациента), поэтому вместо оператора `groupBy`

можно применять `groupByKey`, чтобы избежать ненужного перераспределения данных по разделам.

❷ Для разбиения потока используется переворачивающееся окно протяженностью 60 секунд, что в дальнейшем позволит преобразовать исходные события сердцебиения в частоту сердечных сокращений (в ударах в минуту).

❸ Материализация частоты сердечных сокращений для интерактивных запросов (это потребуется на шаге 8 в нашей топологии процессора на рис. 5.1).

❹ Исклчительно для отладки: `KTable` преобразуется в поток, чтобы вывести содержимое в консоль.

❺ Вывод содержимого оконного потока в консоль. Операторы вывода удобно использовать при разработке на локальной машине, но их следует удалить перед развертыванием приложения в рабочем окружении.

Отметьте один интересный момент в этом примере: тип ключа в `KTable` изменился с `String` на `Windowed<String>`. Дело в том, что оператор `windowedBy` преобразует объекты `KTable` в *оконные* `KTable`. Оконные имеют многомерные ключи, включающие не только исходный ключ записи, но также временной диапазон окна. Это необходимо для группировки ключей в подгруппы (окна), так как при использовании исходного ключа все события сердцебиений окажутся включены в одну подгруппу. Мы можем увидеть, как выглядят полученные многомерные ключи, создав несколько записей в теме `pulse-events` и просмотрев вывод оконного потока. Возьмем для примера несколько исходных записей (здесь ключ и значение записи отделены символом вертикальной черты |):

```
1|{"timestamp": "2020-11-12T09:02:00.000Z"}
1|{"timestamp": "2020-11-12T09:02:00.500Z"}
1|{"timestamp": "2020-11-12T09:02:01.000Z"}
```

В ходе оконной обработки получится результат, показанный в примере 5.3. Обратите внимание, что старый ключ в каждой из этих записей (1) был преобразован в следующий формат:

```
[<старый_ключ>@<начало_окна_мс>/<конец_окна_мс>]
```

Пример 5.3. Вывод оператора `print`, показывающий, как выглядят многомерные ключи в таблице `pulseCounts`, разбитой на окна

```
[pulse-counts]: [1@1605171720000/1605171780000], 1 ❶
[pulse-counts]: [1@1605171720000/1605171780000], 2
[pulse-counts]: [1@1605171720000/1605171780000], 3
```

❶ Многомерный ключ записи, содержащий и исходный ключ, и границы окна.

Помимо логики преобразования ключей, предыдущий вывод демонстрирует поведение Kafka Streams. Счетчик частоты сердечных сокращений, вычисляемый в ходе оконного агрегирования, обновляется с каждым новым событием сердцебиения. Это можно видеть в примере 5.3, где первое значение частоты сердечных сокращений равно 1, за ним следуют 2, 3 и т. д. Таким образом, операторы, следующие далее в потоке обработки, будут видеть не только окончательные результаты, подсчитанные для окна (количество ударов в минуту), но также промежуточные (количество *сердечных сокращений*, зафиксированных в текущем 60-секундном окне *на данный момент*). В приложениях с низкой задержкой эти промежуточные или неполные результаты, вычисляемые в текущем окне, могут быть весьма полезны. Однако в нашем случае это не так, потому что нас интересует частота сердечных сокращений в минуту, а не количество сокращений, зафиксированных к данному моменту в незакрытом окне, которое в лучшем случае будет вводить в заблуждение. Посмотрим, почему Kafka Streams генерирует промежуточные результаты и можно ли настроить этот аспект его поведения.

Вывод результатов оконной обработки

Выбор момента, *когда* запускать вычисления в окне, является удивительно сложным для систем потоковой обработки. Сложность обусловлена двумя фактами:

- события в неограниченных потоках могут не всегда следовать в порядке увеличения отметок времени, особенно при использовании семантики времени события¹;



Kafka гарантирует следование событий в *порядке смещений* на уровне раздела. Это означает, что каждый потребитель всегда будет получать события в той же последовательности, в какой они были добавлены в тему (по возрастанию значения смещения).

- иногда события могут *задерживаться*.

Возможность нарушения следования в порядке увеличения отметок времени не позволяет предположить, что, если была получена запись с определенной отметкой времени, значит, были получены все записи, которые должны были

¹ Конечно, если использовать семантику времени приема, задав в параметре `message.timestamp.type` значение `LogAppendTime`, записи всегда будут следовать в порядке соответствующих им отметок времени. Это связано с тем, что отметки времени перезаписываются в момент добавления в тему. Однако время приема несколько не соответствует времени самого события и в лучшем случае является лишь приближением к нему (при условии, что событие записывается в тему сразу после его создания).

поступить до этого момента, и, следовательно, можно вычислить окончательный оконный результат. Кроме того, возможные задержки и внеочередные данные требуют определиться с выбором: ждать еще какое-то время, пока поступят все данные, или выводить оконный результат всякий раз, когда он обновляется (как показано в примере 5.3)? Это компромисс между полнотой и задержкой. Поскольку ожидание данных с большей вероятностью приведет к полному результату, этот подход оптимизирует полноту. С другой стороны, немедленное распространение обновлений вниз по потоку (даже если они могут быть неполными) уменьшает задержку. Вам решать, что оптимизировать, приняв во внимание установленные соглашения об уровне обслуживания.

На рис. 5.7 показано, как могут возникнуть обе эти проблемы. Пациент 1 подключен к машине мониторинга жизненно важных показателей, в которой периодически возникают проблемы с сетью. Это приводит к задержкам поступления некоторых показателей в кластер Kafka. Кроме того, поскольку в тему `pulse-events` поступают данные от нескольких производителей (по числу наблюдаемых пациентов), это приводит к дополнительной неупорядоченности событий относительно отметок времени их событий.

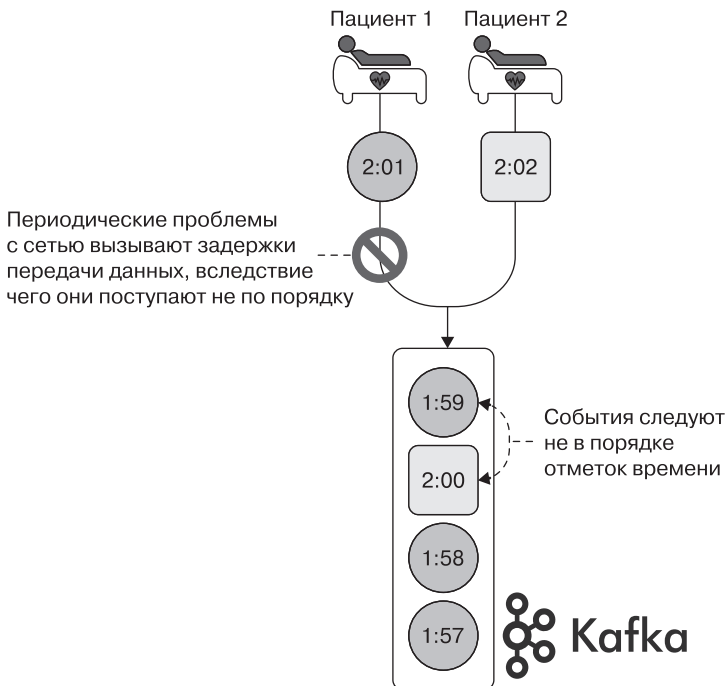


Рис. 5.7. Данные могут поступать не в порядке их отметок времени по многим причинам; одна из них — состояние гонки, возникающей в темах с несколькими производителями

Важно заметить, что причиной нарушения порядка не всегда является какой-то сбой. Это может произойти даже в нормальных рабочих условиях, например, когда несколько производителей пишут в одну тему.

Как упоминалось выше, для преодоления этих проблем необходимо оптимизировать либо задержку, либо полноту. По умолчанию Kafka Streams оптимизирует задержку, используя подход, называемый *непрерывным уточнением* (<https://oreil.ly/-tii3>). Непрерывное уточнение предполагает немедленное выполнение вычислений и выдачу нового результата с добавлением в окно нового события. Вот почему мы видели промежуточные результаты, отправляя записи в нашем приложении мониторинга состояния пациентов (см. пример 5.3). Однако при непрерывном уточнении каждый результат следует рассматривать как потенциально неполный, а сгенерированное событие *не* означает, что были обработаны все записи, которые в конечном итоге попадают в окно. Кроме того, получение задержанных данных может вызывать появление событий в неожиданные моменты времени.

В следующих двух разделах мы поговорим о том, как решить каждую из этих проблем в нашем приложении мониторинга пациентов. Сначала мы обсудим стратегию обработки задержанных данных в Kafka Streams, а потом посмотрим, как подавать промежуточные оконные вычисления с помощью оператора `suppress`.

Период отсрочки

Одна из самых больших проблем, с которыми приходится сталкиваться разработчикам систем потоковой обработки, — задержанные данные. Многие фреймворки, в том числе придерживающиеся устоявшейся модели потока данных (<https://oreil.ly/wAhJZ>, например, Apache Flink), используют *водяные знаки* (watermarks). Водяные знаки применяются для оценки момента, когда должны были поступить все данные для окна (обычно путем настройки *размера окна* и *допустимой задержки* событий). Благодаря этой поддержке пользователи могут указать, как обрабатывать опоздавшие события (определяемые водяным знаком). Наиболее популярный выбор по умолчанию (в Dataflow, Flink и др.) — отбрасывать опоздавшие события.

По аналогии с подходом на основе водяных знаков, Kafka Streams позволяет определить допустимую задержку событий с помощью настройки *периода отсрочки*, в течение которого окно будет оставаться открытым для включения отложенных и следующих не по порядку событий. Например, мы настроили наше переворачивающееся окно, как показано ниже:

```
TimeWindows tumblingWindow =  
    TimeWindows.of(Duration.ofSeconds(60));
```


Чтобы установить пятисекундную задержку для событий сердцебиения (которые используются для вычисления частоты сердечных сокращений), можно задать для нашего окна период отсрочки:

```
TimeWindows tumblingWindow =  
    TimeWindows  
        .of(Duration.ofSeconds(60))  
        .grace(Duration.ofSeconds(5));
```

Период отсрочки можно увеличить, но помните о компромиссах: больший период отсрочки оптимизирует полноту, дольше удерживая окно открытым, но делает это за счет большей задержки окончательных вычислений все на тот же период отсрочки.

Теперь посмотрим, как решить проблему выдачи промежуточных результатов.

Подавление

Как мы узнали в предыдущем разделе, стратегия непрерывного уточнения в Kafka Streams, предполагающая выдачу результатов всякий раз, когда в окно поступают новые данные, идеально подходит для случаев оптимизации задержки и допустимости получения неполных (промежуточных) результатов из окна¹.

Однако в нашем приложении мониторинга пациентов это нежелательно. Частоту сердечных сокращений нельзя рассчитать, используя данные за период менее 60 секунд, поэтому для окна должен выдаваться только окончательный результат. В этом нам поможет оператор `suppress`. Этот оператор позволяет выводить только окончательный результат оконных вычислений и подавляет (то есть сохраняет на время промежуточные результаты в памяти) все другие события. Для применения оператора `suppress` требуется определить:

- стратегию подавления промежуточных вычислений в окне;
- какой объем памяти использовать для буферизации подавленных событий (настраивается выбором *конфигурации буфера*);
- что делать при исчерпании установленного объема памяти (настраивается выбором *стратегии переполнения буфера*).

Сначала рассмотрим две стратегии подавления. В табл. 5.1 перечислены стратегии, доступные в Kafka Streams. Обратите внимание, что выбор стратегии осуществляется вызовом метода класса `Suppressed`.

¹ Этим Kafka Streams отличается от многих других потоковых систем, в которых вычисления выполняются только при закрытии окна.

Таблица 5.1. Стратегии подавления оконных вычислений

Стратегия	Описание
<code>Suppressed.untilWindowCloses</code>	Выдавать только окончательные результаты окна
<code>Suppressed.untilTimeLimit</code>	Выдавать результаты окна по истечении заданного периода времени после получения последнего события. Если другое событие с тем же ключом придет до истечения лимита времени, оно заменяет первое событие в буфере (обратите внимание, что при этом таймер не перезапускается). Это дает эффект ограничения частоты обновления

В нашем приложении мониторинга пациентов результаты вычисления частоты сердцебиения должны выводиться только по истечении полных 60 секунд. Поэтому мы используем стратегию подавления `Suppress.untilWindowCloses`. Однако, прежде чем использовать эту стратегию, мы должны сообщить Kafka Streams, как буферизовать неотправленные результаты в памяти. В конце концов, подавленные результаты *не удаляются*; вместо этого для каждого ключа в данном окне сохраняется последняя неотправленная запись, пока не придет время выдать результат. Память — ограниченный ресурс, поэтому Kafka Streams требует четко указать, какой ее объем можно использовать для этой потенциально расточительной задачи подавления обновлений¹. Стратегия буферизации определяется с помощью конфигурации буфера. В табл. 5.2 перечислены все возможные конфигурации, доступные в Kafka Streams.

Таблица 5.2. Настройки конфигурации буфера

Конфигурация буфера	Описание
<code>BufferConfig.maxBytes()</code>	Буфер в памяти для хранения подавленных событий, ограниченный заданным количеством байтов
<code>BufferConfig.maxRecords()</code>	Буфер в памяти для хранения подавленных событий, ограниченный заданным количеством ключей
<code>BufferConfig.unbounded()</code>	Буфер в памяти для хранения подавленных событий будет использовать столько места в куче, сколько потребуется. В случае исчерпания свободного места в куче будет сгенерировано исключение <code>OutOfMemoryError</code>

¹ Поскольку при использовании оператора `suppress` Kafka Streams хранит последнюю запись для каждого ключа, требуемый объем памяти зависит от пространства ключей: чем меньше пространство ключей, тем меньше памяти требуется. Период отсрочки тоже может повлиять на объем потребляемой памяти.

Наконец, Kafka Streams предлагает несколько стратегий реакции на переполнение буфера. Все они описаны в табл. 5.3.

Таблица 5.3. Стратегии поведения при переполнении буфера

Стратегия при переполнении буфера	Описание
<code>shutDownWhenFull</code>	Завершать работу приложения с выполнением всех необходимых процедур. При использовании этой стратегии вы никогда не получите промежуточных результатов оконных вычислений
<code>emitEarlyWhenFull</code>	Вместо завершения приложения выдавать самые старые результаты. Используя эту стратегию, все еще можно получить промежуточные результаты оконных вычислений

Теперь, выяснив, какие стратегии подавления, конфигурации буфера и поведения при переполнении буфера доступны, определим, какая комбинация подходит для нашего случая. Мы не собираемся ограничивать скорость обновлений, поэтому не будем использовать `untilTimeLimit`. Нам нужны только окончательные результаты оконных вычислений частоты сердечных сокращений, поэтому `untilWindowCloses` лучше подходит для нас. Далее, мы ожидаем, что пространство ключей будет относительно небольшим, поэтому выберем неограниченную конфигурацию буфера `unbounded`. Наконец, нам совершенно не нужны промежуточные результаты, потому что есть риск неточного вычисления частоты сердечных сокращений (как, например, если выдать количество сердечных сокращений по прошествии всего 20 секунд). Поэтому используем стратегию поведения при переполнении буфера `ShutDownWhenFull`.

Теперь можно обновить конфигурацию приложения мониторинга жизненных показателей пациентов, чтобы задействовать в нем оператор `suppress`, как показано в примере 5.4.

Пример 5.4. Применение оператора `suppress` для вывода только окончательных результатов оконных вычислений частоты сердечных сокращений (`tumblingWindow`)

```
TimeWindows tumblingWindow =
    TimeWindows
        .of(Duration.ofSeconds(60))
        .grace(Duration.ofSeconds(5));

KTable<Windowed<String>, Long> pulseCounts =
    pulseEvents
        .groupByKey()
        .windowedBy(tumblingWindow)
        .count(Materialized.as("pulse-counts"))
        .suppress(
            Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull())); ❶
```

❶ Подавить результаты оконных вычислений, чтобы обеспечить получение только окончательных результатов.

Теперь, завершив шаг 4 в реализации нашей топологии процессора (см. рис. 5.1), перейдем к шагам 5 и 6: к фильтрации и изменению ключей в данных `pulse-events` и `body-temp-events`.

Фильтрация и изменение ключей оконных таблиц KTable

Если внимательно посмотреть, как используется `KTable` в примере 5.4, можно заметить, что организация оконных вычислений привела к изменению типа `String` ключа на `Windowed<String>`. Как упоминалось выше, это связано с необходимостью группировать записи по дополнительному измерению: диапазону окна. Итак, чтобы выполнить соединение с потоком `body-temp-events`, нужно изменить ключи в потоке `pulse-events`. Необходимо также отфильтровать оба потока данных, потому что интерес для нас представляют только записи, превышающие предопределенные пороговые значения.

Может показаться, что фильтрацию и изменение ключей можно выполнить в любом порядке, и это правда. И все же первой лучше выполнить фильтрацию. Мы знаем, что для замены ключей записей требуется использовать внутреннюю тему, где производится секционирование, поэтому, выполнив фильтрацию первой, мы уменьшим количество операций чтения/записи с этой внутренней темой, что несколько увеличит производительность приложения.

Операции фильтрации и изменения ключей уже обсуждались в предыдущих главах, поэтому мы не будем углубляться в них. Тем не менее для полноты картины ниже приводится код фильтрации и смены ключей для нашего приложения мониторинга пациентов:

```
KStream<String, Long> highPulse =
    pulseCounts
        .toStream() ❶
        .filter((key, value) -> value >= 100) ❷
        .map(
            (windowedKey, value) -> {
                return KeyValue.pair(windowedKey.key(), value); ❸
            });

KStream<String, BodyTemp> highTemp =
    tempEvents.filter((key, value) -> value.getTemperature() > 100.4); ❹
```

- ❶ Преобразование в поток, чтобы получить возможность применить оператор `map` для изменения ключей в записях.
- ❷ Оставить только результаты измерений частоты сердечных сокращений, превышающие предустановленный порог 100 ударов в минуту.
- ❸ Изменить ключи в потоке, используя оригинальный ключ, доступный как `windowedKey.key()`. Обратите внимание, что `windowedKey` — это экземпляр класса `org.apache.kafka.streams.kstream.Windowed`, предлагающего методы доступа к оригинальному ключу (`windowedKey.key()`), а также к базовому временному окну (`windowedKey.window()`).
- ❹ Оставить только результаты измерений температуры тела, превышающие предустановленный порог в 100,4 °F (38 °C).

Мы выполнили шаги 5 и 6 в топологии нашего процессора и теперь готовы реализовать оконное соединение.

Оконные соединения

Как обсуждалось в пункте «Скольльзящие окна соединений» выше в этой главе, оконные соединения требуют использовать скольльзящие окна соединений. Окна этого вида сравнивают отметки времени событий с обеих сторон соединения и определяют записи для объединения. Оконные соединения необходимы для соединений `KStream-KStream`, потому что потоки не ограничены. Следовательно, для быстрого поиска связанных значений данные должны быть материализованы в локальном хранилище состояний.

Вот как можно создать и использовать скольльзящее окно соединений для соединения потоков с частотой сердцебиения и температурой тела:

```
StreamJoined<String, Long, BodyTemp> joinParams =
    StreamJoined.with(Serdes.String(), Serdes.Long(), JsonSerdes.BodyTemp()); ❶

JoinWindows joinWindows =
    JoinWindows
        .of(Duration.ofSeconds(60)) ❷
        .grace(Duration.ofSeconds(10)); ❸

ValueJoiner<Long, BodyTemp, CombinedVitals> valueJoiner = ❹
    (pulseRate, bodyTemp) -> new CombinedVitals(pulseRate.intValue(), bodyTemp);

KStream<String, CombinedVitals> vitalsJoined =
    highPulse.join(highTemp, valueJoiner, joinWindows, joinParams); ❺
```

- ❶ Указать тип `Serdes` для использования в соединении.
- ❷ Записи с отметками времени, отстоящими друг от друга не далее чем на одну минуту, попадут в одно и то же окно и будут объединены.
- ❸ Допускается задержка до 10 секунд.
- ❹ Объединять частоту сердечных сокращений и температуру тела в объекты `CombinedVitals`.
- ❺ Выполняется присоединение.

Соединения особенно интересны с точки зрения времени, потому что разные стороны соединения могут получать записи с разной скоростью, поэтому необходима дополнительная синхронизация, чтобы произвести соединение событий, происходивших одновременно. К счастью для нас, Kafka Streams поддерживает такую возможность, передавая через топологию нашего процессора отметки времени вместе с данными на каждой стороне соединения. Рассмотрим эти идеи более подробно в следующем разделе.

Потоки данных, управляемые временем

Мы уже видели, как время может влиять на поведение таких операций, как оконное соединение и оконное агрегирование. Однако время также управляет течением данных через наши потоки. Для приложений потоковой обработки важно синхронизировать входные потоки, чтобы обеспечить правильность, особенно при обработке исторических данных из нескольких источников.

С целью такой синхронизации Kafka Streams создает одну *группу разделов* (partition group) для каждой задачи потока. Группа разделов буферизует записи из всех обрабатываемых разделов, используя приоритетную очередь и алгоритм выбора следующей записи из всех входных разделов. Для обработки выбирается запись с наименьшей отметкой времени.



Время потока — это наибольшая отметка времени в определенном разделе темы. Изначально это время неизвестно и может только увеличиваться или оставаться прежним. Время изменяется только с появлением новых данных. Этим время потока отличается от других понятий времени, обсуждавшихся выше, потому что является внутренним для Kafka Streams.

Когда одна задача Kafka Streams потребляет данные из нескольких разделов, например выполняя соединение, Kafka Streams сравнивает отметки времени следующих необработанных записей, называемых *головными записями*, в каждом разделе (очереди записей) и выбирает запись с наименьшей отметкой времени для обработки. Выбранная запись передается соответствующему процессору в топологии. Порядок действий показан на рис. 5.8.

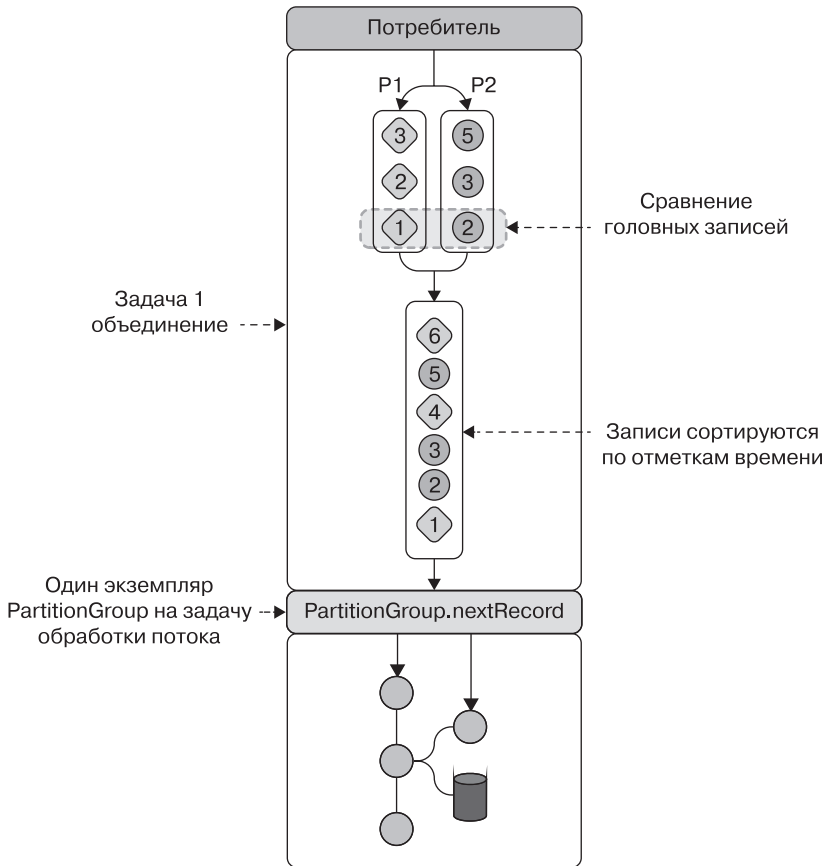


Рис. 5.8. Чтобы определить, как данные должны протекать через приложение Kafka Streams, производится сравнение отметок времени записей

Теперь, когда соединение готово и есть некоторые гарантии, что Kafka Streams постарается обрабатывать записи в порядке следования отметок времени, используя соответствующие механизмы управления на основе

времени¹, можно выполнить последние шаги в нашем процессе мониторинга пациентов. Вспомним, как добавить узел-приемник в топологию, а затем посмотрим, как запрашивать данные из оконного хранилища «ключ — значение».

Приемник предупреждений

Чтобы результаты соединения были доступны для нижестоящих потребителей, обогащенные данные нужно записать обратно в Kafka. Как было показано в предыдущих главах, добавить приемник в Kafka Streams очень просто. Следующий код показывает, как добавить приемник предупреждений с именем `alerts`. Всякий раз, обнаружив риск возникновения синдрома SIRS по пороговым значениям и оконному соединению, наше приложение будет записывать предупреждение в этот приемник:

```
vitalsJoined.to(
    "alerts",
    Produced.with(Serdes.String(), JsonSerdes.CombinedVitals())
);
```

Одно из преимуществ применения экстрактора отметок времени к входным потокам (см. пример 5.2) заключается в том, что выходные записи тоже будут связаны извлеченной отметкой времени. Для обычных потоков/таблиц, не связанных друг с другом, отметка времени распространяется от момента первоначального ее извлечения при регистрации узла-источника. Однако в соединениях, подобных показанному выше, Kafka Streams просматривает отметки времени в каждой записи, участвующей в соединении, и выбирает для выходной записи максимальное значение².

Приемник предупреждений предоставляет данные мониторинга пациентов пользователям темы `alerts` в режиме реального времени. Теперь давайте посмотрим, как запрашивать оконные хранилища «ключ — значение», хранящие

¹ Важно подчеркнуть слово «постарается», потому что один из входных потоков может опустеть. Однако есть возможность использовать параметр конфигурации `max.task.idle.ms`, определяющий интервал времени, в течение которого можно ждать поступления новых записей в опустевший входной поток. Значение по умолчанию равно 0, но его увеличение позволит улучшить синхронизацию времени за счет увеличения времени ожидания.

² До версии 2.3 выходной записи присваивалась отметка времени, равная отметке в любой записи, инициировавшей соединение. Использование максимального значения в более новых версиях Kafka Streams — более совершенное решение, потому что гарантирует присваивание результату одной и той же отметки времени, независимо от упорядоченности данных.

результаты оконного агрегирования (в нашем случае — частоту сердечных сокращений пациента).

Запрос оконных хранилищ «ключ — значение»

В подразделе «Запросы к неоконным хранилищам “ключ — значение”» главы 4 мы видели, как запрашивать неоконные хранилища ключей-значений. В отличие от них *оконные хранилища «ключ — значение»* поддерживают другой набор запросов, потому что в записях используются многомерные ключи, состоящие из исходного ключа и диапазона, охватываемого окном. Для начала рассмотрим сканирование ключей в диапазоне окон.

Сканирование ключей в диапазоне окон

Существует два вида сканирования диапазонов, которые можно использовать с оконными хранилищами «ключ — значение». Первый — поиск определенного ключа в заданном диапазоне окон, который требует трех параметров, таких как:

- искомый ключ для поиска (в нашем приложении мониторинга пациентов он соответствует идентификатору пациента, например, 1);
- нижняя граница диапазона окна в миллисекундах от начала эпохи¹ (например, 1605171720000, что соответствует дате и времени 2020-11-12T09:02:00.00Z);
- верхняя граница диапазона окна в миллисекундах от начала эпохи (например, 1605171780000, что соответствует дате и времени 2020-11-12T09:03:00Z).

Вот как можно реализовать такой вид сканирования и извлечь соответствующие свойства из результата:

```
String key = 1;
Instant fromTime = Instant.parse("2020-11-12T09:02:00.00Z");
Instant toTime = Instant.parse("2020-11-12T09:03:00Z");

WindowStoreIterator<Long> range = getBpmStore().fetch(key, fromTime, toTime); ❶
while (range.hasNext()) {
    KeyValue<Long, Long> next = range.next(); ❷
    Long timestamp = next.key; ❸
    Long count = next.value; ❹
    // выполнить некоторые действия с извлеченными значениями
}

range.close(); ❺
```

¹ 1970-01-01T00:00:00Z (UTC).

- ❶ Возвращает итератор, который можно использовать для обхода ключей в выбранном временном диапазоне.
- ❷ Получить следующий элемент.
- ❸ Отметка времени в записи доступна в виде свойства `key`.
- ❹ Значение в записи доступно в виде свойства `value`.
- ❺ Закрывается итератор, чтобы избежать утечки памяти.

Сканирование диапазонов окон

Второй вид сканирования оконных хранилищ «ключ — значение» предполагает поиск всех ключей в заданном временном диапазоне. Запросы этого вида принимают два параметра:

- нижнюю границу диапазона окна в миллисекундах от начала эпохи¹ (например, `1605171720000`, что соответствует дате и времени `2020-11-12T09:02:00.00Z`);
- верхнюю границу диапазона окна в миллисекундах от начала эпохи (например, `1605171780000`, что соответствует дате и времени `2020-11-12T09:03:00Z`).

Вот как можно реализовать такой вид сканирования и извлечь соответствующие свойства из результата:

```
Instant fromTime = Instant.parse("2020-11-12T09:02:00.00Z");
Instant toTime = Instant.parse("2020-11-12T09:03:00Z");

KeyValueIterator<Windowed<String>, Long> range =
    getBpmStore().fetchAll(fromTime, toTime);

while (range.hasNext()) {
    KeyValue<Windowed<String>, Long> next = range.next();
    String key = next.key.key();
    Window window = next.key.window();
    Long start = window.start();
    Long end = window.end();
    Long count = next.value;
    // выполнить некоторые действия с извлеченными значениями
}

range.close();
```

¹ 1970-01-01T00:00:00Z (UTC).

Все записи

Подобно запросам сканирования диапазона, запрос `all()` возвращает итератор для обхода всех оконных пар «ключ — значение», доступных в локальном хранилище состояний¹. Ниже показано, как выполнить запрос `all()` к локальному оконному хранилищу «ключ — значение». Обход результатов выполняется так же, как при сканировании диапазона, поэтому не буду повторно показывать эту логику:

```
KeyValueIterator<Windowed<String>, Long> range = getBpmStore().all();
```



Важно не забыть закрыть итератор, закончив работать с ним, чтобы избежать утечки памяти. Например, в предыдущем фрагменте кода нужно не забыть вызвать `range.close()`, закончив обход.

Используя эти виды запросов, можете создать интерактивную службу, как обсуждалось в предыдущей главе. Для этого достаточно добавить в приложение службу и клиента RPC, использовать логику обнаружения экземпляров в Kafka Streams для поиска удаленных экземпляров приложений и связать предыдущие запросы к оконному хранилищу «ключ — значение» с конечными точками RPC или RESTful (за дополнительной информацией обращайтесь к подразделу «Удаленные запросы» в главе 4).

Заключение

В этой главе вы узнали, как можно использовать время в более сложных сценариях потоковой обработки. Используя ту или иную семантику времени в определениях топологии, можно добиться детерминированной обработки данных в Kafka Streams. Время не только управляет поведением оконного агрегирования, оконного соединения и других операций, основанных на времени, но также, учитывая, что Kafka Streams пытается синхронизировать входные потоки по времени, контролирует, как и когда данные проходят через наше приложение.

Оконная обработка данных позволяет задавать временные отношения между событиями. Независимо от особенностей использования этих отношений — для агрегирования (как в примере выше, для преобразования исходных событий сердечных сокращений в основанную на времени частоту сердцебиения) или для

¹ В зависимости от количества ключей в хранилище состояний этот запрос может оказаться весьма тяжеловесным.

соединения данных (например, когда мы соединили поток частоты сердечных сокращений с потоком температуры тела) — время открывает дверь для более значимого обогащения данных.

Наконец, в процессе знакомства с приемами запроса данных из оконных хранилищ ключей-значений вы узнали одно важное обстоятельство оконных хранилищ состояний: они используют многомерные ключи (содержащие как исходный ключ, так и временной диапазон окна) и поэтому поддерживают другой набор запросов, включая сканирование по диапазону окон.

В следующей главе мы завершим обсуждение приложений Kafka Streams с состоянием, рассмотрев расширенные задачи управления состоянием.

Расширенное управление состоянием

В двух предыдущих главах мы обсудили потоковую обработку данных в Kafka Streams с сохранением состояния. После знакомства с особенностями агрегирования, соединения и оконных операций стало очевидно, что обработка данных с сохранением состояния не вызывает никаких сложностей.

Однако, как упоминалось ранее, хранилища состояний влекут за собой дополнительные сложности. По мере масштабирования приложения, устранения сбоев и выполнения планового обслуживания вы узнаете, что обработка с сохранением состояния требует глубокого понимания базовой механики, чтобы обеспечить бесперебойную работу приложения с течением времени.

Цель этой главы — глубже изучить хранилища состояний, чтобы дать вам возможность достичь более высокого уровня надежности при создании приложений потоковой обработки с сохранением состояния. Большая часть этой главы посвящена теме *повторной балансировки*, или *переконфигурирования* (rebalancing), которая происходит, когда возникает необходимость перераспределить данные между потребителями в группе. Особенно большое влияние переконфигурирования имеет в приложениях с состоянием, поэтому продолжим укреплять ваше понимание, чтобы подготовить вас к реализации этой возможности в своих приложениях.

Перечислю вопросы, на которые я постараюсь ответить.

- Как хранилища состояний организованы на диске?
- Как в приложениях с состоянием добиться высокой отказоустойчивости?
- Как настраиваются встроенные хранилища состояний?

- Какие виды событий оказывают наибольшее влияние на приложения с состоянием?
- Что можно предпринять, чтобы уменьшить время восстановления задач с состоянием?
- Как не допустить бесконечного роста хранилища состояний?
- Как использовать кэш DSL для ограничения частоты обновлений в нисходящем направлении?
- Как организовать наблюдение за процессом восстановления состояния с помощью обработчика события восстановления состояния?
- Как с помощью обработчика событий изменения состояния обнаруживать случаи переконфигурации?

Начнем с организации хранилищ состояний на диске.

Организация хранилища состояний на диске

Kafka Streams включает два вида хранилищ состояний: временные (в памяти) и постоянные. Постоянные хранилища состояний обычно предпочтительнее, потому что могут помочь сократить время восстановления приложения, когда состояние необходимо инициализировать повторно (например, в случае сбоя или переноса задачи на другой узел).

По умолчанию постоянные хранилища состояний размещаются в каталоге `/tmp/kafka-streams`. Каталог можно изменить, настроив свойство `StreamsConfig.STATE_DIR_CONFIG`, при этом, учитывая эфемерный характер каталога `/tmp` (содержимое этого каталога удаляется во время перезагрузки/сбоя системы), для хранения состояния приложений следует выбрать другой каталог.

Поскольку постоянные хранилища состояний находятся на диске, их легко исследовать¹. Такие исследования позволяют получить удивительно большой объем информации из одних только имен каталогов и файлов. Дерево файлов в примере 6.1 было получено из приложения мониторинга пациентов, созданного в предыдущей главе. Комментарии содержат дополнительные сведения о каталогах и файлах.

¹ Примечание: не пытайтесь изменить содержимое файлов.

Пример 6.1. Организация постоянного хранилища состояний на диске

```
.
├── 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
│   └── ...
```

❶ Каталог верхнего уровня содержит идентификатор приложения. Это помогает понять, какие приложения выполняются на сервере, особенно в общем окружении, где рабочие нагрузки могут планироваться на любом количестве узлов (например, в кластере Kubernetes).

❷ Каждый из каталогов второго уровня соответствует одной задаче Kafka Streams. Имя каталога соответствует идентификатору задачи, состоящему из двух частей: <идентификатор-субтопологии>_<раздел>. Обратите внимание: как обсуждалось в подразделе «Субтопологии» в главе 2, субтопология может обрабатывать данные из одной или нескольких тем, в зависимости от логики программы.

❸ В файлах контрольных точек хранятся смещения в темах журнала изменений (см. подраздел «Темы журналов изменений» далее). Они сообщают Kafka Streams, какие данные прочитаны в локальное хранилище состояний, и играют важную роль в восстановлении хранилища состояний.

❹ Файл блокировки используется в Kafka Streams для блокировки каталога состояния и помогает предотвратить нежелательные исходы, свойственные конкурентному выполнению.

❺ Фактические данные хранятся в именованных каталогах. Здесь имя каталога `pulse-counts` соответствует имени, которое мы задали в настройках материализации хранилища состояний.

Знание структуры хранилищ состояний на диске помогает раскрыть некоторые секреты, связанные с их работой. Важную роль играют также файлы блокировок и контрольных точек, иногда ссылки на них можно увидеть в журналах ошибок (например, проблемы с разрешениями могут проявляться как ошибки записи в файл контрольных точек, а проблемы конкурентного выполнения — как неудачные попытки приобретения блокировки), поэтому полезно знать их назначение и местоположение.

Файл контрольной точки используется для восстановления хранилища состояний. Исследуем подробнее этот аспект, для чего сначала обсудим характеристики отказоустойчивости приложений с состоянием, а затем посмотрим, как контрольные точки помогают сократить время восстановления.

Отказоустойчивость

Своей отказоустойчивостью Kafka Streams во многом обязана уровню хранения Kafka и протоколу управления группами. Например, репликация (копирование) данных на уровне раздела означает, что при отключении брокера данные остаются доступными в одной из реплик (копий) раздела в другом брокере. Кроме того, если при использовании групп потребителей один экземпляр приложения выйдет из строя, работа будет передана какому-то из работоспособных экземпляров.

Однако в отношении приложений с состоянием Kafka Streams принимает дополнительные меры повышения отказоустойчивости, включая использование тем журналов изменений для хранения резервных копий хранилищ состояния и резервных реплик для уменьшения времени повторной инициализации в случае сбоя. Все эти механизмы отказоустойчивости, характерные для Kafka Streams, мы подробно обсудим в следующих разделах.

Темы журналов изменений

Хранилища состояний, если явно не отключены, поддерживаются темами журналов изменений (changelog topics), которые создаются и управляются Kafka Streams. В этих темах фиксируются изменения состояний всех ключей в хранилище, что позволяет в случае сбоя воспроизвести их и восстановить состояние приложения¹. В случае полной потери состояния (или при запуске нового экземпляра) изменения из этой темы воспроизводятся с самого начала.

¹ Для воспроизведения изменений, когда необходимо повторно инициализировать хранилище состояний, используется выделенный потребитель, называемый потребителем восстановления (restore consumer).

Однако если существует файл контрольной точки (см. пример 6.1), то воспроизведение состояния начинается со смещения, заданного контрольной точкой. Оно сообщает, какие данные уже были прочитаны в хранилище состояний. Восстановление с использованием контрольной точки происходит намного быстрее, потому что восстановление части состояния требует меньше времени, чем восстановление полного состояния.

Темы журналов изменений настраиваются с помощью класса `Materialized` в DSL. Например, в предыдущей главе мы материализовали хранилище состояний под названием `pulse-counts`, используя следующий код:

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

Класс `Materialized` имеет дополнительные методы настройки темы журнала изменений. Например, вот как можно полностью отключить журналирование изменений, чтобы создать так называемое *эфмерное хранилище* (то есть хранилище состояний, которое нельзя восстановить в случае сбоя):

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

Однако обычно не рекомендуется отключать журналирование изменений, потому что в этом случае хранилище состояний перестанет быть отказоустойчивым и вы лишитесь возможности использовать резервные реплики. Часто, настраивая темы журналов изменений, либо переопределяют сохранение оконных и сеансовых хранилищ с помощью метода `withRetention` (рассматривается в пункте «Сохранение окон» далее в этой главе), либо добавляют определенные настройки. Например, вот как можно увеличить количество несинхронизированных реплик до двух:

```
Map<String, String> topicConfigs =
  Collections.singletonMap("min.insync.replicas", "2"); ❶
```

```
KTable<Windowed<String>, Long> pulseCounts =
  pulseEvents
    .groupByKey()
    .windowedBy(tumblingWindow)
    .count(
      Materialized.<String, Long, WindowStore<Bytes, byte[]>>
        as("pulse-counts")
        .withValueSerde(Serdes.Long())
        .withLoggingEnabled(topicConfigs)); ❷
```

❶ Создается представление для хранения конфигурации темы. Записи могут включать любые допустимые настройки темы (https://oreil.ly/L_WOj) и значения.

❷ Настройка темы журнала изменений передачей конфигурации методу `Materialized.withLoggingEnabled`.

Если теперь запросить описание темы, вы увидите, что она настроена соответствующим образом:

```
$ kafka-topics \
  --bootstrap-server localhost:9092 \
  --topic dev-consumer-pulse-counts-changelog \ ❶
  --describe

# вывод
Topic: dev-consumer-pulse-counts-changelog
PartitionCount: 4
ReplicationFactor:1
Configs: min.insync.replicas=2
...
```

❶ Обратите внимание, что имена для тем журналов изменений выбираются с применением следующей схемы:

`<идентификатор_приложения>-<внутреннее_имя_хранилища>-changelog`.

Следует отметить, что на момент написания этих строк таким способом нельзя было перенастроить тему журнала изменений после ее создания¹. Для перенастройки существующей темы журнала изменений нужно использовать сценарии командной строки Kafka. Вот как можно вручную обновить конфигурацию темы журнала изменений после ее создания:

```
$ kafka-configs \ ❶
  --bootstrap-server localhost:9092 \
  --entity-type topics \
  --entity-name dev-consumer-pulse-counts-changelog \
  --alter \
  --add-config min.insync.replicas=1 ❷

# вывод
Completed updating config for topic dev-consumer-pulse-counts-changelog
```

❶ Можно также использовать консольный сценарий `kafka-topics`. Не забудьте добавить расширение файла (`.sh`), если вы используете Kafka не из Confluent Platform.

❷ Обновление настроек темы.

¹ Зарегистрирован запрос на реализацию поддержки перенастройки внутренних тем, состояние запроса можно отслеживать по адресу <https://oreil.ly/OoKBV>.

Теперь, получив представление о назначении тем журналов изменений, а также о переопределении настроек по умолчанию, перейдем к тому, что обеспечивает высокую доступность Kafka Streams: к резервным репликам.

Резервные реплики

Один из способов сократить время простоя при сбое приложения с состоянием — создать и поддерживать копии состояния задачи в нескольких экземплярах приложения.

Kafka Streams автоматически решает эту проблему именно таким образом, если присвоить положительное значение свойству `NUM_STANDBY_REPLICAS_CONFIG`. Например, следующая настройка приведет к созданию двух резервных реплик:

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

Обнаружив настройку, определяющую положительное число резервных реплик, Kafka Streams будет передавать любые невыполненные задачи с состоянием экземпляру с горячим резервированием. Это сократит время простоя приложений за счет устранения повторной инициализации базового хранилища состояний с нуля.

Кроме того, как будет показано ближе к концу главы, более новые версии Kafka Streams позволяют возвращаться к резервным репликам при запросе хранилищ состояния во время перебалансировки. Но перед этим обсудим, что подразумевается под перебалансировкой и почему она является самым большим врагом приложений Kafka Streams с состоянием.

Перебалансировка: враг состояния (хранилища)

Мы узнали, что темы журналов изменений и резервные реплики помогают уменьшить влияние сбоев на приложения с состоянием. Первые помогают Kafka Streams восстанавливать состояние в случае его потери, а вторые — минимизировать время повторной инициализации хранилищ состояний.

Но, несмотря на то что Kafka Streams прозрачно обрабатывает сбои, потеря хранилища состояний, даже временная, не становится менее разрушительной (особенно для приложений с большим объемом состояния). Почему? Потому что

для восстановления состояния из резервной копии необходимо воспроизвести каждое сообщение из основной темы, а если эта тема огромна, то повторное чтение всех записей может занять несколько минут, в некоторых случаях даже часов.

Самая частая причина повторной инициализации состояния — *перевесировка* (rebalancing). Впервые мы использовали этот термин при обсуждении групп потребителей в разделе «Группы потребителей» главы 1. Kafka автоматически распределяет работу между активными членами группы потребителей, но иногда работу приходится перераспределять в ответ на определенные события, в первую очередь при изменении членства в группе¹. Мы не будем подробно рассматривать весь протокол перевесировки, но даже для поверхностного обсуждения необходимо вспомнить значения некоторых терминов:

- *координатор группы* — это выделенный брокер, отвечающий за поддержание членства в группе потребителей (например, путем получения контрольных сигналов и запуска перевесировки при обнаружении изменения членства);
- *лидер группы* — выделенный потребитель в каждой группе, отвечающий за назначение разделов.

Мы встретимся с этими терминами при обсуждении перевесировки в следующих разделах. А пока важно запомнить, что перевесировка обходится дорого, если приводит к переносу задачи с состоянием на другой экземпляр, не имеющий резервной реплики. Есть несколько стратегий решения проблем, сопутствующих перевесировке:

- предотвратить перемещение состояния, если это возможно;
- если состояние нужно переместить или воспроизвести, то следует максимально уменьшить необходимое для этого время.

В обоих случаях Kafka Streams автоматически предпринимает определенные меры, но мы тоже можем предусмотреть некоторые действия, чтобы уменьшить влияние перевесировки. Рассмотрим их подробнее, начав с первой стратегии: предотвращение миграции состояния.

¹ Добавление разделов в исходную тему или их удаление тоже может привести к перевесировке.

Предотвращение миграции состояния

Когда задача с состоянием перемещается на другой экземпляр, туда же переносится и ее состояние. Для приложений с объемным состоянием воссоздание хранилища состояний на целевом узле может занять много времени, поэтому по возможности такого переноса следует избегать.

За распределение работы между активными потребителями отвечает лидер группы (один из потребителей), поэтому на библиотеку Kafka Streams, реализующую логику балансировки нагрузки, возлагается определенная ответственность за предотвращение ненужной миграции хранилища состояний. Одним из способов достижения этого является так называемое *закрепленное назначение* (sticky assignment), автоматически поддерживаемое в Kafka Streams. Подробнее этот механизм рассматривается в следующем разделе.

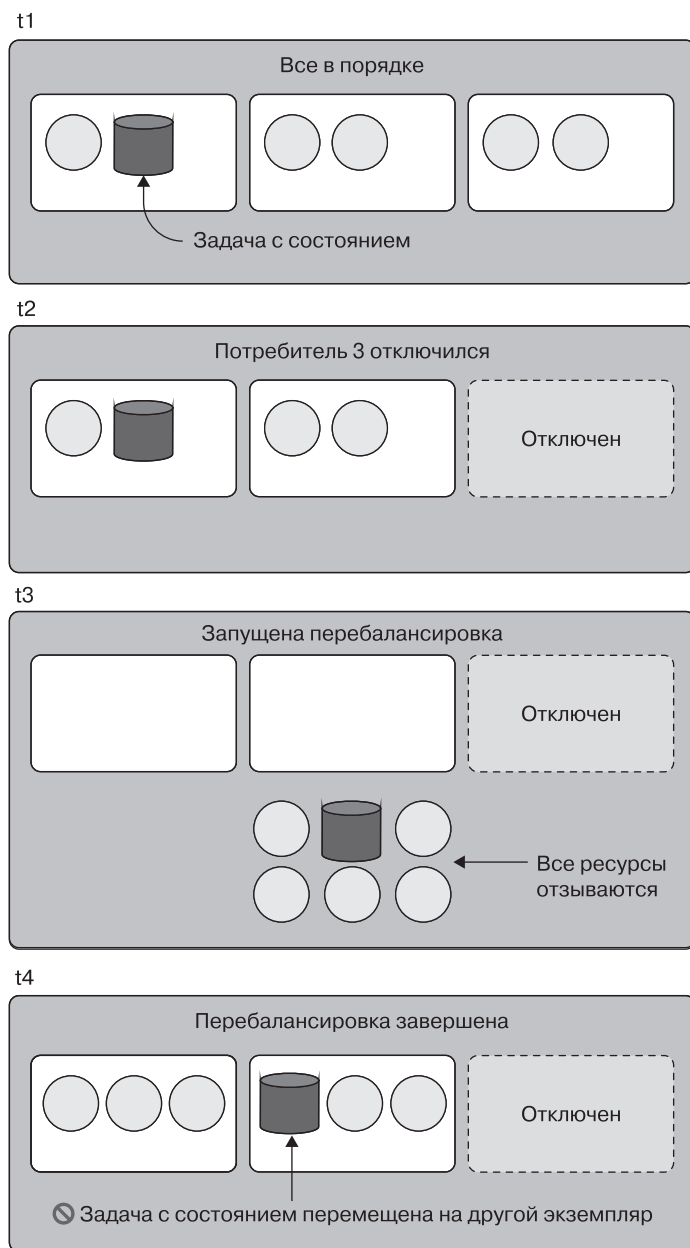
Закрепленное назначение

Для предотвращения перемещения задач с состоянием в Kafka Streams используется настраиваемая стратегия назначения разделов¹, согласно которой задачи назначаются экземплярам, которые прежде выполняли эту задачу и, следовательно, должны иметь копию базового хранилища состояний. Эта стратегия называется *закрепленным назначением* (sticky assignment).

Чтобы понять проблему, решаемую в Kafka Streams с помощью поддержки закрепленного состояния, рассмотрим стратегию переконфигурирования по умолчанию для других типов клиентов Kafka. На рис. 6.1 показано, что при переконфигурировании задача с состоянием может быть передана другому экземпляру приложения, что будет стоить чрезвычайно дорого.

Механизм закрепленного назначения разделов в Kafka Streams решает эту проблему, запоминая, какой задаче принадлежат каждый раздел и связанные с ним хранилища состояний, и назначая задачу с состоянием ее предыдущему владельцу. Как показано на рис. 6.2, это значительно повышает доступность нашего приложения, позволяя избежать ненужной повторной инициализации потенциально больших хранилищ состояний.

¹ Внутренний класс реализации называется StickyTaskAssignor. Обратите внимание, что в Kafka Streams невозможно переопределить механизм распределения разделов по умолчанию.

**Рис. 6.1.** Незакрепленное назначение раздела

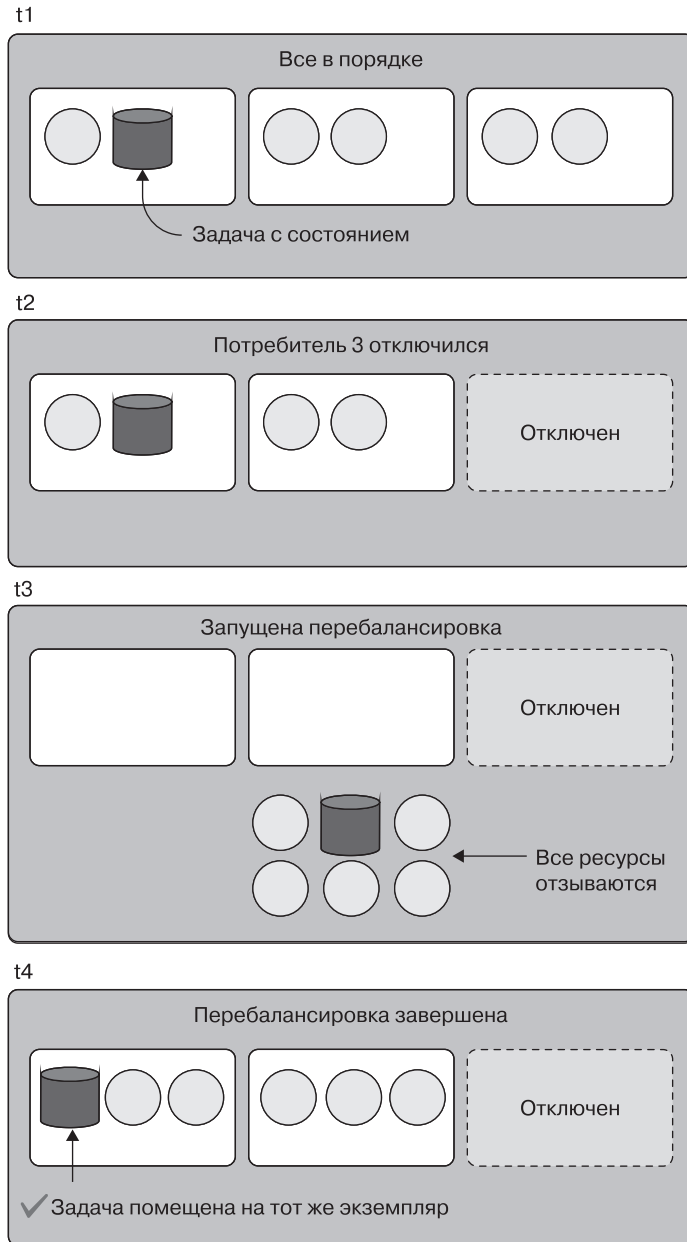


Рис. 6.2. Закрепленное назначение раздела с использованием внутреннего механизма Kafka Streams

Несмотря на то что механизм закрепленного назначения помогает назначать задачи их предыдущим владельцам, хранилища состояний все еще могут переноситься в случае временного отключения клиентов Kafka Streams. А теперь обсудим, что мы, как разработчики Kafka Streams, можем предпринять, чтобы избежать перебалансировки в периоды временного отключения.

Статическое членство

Ненужная перебалансировка — одна из проблем, которая может привести к перемещению состояния. Иногда даже самое обычное событие, например изменение состава группы потребителей, может вызвать несколько перебалансировок. Обнаружив факт отключения любого из членов группы, ее координатор инициирует перебалансировку и немедленно перераспределяет работу между оставшимися экземплярами приложения.

Когда работоспособность экземпляра восстанавливается после короткого периода отключения, координатор не распознает его, потому что идентификатор члена группы (уникальный идентификатор, назначаемый координатором при регистрации потребителя) оказывается удален, а рассматривает как нового члена группы, что опять же может вызвать перераспределение работы.

Для смягчения этой проблемы можно использовать механизм *статического членства*, главная цель которого — сократить количество перебалансировок из-за временного отключения за счет использования жестко заданного идентификатора для обозначения каждого уникального экземпляра приложения. Задать идентификатор можно с помощью следующего свойства конфигурации:

```
group.instance.id = app-1 ❶
```

❶ В этом случае задается идентификатор **app-1**. Чтобы запустить другой экземпляр, мы могли бы назначить ему другой уникальный идентификатор (например, **app-2**). Этот идентификатор должен быть уникальным в пределах всего кластера, даже среди разных приложений Kafka Streams, независимо от **application.id** и других потребителей, а также независимо от **group.id**.

Жестко заданный идентификатор экземпляра обычно используется в сочетании с увеличением времени таймаута сеанса¹, чтобы дать приложению больше времени для перезапуска, в течение которого координатор не будет считать экземпляр-потребитель вышедшим из строя при отключении на короткий период.

Статическое членство доступно только в версиях Kafka ≥ 2.3 , поэтому, если ваш клиент или брокеры используют более старую версию, вам придется сначала обновить версию библиотеки. Однако имейте в виду, что увеличение времени

¹ Здесь имеется в виду конфигурационное свойство потребителя `session.timeout.ms`.

таймаута сеанса — палка о двух концах. Это может помешать координатору ошибочно предположить, что экземпляр вышел из строя, когда он всего лишь отключился на короткое время, но, с другой стороны, может привести к существенному запаздыванию в обнаружении фактических сбоев.

Теперь, узнав, как использовать статическое членство, чтобы избежать ненужных перебалансировок, посмотрим, как ослабить отрицательное влияние перебалансировки, когда она случается. И снова Kafka Streams автоматически предпринимает некоторые меры и предлагает средства ручного воздействия, помогающие уменьшить это влияние. Обсудим их в следующих разделах.

Уменьшение влияния перебалансировки

Статическое членство позволяет избежать ненужных перебалансировок, но иногда перебалансировки неизбежны. В конце концов, распределенные системы по своей природе предполагают возможность сбоя. Перебалансировка всегда была очень дорогостоящей процедурой, потому что в начале каждого раунда перебалансировки все клиенты возвращали все свои ресурсы. Мы видели это на рис. 6.2. Более подробное представление этого шага перебалансировки показано на рис. 6.3.

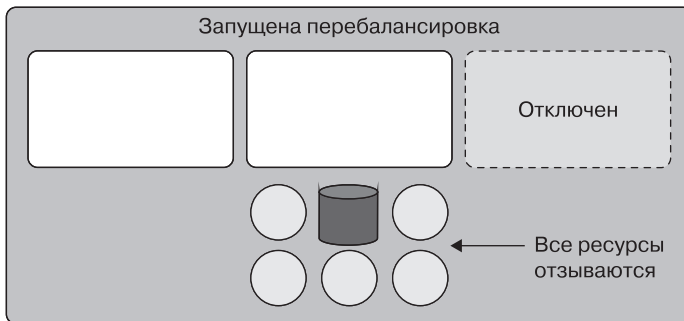


Рис. 6.3. Отзыв всех ресурсов в начале перебалансировки — крайне неэффективный шаг

Такая стратегия перебалансировки называется *жадной перебалансировкой* (eager rebalancing), и она очень негативно воздействует на происходящее по двум причинам:

- так называемый эффект остановки мира, когда все клиенты возвращают свои ресурсы, из-за чего приложение может очень быстро отстать, в связи с полной остановкой обработки данных;
- если задача с состоянием переносится на новый экземпляр, то перед началом обработки необходимо воссоздать состояние до перерыва, что приводит к увеличению времени простоя.

Напомним, что Kafka Streams помогает смягчить вторую проблему (миграции задач с состоянием), предлагая настраиваемый механизм закрепленного назначения разделов. Исторически сложилось, что она полагается на свою собственную реализацию привязки задач, а не на протокол перебалансировки. Но в версии 2.4 в протокол перебалансировки были добавлены дополнительные меры, помогающие уменьшить отрицательное влияние перебалансировок. Мы обсудим этот новый протокол, называемый *пошаговой кооперативной перебалансировкой* (incremental cooperative rebalancing), в следующем разделе.

Пошаговая кооперативная перебалансировка

Пошаговая кооперативная перебалансировка — протокол перебалансировки более эффективный, чем жадная перебалансировка. Она включена по умолчанию в версиях ≥ 2.4 . Если вы используете более старую версию Kafka Streams, ее нужно обновить, чтобы воспользоваться преимуществами этой функции, перечисленными ниже:

- один глобальный раунд перебалансировки заменяется несколькими меньшими (*пошаговыми*) раундами;
- клиенты удерживают ресурсы (задачи), которым не нужно менять владельца, и прекращают обработку только тех задач, которые переносятся (*кооперация*).

На рис. 6.4 показана работа пошаговой кооперативной перебалансировки, когда экземпляр приложения отключается на длительное время, то есть на интервал, превышающий значение конфигурационного свойства `session.timeout.ms`, которое используется для обнаружения отказов потребителей.

Как можно видеть, исправным экземплярам приложения (в том числе экземпляру, выполняющему задачу с состоянием) не нужно возвращать свои ресурсы при запуске перебалансировки. Это огромное улучшение по сравнению со стратегией жадной перебалансировки, потому что приложение может продолжить работу во время перебалансировки.

Мы не будем рассматривать пошаговую кооперативную перебалансировку во всех деталях, но важно знать, что новые версии Kafka Streams реализуют этот протокол по умолчанию, поэтому вам достаточно убедиться, что вы используете соответствующую версию Kafka Streams¹.

¹ Версии ≥ 2.4 используют улучшенную стратегию перебалансировки. Дополнительную информацию о пошаговой кооперативной перебалансировке можно найти по адресу <https://oreil.ly/P3iVG>.

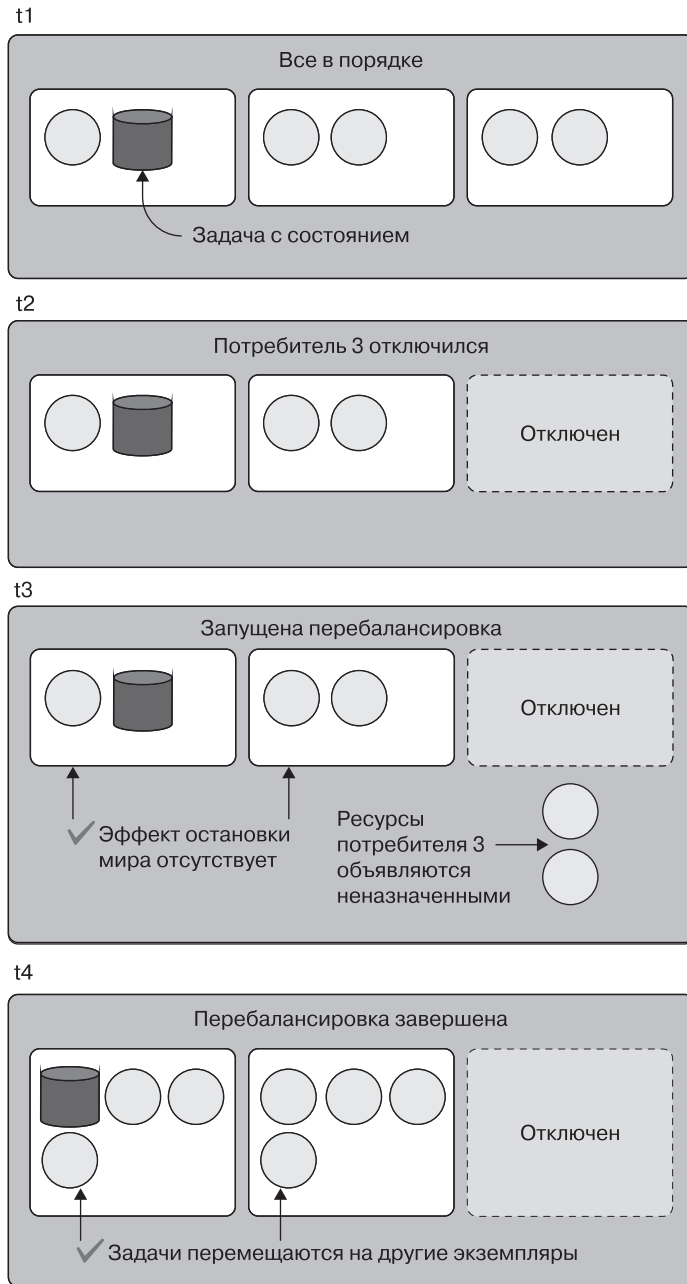


Рис. 6.4. Пошаговая кооперативная перебалансировка помогает избежать эффекта остановки мира

Теперь, узнав, что пошаговая кооперативная перебалансировка может помочь ослабить отрицательное влияние перебалансировок, посмотрим, что может предпринять разработчик приложений, чтобы сделать перебалансировку менее болезненной.

Управление размером состояния

При неосторожном обращении с хранилищами состояний они могут неограниченно расти и вызывать проблемы в работе. Например, представьте, что пространство ключей в сжатой теме журнала изменений очень велико (скажем, один миллиард ключей), а состояние приложения равномерно распределено по десяти физическим узлам¹. Если один из узлов отключится, то потребуется воспроизвести минимум 100 миллионов записей, чтобы воссоздать состояние. Это занимает много времени и может привести к проблемам с доступностью. Нецелесообразно также использовать вычислительные или дисковые ресурсы для хранения большего, чем нужно, количества данных.

В зависимости от варианта использования может и не понадобиться сохранять все состояние приложения на неопределенный срок. Возможно также, что каждая запись имеет ограниченный срок действия. Например, в Mailchimp мы следим за количеством активных сообщений электронной почты, которые еще не были доставлены, и выполняем агрегирование по ключу, чтобы предоставить статистику об этих сообщениях. Однако со временем сообщения электронной почты становятся неактивными (то есть они доставляются или возвращаются), и их больше не нужно отслеживать. В подобных сценариях требуется активная чистка хранилищ состояний. Удаление ненужных данных для уменьшения размера хранилища состояний значительно снижает отрицательное влияние перебалансировки. Если потребуется перенести задачу с состоянием, то гораздо проще воссоздать небольшое хранилище, чем избыточно большое. Но как удалить ненужное состояние в Kafka Streams? Мы используем *надгробия* (tombstones) — маркеры удаления.

Надгробия

Надгробия — это специальные записи, указывающие, что какое-то состояние нужно удалить. Их иногда называют маркерами удаления, и они всегда имеют ключ и пустое значение (`null`). Как упоминалось выше, хранилища состояний основаны на ключах, поэтому ключ в маркере удаления указывает, какую запись нужно удалить из хранилища состояний.

¹ Маловероятно, что в реальности пространство ключей будет разделено совершенно равномерно, но такое представление удобно для обсуждения и не меняет сути.

Ниже показан пример создания маркера удаления. В этом гипотетическом сценарии выполняется агрегирование некоторых данных о пациентах в больнице, но после выписки пациента дальнейших событий, связанных с ним, не ожидается, поэтому его данные удаляются из хранилища состояний:

```
StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream("patient-events");
```

```
stream
    .groupByKey()
    .reduce(
        (value1, value2) -> {
            if (value2.equals(PATIENT_CHECKED_OUT)) {
                // создать маркер удаления
                return null; ❶
            }
            return doSomething(value1, value2); ❷
        });
```

❶ Вернуть `null`, чтобы создать маркер удаления, когда пациент выписывается из больницы. Это повлечет удаление связанного ключа из хранилища состояний.

❷ Если текущее событие не связано с выпиской пациента, выполнить логику агрегирования.

Маркеры удаления помогают ограничить размеры хранилищ «ключ — значение», но есть еще один метод удаления ненужных данных из оконных хранилищ. Мы обсудим его в следующем разделе.

Сохранение окон

Оконные хранилища имеют настраиваемый период хранения, позволяющий ограничить размер хранилищ состояний. Например, в предыдущей главе мы создали оконное хранилище для приложения мониторинга пациентов, с помощью которого преобразовывали исходные события сердцебиения в частоту сердечных сокращений. Вот соответствующий код:

```
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")) ❶
        .suppress(
            Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull()));
```

❶ Материализовать оконное хранилище с ограниченным сроком хранения по умолчанию (в данном случае один день, потому что это значение не переопределяется явно).

Класс `Materialized` имеет метод `withRetention`, вызовом которого можно указать, как долго библиотека Kafka Streams должна хранить записи в оконном хранилище. В следующем примере показано, как это сделать:

```
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))) ❶
        .suppress(
            Suppressed.untilWindowCloses(BufferConfig.unbounded().shutDownWhenFull()));
```

❶ Материализовать оконное хранилище с продолжительностью хранения шесть часов.

Обратите внимание, что срок хранения должен быть больше размера окна *и* периода отсрочки, вместе взятых. В предыдущем примере срок хранения должен быть больше 65 секунд (60 секунд — размер окна + 5 секунд — период отсрочки). По умолчанию срок хранения окна составляет один день. Очевидно, что уменьшение этого значения может уменьшить размер оконного хранилища состояния (и соответствующих разделов журналов изменений) и сократить время восстановления.

Мы обсудили два метода, способствующих уменьшению размеров хранилищ состояний (а именно, создание маркеров удаления и ограничение периода хранения оконных хранилищ). Теперь рассмотрим еще один метод, помогающий уменьшить размеры тем журналов изменений: агрессивное сжатие тем.

Агрессивное сжатие темы

Темы журналов изменений по умолчанию сжимаются, то есть для каждого ключа сохраняется только последнее значение, а при добавлении маркера удаления значение соответствующего ключа полностью удаляется. Однако даже притом, что хранилище состояний немедленно отражает факт сжатия или удаления значения, содержащая его тема может иметь размер больше, чем необходимо, из-за сохранения несжатых/удаленных значений в течение длительного времени.

Причина такого поведения обусловлена особенностями представления тем Kafka на диске. Мы уже говорили, что темы разбиваются на разделы и эти разделы превращаются в единицы работы в Kafka Streams. Разделы — самая низкоуровневая абстракция темы, с которой мы обычно имеем дело на стороне приложения (мы рассуждаем в терминах разделов, пытаемся определить необходимое количество потоков выполнения, а также выясняя, как данные должны маршрутизироваться или использоваться совместно с другими данными), однако на стороне брокера Kafka есть еще более низкоуровневая абстракция: *сегменты*.

Сегменты — это файлы, содержащие подмножество сообщений для данного раздела темы. В любой момент времени всегда есть активный сегмент, или файл, куда в настоящий момент записываются данные соответствующего раздела. Со временем активные сегменты достигают предельного размера и становятся неактивными. И только когда сегмент неактивен, он может быть очищен.



Несжатые записи иногда называют *грязными*. Чистка журналов — это процесс сжатия грязных журналов, который увеличивает объем дискового пространства, доступного брокерам и клиентам Kafka Streams, за счет уменьшения количества записей, воспроизводимых при восстановлении хранилища состояний.

Поскольку активный сегмент не подвергается чистке и, следовательно, может включать большое количество несжатых записей и маркеров удаления, которые необходимо будет воспроизвести при инициализации хранилища состояний, иногда полезно уменьшить размер сегмента, чтобы обеспечить более агрессивное сжатие тем¹. Кроме того, чистка журнала также не будет выполняться, если более 50 % журнала уже очищено/сжато. Этот параметр тоже настраивается, и его значение можно скорректировать в сторону увеличения частоты чистки журнала.

В табл. 6.1 перечислены конфигурационные параметры, помогающие настроить более агрессивное сжатие и уменьшить количество записей, которые понадобится воспроизвести при повторной инициализации хранилища состояний².

¹ Дополнительные сведения можно найти в статье Левани Кохреидзе (Levani Kokhreidze) *Achieving High Availability with Stateful Kafka Streams Applications* (<https://oreil.ly/ZR4A6>).

² Описание параметров взято из официальной документации Kafka.

Таблица 6.1. Конфигурационные параметры для настройки более частой очистки/сжатия журнала

Параметр	Значение по умолчанию	Описание
<code>segment.bytes</code>	1 073 741 824 (1 Гбайт)	Этот параметр управляет размером файла сегмента для журнала. Очистка файлов сегментов всегда выполняется по одному, поэтому, чем больше размер сегмента, тем меньше количество файлов и тем дольше сохраняются записи, подлежащие сжатию/удалению
<code>segment.ms</code>	604 800 000 (7 дней)	Этот параметр управляет периодом времени, по истечении которого Kafka принудительно закрывает сегмент, даже если размер файла еще не достиг предельного размера, чтобы быстрее удалить или сжать старые данные
<code>min.cleanable.dirty.ratio</code>	0,5	Этот параметр определяет, как часто должно выполняться сжатие журнала (при условии, что оно включено). По умолчанию чистка журнала не производится, если более 50 % журнала сжато. Это соотношение ограничивает пространство, занимаемое дубликатами в журнале (при значении по умолчанию дубликаты могут содержаться не более чем в половине журнала). Большее значение коэффициента означает менее эффективную чистку и больший объем впустую расходуемого места в журнале. Если также определен параметр <code>max.compaction.lag.ms</code> или <code>min.compaction.lag.ms</code> , то процедура сжатия будет считать журнал подлежащим сжатию, при достижении им порога степени загрязнения (данный параметр) и при наличии в журнале грязных (несжатых) записей: 1) хранящихся не менее <code>min.compaction.lag.ms</code> или 2) более <code>max.compaction.lag.ms</code>
<code>max.compaction.lag.ms</code>	<code>Long.MAX_VALUE - 1</code>	Максимальное время, в течение которого сообщение может оставаться несжатым. Применимо только для сжимаемых журналов
<code>min.compaction.lag.ms</code>	0	Минимальное время, в течение которого сообщение должно оставаться несжатым. Применимо только для сжимаемых журналов

Ниже показан пример изменения двух из этих параметров в материализованном хранилище. С помощью этих параметров можно настроить более частую чистку журнала путем уменьшения размера сегмента и минимального значения коэффициента загрязнения, при котором должен происходить запуск чистки:


```

Map<String, String> topicConfigs = new HashMap<>();
topicConfigs.put("segment.bytes", "536870912"); ❶
topicConfigs.put("min.cleanable.dirty.ratio", "0.3"); ❷

StreamsBuilder builder = new StreamsBuilder();
KStream<byte[], String> stream = builder.stream("patient-events");

KTable<byte[], Long> counts =
    stream
        .groupByKey()
        .count(
            Materialized.<byte[], Long, KeyValueStore<Bytes, byte[]>>as("counts")
                .withKeySerde(Serdes.ByteArray())
                .withValueSerde(Serdes.Long())
                .withLoggingEnabled(topicConfigs));

```

❶ Уменьшить размер сегмента до 512 Мбайт.

❷ Уменьшить коэффициент загрязнения для запуска чистки до 30 %.

Другой подход к проблеме уменьшения размера хранилища состояний — использовать структуру данных фиксированного размера. Этот подход имеет некоторые недостатки, однако Kafka Streams позволяет решить проблему ограничения размеров хранилищ состояний по этому сценарию. Мы обсудим его далее.

Кэш LRU фиксированного размера

Менее распространенный метод предотвращения неограниченного роста хранилищ состояний — использование кэша LRU (Least Recently Used — «наиболее давно использовавшийся») в памяти. Это простое хранилище пар «ключ — значение» с настраиваемой фиксированной емкостью (максимальным количеством записей), которое автоматически удаляет наиболее давно использовавшиеся записи, когда объем состояния превышает заданный размер. Кроме того, при удалении записи из хранилища в памяти в тему журнала изменений автоматически отправляется соответствующий маркер удаления.

Вот пример использования кэша LRU в памяти:

```

KeyValueBytesStoreSupplier storeSupplier = Stores.lruMap("counts", 10); ❶

StreamsBuilder builder = new StreamsBuilder();
KStream<String, String> stream = builder.stream("patient-events");

stream
    .groupByKey()
    .count(
        Materialized.<String, Long>as(storeSupplier) ❷
            .withKeySerde(Serdes.String())
            .withValueSerde(Serdes.Long()));

return builder.build();

```

❶ Создать кэш LRU с именем `counts` для хранилища в памяти, способного вместить десять записей.

❷ Материализовать хранилище, используя кэш.

Чтобы добиться того же эффекта с применением Processor API, можно воспользоваться построителем, как показано ниже:

```
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);
```

Как отмечалось выше, в пункте «Агрессивное сжатие темы», сжатие и удаление в теме журнала изменений, поддерживающей кэш LRU, происходят не немедленно. Поэтому в случае сбоя время восстановления может увеличиться, по сравнению со случаем использования постоянного хранилища состояний, из-за необходимости воспроизвести всю тему (даже в том случае, когда хранилище хранит не более десяти записей в кэше LRU!). Это серьезный недостаток хранилищ в памяти, который мы уже обсудили в подразделе «Постоянные хранилища и хранилища в оперативной памяти» главы 4. Их следует использовать с учетом всех достоинств и недостатков.

На этом мы завершаем обсуждение освобождения от ненужных записей хранилищ состояний и лежащих в их основе тем журналов. Далее рассмотрим стратегию, которой можно следовать, если окажется, что хранилища состояний страдают большой задержкой чтения или записи.

Исключение повторных операций записи с помощью кэширования

Как отмечалось в подразделе «Подавление» главы 5, существуют методы DSL (в частности, `suppress` в сочетании с параметрами настройки буфера; см. табл. 5.2), ограничивающие частоту обновления оконного хранилища. Имеются также оперативные параметры¹ для управления частотой записи обновлений в хранилища состояний и в нижестоящие узлы. Они перечислены в табл. 6.2.

¹ Различие между этими оперативными параметрами и приемами на основе бизнес-логики, предлагаемой методом `suppress`, обсуждается в статье *Watermarks, Tables, Event Time and the Dataflow Model* Эно Теребка (Eno Thereska) и его коллег в блоге Confluent (<https://oreil.ly/MTN-R>).

Таблица 6.2. Параметры настройки тем, уменьшающие количество операций записи в хранилище состояний и нижестоящие узлы

Параметр настройки	Свойство <code>StreamsConfig</code>	Значение по умолчанию	Описание
<code>cache.max.bytes.buffering</code>	<code>CACHE_MAX_BYTES_BUFFERING_CONFIG</code>	1 048 576 (10 Мбайт)	Максимальный объем памяти, выделяемой для буферизации для всех потоков выполнения, в байтах
<code>commit.interval.ms</code>	<code>COMMIT_INTERVAL_MS_CONFIG</code>	30 000 (30 с)	Частота сохранения позиции обработчика

Большой размер кэша и более долгий интервал сохранения могут помочь уменьшить количество повторных операций записи при обновлении одного и того же ключа. Это дает несколько преимуществ, в том числе:

- уменьшение задержки чтения;
- уменьшение объема записываемых данных:
 - в хранилища состояний;
 - соответствующие темы журналов изменений (если включены);
 - нижестоящие узлы.

Поэтому, если узким местом являются операции чтения/записи с хранилищем состояний или сетевой ввод/вывод (который может быть побочным результатом частых обновлений темы журнала изменений), следует подумать о настройке этих параметров. Конечно, кэш большого размера имеет свои недостатки:

- занимает больше памяти;
- дает большую задержку (записи выводятся в хранилище реже).

Что касается первого пункта, объем памяти, заданный параметром `cache.max.bytes.buffering`, распределяется между всеми потоками выполнения. Пул памяти будет делиться равномерно, поэтому потоки, обрабатывающие горячие разделы (то есть разделы с относительно большим объемом данных по сравнению с другими), будут выталкивать свой кэш чаще. Независимо от размера кэша или интервала записи окончательные вычисления с состоянием будут одинаковыми.

Большие интервалы записи имеют еще один недостаток — после сбоя придется проделать работы тем больше, чем больше будет значение этого параметра.

Наконец, порой желательно наблюдать промежуточные состояния без всякого кэширования. На самом деле иногда плохо знакомые с `Kafka Streams` разработчики, наблюдая отсутствие повторяющихся операций записи или задержку выталкивания кэша, думают, что что-то не так с их топологией. Пытаясь отладить

мнимую проблему, они создают определенное количество записей в исходящей теме и наблюдают, как в хранилище сбрасывается только часть изменений состояния (возможно, с задержкой в несколько секунд, а не сразу). Они полностью отключают кэширование и получают уменьшение интервала записи в окружении разработки. Но будьте осторожны, не поступайте так в промышленных окружениях, потому что может пострадать производительность.

Мониторинг хранилища состояний

Перед развертыванием приложения в промышленном окружении важно убедиться в доступности всех его аспектов для наблюдения. В этом разделе мы обсудим основные подходы к мониторингу приложений с хранимым состоянием, помогающие меньшими усилиями получать больше информации, необходимой для отладки при появлении ошибок.

Обработка событий изменения состояния

Приложение Kafka Streams может находиться в одном из множества состояний (не путайте с хранимыми состояниями). На рис. 6.5 показаны некоторые из этих состояний и переходы между ними.

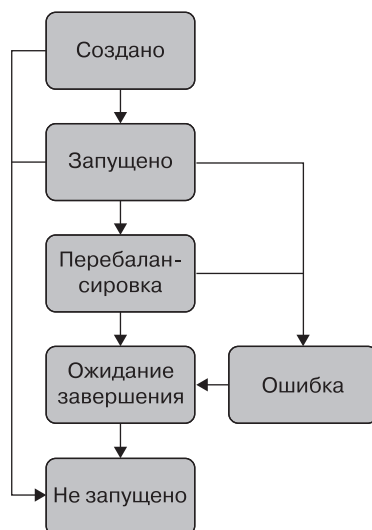


Рис. 6.5. Состояния приложения Kafka Streams и переходы между ними

Как упоминалось выше, состояние перебалансировки оказывает, пожалуй, наибольшее влияние на приложения Kafka Streams с хранимым состоянием, поэтому возможность отмечать моменты, когда приложение переходит в состояние перебалансировки, и отслеживать, как часто это происходит, может быть полезна для мониторинга. К счастью, Kafka Streams позволяет мониторить изменения состояния приложения с помощью *обработчика событий изменения состояния* (State Listener). Обработчик событий изменения состояния (или просто прослушиватель) — это всего лишь метод обратного вызова, к которому происходит автоматическое обращение при изменении состояния приложения.

В зависимости от логики работы приложение может предпринять некоторые действия при переходе в состояние перебалансировки. Например, в Mailchimp мы создаем специальную метрику в наших приложениях Kafka Streams, которая увеличивается, когда запускается перебалансировка. Эта метрика передается в нашу систему мониторинга (Prometheus), где ее можно запросить или использовать для создания оповещений.

Следующий пример показывает, как добавить в топологию Kafka Streams обработчик, определяющий переход в состояние перебалансировки:

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

streams.setStateListener( ❶
    (oldState, newState) -> { ❷
        if (newState.equals(State.REBALANCING)) { ❸
            // сделать что-то
        }
    });
```

❶ Установить обработчик событий изменения состояния приложения с помощью метода `KafkaStreams.setStateListener`.

❷ Сигнатура метода-обработчика класса `StateListener` подсказывает, что он получает как старое, так и новое состояние.

❸ Выполнение некоторого действия, когда приложение переходит в состояние перебалансировки.

Обработчики событий изменения состояния очень полезны, но это не единственный инструмент, доступный в приложениях Kafka Streams. В следующем разделе демонстрируется другой метод, помогающий улучшить наблюдаемость приложений с хранимым состоянием.

Обработка событий восстановления хранимого состояния

В предыдущем разделе вы узнали, как обрабатывать переход приложения Kafka Streams в состояние перебалансировки. Однако событие перебалансировки представляет интерес, только когда оно вызывает повторную инициализацию хранилища состояний. Kafka Streams поддерживает еще один обработчик — *обработчик события восстановления хранимого состояния* (State Restore Listener), вызываемый при повторной инициализации хранилища состояний. В следующем фрагменте показано, как добавить такой обработчик в приложение Kafka Streams:

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

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

Определение класса `MyRestoreListener`, наследующего `StateRestoreListener`, представлено в следующем фрагменте кода. В отличие от обработки событий изменения состояния, описанной в предыдущем разделе, обработка событий восстановления состояния требует реализации трех методов, каждый из которых связан с определенным отрезком жизненного цикла процесса восстановления состояния. Комментарии, следующие за фрагментом кода ниже, описывают, для чего используется каждый метод:

```
class MyRestoreListener implements StateRestoreListener {

    private static final Logger log =
        LoggerFactory.getLogger(MyRestoreListener.class);

    @Override
    public void onRestoreStart( ❶
        TopicPartition topicPartition,
        String storeName,
        long startingOffset,
        long endingOffset) {

        log.info("The following state store is being restored: {}", storeName);
    }

    @Override
    public void onRestoreEnd( ❷
        TopicPartition topicPartition,
        String storeName,
        long totalRestored) {

        log.info("Restore complete for the following state store: {}", storeName);
    }
}
```

```
@Override
public void onBatchRestored( ❸
    TopicPartition topicPartition,
    String storeName,
    long batchEndOffset,
    long numRestored) {

    // вызывается очень часто, поэтому журналирование не производится
}
}
```

❶ Метод `onRestoreStart` вызывается перед началом повторной инициализации состояния. Параметр `startOffset` представляет особый интерес: он указывает на необходимость воспроизведения всего состояния целиком (это наиболее затратный вид повторной инициализации, который случается при использовании хранилищ в памяти или при использовании постоянных хранилищ и потере предыдущего состояния). Значение `0` в `startOffset` говорит о необходимости полной повторной инициализации. Значение больше `0` сообщает о необходимости только частичного восстановления.

❷ Метод `onRestoreEnd` вызывается после завершения восстановления хранилища.

❸ Метод `onBatchRestored` вызывается всякий раз, когда восстанавливается один пакет записей. Максимальный размер пакета определяется конфигурационным параметром `MAX_POLL_RECORDS`. Этот метод может вызываться много раз, поэтому будьте осторожны, выполняя любую синхронную обработку в нем, потому что это может замедлить процесс восстановления. Обычно я ничего не делаю в этом методе, даже отказываюсь от журналирования, чтобы не захламлять журнал множеством малоинтересных записей.

Встроенные метрики

Основное обсуждение мониторинга приложений Kafka Streams мы отложим до главы 12 (см. раздел «Мониторинг»). Однако важно отметить, что Kafka Streams включает набор встроенных метрик JMX, многие из которых характеризуют работу хранилищ состояний.

Например, есть метрики, характеризующие частоту выполнения определенных операций и запросов (такие как `get`, `put`, `delete`, `all`, `range`), среднее и максимальное время выполнения этих операций и размер буфера подавления. Существует также ряд метрик, поддерживаемых хранилищами на основе RocksDB, в числе которых можно назвать `bytes-written-rate` и `bytes-read-rate` — особенно полезные при исследовании трафика ввода/вывода на уровне байтов.

Полный перечень метрик можно найти в документации по мониторингу Confluent (<https://oreil.ly/vpBn0>). На практике я обычно использую высокоуровневые метрики (например, характеризующие запаздывание потребителей) для оповещения, но и низкоуровневые метрики могут пригодиться для устранения неполадок.

Интерактивные запросы

До версии Kafka Streams 2.5 перебалансировка была особенно болезненной в применении к приложениям, обеспечивающим доступность своего хранимого состояния для интерактивных запросов. В этих старых версиях библиотеки отключение или перебалансировка разделов приводили к сбою интерактивных запросов. Даже обычная перебалансировка (например, вызванная простым обновлением) могла привести к проблемам с доступностью, что стало в свое время препятствием для микросервисов, требующих высокой доступности.

Однако, начиная с Kafka Streams 2.5 (<https://oreil.ly/1nY9t>), появилась возможность использовать резервные реплики для возврата устаревших результатов, пока не закончилась инициализация только что перенесенного хранилища состояний. Она обеспечивает высокую доступность API, даже когда приложение входит в состояние перебалансировки. Вспомните, как в примере 4.11 мы научились извлекать метаданные для заданного ключа в нашем хранилище состояний. В том начальном примере мы извлекли *активный экземпляр Kafka Streams*:

```
KeyQueryMetadata metadata =  
    streams.queryMetadataForKey(storeName, key, Serdes.String().serializer()); ❶  
  
String remoteHost = metadata.activeHost().host(); ❷  
int remotePort = metadata.activeHost().port(); ❸
```

❶ Получить для указанного ключа метаданные, включающие имя хоста и номер порта, на котором должен быть доступен конкретный ключ, если он существует.

❷ Извлечь имя хоста активного экземпляра Kafka Streams.

❸ Извлечь номер порта активного экземпляра Kafka Streams.

Начиная с версии 2.5, можно получить резервные хосты:

```
KeyQueryMetadata metadata =  
    streams.queryMetadataForKey(storeName, key, Serdes.String().serializer()); ❶  
  
if (isAlive(metadata.activeHost())) { ❷  
    // передать запрос активному хосту
```



```
} else {  
    // передать запрос резервным хостам  
    Set<HostInfo> standbys = metadata.standbyHosts(); ❸  
}
```

❶ Вызвать метод `KafkaStreams.queryMetadataForKey`, чтобы получить активный и резервный хосты для данного ключа.

❷ Проверить доступность активного хоста. Эту проверку вам придется реализовать самостоятельно, но в общем случае можно добавить обработчик событий изменения состояния (как описано в подразделе «Обработка событий изменения состояния» выше) и затем использовать соответствующую конечную точку API на вашем сервере RPC для получения текущего состояния приложения. Метод `isAlive` должен возвращать значение `true`, когда приложение находится в состоянии *выполнения*.

❸ Если активный хост недоступен, то извлечь резервные хосты, чтобы можно было послать запрос одной из копий хранилища состояний. Имейте в виду, что, если резервные копии не настроены, этот метод вернет пустое множество.

Как видите, возможность передавать запросы резервным копиям обеспечивает доступность приложения, даже когда активный экземпляр не работает или не может обслуживать запросы. На этом мы завершаем обсуждение способов смягчения последствий перебалансировки. Далее перейдем к обсуждению нестандартных хранилищ состояний.

Нестандартные хранилища состояний

При желании можно создать свое, нестандартное хранилище состояний. Для этого нужно реализовать интерфейс `StateStore` напрямую или опосредованно, создав один из интерфейсов более высокого уровня, таких как `KeyValueStore`, `WindowStore` или `SessionStore`, добавляющих дополнительные методы, характерные для конкретного применения хранилища¹.

Кроме `StateStore`, также потребуется реализовать интерфейс `StoreSupplier` и определить логику создания новых экземпляров хранилища состояний. Поскольку производительность встроенных хранилищ состояний на основе RocksDB едва ли удастся превзойти, обычно нет необходимости тратить время на решение сложной и подверженной ошибкам задачи реализации собственного хранилища. По этой причине, а также учитывая огромный объем кода, который

¹ Например, кроме всего прочего, интерфейс `KeyValueStore` добавляет метод `void put(K key, V value)`, поскольку предполагает, что хранилищу потребуется записывать пары «ключ — значение» с помощью базового механизма хранения.

придется написать, чтобы получить даже очень простое хранилище, я просто дам вам ссылку на один из немногих примеров нестандартных хранилищ на GitHub: <https://oreil.ly/pZf9g>.

Если вы все же решитесь реализовать свое хранилище, то имейте в виду, что любое решение для хранения данных, обслуживающее сетевые запросы, может сильно влиять на производительность. Одна из причин, почему RocksDB или локальное хранилище в памяти считаются хорошим выбором, заключается в размещении их поблизости от задачи потоковой обработки. Конечно, ваша ситуация может отличаться, в зависимости от требований проекта, поэтому обязательно заранее определите целевую производительность и выберите подходящее хранилище состояний.

Заключение

Теперь у вас должно сложиться более полное представление об особенностях управления хранилищами состояний в Kafka Streams и вариантах, доступных вам как разработчику. Вы должны уметь определять, какие варианты помогут обеспечить бесперебойную работу ваших приложений с состоянием. К ним относятся маркеры удаления, агрессивное сжатие тем и другие способы удаления старых данных из хранилищ состояний (и сокращения времени повторной инициализации состояния). С помощью резервных копий можно сократить время обработки отказа в задачах с состоянием и обеспечить высокую доступность при перебалансировке. Наконец, в некоторых случаях можно избежать перебалансировки с помощью статического членства, а ее негативное влияние ослабить, используя версию Kafka Streams, поддерживающую улучшенный протокол пошаговой кооперативной перебалансировки.

Processor API

Совсем недавно мы отправились в путешествие по миру Kafka Streams. Начали с высокоуровневого предметного языка (DSL) Kafka Streams, позволяющего создавать приложения потоковой обработки, используя удобный функциональный интерфейс, а также комбинировать и объединять функции потоковой обработки с использованием встроенных операторов (таких как `filter`, `flatMap`, `groupBy` и т. д.) и абстракций (`KStream`, `KTable`, `GlobalKTable`).

В этой главе мы исследуем низкоуровневый API, доступный в Kafka Streams, — Processor API (иногда его называют PAPI). Processor API менее абстрагированный, чем высокоуровневый DSL, он предлагает императивный стиль программирования, пусть и требующий описания мелких деталей, зато более мощный и гибкий, позволяющий определять порядок прохождения данных через наши топологии, связи между узлами-обработчиками потоков, особенности создания и поддержки состояния и даже способы синхронизации тех или иных операций.

Вот вопросы, на которые вы получите ответы в этой главе.

- Когда использовать Processor API?
- Как с помощью Processor API добавить узлы источников, приемников и обработчиков?
- Как запланировать периодическое выполнение действий?
- Можно ли комбинировать Processor API и DSL?
- В чем разница между обработчиками и преобразователями?

Как обычно, мы исследуем основы API в практическом примере и попутно ответим на вопросы, перечисленные выше. Однако прежде чем говорить о том, *как* использовать Processor API, перечислю случаи, *когда* следует его использовать.

Когда использовать Processor API

Разрабатывая приложение потоковой обработки, важно выбрать правильный уровень абстракции. Для всякого усложнения проекта должна быть веская причина. Конечно, Processor API не особенно сложен, но его низкоуровневый характер (по сравнению с DSL и `ksqlDB`) и меньшее количество абстракций могут привести к увеличению объема кода и, если не проявить осторожность, к большему количеству ошибок.

В общем случае Processor API следует использовать, когда есть потребность в одном из его преимуществ, а их перечень таков:

- доступ к метаданным записей (тема, раздел, смещение, заголовки и т. д.);
- возможность планировать периодическое выполнение действий;
- более полный контроль над пересылкой записей нижестоящим узлам-обработчикам;
- более детальный доступ к хранилищам состояний;
- возможность обхода любых ограничений, возникающих в DSL (позже мы увидим пример).

С другой стороны, Processor API имеет некоторые недостатки, в том числе:

- более подробный и менее удобочитаемый код, требующий больше усилий для обслуживания;
- более высокие требования к новым сотрудникам, присоединяющимся к проекту;
- больше подводных камней, в их числе случайное повторение функций или абстракций DSL, необычное формулирование задач¹ и ловушки, ухудшающие производительность².

К счастью, Kafka Streams позволяет смешивать DSL и Processor API, поэтому вам не придется выбирать один из вариантов. Вы сможете использовать DSL для простых и стандартных операций, а Processor API — для более сложных

¹ DSL обладает мощными выразительными средствами в виде встроенных операторов, упрощающих формулировку задач. Потеря части этой выразительности и самих операторов может сделать формулировку проблем менее стандартной и усложнить описание решений.

² Например, слишком агрессивная фиксация или применение методов доступа к хранилищам состояний, способных повлиять на производительность

или необычных действий, требующих низкоуровневого доступа к контексту обработки, состоянию или метаданным записи. Мы обсудим способы комбинирования DSL и Processor API в конце этой главы, но сначала посмотрим, как реализовать приложение, используя только Processor API.

А теперь обсудим приложение, которое будем создавать в этой главе.

Служба цифровых двойников IoT

В этом проекте мы с помощью Processor API создадим службу *цифрового двойника* для ветряной электростанции. Цифровые двойники (иногда их называют теневыми устройствами) популярны в Интернете вещей (Internet of Things, IoT) и в промышленном Интернете вещей (Industrial IoT, IIoT)¹ и отражают состояние физического объекта в виде цифровой копии. Это отличный вариант использования Kafka Streams, способной принимать и обрабатывать большие объемы данных с датчиков, фиксировать состояние физического объекта с помощью хранилищ состояний и отображать это состояние с помощью интерактивных запросов.

Сформулируем задачу, чтобы получить общее представление о цифровых двойниках (это сделает идею проекта более понятной). Имеется ветряная электростанция с 40 ветряками. Всякий раз, когда один из ветряков сообщает о своем текущем состоянии (скорость ветра, температура, признак «включен/выключен» и т. д.), эта информация сохраняется в хранилище ключей-значений. Ниже показан пример записи с состоянием:

```
{
  "timestamp": "2020-11-23T09:02:00.000Z",
  "wind_speed_mph": 40,
  "temperature_fahrenheit": 60,
  "power": "ON"
}
```

Обратите внимание, что идентификатор устройства передается через ключ записи (например, предыдущая запись может соответствовать устройству

¹ В этой главе мы будем использовать более широкий термин IoT (Интернет вещей), несмотря на применение некоторых двойников в промышленности. В числе ярких примеров цифровых двойников можно назвать Tesla Motors. «Tesla создает цифровой двойник для каждого проданного автомобиля. Затем Tesla обновляет программное обеспечение на основе данных, полученных с датчиков отдельных автомобилей, и загружает обновления в свои продукты». Дополнительную информацию вы найдете по ссылке https://oreil.ly/j6_mj.

с идентификатором `abc123`). Это позволяет различать события изменения состояния для разных устройств.

Взаимодействия с конкретным ветряком¹ осуществляются не напрямую. Устройства IoT часто отключаются, поэтому, чтобы повысить доступность и уменьшить количество ошибок, мы будем взаимодействовать только с цифровой копией (двойником) физического устройства.

Например, если понадобится выключить ветряк, то вместо отправки соответствующего сигнала напрямую устройству мы установим так называемое *желаемое состояние* в цифровой копии. Физическое устройство впоследствии синхронизирует свое состояние (то есть переведет лопасти в нерабочее положение), когда вновь подключится к управляющей сети и после этого — через определенные промежутки времени. Соответственно, запись цифрового двойника должна включать не только *сообщаемое*, но и *желаемое* состояние, поэтому мы будем создавать и использовать записи цифровых двойников, как показано ниже:

```
{
  "desired": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "power": "OFF"
  },
  "reported": {
    "timestamp": "2020-11-23T09:00:01.000Z",
    "windSpeedMph": 68,
    "power": "ON"
  }
}
```

Таким образом, приложение должно принимать поток данных от нескольких ветряков², обрабатывать их и поддерживать информацию о текущем состоянии каждого ветряка в постоянном хранилище «ключ — значение». Эту информацию затем можно будет извлекать с помощью функции интерактивных запросов Kafka Streams.

Мы постараемся избегать технических подробностей, связанных с интерактивными запросами, так как они уже были рассмотрены в предыдущих главах, и представим лишь некоторые дополнительные сведения, касающиеся поддержки интерактивных запросов в IoT.

¹ Под взаимодействием обычно понимается получение последнего состояния устройства или его изменение путем обновления одного из изменяемых свойств состояния, например состояния «включено».

² В IoT данные с датчиков часто передаются по протоколу MQTT. Один из способов передать эти данные в Kafka — использовать коннектор Confluent MQTT Connector (<https://oreil.ly/pTRmb>).

На рис. 7.1 показана топология, которую нам предстоит реализовать в этой главе. Каждый шаг подробно описан ниже.



Рис. 7.1. Топология службы цифровых двойников IoT

❶ Наш кластер содержит две темы, поэтому мы должны научиться добавлять узлы-источники с помощью Processor API. Ниже приведу перечень этих тем.

- Каждый ветряк (конечный узел) оснащен набором датчиков, измеряющих параметры окружающей среды, и эти данные (например, скорость ветра) вместе с некоторыми метаданными о самой турбине (например, состояние «включено/выключено») периодически отправляются в тему `reported-state-events`.
- Запись данных в тему `desired-state-events` происходит всякий раз, когда пользователь или процесс пытается изменить состояние турбины (то есть включить или выключить ее).

❷ Поскольку данные с датчиков передаются через тему `reported-state-events`, мы добавим узел-обработчик, сравнивающий сообщаемую скорость ветра с максимальным безопасным уровнем¹ и автоматически генерирующий сигнал

¹ На практике может использоваться два порога скорости ветра: высокая скорость ветра создает опасные условия эксплуатации, а низкая может не оправдывать накладные расходы.

отключения при его превышении. Для этого мы должны узнать, как с помощью Processor API добавить обработчик потока без хранимого состояния.

❸ Третий шаг делится на два этапа.

- На первом этапе события двух типов (сообщаемые и желаемые) объединяются в так называемую запись цифрового двойника. Эти записи будут обработаны и записаны в постоянное хранилище «ключ — значение» **digital-twin-store**. Реализуя этот шаг, мы покажем, как подключаться к хранилищам состояний и взаимодействовать с ними с помощью Processor API, и как получать определенные метаданные записей, недоступные через DSL.
- На втором этапе происходит планирование периодического выполнения функции, называемой *пунктуатором* (punctuator, или *разделителем*), для удаления старых записей цифровых двойников, не обновлявшихся дольше семи дней. Для реализации этого этапа мы познакомимся с интерфейсом пунктуатора в Processor API, а также исследуем альтернативный метод удаления ключей из хранилищ состояний¹.

❹ Каждая запись цифровых двойников будет передаваться в выходную тему **digital-twins** для целей анализа. На этом шаге вы узнаете, как добавлять узлы-приемники с помощью Processor API.

❺ Доступ к записям цифровых двойников будет предоставляться посредством интерактивных запросов. Каждые несколько секунд микроконтроллер турбины будет пытаться синхронизировать свое состояние с желаемым, полученным из Kafka Streams. Например, если мы сгенерируем сигнал выключения на шаге 2 (который установит желаемое состояние **power=OFF**), то микроконтроллер увидит это желаемое состояние и повернет лопасти в нерабочее положение.

Теперь, получив представление о том, что предстоит создать (и чему нужно будет научиться на каждом этапе), перейдем к настройке проекта.

Настройка проекта

Код для этой главы доступен по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>.

Если вы решите опробовать этот код в процессе обсуждения каждого шага топологии, то клонируйте репозиторий и перейдите в каталог с проектом

¹ Альтернативный метод описан в пункте «Надгробия» главы 6.

для рассматриваемой главы. Для этого достаточно выполнить следующие команды:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-07/digital-twin
```

Собрать проект можно в любой момент командой:

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

Закончив настройку проекта, перейдем к обсуждению реализации приложения цифровых двойников.

Модели данных

Как обычно, прежде чем реализовать топологию, определим модели данных. Примеры записей и определение класса, показанные в табл. 7.1, соответствуют данным в наших входных темах (см. шаг 1 на рис. 7.1).

Обратите внимание, что данные, поступающие через обе темы, для простоты представлены в формате JSON, и оба типа записей реализованы в общем классе `TurbineState`. Для краткости мы опустили функции доступа к членам в классе `TurbineState`.

Таблица 7.1. Примеры записей и классы данных для обеих входных тем

Тема Kafka	Пример записи	Класс данных
reported-state-events	<pre>{ "timestamp": "...", "wind_speed_mph": 40, "power": "ON" }</pre>	<pre>public class TurbineState { private String timestamp; private Double windSpeedMph; public enum Power { ON, OFF } public enum Type { DESIRED, REPORTED } private Power power; private Type type; }</pre>
desired-state-events	<pre>{ "timestamp": "...", "power": "OFF" }</pre>	Тот же класс данных, что и для темы reported-state-events

Как упоминалось в описании проекта, для создания цифрового двойника нужно объединить записи с сообщаемым и желаемым состоянием. Поэтому нам также нужен класс данных, представляющий комбинированные записи. В следующей таблице показана структура комбинированной записи цифрового двойника в формате JSON и соответствующий класс данных.

Пример записи	Класс данных
<pre>{ "desired": { "timestamp": "2020-11-23T09:02:01.000Z", "power": "OFF" }, "reported": { "timestamp": "2020-11-23T09:00:01.000Z", "windSpeedMph": 68, "power": "ON" } }</pre>	<pre>public class DigitalTwin { private TurbineState desired; private TurbineState reported; // методы доступа опущены // для краткости }</pre>

Как видно в примере записи, согласно желаемому состоянию турбина должна быть выключена, а согласно последнему сообщаемому состоянию она включена. В какой-то момент микроконтроллер турбины синхронизирует свое состояние с цифровым двойником и переведет лопасти в нерабочее положение.

Возможно, вам интересно, как производится сериализация и десериализация записей в Processor API по сравнению с высокоуровневым DSL. Другими словами: как на самом деле преобразуются исходные байты записей в темах Kafka в классы данных, показанные в табл. 7.1? В разделе «Сериализация/десериализация» главы 3 рассказывалось о применении классов *Serdes* в DSL — классов-обертки, реализующих сериализацию и десериализацию. Многие операторы DSL, такие как *stream*, *table*, *join* и т. д., принимают экземпляр *Serdes*, и потому они часто встречаются в приложениях, использующих DSL.

Методам в Processor API требуется только базовый сериализатор или десериализатор, который обычно содержит экземпляр *Serdes*. Тем не менее часто бывает удобно определить *Serdes* для классов данных, потому что:

- 1) всегда можно извлечь базовый сериализатор/десериализатор, чтобы удовлетворить требования сигнатуры метода Processor API;
- 2) *Serdes* часто полезны для тестирования.

С учетом сказанного в этом проекте мы используем классы *Serdes*, показанные в примере 7.1.

Пример 7.1. Класс `Serde` для сериализации/десериализации цифровых двойников и записей о состоянии

```
public class JsonSerdes {

    public static Serde<DigitalTwin> DigitalTwin() { ❶
        JsonSerializer<DigitalTwin> serializer = new JsonSerializer<>();
        JsonDeserializer<DigitalTwin> deserializer =
            new JsonDeserializer<>(DigitalTwin.class);

        return Serdes.serdeFrom(serializer, deserializer);
    }

    public static Serde<TurbineState> TurbineState() { ❷
        JsonSerializer<TurbineState> serializer = new JsonSerializer<>();
        JsonDeserializer<TurbineState> deserializer =
            new JsonDeserializer<>(TurbineState.class);

        return Serdes.serdeFrom(serializer, deserializer); ❸
    }
}
```

❶ Метод получения `DigitalTwin` из исходной записи в теме.

❷ Метод получения `TurbineState` из цифрового двойника.

❸ В предыдущих проектах мы реализовали интерфейс `Serde` напрямую (пример в подразделе «Класс `Serde` для твитов» главы 3). Сейчас в рассматриваемом проекте используется альтернативный подход, заключающийся в использовании метода `Serde.serdeFrom` для создания `Serde` из экземпляров сериализатора и десериализатора.

В следующем разделе вы узнаете, как добавлять узлы-источники и десериализовать входные записи с помощью Processor API.

Добавление узлов-источников

Теперь, определив классы данных, можно приступить к реализации шага 1 в нашей топологии (см. рис. 7.1). Для этого нужно добавить два узла-источника для передачи данных из входных тем в приложение Kafka Streams. В примере 7.2 показано, как это сделать с помощью Processor API.

Пример 7.2. Начальная реализация топологии с двумя узлами-источниками

```
Topology builder = new Topology(); ❶

builder.addSource( ❷
    "Desired State Events", ❸
```

```
Serdes.String().deserializer(), ❹  
JsonSerdes.TurbineState().deserializer(), ❺  
"desired-state-events"); ❻  
  
builder.addSource( ❼  
    "Reported State Events",  
    Serdes.String().deserializer(),  
    JsonSerdes.TurbineState().deserializer(),  
    "reported-state-events");
```

❶ Создание экземпляра **Topology** напрямую. Он будет использоваться для добавления и подключения узлов источников, приемников и обработчиков. Обратите внимание, что порядок создания экземпляра **Topology** напрямую отличается от порядка, принятого в DSL, где требуется создать экземпляр объекта **StreamsBuilder**, добавить операторы DSL (например, **map**, **flatMap**, **merge**, **branch** и т. д.) в экземпляр **StreamsBuilder** и, наконец, создать экземпляр **Topology** вызовом метода **StreamsBuilder#build**.

❷ Создание узла-источника вызовом метода **addSource**. Этот метод имеет множество перегруженных версий, включая поддерживающие стратегии сброса смещения, шаблоны тем и др. Поэтому загляните в документацию Javadocs Kafka Streams или в определение класса **Topology**, чтобы выбрать версию **addSource**, лучше всего соответствующую вашим потребностям.

❸ Имя узла-источника. Каждый узел должен иметь уникальное имя, потому что внутри Kafka Streams хранит эти имена в топологически отсортированной карте (именно поэтому все ключи должны быть уникальными). Как будет показано ниже, имена играют важную роль в Processor API, потому что используются для подключения дочерних узлов. И снова мы наблюдаем существенное отличие от порядка, в каком соединения устанавливаются в DSL, где для объявления отношений между узлами не требуется явно определять имена (по умолчанию DSL генерирует их автоматически). Рекомендуется использовать описательное имя, чтобы улучшить читаемость кода.

❹ Десериализатор ключей. Здесь применяется встроенный десериализатор **String**, потому что наши ключи представлены строками. Это еще одно отличие Processor API от DSL. Последний требует передать **Serdes** (объект, содержащий сериализатор и десериализатор записи), тогда как в Processor API требуется только десериализатор (который можно извлечь непосредственно из **Serdes**, как сделано в этом примере).

❺ Десериализатор значений. Здесь используется нестандартный **Serdes** (его можно найти в исходном коде этого проекта) для преобразования записей в объекты **TurbineState**. К этому десериализатору относятся все дополнительные примечания, которые были сделаны в описании десериализатора ключей.

❖ Имя темы, откуда узел-источник извлекает данные.

❗ Добавляется второй узел-источник для темы `reported-state-events`. Я не буду повторно описывать каждый параметр, потому что они соответствуют параметрам предыдущего узла-источника.

Особо следует отметить отсутствие в примере упоминаний о потоках или таблицах. Эти абстракции не существуют в Processor API. Однако с концептуальной точки зрения оба наших узла-источника представляют поток. Это связано с тем, что узлы не имеют хранимого состояния (то есть они не подключены к хранилищу состояний) и, следовательно, не запоминают последнее состояние/представление данного ключа.

Мы увидим табличное представление нашего потока, когда перейдем к шагу 3 в топологии, а пока завершим шаг 1 и посмотрим далее, как добавить узел-обработчик потока, генерирующий сигналы выключения, когда микроконтроллер нашей турбины сообщает о превышении опасной отметки скорости ветра.

Добавление узлов-обработчиков без состояния

Следующий шаг в нашей топологии предполагает автоматическое генерирование сигнала выключения, когда скорость ветра, зарегистрированная данной турбиной, превышает безопасный порог (65 миль в час \approx 105 км/час). Очевидно, нужно научиться добавлять узлы-обработчики потока с помощью Processor API. Для этой цели можно использовать метод `addProcessor`, как показано ниже:

```
builder.addProcessor(
    "High Winds Flatmap Processor", ❶
    HighWindsFlatmapProcessor::new, ❷
    "Reported State Events"); ❸
```

❶ Имя узла-обработчика потока.

❷ Во втором аргументе ожидается экземпляр `ProcessSupplier` — функциональный интерфейс, возвращающий экземпляр `Processor`, который содержит всю логику обработки/преобразования данных для данного узла-обработчика. В следующем разделе мы определим класс `HighWindsFlatmapProcessor`, реализующий интерфейс `Processor`. Поэтому здесь мы можем просто использовать ссылку на конструктор этого класса.

❸ Имена родительских узлов. В этом случае имеется только один родительский узел — узел с именем `"Reported State Events"`, созданный в примере 7.2.

Узлы-обработчики потоков могут подключаться к одному или нескольким родительским узлам.

Всякий раз, добавляя узел-обработчик потока, необходимо реализовать интерфейс `Processor`. Это не функциональный интерфейс (в отличие от `ProcessSupplier`, который мы только что обсудили), поэтому нельзя просто передать лямбда-выражение в метод `addProcessor`, как это часто делается с операторами DSL. Эта процедура немного сложнее и требует больше кода, чем мы привыкли, поэтому в следующем разделе посмотрим, как это осуществить.

Создание узлов без состояния

При вызове метода `addProcessor` из `Processor API`, ему нужно передать реализацию интерфейса `Processor`, содержащую логику обработки и преобразования записей в потоке. Интерфейс имеет три метода:

```
public interface Processor<K, V> { ❶  
  
    void init(ProcessorContext context); ❷  
  
    void process(K key, V value); ❸  
  
    void close(); ❹  
}
```

❶ Обратите внимание, что интерфейс `Processor` имеет два параметра типа: один определяет тип ключа (`K`), а другой — тип значения (`V`). Мы увидим, как их использовать, когда займемся реализацией своего узла-обработчика ниже в этом разделе.

❷ Метод `init` вызывается при создании экземпляра `Processor`. Если узлу необходимо выполнить какие-то действия, связанные с инициализацией, то их следует поместить в этот метод. Параметр `ProcessorContext`, передаваемый методу `init`, содержит множество методов, которые мы рассмотрим в этой главе, и может очень пригодиться.

❸ Метод `process` вызывается после получения узлом новой записи. Он должен содержать логику преобразования/обработки каждой записи. В нашем примере именно сюда мы добавим логику определения превышения скоростью ветра своего порогового значения.

❹ Метод `close` вызывается, когда Kafka Streams завершает работу с этим оператором (например, завершая работу приложения). Обычно в этот метод помещается логика очистки ресурсов, использовавшихся узлом. Однако не сле-

дует пытаться освобождать какие-либо ресурсы, управляемые Kafka Streams, например хранилища состояний, так как эти ресурсы обрабатываются самой библиотекой.

Теперь напишем свою реализацию `Processor`, генерирующую сигнал выключения, когда скорость ветра достигает опасной величины. Ее код показан в примере 7.3.

Пример 7.3. Реализация интерфейса `Processor`, определяющая опасную скорость ветра

```
public class HighWindsFlatmapProcessor
    implements Processor<String, TurbineState, String, TurbineState> { ❶
    private ProcessorContext<String, TurbineState> context;

    @Override
    public void init(ProcessorContext<String, TurbineState> context) { ❷
        this.context = context; ❸
    }

    @Override
    public void process(Record<String, TurbineState> record) {
        TurbineState reported = record.value();
        context.forward(record); ❹

        if (reported.getWindSpeedMph() > 65 && reported.getPower() == Power.ON) { ❺
            TurbineState desired = TurbineState.clone(reported); ❻
            desired.setPower(Power.OFF);
            desired.setType(Type.DESIRED);

            Record<String, TurbineState> newRecord = ❼
                new Record<>(record.key(), desired, record.timestamp());
            context.forward(newRecord); ❸
        }
    }

    @Override
    public void close() {
        // ничего не нужно делать ❾
    }
}
```

❶ Напомню, что интерфейс `Processor` имеет четыре параметра типа. Первые два (в данном случае `Processor<String, TurbineState, ..., ...>`) определяют типы *входных* ключей и значений, а последние (`Processor<..., ..., String, TurbineState>`) — типы *выходных* ключей и значений.

❷ Параметры типа в интерфейсе `ProcessorContext` определяют типы выходных ключей и значений (в данном случае `ProcessorContext<String, TurbineState>`).

❸ Обычно контекст узла-обработчика сохраняется в виде свойства экземпляра, как в этом случае, чтобы иметь к нему доступ позже (например, из методов `process` и/или `close`).

❹ Для отправки записи нижестоящим обработчикам можно вызвать метод `forward` экземпляра `ProcessorContext` (мы сохранили его в свойстве `context`). Этот метод принимает запись для передачи. Наша реализация узла-обработчика всегда должна пересылать записи с сообщаемым состоянием, поэтому мы вызываем метод `context.forward` и передаем ему исходную запись.

❺ Проверяем, выполняются ли условия, когда требуется отправить сигнал выключения. В этом случае скорость ветра должна превысить пороговое значение (65 миль в час), и турбина должна быть включена.

❻ Если предыдущие условия выполняются, то генерируется новая запись, содержащая информацию о желаемом состоянии выключения. Мы уже отправили исходную запись с сообщаемым состоянием и теперь генерируем запись желаемого состояния, поэтому данная операция фактически является операцией `flatMap` (узел создает две выходные записи из одной входной).

❼ Создается выходная запись с желаемым состоянием, которое мы сохранили в хранилище состояний. Ключ записи и метка времени копируются из входной записи.

❽ Вызов метода `context.forward` для отправки новой записи (с сигналом выключения) нижестоящим узлам.

❾ При закрытии этого узла не требуется выполнять никакой особой логики.

Как видите, узлы-обработчики реализуются довольно просто. Однако обратите внимание: при создании подобных узлов обычно не приходится беспокоиться о том, *куда* будут передаваться выходные записи (можно определить поток данных, задав имя родителя в нижестоящем узле). Исключением является версия `ProcessorContext#forward`, принимающая список имен нижестоящих узлов, посредством которого можно сообщить Kafka Streams, в какие дочерние узлы должны передаваться выходные данные. Например:

```
context.forward(newRecord, "some-child-node");
```

Выбор этой версии метода `forward` зависит от необходимости рассылать выходные данные всем или конкретным нижестоящим узлам. Например, метод `branch` в DSL использует описанную выше версию, только если требуется передать свой вывод ограниченному подмножеству доступных нижестоящих узлов.

На этом мы завершаем второй шаг в нашей топологии (см. рис. 7.1). Далее нам предстоит реализовать узел-обработчик потока с состоянием, который создает и сохраняет записи цифровых двойников в хранилище ключей-значений.

Создание узлов с состоянием

В разделе «Хранилища состояний» главы 4 мы узнали, что операциям с состоянием в Kafka Streams требуются так называемые хранилища состояний, в которых запоминаются ранее обрабатывавшиеся данные. Для создания записей цифровых двойников нужно объединить желаемое и сообщаемое состояние в одну запись. Поскольку эти записи будут поступать в разное время, последние сообщаемое и желаемое состояния должны сохраняться для каждой турбины.

До настоящего момента основное наше внимание уделялось использованию хранилищ состояний в DSL. Кроме того, DSL предлагает несколько вариантов использования хранилищ состояний. Мы можем использовать внутреннее хранилище состояний по умолчанию, просто применив оператор с состоянием, не указывая хранилище состояний, например, так:

```
grouped.aggregate(initializer, adder);
```

Или с помощью класса фабрики `Stores` создать *поставщик хранилища* и материализовать хранилище состояний с помощью класса `Materialized` в сочетании с оператором с состоянием, как показано ниже:

```
KeyValueBytesStoreSupplier storeSupplier =
    Stores.persistentTimestampedKeyValueStore("my-store");

grouped.aggregate(
    initializer,
    adder,
    Materialized.<String, String>as(storeSupplier));
```

Порядок использования хранилищ состояний в Processor API немного отличается. В отличие от DSL, Processor API не создает внутреннего хранилища состояний автоматически, поэтому его всегда следует создавать вручную и подключать к соответствующим узлам-обработчикам потока, когда требуется выполнять операции с состоянием. Кроме того, для создания хранилищ состояний мы будем использовать другой набор методов все того же фабричного класса `Stores`: вместо одного из методов, возвращающих *поставщик хранилища*, мы будем использовать методы для создания *построителя хранилища*.

Например, для хранения записей цифровых двойников достаточно простого хранилища ключей и значений. Фабричный метод для получения *построителя* хранилища ключей и значений называется `keyValueStoreBuilder`, и ниже показано, как можно использовать этот метод, чтобы создать хранилище для наших цифровых двойников:

```
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("digital-twin-store"),
        Serdes.String(), ❶
        JsonSerdes.DigitalTwin()); ❷
```

❶ Для сериализации/десериализации ключей будет использоваться встроенный класс `Serdes`, параметризованный типом `String`.

❷ Для сериализации/десериализации значений будет использоваться класс `Serdes`, представленный в примере 7.1.

После создания построителя хранилища для узла-обработчика с состоянием самое время реализовать интерфейс `Processor`. Это очень похоже на реализацию узла без процессора состояния, как было показано в разделе «Создание узлов без состояния» выше. Разница лишь в том, что метод `addProcessor` используется несколько иначе, как показано ниже:

```
builder.addProcessor(
    "Digital Twin Processor", ❶
    DigitalTwinProcessor::new, ❷
    "High Winds Flatmap Processor", "Desired State Events"); ❸
```

❶ Имя этого узла-обработчика потока.

❷ Класс, производный от `ProcessSupplier`, и его метод, который можно использовать для получения экземпляра `Processor`. Класс `DigitalTwinProcessor` мы реализуем чуть ниже.

❸ Имена родительских узлов. Указав несколько родителей, мы фактически выполняем операцию `merge` из DSL.

Прежде чем реализовать `DigitalTwinProcessor`, добавим в топологию новое хранилище состояний. Сделать это можно с помощью метода `Topology#addStateStore`, как показано в примере 7.4.

Пример 7.4. Пример использования метода `addStateStore`

```
builder.addStateStore(
    storeBuilder, ❶
    "Digital Twin Processor" ❷
);
```

❶ Построитель хранилищ, с помощью которого можно получить хранилище состояний.

❷ *При необходимости* можно указать имена узлов, которым должно быть доступно это хранилище. В данном случае хранилище должно быть доступно узлу `DigitalTwinProcessor`, созданному выше. Здесь можно передать имена нескольких узлов, если они имеют общее состояние. Наконец, опустив этот необязательный аргумент, можно использовать `Topology#connectProcessorAndState` для подключения хранилища состояний к узлу *после* добавления хранилища в топологию, а не во время, как здесь.

Последний шаг — реализация нового узла с состоянием: `DigitalTwinProcessor`. По аналогии с узлами без состояния мы должны реализовать интерфейс `Processor`. Однако на этот раз реализация будет несколько сложнее, потому что этот узел должен взаимодействовать с хранилищем состояний. В примере 7.5 приводится код реализации, а ее описание — в комментариях ниже.

Пример 7.5. Узел с состоянием для создания записей цифровых двойников

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> { ❶
    private ProcessorContext<String, DigitalTwin> context;
    private KeyValueStore<String, DigitalTwin> kvStore;

    @Override
    public void init(ProcessorContext<String, DigitalTwin> context) { ❷
        this.context = context; ❸
        this.kvStore = (KeyValueStore) context.getStateStore("digital-twin-store"); ❹
    }

    @Override
    public void process(Record<String, TurbineState> record) {
        String key = record.key(); ❺
        TurbineState value = record.value();
        DigitalTwin digitalTwin = kvStore.get(key); ❻
        if (digitalTwin == null) { ❼
            digitalTwin = new DigitalTwin();
        }

        if (value.getType() == Type.DESIRED) { ❽
            digitalTwin.setDesired(value);
        } else if (value.getType() == Type.REPORTED) {
            digitalTwin.setReported(value);
        }

        kvStore.put(key, digitalTwin); ❾

        Record<String, DigitalTwin> newRecord =
            new Record<>(record.key(), digitalTwin, record.timestamp()); ❿
        context.forward(newRecord); ⓫
    }
}
```

```
@Override
public void close() {
    // ничего не нужно делать
}
}
```

❶ Первые два параметра типа в интерфейсе `Processor` (в данном случае `Processor<String, TurbineState, ..., ...>`) определяют типы *входных* ключей и значений, а последние два (`Processor<..., ..., String, DigitalTwin>`) — типы *выходных* ключей и значений.

❷ Параметры типа в интерфейсе `ProcessorContext` (`ProcessorContext<String, DigitalTwin>`) определяют типы выходных ключей и значений.

❸ Мы сохраним экземпляр `ProcessorContext` (на который ссылается свойство `context`), чтобы получить к нему доступ позже.

❹ Метод `getStateStore` класса `ProcessorContext` позволяет получить хранилище состояний, подключенное ранее к узлу-обработчику потока. Мы будем взаимодействовать с этим хранилищем при обработке записей, поэтому сохраним ссылку на него в свойстве экземпляра с именем `kvStore`.

❺ Эта и следующая строка показывают, как получить ключ и значение входной записи.

❻ Поиск ключа текущей записи в хранилище ключей и значений. Если искомым ключом сохранялся прежде, то этот вызов вернет ранее сохраненную запись цифрового двойника.

❼ Если поиск не дал результатов, то создается новая запись цифрового двойника.

❽ Этот блок кода устанавливает соответствующее значение в записи цифрового двойника в зависимости от типа текущей записи (сообщаемое или желаемое состояние).

❾ Сохранение записи цифрового двойника в хранилище состояний вызовом метода `put`.

❿ Создание выходной записи, включающей экземпляр цифрового двойника, находящийся в хранилище состояний. Ключ и отметка времени копируются из входной записи.

⓫ Передача выходной записи нижестоящим узлам.

Итак, мы реализовали первую часть шага 3 в нашей топологии. Вторая часть шага 3 (см. рис. 7.1) познакомит вас с очень важной особенностью `Processor API`, не имеющей эквивалента в `DSL`, — планированием выполнения периодических функций.

Периодическое выполнение функций с Punctuate

В зависимости от конкретного случая в приложении Kafka Streams может потребоваться выполнять некоторые действия периодически. Это одна из областей, где Processor API действительно предстает во всей красе, позволяя планировать задачи с помощью метода `ProcessorContext#schedule`. В пункте «Надгробия» главы 6 мы обсуждали, как уменьшить размер хранилища состояний, удалив ненужные записи. В текущем проекте демонстрируется еще один метод очистки хранилищ состояний, использующий возможность планирования задач. В частности, мы реализуем удаление записей цифровых двойников, состояние которых не обновлялось в последние семь дней, предположив, что соответствующие ветряки выведены из эксплуатации или находятся в длительном ремонте.

В главе 5 мы видели, что понятие *времени* в обработке потоковых данных — сложная штука. Теперь, вплотную подойдя к вопросу о том, *когда* в Kafka Streams будет выполняться периодическая функция, нам придется вспомнить об этих сложностях. Существует два *типа определения периодов* (то есть стратегий исчисления времени), описываемых в табл. 7.2.

Таблица 7.2. Типы определения периодов, доступные в Kafka Streams

Тип	Элемент перечисления	Описание
Время потока	<code>PunctuationType.STREAM_TIME</code>	Время потока — это наибольшая отметка времени, наблюдаемая в конкретной теме-разделе. Изначально это время неизвестно и может только увеличиваться или оставаться прежним. Оно увеличивается, только когда появляются новые данные, поэтому при использовании этого типа определения периодов функция будет выполняться, только если данные поступают постоянно
Системное время	<code>PunctuationType.WALL_CLOCK_TIME</code>	Локальное время в системе, которое увеличивается с каждой итерацией в методе опроса потребителей. Максимальная частота обновления определяется параметром <code>StreamsConfig#POLLMSECONFIG</code> , содержащим максимальную продолжительность (в миллисекундах) блокирования базового метода опроса в ожидании новых данных. Это означает, что периодические функции будут продолжать выполняться, независимо от получения новых сообщений

Поскольку периодичность выполнения нашей функции не должна зависеть от поступления новых данных (на самом деле само наличие функции TTL (time to live — «время жизни») основано на предположении, что данные могут перестать поступать), для определения периодов мы будем использовать системное время. Теперь, после выбора нужной абстракции, нам остается только реализовать функцию и запланировать ее выполнение.

Ниже показана наша реализация:

```
public class DigitalTwinProcessor
    implements Processor<String, TurbineState, String, DigitalTwin> {

    private Cancellable punctuator; ❶

    // остальные поля опущены для краткости

    @Override
    public void init(ProcessorContext<String, DigitalTwin> context) {

        punctuator = this.context.schedule(
            Duration.ofMinutes(5),
            PunctuationType.WALL_CLOCK_TIME, this::enforceTtl); ❷

        // ...
    }

    @Override
    public void close() {
        punctuator.cancel(); ❸
    }

    public void enforceTtl(Long timestamp) {
        try (KeyValueIterator<String, DigitalTwin> iter = kvStore.all()) { ❹

            while (iter.hasNext()) {
                KeyValue<String, DigitalTwin> entry = iter.next();
                TurbineState lastReportedState = entry.value.getReported(); ❺
                if (lastReportedState == null) {
                    continue;
                }

                Instant lastUpdated = Instant.parse(lastReportedState.getTimestamp());
                long daysSinceLastUpdate =
                    Duration.between(lastUpdated, Instant.now()).toDays(); ❻
                if (daysSinceLastUpdate >= 7) {
                    kvStore.delete(entry.key); ❼
                }
            }
        }
    }

    // ...
}
```

- ❶ Операция планирования периодической функции возвращает объект `Cancellable`, который можно использовать позже для остановки запланированной функции. Мы сохраняем полученный объект в переменной с именем `punctuator`.
- ❷ Планирование выполнения периодической функции раз в пять минут по системным часам и сохранение возвращаемого объекта `Cancellable` в свойстве `punctuator` (см. предыдущий комментарий).
- ❸ Отмена периодического выполнения функции при закрытии узла-обработчика, например, когда приложение `Kafka Streams` завершается.
- ❹ В каждом вызове функции извлечь все значения из хранилища состояний. Обратите внимание, что здесь используется оператор `try-with-resources`, чтобы гарантировать правильное закрытие итератора и предотвратить утечку ресурсов.
- ❺ Получить последнее сообщаемое состояние из текущей записи (которое соответствует физическому ветряку).
- ❻ Определить время (в днях), прошедшее с момента, как эта турбина в последний раз сообщила свое состояние.
- ❼ Удалить запись из хранилища состояний, если она устарела (то есть не обновлялась по меньшей мере семь дней).



Функция `process` и любые запланированные периодические функции будут выполняться в одном потоке выполнения (то есть для периодических функций не существует отдельного фонового потока), поэтому можно не беспокоиться о проблемах параллелизма.

Как видите, в планировании периодического выполнения функций нет ничего сложного. Теперь исследуем еще одну область, в которой `Processor API` имеет неоспоримые преимущества: доступ к метаданным записей.

Доступ к метаданным записей

Когда мы используем `DSL`, то обычно имеем доступ только к ключу и значению записи. Однако запись сопровождается большим количеством дополнительной информации, недоступной в `DSL`, но доступной в `Processor API`. Некоторые из наиболее ярких примеров метаданных записей, которые можно получить, перечислены в табл. 7.3. Обратите внимание, что переменная `context` в этой таблице

ссылается на экземпляр `ProcessorContext`, который доступен в методе `init`, как было показано в примере 7.3.

Таблица 7.3. Методы доступа к дополнительным метаданным записей

Метаданные	Пример
Заголовки записи	<code>context.headers()</code>
Смещение	<code>context.offset()</code>
Раздел	<code>context.partition()</code>
Отметка времени	<code>context.timestamp()</code>
Тема	<code>context.topic()</code>



Методы, перечисленные в табл. 7.3, извлекают метаданные о текущей записи и могут использоваться внутри функции `process()`. Однако, когда вызываются функции `init()` и `close()` или когда выполняется периодическая функция, текущая запись недоступна, поэтому метаданные для извлечения отсутствуют.

Где можно использовать эти метаданные? Один из вариантов — сопроводить значения записей дополнительным контекстом перед записью в какую-либо нижестоящую систему. Эту информацию также можно добавить в журналы приложений для целей отладки. Например, столкнувшись с искаженной записью, можно зарегистрировать ошибку, добавив в описание раздел и смещение записи, и использовать эту информацию как основу для дальнейшего устранения неполадок.

Заголовки записей тоже представляют определенный интерес, потому что их можно использовать для вставки дополнительных метаданных (например, контекста трассировки, который можно использовать для распределенной трассировки (<https://oreil.ly/ZshyG>)). Вот несколько примеров работы с заголовками записей:

```
Headers headers = context.headers();
headers.add("hello", "world".getBytes(StandardCharsets.UTF_8)); ❶
headers.remove("goodbye"); ❷
headers.toArray(); ❸
```

❶ Добавление заголовка с именем `hello`. Этот заголовок будет передан нижестоящим узлам-обработчикам.

❷ Удаление заголовка с именем `goodbye`.

❸ Получение массива всех доступных заголовков. Получив массив, можно обойти все его элементы и выполнить некоторую операцию с каждым из них.

Наконец, можно использовать метод `topic()`, чтобы проследить происхождение записи. В текущем проекте нам это не понадобится, как и доступ к метаданным вообще, но теперь у вас должно сложиться хорошее представление о том, как получить дополнительные метаданные на случай, если в будущем это потребуется.

Теперь мы готовы перейти к следующему шагу в нашей топологии и узнать, как с помощью Processor API добавить узел-приемник.

Добавление узла-приемника

Шаг 4 в нашей топологии (см. рис. 7.1) предполагает добавление узла-приемника, который будет сохранять все записи цифровых двойников в выходную тему `digital-twins`. Сделать это с помощью Processor API очень просто, поэтому раздел будет коротким. Мы просто должны вызвать метод `addSink` и указать несколько дополнительных параметров, которые подробно описаны в следующем коде:

```
builder.addSink(
    "Digital Twin Sink", ❶
    "digital-twins", ❷
    Serdes.String().serializer(), ❸
    JsonSerializer.DigitalTwin().serializer(), ❹
    "Digital Twin Processor"); ❺
```

❶ Имя узла-приемника.

❷ Имя выходной темы.

❸ Сериализатор для ключей.

❹ Сериализатор для значений.

❺ Имя одного или нескольких родительских узлов для этого узла-приемника.

Описанных действий достаточно для добавления простого узла-приемника. Конечно, этот метод, как и многие другие в Kafka Streams, имеет несколько перегруженных версий. Например, одна из версий позволяет указать пользовательский `StreamPartitioner` для сохранения выходной записи в раздел

с определенным номером. Еще одна версия позволяет исключить сериализаторы ключей и значений и использовать сериализаторы по умолчанию, определяемые свойством `DEFAULT_KEY_SERDE_CLASS_CONFIG`. Однако, независимо от выбора перегруженной версии, добавление узла-приемника — довольно простая операция.

Теперь перейдем к последнему шагу передачи записей цифровых двойников внешним службам (включая сами ветряки, которые будут синхронизировать свое состояние с записями цифровых двойников, находящимися в хранилище состояний).

Интерактивные запросы

Мы выполнили шаги с 1-го по 4-й в нашей топологии (см. рис. 7.1). Пятый шаг включает получение записей цифровых двойников с помощью интерактивных запросов. Мы уже рассмотрели эту тему в разделе «Интерактивные запросы» главы 4, поэтому не будем углубляться в детали, исследуя полную реализацию. В примере 7.6 показана очень простая служба REST, использующая интерактивные запросы для извлечения последней записи цифрового двойника. Обратите внимание, что в самом этом примере не показаны удаленные запросы, но вы можете заглянуть в его исходный код, чтобы увидеть их.

Важно отметить, что с точки зрения интерактивных запросов Processor API мало чем отличается от DSL.

Пример 7.6. Пример REST-службы, получающей записи цифровых двойников

```
class RestService {
    private final HostInfo hostInfo;
    private final KafkaStreams streams;

    RestService(HostInfo hostInfo, KafkaStreams streams) {
        this.hostInfo = hostInfo;
        this.streams = streams;
    }

    ReadOnlyKeyValueStore<String, DigitalTwin> getStore() {
        return streams.store(
            StoreQueryParameters.fromNameAndType(
                "digital-twin-store", QueryableStoreTypes.keyValueStore()));
    }

    void start() {
        Javalin app = Javalin.create().start(hostInfo.port());
        app.get("/devices/:id", this::getDevice);
    }
}
```

```

void getDevice(Context ctx) {
    String deviceId = ctx.pathParam("id");
    DigitalTwin latestState = getStore().get(deviceId);
    ctx.json(latestState);
}
}

```

Этот шаг завершает реализацию нашей топологии (см. рис. 7.1). Теперь объединим все, что мы сделали до сих пор.

Все вместе

Ниже показана полная реализация нашей топологии:

```

Topology builder = new Topology();

builder.addSource( ❶
    "Desired State Events",
    Serdes.String().deserializer(),
    JsonSerdes.TurbineState().deserializer(),
    "desired-state-events");

builder.addSource( ❷
    "Reported State Events",
    Serdes.String().deserializer(),
    JsonSerdes.TurbineState().deserializer(),
    "reported-state-events");

builder.addProcessor( ❸
    "High Winds Flatmap Processor",
    HighWindsFlatmapProcessor::new,
    "Reported State Events");

builder.addProcessor( ❹
    "Digital Twin Processor",
    DigitalTwinProcessor::new,
    "High Winds Flatmap Processor",
    "Desired State Events");

StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
    Stores.keyValueStoreBuilder( ❺
        Stores.persistentKeyValueStore("digital-twin-store"),
        Serdes.String(),
        JsonSerdes.DigitalTwin());

builder.addStateStore(storeBuilder, "Digital Twin Processor"); ❻

builder.addSink( ❼
    "Digital Twin Sink",

```

```
"digital-twins",
Serdes.String().serializer(),
JsonSerdes.DigitalTwin().serializer(),
"Digital Twin Processor");
```

❶ Создание *узла-источника* с именем `Desired State Events`, который извлекает данные из темы `desired-state-events`. Это эквивалент *потока* в DSL.

❷ Создание *узла-источника* с именем `Reported State Events`, который извлекает данные из темы `reported-state-events`. Он тоже является эквивалентом *потока* в DSL.

❸ Добавление *узла-обработчика потока* с именем `High Winds Flatmap Processor`, генерирующего сигнал отключения при сильном ветре. Этот узел-обработчик получает события от узла `Reported State Events` и эквивалентен операции `flatMap` в DSL, потому что реализует отношение 1:N между входными и выходными записями. Реализация этого узла показана в примере 7.3.

❹ Добавление *узла-обработчика потока* с именем `Digital Twin Processor`, который создает записи цифровых двойников, используя данные, полученные от узлов `High Winds Flatmap Processor` и `Desired State Events`. Этот узел эквивалентен операции `merge` в DSL, потому что задействует несколько источников. Кроме того, поскольку этот узел имеет хранимое состояние, он также эквивалентен агрегированной *таблице* в DSL. Реализация этого узла показана в примере 7.5.

❺ Применение фабричного класса `Stores` для создания *построителя хранилища*, с помощью которого Kafka Streams будет строить постоянные хранилища ключей и значений, доступных узлу `Digital Twin Processor`.

❻ Добавление хранилища состояний в топологию и его подключение к узлу `Digital Twin Processor`.

❼ Создание *узла-приемника* с именем `Digital Twin Sink`, сохраняющего все записи цифровых двойников, отправленных узлом `Digital Twin Processor`, в выходную тему с именем `digital-twins`.

Итак, можно запустить наше приложение, записать некоторые тестовые данные в кластер Kafka и послать запрос службе цифровых двойников. Запуск этого приложения ничем не отличается от других, которые мы видели в предыдущих главах:

```
Properties props = new Properties();
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev-consumer"); ❶
// ...
```

```
KafkaStreams streams = new KafkaStreams(builder, props); ❷  
streams.start(); ❸  
  
Runtime.getRuntime().addShutdownHook(new Thread(streams::close)); ❹  
  
RestService service = new RestService(hostInfo, streams); ❺  
service.start();
```

❶ Настройка приложения Kafka Streams. Действует точно так же, как при использовании DSL. Большинство настроек опущено для краткости.

❷ Создание нового экземпляра `KafkaStreams` для выполнения нашей топологии.

❸ Запуск приложения Kafka Streams.

❹ Добавление обработчика завершения, чтобы корректно освободить ресурсы, занятые приложением Kafka Streams, при получении глобального сигнала остановки.

❺ Создание экземпляра и запуск в следующей строке службы REST, реализованной в примере 7.6.

Наше приложение читает данные из нескольких входных тем, но для демонстрации на страницах книги мы создадим тестовые данные только для темы `report-state-events` (более полный пример вы найдете в репозитории с примерами <https://oreil.ly/LySHt>). Чтобы проверить, генерирует ли приложение сигнал выключения турбины, добавим одну запись, сообщающую о скорости ветра, превышающей рабочий порог 65 миль в час. Ниже показаны наши тестовые данные с ключами и значениями, разделенными символом `|` (отметки времени опущены для краткости):

```
1|{"timestamp": "...", "wind_speed_mph": 40, "power": "ON", "type": "REPORTED"}  
1|{"timestamp": "...", "wind_speed_mph": 42, "power": "ON", "type": "REPORTED"}  
1|{"timestamp": "...", "wind_speed_mph": 44, "power": "ON", "type": "REPORTED"}  
1|{"timestamp": "...", "wind_speed_mph": 68, "power": "ON", "type": "REPORTED"} ❶
```

❶ Согласно этим данным, полученным от датчика, скорость ветра составляет 68 миль в час. Когда приложение встретит эту запись, оно должно сгенерировать сигнал выключения турбины, создав новую запись `TurbineState` с желаемым состоянием `"power": "OFF"`.

Если создать эти тестовые данные в теме `reported-state-events`, а затем послать запрос службе цифровых двойников, то мы увидим, что наше приложение Kafka Streams не только обработало сообщенные состояния ветряка, но также создало

запись желаемого состояния со значением "power": "OFF". Ниже показан пример запроса нашей службе REST и ее ответ:

```
$ curl localhost:7000/devices/1 | jq '.'
{
  "desired": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "windSpeedMph": 68,
    "power": "OFF",
    "type": "DESIRED"
  },
  "reported": {
    "timestamp": "2020-11-23T09:02:01.000Z",
    "windSpeedMph": 68,
    "power": "ON",
    "type": "REPORTED"
  }
}
```

Теперь наши ветряные турбины могут обращаться к службе REST и синхронизировать свое состояние с желаемым, которое было получено (из темы `desired-state-events`) или принудительно отправлено (узлом `High Winds Flatmap Processor`) с помощью Kafka Streams.

Объединение Processor API и DSL

Мы убедились в нормальной работе приложения. Однако если внимательно рассмотреть код, можно заметить, что только один шаг в топологии требует низкоуровневого доступа, который предлагает Processor API. Шаг, о котором я говорю, — это узел-обработчик `Digital Twin Processor` (см. шаг 3 на рис. 7.1), использующий возможность периодического выполнения функции.

Kafka Streams позволяет комбинировать Processor API и DSL, благодаря чему легко можно реорганизовать приложение так, что оно будет использовать Processor API *только* на шаге `Digital Twin Processor` и DSL — на всех остальных. Самое большое преимущество такой реорганизации — возможность упростить все остальные шаги потоковой обработки. Наибольшего упрощения в этом проекте можно добиться в реализации `High Winds Flatmap Processor`, но в более крупных приложениях подобная реорганизация уменьшает сложность еще больше.

Первые два шага в нашей топологии (регистрация узлов-источников и генерирование сигнала выключения с использованием `flatMap`-подобной операции)

можно реорганизовать с использованием операторов, уже обсуждавшихся в этой книге. В частности, можно внести следующие изменения.

Processor API	DSL
Topology builder = new Topology();	StreamsBuilder builder = new StreamsBuilder();
builder.addSource("Desired State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState(). deserializer(), "desired-state-events");	KStream<String, TurbineState> desiredStateEvents = builder.stream("desired-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState())));
builder.addSource("Reported State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState(). deserializer(), "reported-state-events");	KStream<String, TurbineState> highWinds = builder.stream("reported-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState())) .flatMapValues((key, reported) -> ...) .merge(desiredStateEvents);
builder.addProcessor("High Winds Flatmap Processor", HighWindsFlatmapProcessor::new, "Reported State Events");	

Как видите, изменения довольно просты. Однако шаг 3 в нашей топологии действительно требует использовать Processor API, но как совместить DSL и Processor API на этом шаге? Для этого можно использовать особый набор операторов DSL, которые мы рассмотрим далее.

Обработчики и преобразователи

DSL поддерживает набор специальных операторов, позволяющих использовать Processor API для низкоуровневого доступа к хранилищам состояний, метаданным записей и контексту узла-обработчика (который, кроме прочего, можно использовать для планирования периодических функций). Специальные операторы делятся на две категории: *обработчики* (processors) и *преобразователи* (transformers). Ниже перечислены различия между этими категориями.

- *Обработчик* (processor) — это *терминальная операция* (возвращающая значение `void` и не позволяющая добавить последующие операторы в цепочку), поэтому вычислительная логика должна быть реализована с использованием

интерфейса `Processor` (который мы впервые обсудили в разделе «Добавление узлов-обработчиков без состояния» выше в этой главе). Обработчики следует использовать всегда, когда возникает потребность вызвать `Processor API` из DSL и не нужно добавлять в цепочку последующие операторы. В настоящее время существует только один вариант операторов этого типа.

Оператор DSL	Интерфейс для реализации	Описание
<code>process</code>	<code>Processor</code>	Применяет реализацию <code>Processor</code> к каждой записи

- Категория *преобразователей* (transformers) включает разнообразный набор операторов, способных возвращать одну или несколько записей в зависимости от используемого варианта, поэтому они лучше подходят для случаев, когда требуется добавить последующие операторы. Варианты преобразователей перечислены в табл. 7.4.

Таблица 7.4. Операторы-преобразователи, доступные в Kafka Streams

Оператор DSL	Интерфейс для реализации	Описание	Отношение ввода/вывода
<code>transform</code>	<code>Transformer</code>	Применяет реализацию <code>Transformer</code> к каждой записи и создает одну или несколько выходных записей. Метод <code>Transformer#transform</code> может возвращать одиночные записи, а с помощью <code>ProcessorContext#forward¹</code> можно отправить несколько значений. Преобразователь имеет доступ к ключу записи, значению, метаданным, контексту обработчика (который можно использовать для планирования периодических функций) и подключенным хранилищам состояния	1:N
<code>transform-Values</code>	<code>ValueTransformer</code>	Во многом похож на <code>transform</code> , но <i>не имеет</i> доступа к ключу записи и не может пересылать несколько записей с помощью <code>ProcessorContext#forward</code> (при попытке переслать несколько записей вы	1:1

¹ Несмотря на поддержку преобразований 1:N, `transform` лучше подходит для преобразований 1:1 или 1:0, когда возвращается одна запись, потому что метод `ProcessorContext#forward` не является типобезопасным. Поэтому, если вам понадобится переслать из преобразователя несколько записей, используйте `flatTransform`, безопасный по типам.

Оператор DSL	Интерфейс для реализации	Описание	Отношение ввода/вывода
		получите исключение <code>StreamsException</code>). Поскольку операции с хранилищем состояний основаны на ключах, этот оператор не подходит для случаев, когда требуется выполнить поиск в хранилище. Кроме того, выходные записи будут иметь тот же ключ, что и входные, а автоматическое секционирование нисходящего потока не будет выполнено, потому что ключ нельзя изменить (что, в общем-то, выгодно, потому что позволяет избежать лишних пересылок данных по сети)	
<code>transform-Values</code>	<code>ValueTransformerWithKey</code>	Напоминает <code>transform</code> , но ключ записи <i>доступен только для чтения и не должен изменяться</i> . Кроме того, с помощью <code>ProcessorContext#forward</code> нельзя переслать несколько записей (при попытке сделать это вы получите исключение <code>StreamsException</code>)	1:1
<code>flatMap</code>	<code>Transformer</code> (с итерируемым возвращаемым значением)	Напоминает <code>transform</code> , но вместо использования <code>ProcessorContext#forward</code> для возврата нескольких записей можно просто вернуть набор значений. Поэтому, когда нужно создать несколько записей, рекомендуется использовать <code>flatMap</code> вместо <code>transform</code> , так как этот метод является типобезопасным, в отличие от <code>ProcessorContext#forward</code>	1:N
<code>flatMapValues</code>	<code>ValueTransformer</code> (с итерируемым возвращаемым значением)	Применяет реализацию <code>Transformer</code> к каждой записи и возвращает одну или несколько выходных записей непосредственно из метода <code>ValueTransformer#transform</code>	1:N
<code>flatMapValues</code>	<code>ValueTransformerWithKey</code> (с итерируемым возвращаемым значением)	Версия <code>flatMapValues</code> с состоянием. Передает методу <code>transform</code> ключ, доступный только для чтения, который можно использовать для поиска состояния. Возвращает одну или несколько выходных записей непосредственно из метода <code>ValueTransformerWithKey#transform</code>	1:N

Независимо от выбранного варианта, если оператор поддерживает состояние, то перед его добавлением вам придется подключить хранилище состояний

к построителю топологии. Напомню, что мы собрались реорганизовать шаг `Digital Twin Processor`, имеющий хранимое состояние, давайте сделаем это:

```
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder =
    Stores.keyValueStoreBuilder(
        Stores.persistentKeyValueStore("digital-twin-store"),
        Serdes.String(),
        JsonSerdes.DigitalTwin());

builder.addStateStore(storeBuilder); ❶
```

❶ В примере 7.4 мы обсудили необязательный второй параметр метода `Topology#addStateStore`, сообщаящий имена узлов, которые должны быть подключены к хранилищу состояний. Здесь мы опустили второй параметр, поэтому хранилище пока остается в подвешенном состоянии (мы подключим его в следующем блоке кода).

Теперь нам нужно решить, что использовать на шаге `Digital Twin Processor`: обработчик или преобразователь? Прочитав описания в предыдущих таблицах, мы можем захотеть использовать оператор `process`, так как к этому времени уже реализовали интерфейс `Processor` в версии приложения с `Processor API` (см. пример 7.5). Если выбрать этот подход (что довольно проблематично по причинам, которые обсудим ниже), то мы получим следующую реализацию:

```
highWinds.process(
    DigitalTwinProcessor::new, ❶
    "digital-twin-store"); ❷
```

❶ `ProcessSupplier`, используемый для получения экземпляра `DigitalTwinProcessor`.

❷ Имя хранилища состояний, с которым будет взаимодействовать обработчик.

К сожалению, это не лучший выбор, потому что нам нужно подключить узел-приемник к этому узлу, а оператор `process` является терминальной операцией. Поэтому здесь лучше подойдет один из операторов преобразователей, поскольку они, как мы вскоре увидим, позволяют подключить узел-приемник. Теперь, взглянув на табл. 7.4, выберем оператор, удовлетворяющий нашим требованиям:

- каждая входная запись всегда будет создавать одну выходную запись (отношение 1:1);
- ключ записи должен быть доступен только для чтения, чтобы иметь возможность поиска точек в хранилище состояний, но нам не нужно каким-либо образом изменять ключ.

Лучше всего этим требованиям отвечает оператор `transformValues` (вариант, использующий `ValueTransformerWithKey`). Мы уже реализовали вычислительную логику для этого шага с помощью `Processor` (см. пример 7.5), поэтому нужно лишь реализовать интерфейс `ValueTransformerWithKey` и скопировать логику из метода `process` в примере 7.5 в метод `transform`, как показано ниже. Большая часть кода в этом фрагменте опущена, потому что она совпадает с реализацией обработчика. Изменения описаны в комментариях после листинга:

```
public class DigitalTwinValueTransformerWithKey
    implements ValueTransformerWithKey<String, TurbineState, DigitalTwin> { ❶

    @Override
    public void init(ProcessorContext context) {
        // ...
    }

    @Override
    public DigitalTwin transform(String key, TurbineState value) {
        // ...
        return digitalTwin; ❷
    }

    @Override
    public void close() {
        // ...
    }

    public void enforceTtl(Long timestamp) {
        // ...
    }
}
```

❶ Реализует интерфейс `ValueTransformerWithKey`. `String` — это тип ключа, `TurbineState` — тип значения входной записи, а `DigitalTwin` — тип значения выходной записи.

❷ Для отправки записей нижестоящим узлам не требуется использовать `context.forward`, достаточно просто вернуть их из метода `transform`. Как видите, этот код больше похож на DSL.

Имея такую реализацию преобразователя, можно добавить в приложение следующую строку:

```
highWinds
    .transformValues(DigitalTwinValueTransformerWithKey::new, "digital-twin-store")
    .to("digital-twins", Produced.with(Serdes.String(), JsonSerdes.DigitalTwin()));
```

Все вместе: реорганизация

Теперь, обсудив отдельные шаги по реорганизации кода для возврата к DSL, сравним две реализации нашего приложения, как показано в табл. 7.5.

Таблица 7.5. Две реализации топологии службы цифровых двойников

Только Processor API	DSL + Processor API
Topology builder = new Topology();	StreamsBuilder builder = new StreamsBuilder();
builder.addSource("Desired State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState(). deserializer(), "desired-state-events");	KStream<String, TurbineState> desiredStateEvents = builder.stream("desired-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState()));
builder.addSource("Reported State Events", Serdes.String().deserializer(), JsonSerdes.TurbineState(). deserializer(), "reported-state-events");	KStream<String, TurbineState> highWinds = builder.stream("reported-state-events", Consumed.with(Serdes.String(), JsonSerdes.TurbineState())) .flatMapValues((key, reported) -> ...) .merge(desiredStateEvents);
builder.addProcessor("High Winds Flatmap Processor", HighWindsFlatmapProcessor::new, "Reported State Events");	
builder.addProcessor("Digital Twin Processor", DigitalTwinProcessor::new, "High Winds Flatmap Processor", "Desired State Events");	// пустое пространство, чтобы упростить // сопоставление шагов в топологии
StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder = Stores.keyValueStoreBuilder(Stores. persistentKeyValueStore("digital-twin-store"), Serdes.String(), JsonSerdes.DigitalTwin());	StoreBuilder<KeyValueStore<String, DigitalTwin>> storeBuilder = Stores.keyValueStoreBuilder(Stores.persistentKeyValueStore("digital-twin-store"), Serdes.String(), JsonSerdes.DigitalTwin());
builder. addStateStore(storeBuilder, "Digital Twin Processor");	builder.addStateStore(storeBuilder);

Только Processor API	DSL + Processor API
<pre>builder.addSink("Digital Twin Sink", "digital-twins", Serdes.String().serializer(), JsonSerdes.DigitalTwin(). serializer(), "Digital Twin Processor");</pre>	<pre>highWinds .transformValues(DigitalTwinValueTransformerWithKey::new, "digital-twin-store") .to("digital-twins", Produced.with(Serdes.String(), JsonSerdes.DigitalTwin()));</pre>

Обе реализации одинаково хороши. Но, возвращаясь к тому, о чем говорилось выше, отмечу, что нет причин вводить дополнительную сложность, если для этого нет веских причин.

Вот основные преимущества гибридной реализации DSL + Processor API.

- Проще построить мысленную карту потоков данных, имея перед глазами цепочки операторов, чем выясняя взаимосвязи между узлами, используя их имена.
- Большинство операторов DSL поддерживают лямбда-выражения, что удобно для реализации простых преобразований (Processor API требует реализации интерфейса `Processor` даже для самых простых операций, это может быть утомительно).
- В этом проекте нам не потребовалось повторно вводить какие-либо записи, однако при необходимости сделать это в Processor API гораздо сложнее. Для простой операции смены ключа придется не только реализовать интерфейс `Processor`, но и записать данные в промежуточную тему, где произойдет секционирование (это включает явное добавление дополнительных узлов приемника и источника, что может чересчур усложнить код).
- Операторы DSL дают стандартный словарь, помогающий определить происходящее на данном этапе обработки потока. Например, встретив оператор `flatMap`, даже не заглядывая в вычислительную логику, мы можем сделать вывод, что количество записей на выходе может отличаться от количества записей на входе. С другой стороны, Processor API способствует сокрытию природы реализации узла, что ухудшает читабельность кода и может негативно сказаться на его сопровождении.
- DSL дает типовой словарь для различных видов потоков. К ним относятся чистые потоки записей, локальные агрегированные потоки (которые мы обычно называем таблицами) и глобальные агрегированные потоки (которые мы называем глобальными таблицами).

Учитывая все это, я обычно рекомендую, когда нужен низкоуровневый доступ, применять специальные операторы DSL для работы с Processor API, а не пытаться реализовать приложение исключительно на уровне Processor API.

Заключение

В этой главе вы узнали, как использовать Processor API для низкоуровневого доступа к записям в Kafka и контексту узла Kafka Streams. Мы также обсудили возможности прикладного интерфейса Processor API, позволяющего планировать периодические функции, коснулись различных понятий времени, связанных с этим. Наконец, вы увидели, что комбинирование Processor API и высокоуровневого DSL позволяет использовать преимущества обоих API. В следующей главе мы начнем изучать ksqlDB, противоположную сторону спектра с точки зрения простоты: это самый простой вариант для создания приложений потоковой обработки, который мы обсудим в этой книге, и, возможно, вообще самый простой вариант.

Часть III

ksqlDB

СИНТАКСИЧЕСКАЯ НОТАЦИЯ SQL

Главы в этой части книги будут включать описание синтаксиса операторов SQL в ksqlDB. Например, вот как выглядит описание синтаксиса оператора, возвращающего список запросов, запущенных на сервере ksqlDB:

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

Для всех описаний синтаксических конструкций SQL в этой части будут использоваться следующие обозначения:

- квадратные скобки ([]) заключают необязательные элементы или операторы;
- фигурные скобки ({ }) заключают набор альтернативных вариантов;
- круглые скобки (()) обозначают сами круглые скобки;
- вертикальная черта (|) представляет логическое ИЛИ;
- многоточие с предшествующей запятой в квадратных скобках ([, . . .]) указывает, что предыдущий элемент может повторяться в списке через запятую.

Например, синтаксис, приведенный выше, говорит нам, что описываемый им оператор может начинаться с SHOW или LIST, а в конце может быть добавлено необязательное ключевое слово EXTENDED. Следовательно, вот такой оператор ksqlDB является допустимым:

```
SHOW QUERIES ;
```

Имейте в виду, что мы будем следовать этому шаблону, знакомясь с каждым новым оператором ksqlDB.

ГЛАВА 8

Знакомство с ksqlDB

История базы данных ksqlDB — это история упрощения и эволюции. Она создавалась с той же целью, что и Kafka Streams: упростить процесс создания приложений для потоковой обработки данных. Однако по мере развития ksqlDB стало ясно, что ее цели гораздо более амбициозные, чем даже у Kafka Streams, потому что она упрощает не только создание приложений для потоковой обработки, но и их интегрирование с другими системами (в том числе внешними по отношению к Kafka). Все это она делает с помощью интерфейса SQL, что позволяет и новичкам, и экспертам с легкостью использовать возможности Kafka.

Я догадываюсь, о чем вы сейчас подумали: зачем изучать и Kafka Streams, и ksqlDB? Может быть, можно оставить себе один из разделов этой книги, а остальное продать на Craigslist, чтобы вернуть часть потраченных денег? На самом деле и Kafka Streams, и ksqlDB — отличные инструменты для потоковой обработки, которые неплохо дополняют друг друга. ksqlDB можно использовать для создания приложений, логику которых легко выразить на SQL, а также для настройки простых источников и приемников данных и создания сквозных конвейеров обработки данных с помощью одного инструмента. С другой стороны, с помощью Kafka Streams можно создавать более сложные приложения, и знание этой библиотеки только углубит ваше понимание ksqlDB, потому что она фактически основана на Kafka Streams.

Я хотел утаить, что ksqlDB основана на Kafka Streams, чтобы потом представить это как большое откровение, которое, как мне хотелось, поразило бы вас, но не смог утерпеть и двух абзацев, чтобы не поделиться этим. Это огромное преимущество и одна из причин, почему, как мне кажется, вам понравится работать с ksqlDB. Большинство баз данных поражают своей сложностью, стоит только заглянуть внутрь. Это затрудняет освоение заложенной в них технологии, приходится посвящать предварительно месяцы утомительному изучению ее внутренней работы. Однако тот факт, что в основе ksqlDB лежит

Kafka Streams, означает, что эта база данных основана на хорошо продуманных и понятных уровнях абстракции, и это позволяет глубже погрузиться во внутренности и узнать, причем в увлекательной и доступной форме, как в полной мере использовать возможности этой технологии. На самом деле часть этой книги, посвященную Kafka Streams, можно рассматривать как введение во внутреннее устройство ksqlDB.

ksqlDB обладает таким большим количеством замечательных возможностей, что я решил посвятить им несколько глав. В этой первой главе вы познакомитесь с самыми основами технологии и узнаете ответы на некоторые важные вопросы, для начала сведу их в перечень.

- Что такое ksqlDB?
- Когда следует использовать ksqlDB?
- Как менялись возможности ksqlDB с течением времени?
- В каких областях ksqlDB предлагает упрощение?
- Из каких ключевых компонентов складывается архитектура ksqlDB?
- Как установить и запустить ksqlDB?

А теперь без лишних разглагольствований приступим к делу и узнаем, что такое ksqlDB в действительности и на что она способна.

Что такое ksqlDB

ksqlDB — это *база данных потоковой передачи событий* с открытым исходным кодом, выпущенная компанией Confluent в 2017 году (чуть больше года после появления Kafka Streams в экосистеме Kafka). Она упрощает создание, развертывание и обслуживание приложений потоковой обработки за счет интеграции двух специализированных компонентов экосистемы Kafka (Kafka Connect и Kafka Streams) в единую систему и предоставления высокоуровневого интерфейса SQL для взаимодействия с этими компонентами. Вот некоторые возможности ksqlDB.

- Моделирование данных в виде потоков или таблиц (и те и другие в ksqlDB считаются *коллекциями*) с помощью SQL.
- Применение большого количества конструкций SQL (например, для объединения, агрегирования, преобразования, фильтрации и оконной обработки данных) для создания новых производных представлений данных; иногда это можно сделать, не написав ни строчки кода на Java.

- Выполнение *запросов на передачу данных (push-запросов)*, которые выполняются непрерывно и выдают/отправляют результаты клиентам всякий раз, когда появляются новые данные. Внутри push-запросы компилируются в приложения Kafka Streams и идеально подходят для реализации микросервисов, управляемых событиями. Приложения должны следить за событиями и быстро реагировать на них.
- Создание *материализованных представлений* из потоков и таблиц и запрос этих представлений с помощью *запросов на получение данных (pull-запросов)*. Запросы на получение действуют подобно поиску по ключу в традиционных базах данных SQL, внутри они работают с потоками Kafka и хранилищами состояний. Запросы на получение данных могут использоваться клиентами, работающими с ksqlDB в синхронном режиме.
- Определение *коннекторов* для интеграции ksqlDB с внешними хранилищами данных; коннекторы позволяют работать с широким спектром источников и приемников данных. Их также можно комбинировать с таблицами и потоками для создания сквозных конвейеров потоковой обработки ETL¹.

Мы подробно рассмотрим все эти возможности в следующих главах. Но стоит отметить, что ksqlDB основана на зрелых решениях Kafka Connect и Kafka Streams, поэтому в ее лице вы автоматически получаете стабильность и мощность этих инструментов, а также все преимущества более удобного интерфейса. Мы обсудим преимущества использования ksqlDB в следующем разделе, который поможет вам понять, когда именно следует ее использовать.

Когда следует использовать ksqlDB

Никого не удивляет, что с высокоуровневыми абстракциями часто легче работать, чем с их низкоуровневыми аналогами. Однако если бы я просто сказал: «Писать код на SQL проще, чем на Java», я бы упустил из виду многие преимущества ksqlDB, вытекающие из ее более простого интерфейса и архитектуры. Перечислю эти преимущества.

- *Интерактивность*, обусловленная использованием управляемой среды выполнения, которая может компоновать и анализировать приложения потоковой обработки по запросу с помощью встроенного интерфейса командной строки (CLI) и службы REST.

¹ Аббревиатура ETL расшифровывается как extract, transform, load («извлечь, преобразовать, загрузить»).

- *Меньший объем кода*, потому что топологии потоковой обработки выражены на SQL, а не на языке JVM.
- *Более низкий порог входа* и меньшее количество новых понятий для изучения, особенно для тех, кто знаком с традиционными базами данных SQL, но плохо знаком с потоковой обработкой. Это способствует более простой интеграции новых разработчиков в проекты, а также упрощает сопровождение созданных систем.
- *Упрощенная архитектура*, так как интерфейс управления коннекторами (позволяющими интегрировать внешние источники данных в Kafka) и преобразование данных объединены в общую систему. Существует также возможность запуска Kafka Connect в той же JVM, которая выполняет ksqlDB¹.
- *Повышение производительности труда разработчиков*, поскольку требуется писать меньше кода, а низкоуровневые сложности скрыты за новыми уровнями абстракции. Кроме того, интерактивность обеспечивает более быструю обратную связь и в комплект входит тестовая система, чрезвычайно упрощающая проверку запросов². Все это делает процесс разработки приятным и продуктивным.
- *Согласованность между проектами*. Благодаря декларативному синтаксису языка SQL приложения потоковой обработки, написанные на SQL, меньше страдают от специализации, то есть от необходимости разработки уникальных функций, отличающих проекты друг от друга. Kafka Streams отлично с этим справляется, вводя стандартный набор операторов DSL для работы с потоками событий, но оставляет достаточно свободы для управления остальным кодом приложения, что может привести к созданию приложений с уникальными характеристиками.
- *Простота установки и развертывания «под ключ»* благодаря нескольким вариантам распространения, включая официально поддерживаемые образы Docker. Для тех, кому действительно нужен упрощенный путь к интеграции в производство, есть также полностью управляемые облачные предложения ksqlDB (например, Confluent Cloud).

¹ Применимость этой возможности в рабочем окружении зависит от рабочей нагрузки. Для серьезных рабочих нагрузок предпочтительнее запускать Connect отдельно, чтобы иметь возможность масштабировать эти компоненты независимо, так, как это будет показано в главе 9. Однако в разработке такое комбинирование чрезвычайно удобно.

² Обратите внимание, что приложения Kafka Streams также легко поддаются тестированию, как будет показано в главе 12.

- *Улучшенная поддержка анализа данных.* ksqlDB позволяет легко извлекать и выводить содержимое разделов, а также быстро создавать и запрашивать материализованные представления данных. Эти варианты исследования данных отлично укладываются в идеологию ksqlDB.

Теперь, перечислив некоторые преимущества ksqlDB, я предлагаю посмотреть, в каких случаях следует использовать ksqlDB вместо Kafka Streams. Самый распространенный ответ, который можно услышать, — применять ksqlDB всегда, когда логику приложения потоковой обработки можно выразить на SQL. Однако такой ответ мне кажется неполным. Поэтому я предлагаю использовать ksqlDB всегда, когда для проекта важно любое из перечисленных выше преимуществ *и* когда логику приложения потоковой обработки можно простым и естественным образом выразить на SQL.

Например, одна из самых замечательных особенностей ksqlDB — возможность расширения встроенной библиотеки с пользовательскими функциями на Java. Возможно, вы захотите использовать эту возможность на регулярной основе, но если обнаружится, что вы часто работаете на уровне JVM, то стоит оценить, насколько правильно вы выбрали уровень абстракции (например, иногда может оказаться предпочтительнее использовать Kafka Streams).

Есть еще несколько случаев, когда лучше подходит Kafka Streams. Например, если нужен низкоуровневый доступ к состоянию приложения, требуется периодически запускать функции для обработки данных, данные хранятся в формате, который не поддерживается в ksqlDB, нужна большая гибкость для профилирования/мониторинга приложений (например, распределенная трассировка, сбор нестандартных метрик и т. д.) или имеется много бизнес-логики, которую нелегко выразить в SQL, то тогда лучше использовать Kafka Streams.

Мы обсудили варианты, когда предпочтительнее использовать ksqlDB, и отметили, какие преимущества она предлагает. Далее, чтобы лучше понять суть отдельных компонентов (Kafka Streams и Kafka Connect), посмотрим, как ksqlDB способствовала их развитию и улучшению.

Эволюция базы данных нового типа

Знание, как ksqlDB развивалась и какие возможности приобрела, будет очень полезно. Конечно, эволюция ksqlDB интересна сама по себе, но этот раздел служит более важной цели, чем просто обзор ее истории. Поскольку ksqlDB раньше была известна под другим названием (KSQL), знание, когда появились те или иные функции, поможет вам различать разные поколения этой технологии.

Для начала посмотрим, как развивалась интеграция ksqlDB с Kafka Streams и как Kafka Streams поддерживает одну из самых фундаментальных возможностей ksqlDB: запросы данных.

Интеграция с Kafka Streams

Первые два года своего существования ksqlDB была известна как *KSQL*. В самом начале разработка была сосредоточена на основной функции: потоковом механизме SQL, способном анализировать и компилировать операторы SQL в полноценные приложения потоковой обработки. В ту пору KSQL концептуально представляла собой смесь традиционной базы данных SQL и Kafka Streams, заимствуя функциональную основу из реляционных баз данных (RDBMS) и используя Kafka Streams для выполнения потоковой обработки (рис. 8.1).

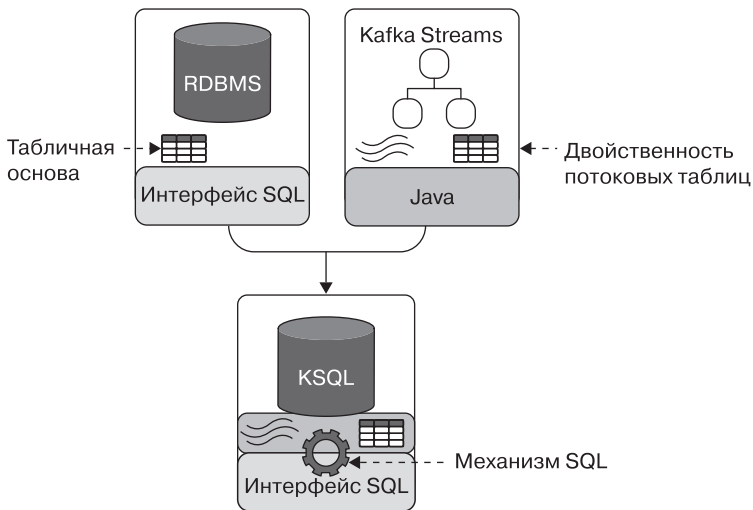


Рис. 8.1. На первом этапе развития ksqlDB являла собой сочетание Kafka Streams с функциями традиционных баз данных SQL, включая интерфейс SQL

Самая примечательная особенность, которую KSQL заимствовала из эволюционного дерева RDBMS, — это интерфейс SQL. Он устранил языковой барьер для создания приложений потоковой обработки в экосистеме Kafka, благодаря чему исчезла необходимость использовать язык JVM, такой как Java или Scala, для программирования взаимодействий с Kafka Streams.

ПОЧЕМУ SQL

Язык SQL сам по себе является результатом длительной разработки и упрощения продукта. Извлечение данных из хранилищ раньше требовало писать длинные программы на сложных языках. Однако благодаря применению математических обозначений и декларативных конструкций реляционные языки стали гораздо более компактными и эффективными. Дополнительные раунды лингвистической оптимизации сделали новый язык SQL более похожим на естественный английский язык¹. В результате всех этих раундов упрощения получился очень доступный для понимания, пользующийся широкой популярностью по сей день язык написания запросов к хранилищам данных. Адаптировав SQL к потоковой обработке, ksqlDB автоматически получила все преимущества классического SQL:

- лаконичный и выразительный синтаксис, который воспринимается как предложения на английском языке;
- декларативный стиль программирования;
- пологую кривую обучения.

В основу грамматики SQL лег ANSI SQL, однако для моделирования данных в потоках и в таблицах потребовался специальный диалект. Традиционные базы данных SQL в первую очередь основаны на табличном представлении и не имеют встроенной поддержки неограниченных наборов данных (потоков). В классическом SQL это проявляется в виде двух типов операторов:

- классические операторы языка определения данных (Data Definition Language, DDL) ориентированы на создание и уничтожение объектов базы данных (таблиц, самих баз данных, представлений и т. д.):

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

- классические операторы языка манипулирования данными (Data Manipulation Language, DML) ориентированы на чтение и манипулирование данными в таблицах:

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

Диалект SQL, реализованный в KSQL (и, соответственно, в ksqlDB), добавляет к классическому SQL поддержку потоков. В следующей главе мы подробно рассмотрим дополнительные операторы DDL и DML, но уже сейчас стоит от-

¹ Chamberlin D. D. Early History of SQL // IEEE Annals of the History of Computing. Vol. 34. No. 4. Oct. — Dec. 2012. P. 78–82.

метить, что операторы `CREATE TABLE` и `DROP TABLE` имеют эквиваленты для работы с потоками (`CREATE STREAM`, `DROP STREAM`), а дополнительные операторы DML позволяют описывать запросы и к потокам, и к таблицам.

Также важно, что более ранняя форма этой технологии, KSQL, использовала Kafka Streams в основном для поддержки запросов на передачу данных (push-запросов), которые выполняются постоянно, поддерживают как потоки, так и таблицы и выдают (или *передают*) результаты клиенту всякий раз, когда появляются новые данные. На рис. 8.2 показан поток данных, генерируемый push-запросами.



Запросы в KSQL еще не назывались push-запросами. Эта терминология появилась позже, хотя в этой книге мы используем ее для описания более ранней формы запросов KSQL, потому что она достаточно точно описывает порядок передачи результатов клиентам.

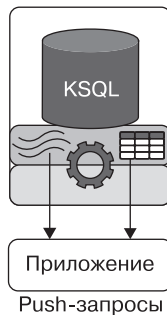


Рис. 8.2. Push-запросы автоматически передают результаты всякий раз, когда появляются новые данные; это позволяет приложениям/клиентам просто ждать и получать новые данные

Со временем механизм SQL приобрел дополнительные возможности, и, когда KSQL была переименована в ksqlDB, вместе с изменением названия появилась важная функция: возможность выполнять *запросы на получение* (pull-запросы). Pull-запросы очень похожи на обычные запросы в традиционных базах данных: они выполняются однократно и применяются для поиска данных по ключу. Внутри для выполнения pull-запросов используются Kafka Streams и хранилища состояний. Как рассказывалось в главе 4, хранилища состояний — это локальные хранилища ключей и значений, которые обычно поддерживаются RocksDB. На рис. 8.3 показаны потоки данных, порождаемые push- и pull-запросами, доступными в ksqlDB.

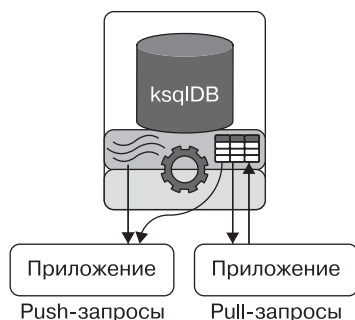


Рис. 8.3. ksqlDB поддерживает не только push-, но и pull-запросы

Оба типа запросов в значительной степени зависят от Kafka Streams, а также от собственного механизма SQL в ksqlDB, который мы подробнее рассмотрим ниже, в разделе «Архитектура». А теперь, познакомившись с интеграцией с Kafka Streams, перейдем к интеграции с Kafka Connect и узнаем, как она решает другие задачи потоковой обработки, отличные от задач, решаемых интеграцией с Kafka Streams.

Интеграция с Kafka Connect

Как рассказывалось в главе 2, приложения Kafka Streams читают и записывают данные в темы Kafka. Поэтому если данные для обработки хранятся вне Kafka или если требуется поместить результаты во внешнее хранилище, то необходимо создать конвейер данных, который переместит данные из соответствующих систем или в них. Эти процессы ETL обычно реализуются с применением отдельного компонента экосистемы Kafka: Kafka Connect. Поэтому в случае, когда используется только Kafka Streams, вам придется самостоятельно развернуть Kafka Connect и настроить соответствующие коннекторы для приема/передачи.

Первоначально KSQL накладывала те же ограничения, что и Kafka Streams. Интеграция источников данных, отличных от Kafka, требовала дополнительных архитектурных сложностей и операционных издержек, потому что управление коннекторами должно было осуществляться отдельной системой. Однако с превращением KSQL в более продвинутую форму ksqlDB появились новые возможности ETL, обусловленные добавлением интеграции с Kafka Connect. Эта интеграция включает в себя следующее.

- Дополнительные конструкции SQL для определения коннекторов, соединяющих источники с приемниками. Примеры мы представим в следующей главе, но вот для начала фрагмент кода на расширенном языке DDL:


```
CREATE SOURCE CONNECTOR `jdbc-conector` 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'  
);
```

- Возможность управлять коннекторами и выполнять их в кластере Kafka Connect или запускать распределенный кластер Kafka Connect вместе с ksqlDB для упрощения настройки.

Интеграция с Kafka Connect позволяет ksqlDB поддерживать полный жизненный цикл ETL потока событий, а не только часть процесса ETL, отвечающую за преобразование, которая поддерживается интеграцией Kafka Streams. На рис. 8.4 показан обновленный взгляд на ksqlDB, включающий возможности интеграции и преобразования данных.

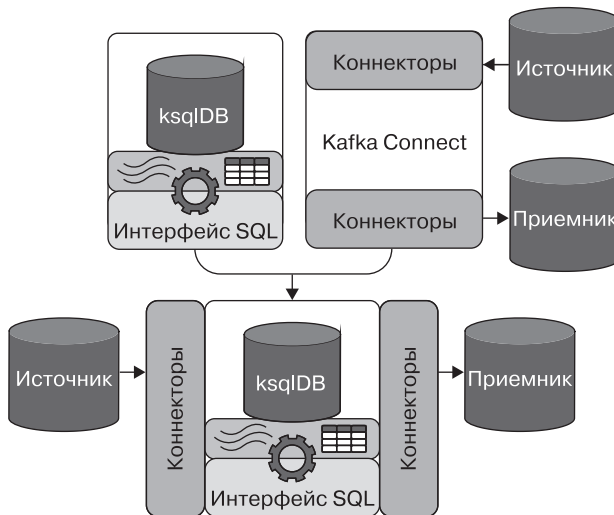


Рис. 8.4. ksqlDB превратилась в систему, которая поддерживает как преобразование данных, так и интеграцию

Обзор интеграции с Kafka Streams, обеспечивающей поддержку запросов в ksqlDB, и интеграции с Kafka Connect, помогающей сделать доступными для запросов дополнительные источники данных, позволяет понять, насколько широкими возможностями обладает ksqlDB.

Сравнение ksqlDB с традиционной базой данных SQL

Теперь, увидев, как ksqlDB превратилась в базу данных нового типа (*базу данных потоковой передачи событий*), сравним ее с традиционной базой данных SQL. В конце концов, узнав, что вы запускаете запросы, которые могут продолжать выполняться в течение нескольких месяцев (push-запросы), ваш администратор баз данных захочет получить ответы на некоторые вопросы. Начнем с перечисления сходств между ksqlDB и традиционной базой данных SQL.

Сходства

ksqlDB ориентирована на потоковую передачу данных, но, несмотря на это, она обладает многими чертами традиционной базы данных SQL.

SQL-интерфейс

ksqlDB, как и традиционная база данных SQL, имеет грамматику SQL, синтаксический анализатор и механизм выполнения. Это означает, что взаимодействие с данными в системах обоих типов можно осуществлять с использованием SQL — декларативного языка высокого уровня. Дialect SQL в ksqlDB содержит ожидаемые языковые конструкции, в том числе **SELECT** для проецирования, **FROM** для определения источников, **WHERE** для фильтрации, **JOIN** для создания соединений и т. д.

Операторы DDL и DML

DDL и DML — это две широкие категории операторов, поддерживаемые как традиционными базами данных SQL, так и ksqlDB. Операторы DDL отвечают за управление объектами базы данных (например, таблицами в традиционных базах данных, таблицами и потоками в ksqlDB), а операторы DML применяются для чтения и управления данными.

Сетевая служба и клиенты для отправки запросов

Имеющие опыт работы с традиционными базами данных SQL наверняка будут ожидать поддержки возможности подключения к базе данных по сети и наличия клиентских реализаций по умолчанию (например, CLI) для отправки запросов. ksqlDB обладает всеми этими возможностями: в ней имеется своя сетевая служба, реализованная как REST API, а также графический интерфейс и интерфейс командной строки для интерактивной отправки запросов. Существует также клиент для Java (<https://oreil.ly/s6dDf>), с помощью которого можно взаимодействовать с сервером ksqlDB.

Схемы

Коллекции, с которыми вы взаимодействуете, определяются схемами, включающими имена и типы полей. Более того, подобно некоторым гибким системам баз данных (например, Postgres), ksqlDB поддерживает определяемые пользователем типы.

Материализованные представления

Для оптимизации производительности операций чтения пользователи традиционных баз данных иногда создают материализованные представления — именованные объекты, содержащие результаты запроса. В традиционных системах эти представления могут обновляться либо *пассивно* (обновление представления ставится в очередь или осуществляется вручную, по запросу), либо *активно* (при каждом поступлении новых данных). Активно обновляемые представления похожи на представление данных в ksqlDB: обновление потоков и таблиц выполняется сразу же, как только новые данные становятся доступными.

Встроенные функции и операторы для преобразования данных

ksqlDB, подобно многим традиционным базам данных SQL, имеет богатый набор функций и операторов для работы с данными. Мы подробно обсудим функции и операторы в разделе «Функции и операторы» главы 11, а пока лишь отмечу, что существует широкий спектр строковых и математических функций, функций для работы с временем, табличных и геопространственных функций и много других. Также поддерживается ряд операторов (+, -, /, *, %, || и т. д.) и имеется даже интерфейс для определения и подключения нестандартных функций на Java.

Репликация данных

В большинстве традиционных баз данных используется репликация с ведущим узлом, когда данные, записанные на ведущем узле, распространяются на *узел-последователь* (или в реплику). ksqlDB наследует эту стратегию репликации (<https://oreil.ly/k92Sd>) от Kafka (для данных базовой темы) и от Kafka Streams (для таблиц данных с состоянием через резервные реплики, как описано в подразделе «Резервные реплики» главы 6). В интерактивном режиме ksqlDB также использует *репликацию операторов*, записывая запросы во внутреннюю тему, называемую *темой команд*, гарантируя возможность выполнения одного и того же запроса несколькими узлами в кластере ksqlDB.

Как видите, ksqlDB обладает многими чертами, присущими традиционным базам данных SQL.

Далее рассмотрим черты, отличающие ksqlDB от других систем баз данных. Это поможет вам объяснить вашему другу, администратору баз данных, что опрос традиционной базы данных в бесконечном цикле — это не то же самое,

что использование базы данных потоковой передачи событий, такой как ksqlDB, имеющей встроенную поддержку неограниченных наборов данных. А также изложение отличий поможет разобраться вам самим, в каких случаях лучше использовать другие системы.

Отличия

Несмотря на множественные сходства с традиционными базами данных SQL, ksqlDB имеет некоторые существенные отличия от них.

Расширенные операторы DDL и DML

Классические операторы DDL и DML, поддерживаемые традиционными базами данных, ориентированы на моделирование и запросы данных в таблицах. Однако ksqlDB как база данных потоковой передачи событий имеет свой взгляд на мир. Она распознает двойственность потоков/таблиц, описанную в подразделе «Потоково-табличный дуализм» главы 2, и поэтому ее диалект SQL поддерживает моделирование и запросы данных из потоков и таблиц. Она также поддерживает новые объекты базы данных, обычно отсутствующие в других системах: *коннекторы*.

Push-запросы

В большинстве традиционных баз данных SQL запросы выполняются к текущему моментальному снимку данных и завершаются, как только будут выполнены или возникнут ошибки. Такие короткоживущие запросы поддерживаются и в ksqlDB, но, так как ksqlDB способна работать с неограниченными потоками событий, она дополнительно поддерживает запросы, способные непрерывно выполняться месяцами или даже годами, выдавая результаты при получении новых данных. Это означает, что ksqlDB имеет лучшую поддержку клиентов, позволяя им подписаться на получение изменений данных.

Простота поддержки запросов

ksqlDB — это узкоспециализированная база данных для обработки запросов к материализованным представлениям, постоянно поддерживаемым в актуальном состоянии с помощью push-запросов или (в интерактивном режиме) с помощью pull-запросов. Она не пытается обеспечить те же возможности, которыми обладают аналитические хранилища (например, Elasticsearch), реляционные системы (например, Postgres, MySQL) или другие типы специализированных хранилищ данных¹. Ее механизм запросов адаптирован для более узкого круга вариантов использования, включая потоковую обработку, материализованные кэши и микросервисы, управляемые событиями.

¹ Дополнительную информацию вы найдете в статье Джея Крепса (Jay Kreps) *Introducing ksqlDB* на сайте Confluent (<https://oreil.ly/lo9LY>).

Более сложные стратегии управления схемами

Схемы можно определять, используя сам SQL, как можно было бы ожидать. Однако их можно также хранить в отдельном реестре схем (Confluent Schema Registry), обладающем дополнительными преимуществами, включая гарантии поддержки/совместимости схем на протяжении их эволюции, уменьшенный размер данных (за счет замены схемы ее идентификатором в сериализованных записях), автоматическое определение имени столбца/типа данных и упрощенную интеграцию с другими системами (так как нижестоящие приложения тоже могут извлекать схемы из реестра для десериализации данных, обрабатываемых ksqlDB).

SQL основан на стандарте ANSI, но не полностью совместим с ним

Попытки стандартизировать потоковый SQL начали предприниматься относительно недавно, поэтому диалект SQL в ksqlDB содержит конструкции, отсутствующие в стандарте.

Высокая доступность, отказоустойчивость и аварийное переключение работают гораздо более плавно

Это не отдельные надстройки или корпоративные функции, как в некоторых системах. Они встроены в ksqlDB и легко настраиваются¹.

Локальное и удаленное хранилище

Данные, отображаемые средствами ksqlDB, хранятся в Kafka, а при использовании таблиц материализуются в локальных хранилищах состояний. Здесь важно сделать пару интересных замечаний. Например, синхронизация/фиксация обрабатывается самой Kafka, а слой хранения может масштабироваться независимо от механизма SQL. Кроме того, совместное размещение вычислительных ресурсов с данными (то есть хранилищ состояний) дает дополнительные преимущества производительности, а также улучшенную надежность и масштабируемость собственного распределенного хранилища Kafka.

Модель согласованности

ksqlDB придерживается асинхронной модели, которая приводит к согласованности в конечном итоге, тогда как многие традиционные системы более тесно придерживаются модели ACID (atomicity, consistency, isolation, durability — «атомарность, непротиворечивость, изоляция, долговечность»).

Итак, какой вывод можно сделать после знакомства со сходствами и различиями между ksqlDB и традиционными базами данных SQL? ksqlDB обладает многими чертами традиционных баз данных, но она не стремится их заменить.

¹ Например, горячее резервирование поддерживается через настройку резервной реплики в Kafka Streams.

Это специализированный инструмент, который можно использовать для потоковой передачи, а когда нужны возможности другой системы, интеграция с Kafka Connect поможет вам переместить любые обогащенные, преобразованные или иным образом обработанные данные в хранилище данных по вашему выбору.

Прежде чем заняться установкой ksqlDB, рассмотрим кратко ее архитектуру.

Архитектура

ksqlDB основана на Kafka Streams, поэтому достаточно просмотреть обсуждение архитектуры потоковой передачи в главе 2, чтобы понять, как работает интеграция с Kafka Streams на более низком уровне. В этом разделе основное внимание уделяется компонентам архитектуры, характерным для ksqlDB, которые разбиты на две основные группы: серверы и клиенты ksqlDB.

Сервер ksqlDB

Сервер ksqlDB отвечает за выполнение приложений потоковой обработки (которые в ksqlDB выглядят как набор запросов, выполняемых для решения общей бизнес-задачи). Каждый сервер концептуально подобен одному экземпляру приложения Kafka Streams, а рабочие нагрузки, созданные набором запросов, могут распределяться между несколькими серверами ksqlDB с одинаковой конфигурацией `ksql.service.id`. Так же как приложения Kafka Streams, серверы ksqlDB развертываются отдельно от кластера Kafka (обычно на машинах/контейнерах отдельно от брокеров).

Группа взаимодействующих серверов ksqlDB называется *кластером ksqlDB*, и обычно рекомендуется изолировать рабочие нагрузки одного приложения на уровне кластера. Например, на рис. 8.5 показаны два кластера ksqlDB, каждый со своим идентификатором службы, выполняющей изолированные рабочие нагрузки, которые могут масштабироваться и управляться независимо друг от друга.

Если потребуется увеличить мощность кластера ksqlDB, можете развернуть больше серверов ksqlDB. Масштабирование можно также выполнять в обратную сторону, удаляя серверы ksqlDB. Поскольку серверы ksqlDB с одинаковым идентификатором службы являются членами одной и той же группы потребителей, Kafka автоматически обрабатывает переназначение/распределение работы при добавлении или удалении новых серверов ksqlDB (удаление может производиться вручную или автоматически, например, в результате системного сбоя).

Каждый сервер ksqlDB состоит из двух компонентов: ядра SQL и службы REST. Мы обсудим эти компоненты в следующих разделах.

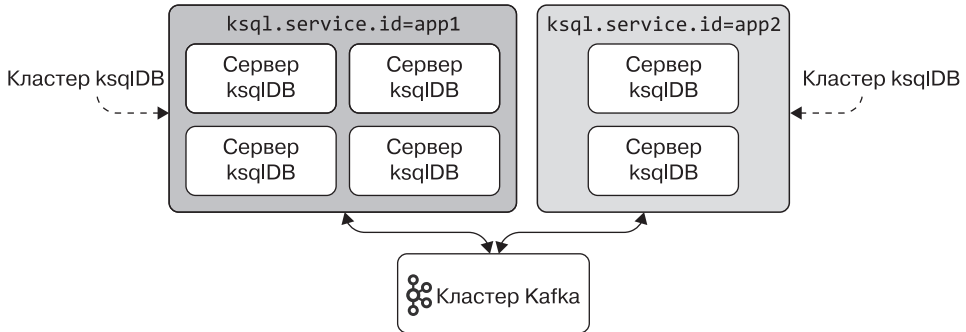


Рис. 8.5. Два кластера ksqlDB независимо обрабатывают данные из Kafka

Механизм SQL

Механизм SQL отвечает за синтаксический анализ операторов SQL, их преобразование в одну или несколько топологий Kafka Streams и, наконец, запуск приложений Kafka Streams. Этот процесс показан на рис. 8.6.

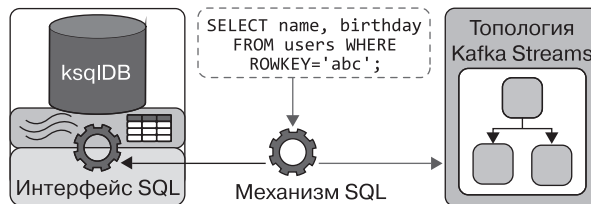


Рис. 8.6. Механизм SQL преобразует операторы SQL в топологии Kafka Streams

Сам синтаксический анализатор использует инструмент ANTLR (<https://antlr.org/>), который преобразует инструкцию SQL в абстрактное синтаксическое дерево (Abstract Syntax Tree, AST; <https://oreil.ly/8ScW6>), каждый узел которого представляет опознанную фразу или токен в исходном запросе. ksqlDB обходит все узлы в дереве синтаксического анализа и выстраивает топологию Kafka Streams, используя найденные токены. Например, если в запросе включен оператор `WHERE`, то ksqlDB использует оператор `filter` без состояния. Точно так же если запрос включает условие соединения (например, `LEFT JOIN`), то ksqlDB добавит в топологию оператор `left Join`. Узлы-источники определяются по значению `FROM`, а для создания проекции используется оператор `SELECT`.

После создания топологии обработки, необходимой для выполнения запроса, запускается получившееся в результате приложение Kafka Streams. Теперь давайте взглянем на компонент, передающий запросы механизму SQL: службу REST.

Служба REST

ksqlDB имеет интерфейс REST, позволяющий клиентам взаимодействовать с механизмом SQL. В основном он используется интерфейсом командной строки ksqlDB, графическим интерфейсом ksqlDB и другими клиентами для передачи запросов механизму (то есть инструкций DML, начинающихся с оператора `SELECT`), выполнения других типов инструкций (например, инструкций DDL), проверки состояния кластера и т. д. По умолчанию служба прослушивает порт 8088 и обменивается данными по протоколу HTTP, но при желании можно изменить конечную точку с помощью конфигурационного параметра `listeners` и включить поддержку протокола HTTPS с помощью конфигурационных параметров `ssl`. Ниже показаны эти два набора конфигураций:

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

REST API является необязательным и, в зависимости от режима работы, как описывается в разделе «Режимы развертывания» ниже, его можно полностью отключить. Однако API необходим для интерактивной работы с ksqlDB независимо от того, какой из клиентов используется — интерфейс командной строки, графический интерфейс ksqlDB (которые мы обсудим в следующем разделе) или пользовательский клиент. Запросы можно отправлять даже с помощью `curl`, как показано ниже:

```
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": {}
  }'
```

При непосредственном взаимодействии с API следует обращаться к самой свежей справке по REST API в документации ksqlDB (https://oreil.ly/J_wNH). ksqlDB все еще продолжает активно развиваться, и, как мне кажется, это одна из областей, в которой в будущем произойдут изменения, основанные на некоторых предложениях, представленных в проекте ksqlDB. В большинстве примеров этой книги API будет использоваться косвенно через один из официально поддерживаемых клиентов, о котором мы поговорим далее.

Клиенты ksqlDB

В предыдущем разделе вы узнали, что серверы `ksqlDB` поддерживают интерфейс `REST` для отправки запросов и получения информации о кластере `ksqlDB`. Вы также узнали, что со службой `REST` можно взаимодействовать с помощью `curl` или пользовательских клиентов, но в большинстве случаев для взаимодействия с серверами `ksqlDB` применяются официальные клиенты. В этом разделе мы поговорим о таких клиентах, начав с `ksqlDB CLI`.

Интерфейс командной строки ksqlDB CLI

Интерфейс командной строки `ksqlDB CLI` — это приложение, позволяющее взаимодействовать с сервером `ksqlDB`. Оно отлично подходит для экспериментов, позволяя отправлять запросы, просматривать темы, настраивать `ksqlDB` и многое другое в интерактивном режиме. Этот клиент распространяется как образ `Docker` (`confluentinc/ksqldb-cli`), а также как часть платформы `Confluent` (<https://oreil.ly/eMqcJ>; полностью управляемой `Confluent Cloud` или развертываемой самостоятельно).

Вызов интерфейса командной строки предполагает запуск команды `ksql` и передачу ей комбинации «хост/порт» сервера `ksqlDB` (это соответствует конфигурационному параметру `listeners`, как описано в пункте «Служба REST» выше):

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

После запуска команда `ksq1` выводит приглашение, как показано ниже:

```
=====
=
=      | | _____ | | | | | | )
=      | / / _ / \ | | | | | | \
=      | < \ \ ( | | | | | | ) |
=      | \ \ \ \ , | | | | / |
=              | |
=
= 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>

В этой книге такое приглашение будет служить отправной точкой при изучении диалекта SQL и работе с учебными проектами. Прежде чем начать использовать интерфейс командной строки, рассмотрим еще один клиент: графический интерфейс ksqlDB.

Графический интерфейс ksqlDB

Для взаимодействия с ksqlDB платформа Confluent Platform включает также графический интерфейс ksqlDB UI. Это коммерческое предложение, поэтому его можно найти только в версии Confluent Platform с коммерческой лицензией, а также в Confluent Cloud (где Confluent Platform запускается в полностью управляемой облачной среде). Помимо возможности отправлять запросы из веб-редактора, также можно визуализировать поток данных, создавать новые потоки и таблицы с помощью веб-форм, просматривать список запущенных запросов и многое другое. Снимок графического интерфейса ksqlDB UI показан на рис. 8.7.

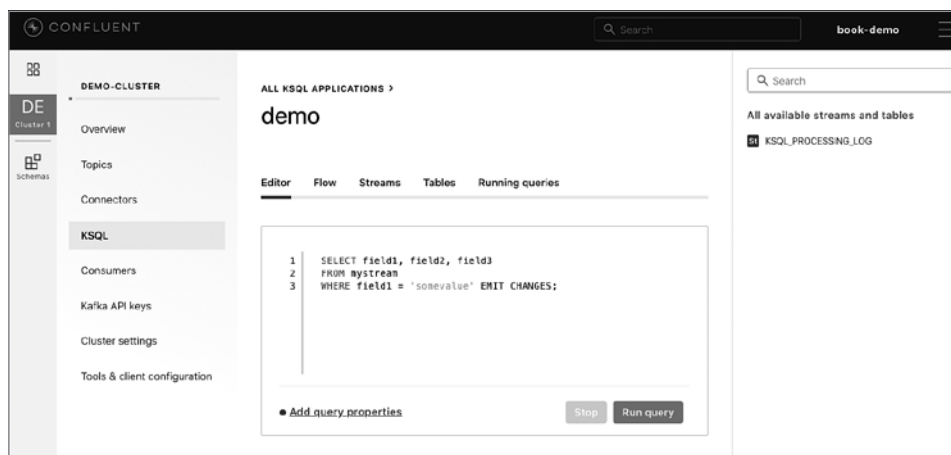


Рис. 8.7. Графический интерфейс ksqlDB UI в Confluent Cloud

Режимы развертывания

ksqlDB поддерживает два режима развертывания в зависимости от уровня интерактивности, выбранного для работы с серверами ksqlDB. В этом разделе описывается каждый из режимов и обсуждается, когда и какой из них лучше использовать.

Интерактивный режим

При запуске ksqlDB в интерактивном режиме клиенты могут отправлять новые запросы в любое время с помощью REST API. Как следует из названия, этот режим позволяет интерактивно взаимодействовать с серверами ksqlDB и вручную создавать и удалять потоки, таблицы, запросы и коннекторы.

На рис. 8.8 показана схема работы ksqlDB в интерактивном режиме. Одна из ключевых особенностей интерактивного режима состоит в том, что все запросы, отправленные в механизм SQL (через REST API), записываются во внутреннюю тему — *тему команд*. Эта тема создается и управляется сервером ksqlDB автоматически и используется для репликации операторов, что позволяет всем серверам ksqlDB в кластере выполнять запрос.

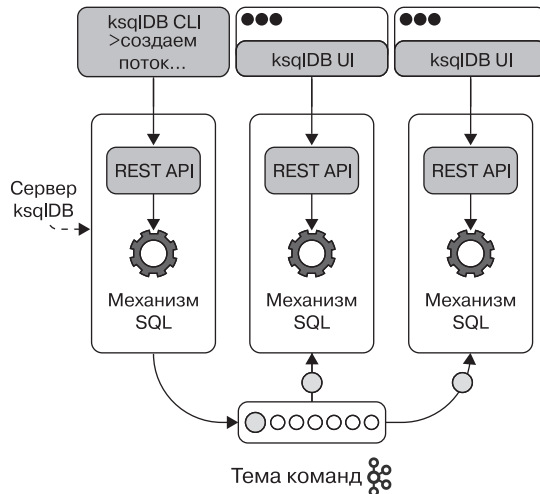


Рис. 8.8. Работая в интерактивном режиме, ksqlDB позволяет клиентам (например, интерфейсу командной строки ksqlDB, графическому интерфейсу, клиенту на Java или даже curl) отправлять запросы через REST API

Интерактивный режим развертывания используется по умолчанию, и для работы в этом режиме не требуется никаких специальных настроек. Однако если вы решите отключить интерактивный режим и использовать автономный, то вам придется выполнить специальные настройки. Мы обсудим автономный режим в следующем разделе.

Автономный режим

В некоторых случаях нежелательно давать возможность клиентам отправлять запросы кластеру ksqlDB в интерактивном режиме. Например, если понадобится заблокировать доступ к разворачиванию в промышленном окружении, вы можете настроить автономный режим (который отключает REST API) и гарантировать невозможность изменения выполняемых запросов. Для перехода в автономный режим нужно создать файл, содержащий любые постоянные запросы, которые должен выполнять механизм SQL, и указать путь к этому файлу, используя конфигурационный параметр `queries.file` сервера ksqlDB. Например:

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

❶ Путь к файлу с запросами, которые могут запускаться на сервере ksqlDB, задается с помощью свойства `queries.file`.

Схема работы ksqlDB в автономном режиме показана на рис. 8.9. Обратите внимание, что, в отличие от интерактивного режима, автономный режим не использует тему команд для репликации операторов. Однако он записывает некоторые внутренние метаданные в тему конфигурации.

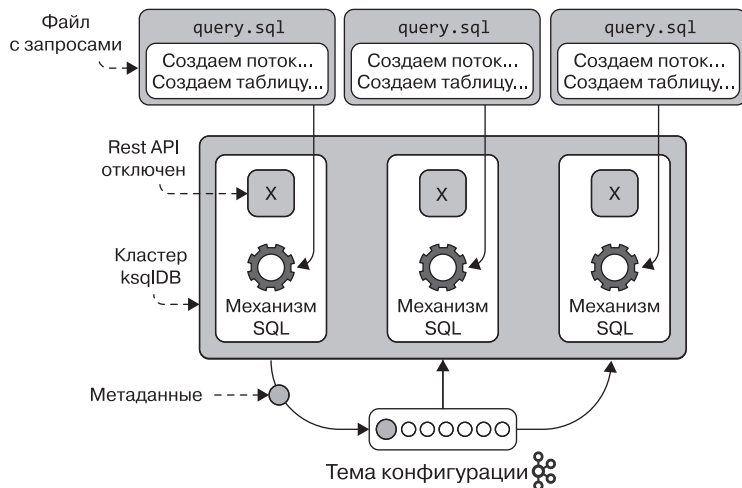


Рис. 8.9. Схема работы ksqlDB в автономном режиме

Большинство учебных проектов в этой книге мы будем выполнять в интерактивном режиме. А теперь давайте наконец перейдем к первому нашему проекту в стиле Hello, world.

Учебный проект

В этом разделе мы рассмотрим очень простой учебный проект Hello, world. То есть создадим простое приложение потоковой обработки с помощью ksqlDB; оно будет приветствовать каждого пользователя, имя которого мы запишем в тему Kafka с именем `users`. А начнем мы с установки ksqlDB.

Установка ksqlDB

Начать работу с ksqlDB можно несколькими способами. В табл. 8.1 перечислены наиболее популярные из них.

Таблица 8.1. Способы начать работу с ksqlDB

Метод установки	Ссылка	Примечания
Загрузить Confluent Platform	https://www.confluent.io/download	ksqlDB — это программный компонент с общедоступной лицензией при установке в составе автономной версии Confluent Platform
Использовать Confluent Cloud	https://confluent.cloud	Ничего загружать не требуется, просто нужно создать учетную запись
Загрузить и запустить официальный образ Docker	Образы сервера и клиента командной строки ksqlDB: <ul style="list-style-type: none"> https://hub.docker.com/r/confluentinc/ksqldb-server https://hub.docker.com/r/confluentinc/ksqldb-cli 	Этот метод потребует установить и запустить все необходимые зависимости (Kafka, Schema Registry, ksqlDB CLI)
Клонировать репозиторий на GitHub и собрать компоненты из исходного кода	https://github.com/confluentinc/ksql	Самый сложный вариант

Самый простой способ из перечисленных в таблице выше — использовать официальные образы Docker. Репозиторий с примерами для этой книги включает развертывание Docker Compose, поддерживающее все связанные службы с помощью соответствующих образов Docker. Полные инструкции вы найдете на GitHub (<https://oreil.ly/Mk4Kt>).

В остальных примерах кода этой главы будут демонстрироваться исходные команды, необходимые для запуска сервера ksqlDB, интерфейса командной строки, предварительного создания темы Kafka и т. д. Там, где это применимо, будут приводиться инструкции, описывающие, как выполнять эти команды с помощью Docker Compose. Они поясняются отдельно, поскольку более характерны для рабочего процесса на основе Docker Compose.

После установки ksqlDB можно запустить сервер ksqlDB. Как это сделать, рассказывается в следующем разделе.

Запуск сервера ksqlDB

После установки нужно создать конфигурацию для сервера ksqlDB и определить несколько конфигурационных свойств, но пока не будем слишком углубляться и просто сохраним два наиболее важных параметра в файл с именем *ksql-server.properties*:

```
listeners=http://0.0.0.0:8088 ❶  
bootstrap.servers=kafka:9092 ❷
```

❶ Конечная точка REST API для сервера ksqlDB. Она привязана ко всем интерфейсам IPv4.

❷ Список пар «хост/порт», соответствующих одному или нескольким брокерам Kafka. Он будет использоваться для установки соединения с кластером Kafka.

После сохранения конфигурации сервера ksqlDB можно запустить сервер ksqlDB, выполнив следующую команду:

```
ksql-server-start ksql-server.properties ❶
```

❶ В рабочем процессе Docker Compose эта команда задается в файле `docker-compose.yml`. Дополнительную информацию ищите в репозитории с примерами кода для этой главы.

После запуска команды в течение загрузки ksqlDB в консоли появится много информации. Среди прочего вы должны увидеть примерно такую строку:

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

Как отмечалось выше, в разделе «Режимы развертывания», по умолчанию развертывается интерактивный режим, и, поскольку мы не установили конфигурационный параметр `queries.file` в файле `ksql-server.properties`, внутри

нашего сервера ksqlDB будет запущена служба REST. В результате мы сможем использовать один из клиентов по умолчанию для отправки запросов. В этом проекте будем применять интерфейс командной строки ksqlDB CLI.

Предварительное создание тем

В следующих главах вы узнаете, как в ksqlDB реализовать автоматическое создание тем, установив определенные параметры в инструкциях SQL. А пока просто создадим тему `users`, выполнив следующую команду:

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

❶ При работе в Docker Compose добавьте к этой команде префикс `docker-compose exec kafka`.

Теперь можно начинать использовать интерфейс командной строки ksqlDB CLI.

Использование интерфейса командной строки ksqlDB CLI

Давайте создадим приложение потоковой обработки `Hello, world`, определив набор запросов для выполнения сервером ksqlDB. Поскольку для отправки запросов будет использоваться клиент командной строки, то первое, что нужно сделать, — это запустить команду `ksql` и передать ей адрес конечной точки REST нашего сервера ksqlDB:

```
ksql http://0.0.0.0:8088 ❶
```

❶ При работе в Docker Compose выполните команду `docker-compose exec ksqldb-cli ksql http://ksqldb-server:8088`.

После этого вы окажетесь в сеансе клиента командной строки. Отсюда можно запускать различные запросы и инструкции, а также настраивать различные конфигурационные параметры ksqlDB. Например, выполните следующую инструкцию `SET`, чтобы гарантировать чтение наших запросов с начала темы в Kafka:

```
SET 'auto.offset.reset' = 'earliest';
```

Мы уже создали тему `users`, поэтому прямо сейчас можем выполнить команду `SHOW TOPICS`, чтобы проверить ее наличие и просмотреть некоторые настройки темы (например, количество разделов и реплик).

Эта команда и ее результат показаны ниже:

```
ksql> SHOW TOPICS ;
Kafka Topic | Partitions | Partition Replicas
-----
users       | 4          | 1
-----
```

Теперь смоделируем данные в теме `users` как поток, выполнив инструкцию DDL `CREATE STREAM`. Вместе с ней передадим дополнительную информацию о типах и форматах данных в этой теме. В результате будет создан поток, который мы можем запросить:

```
CREATE STREAM users (
  ROWKEY INT KEY, ❶
  USERNAME VARCHAR ❷
) WITH ( ❸
  KAFKA_TOPIC='users', ❹
  VALUE_FORMAT='JSON' ❺
);
```

❶ `ROWKEY` соответствует ключу записи в Kafka. В нашем случае ключ имеет тип `INT`.

❷ Здесь указывается, что записи в теме `users` имеют поле `USERNAME` с типом `VARCHAR`.

❸ Оператор `WITH` используется для передачи дополнительных свойств. Перечисленные здесь и другие дополнительные свойства мы обсудим в подразделе «Оператор `WITH`» главы 10.

❹ Поток читает из темы `users`, как определено в свойстве `KAFKA_TOPIC` в операторе `WITH`.

❺ Формат сериализации значений записи в Kafka.

После запуска инструкции `CREATE STREAM` должно появиться подтверждение, как показано ниже:

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


Прежде чем запросить поток `users`, добавим в него некоторые тестовые данные, используя инструкцию `INSERT INTO`:

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

Теперь, после создания и заполнения потока тестовыми данными, можно создать push-запрос, который будет приветствовать каждого пользователя, появляющегося в потоке `users`. Выполните инструкцию из примера 8.1, чтобы создать непрерывный и временный (то есть непостоянный)¹ push-запрос.

Пример 8.1. Push-запрос в стиле Hello, world

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

❶ Добавив оператор `EMIT CHANGES`, мы сообщаем серверу ksqlDB, что выполняется push-запрос, автоматически отправляющий изменения клиенту (в данном случае — клиенту командной строки).

Поскольку мы добавили в поток `users` некоторые тестовые данные и установили значение `earliest` в свойстве `auto.offset.reset`, в консоли немедленно должен появиться следующий вывод:

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

Обратите внимание, что запрос `SELECT` будет продолжать выполняться даже после вывода первоначального набора результатов. Как упоминалось выше, это свойственно push-запросам, которые продолжают выполняться и выдавать результаты, пока вы не завершите их. А если включить в запрос оператор `LIMIT`, то запрос будет продолжать выполняться до достижения указанного предела.

Можно также открыть другой сеанс CLI и выполнить дополнительные инструкции `INSERT INTO`, чтобы добавить больше данных в поток `users`. Всякий раз после добавления новой записи push-запрос, представленный оператором `SELECT`, будет выводить результаты в консоль.

¹ Под словом «временный» подразумевается, что результаты запроса не будут записаны обратно в Kafka. Постоянные запросы мы рассмотрим в следующей главе.

На этом мы завершаем наш простой учебный проект для этой главы, но это лишь верхушка айсберга возможностей ksqlDB. В следующих нескольких главах подробнее изучим язык запросов и поработаем с некоторыми интересными учебными проектами, чтобы вы могли лучше познакомиться с ksqlDB.

Заключение

В этой главе вы получили первое представление об истории развития ksqlDB, узнали, с какой целью была создана эта база данных и когда следует ее использовать. Мы также сравнили ее с традиционными базами данных и взглянули на ее архитектуру. В следующей главе продолжим изучение ksqlDB и рассмотрим способы интеграции с внешними источниками данных.

Интеграция данных в ksqlDB

Первый шаг при создании приложения потоковой обработки с помощью ksqlDB — обзор источников данных для обработки и выяснение, куда в конечном итоге будут записаны результаты обогащения/преобразования. Поскольку внутри ksqlDB использует Kafka Streams, источниками и приемниками данных всегда будут темы Kafka. Кроме того, ksqlDB упрощает интеграцию с другими источниками данных, в том числе с такими популярными системами, как Elasticsearch, PostgreSQL, MySQL, Google PubSub, Amazon Kinesis, MongoDB, и многими другими.

Конечно, если данные уже находятся в Kafka и не планируется записывать результаты обработки во внешнюю систему, то средство интеграции данных в ksqlDB, которые работают под управлением Kafka Connect, не требуется. И все же если вам когда-нибудь понадобится читать или записывать данные во внешние системы, то в этой главе вы найдете необходимые основы, которые помогут вам подключить соответствующие источники и приемники данных с помощью ksqlDB и Kafka Connect.

Эта глава не претендует на роль исчерпывающего руководства по Kafka Connect — отдельному API в экосистеме Kafka, о котором можно рассказывать до бесконечности. Однако здесь будет представлено достаточно информации, чтобы начать работу, и перечислены высокоуровневые абстракции ksqlDB для работы с Connect API. Вот некоторые из тем, которые будут рассмотрены:

- краткий обзор Kafka Connect;
- режимы интеграции в Kafka Connect;
- настройка рабочих процессов Kafka Connect;
- установка коннекторов для связи с источниками и приемниками;
- создание, удаление и анализ коннекторов в ksqlDB;
- анализ коннекторов с помощью Kafka Connect API;
- просмотр схем коннекторов в реестре Confluent Schema Registry.

К концу этой главы вы освоите решение двух задач из трех, связанных с выполнением операций *Streaming ETL* (extract, transform, load — «извлечение, преобразование, загрузка») в ksqlDB, а именно:

- извлечение данных из внешней системы в Kafka;
- выгрузку данных из Kafka во внешнюю систему.

Недостающую часть аббревиатуры ETL — T (transform), то есть преобразование, — мы рассмотрим в следующих двух главах, потому что преобразование данных теснее связано с потоковой обработкой, чем с интеграцией данных. А пока начнем с краткого обзора Kafka Connect, чтобы поближе познакомиться с технологией, которая фактически поддерживает интеграцию данных в ksqlDB.

Обзор Kafka Connect

Kafka Connect — это один из пяти API в экосистеме Kafka¹, он используется для подключения к Kafka внешних хранилищ данных, API и файловых систем. Когда данные находятся в Kafka, их можно обрабатывать, преобразовывать и обогащать с помощью ksqlDB. Перечислю основные компоненты Kafka Connect.

Коннекторы

Коннекторы — это упакованные фрагменты кода, которые можно внедрить в рабочие процессы (обсудим их чуть ниже). Они способствуют перемещению данных между Kafka и другими системами и делятся на две категории:

- коннекторы-источники читают данные из внешних систем в Kafka;
- коннекторы-приемники записывают данные во внешние системы из Kafka.

Задачи

Задачи — это единицы работы внутри коннектора. Количество задач может быть разным, что позволяет контролировать объем работы, выполняемой одним рабочим процессом.

Рабочие процессы

Рабочие процессы (workers) — это процессы JVM, которые выполняют коннекторы. Можно развернуть несколько рабочих процессов, чтобы распараллелить/распределить работу и добиться отказоустойчивости в случае частичного сбоя (например, если один рабочий процесс неожиданно завершится).

¹ Другие четыре API: Consumer, Producer, Streams и Admin.

Конвертеры

Конвертеры — это код, осуществляющий сериализацию/десериализацию данных в Connect. Конвертер по умолчанию (например, `AvroConverter`) должен указываться на уровне рабочего процесса, но также есть возможность задавать конвертеры на уровне коннекторов.

Кластер Connect

Кластер Connect объединяет один или несколько рабочих процессов Kafka Connect, действующих вместе как группа и перемещающих данные в Kafka и из нее.

На рис. 9.1 показана схема работы всех этих компонентов.

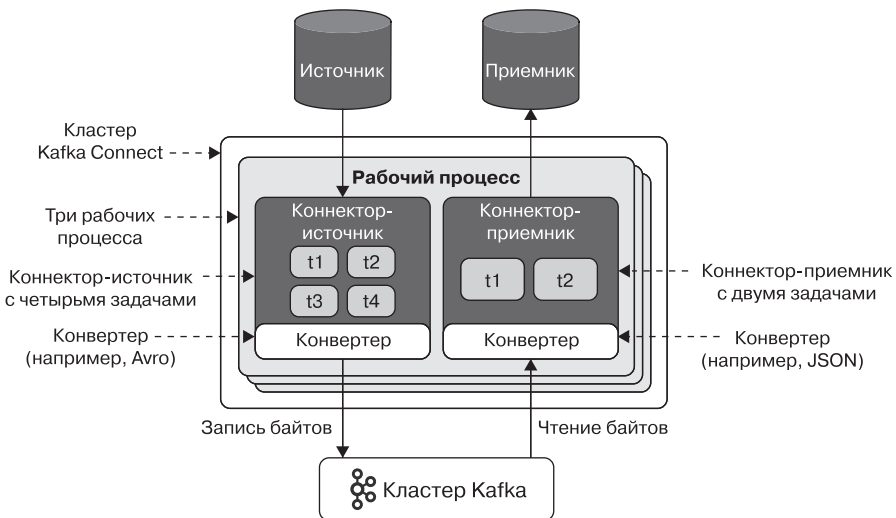


Рис. 9.1. Архитектура Kafka Connect

Может показаться, что все это будет трудно усвоить, но по мере чтения главы вы увидите, что `ksqlDB` значительно упрощает ментальную модель Kafka Connect. А теперь посмотрим на варианты развертывания Kafka Connect для использования с `ksqlDB`.

Внешняя и встроенная интеграция с Connect

Интеграция с Kafka Connect в `ksqlDB` может работать в двух разных режимах. В этом разделе описываются оба режима и рассказывается, когда их использовать. Начнем с внешней интеграции.

Внешняя интеграция

Если у вас уже есть готовый кластер Kafka Connect или вы хотите развернуть Kafka Connect отдельно от ksqlDB, то существует возможность использовать внешнюю интеграцию с Kafka Connect. Для этого необходимо в ksqlDB настроить URL кластера Kafka Connect, определив свойство `ksql.connect.url`. После этого ksqlDB сможет обращаться к внешнему кластеру Kafka Connect напрямую, создавать коннекторы и управлять ими. Пример конфигурации внешнего режима показан ниже (он будет сохранен в файле свойств сервера ksqlDB):

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

При работе в режиме внешней интеграции любые коннекторы (источники и приемники), необходимые приложению, должны действовать во внешних рабочих процессах. Обратите внимание, что при работе в режиме внешней интеграции рабочие процессы, как правило, размещаются отдельно от сервера ksqlDB, потому что одно из основных преимуществ этого режима заключается в отсутствии необходимости использования ресурсов компьютера совместно с ksqlDB. На рис. 9.2 показана схема работы Kafka Connect в режиме внешней интеграции.

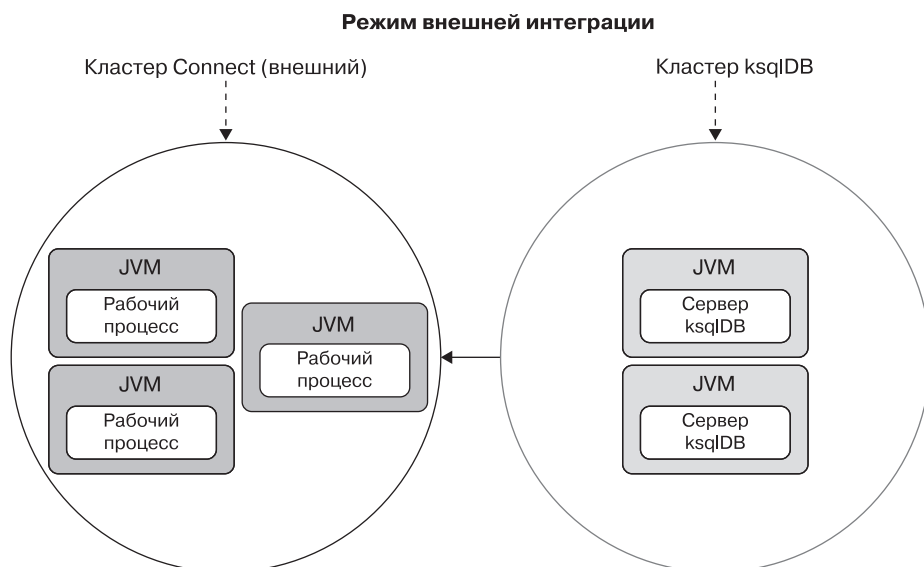


Рис. 9.2. Kafka Connect в режиме внешней интеграции

Вот некоторые ситуации, когда может появиться желание использовать режим внешней интеграции с Kafka Connect:

- требуется независимо масштабировать рабочие нагрузки и ввод/вывод данных и/или изолировать ресурсы для этих различных видов рабочих нагрузок;
- ожидается большой трафик через темы источников/приемников;
- уже есть действующий кластер Kafka Connect.

Далее рассмотрим режим встроенной интеграции, который используем в учебных проектах в этой книге.

Встроенная интеграция

В *режиме встроенной интеграции* рабочий процесс Kafka Connect выполняется под управлением той же JVM, что и сервер ksqlDB, в распределенном режиме Kafka Connect¹. Это означает возможность распределения работы между несколькими взаимодействующими экземплярами рабочего процесса. Количество рабочих процессов Kafka Connect совпадает с количеством серверов ksqlDB в кластере ksqlDB. Режим встроенной интеграции предпочтительнее использовать, когда:

- требуется одновременно масштабировать рабочие нагрузки потоковой обработки и ввода/вывода;
- ожидается небольшой или средний трафик через темы источников/приемников;
- желательны простота поддержки интеграции данных, отсутствие необходимости управлять отдельным развертыванием Kafka Connect и независимо масштабировать рабочие нагрузки интеграции/преобразования данных;
- допускается перезапуск рабочих процессов Kafka Connect с перезапуском серверов ksqlDB;
- допускается совместное использование вычислительных ресурсов/памяти ksqlDB и Kafka Connect².

Поскольку в режиме встроенной интеграции серверы ksqlDB сосуществуют вместе с рабочими процессами Kafka Connect, любые коннекторы источников/

¹ Kafka Connect имеет свои режимы развертывания (распределенный и автономный). Не путайте их с режимами интеграции ksqlDB с Kafka Connect (внешним и встроенным).

² Дополнительную информацию вы найдете по адресу <https://oreil.ly/fK6WQ>.

приемников, необходимые приложению, должны устанавливаться на том же узле, где работают серверы ksqlDB. На рис. 9.3 показана схема работы Kafka Connect в режиме встроенной интеграции.

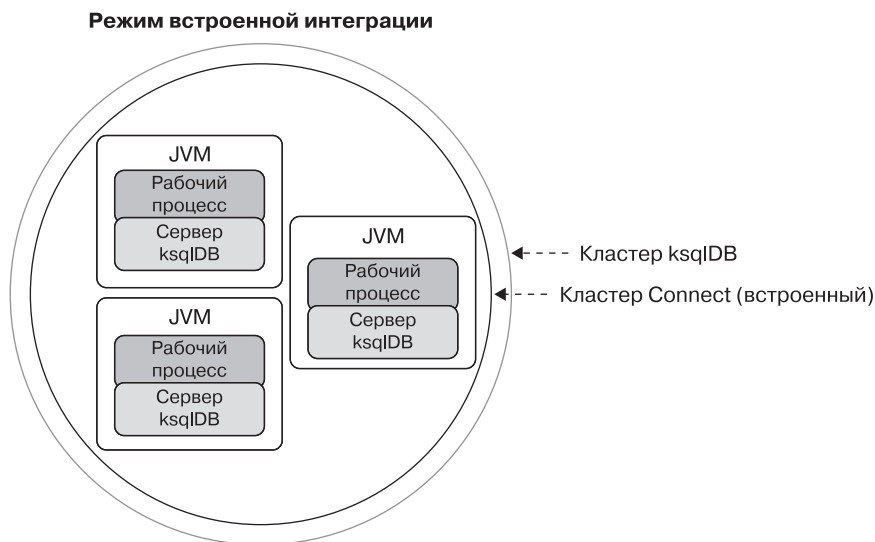


Рис. 9.3. Kafka Connect в режиме встроенной интеграции

Для запуска в режиме встроенной интеграции необходимо установить конфигурационное свойство `ksql.connect.worker.config` сервера ksqlDB, указав путь к конфигурациям рабочих процессов Kafka Connect. Не забывайте, что рабочие процессы — это процессы Kafka Connect, в рамках которых фактически действуют коннекторы источников и приемников. Вот пример настройки этого свойства в файле свойств сервера ksqlDB:

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

Но какая информация должна быть определена в конфигурационном файле рабочего процесса, на который ссылается свойство `ksql.connect.worker.config`? Мы поговорим об этом в следующем разделе.

Настройка рабочих процессов Connect

Kafka Connect имеет множество параметров настройки, подробно описанных в официальной документации Apache Kafka (<https://oreil.ly/UWnW3>). В этом разделе будут представлены только наиболее важные из них на примере настройки

рабочего процесса Kafka Connect. При запуске в режиме встроенной интеграции настройки следует определить в файле (например, `connect.properties`) и сослаться на него в свойстве `ksql.connect.worker.config` в конфигурации сервера ksqlDB. При запуске в режиме внешней интеграции настройки рабочего процесса передаются в аргументах запуска Kafka Connect. Пример конфигурации показан в следующем листинге:

```
bootstrap.servers=localhost:9092 ❶
group.id=ksql-connect-cluster ❷

key.converter=org.apache.kafka.connect.storage.StringConverter ❸
value.converter=org.apache.kafka.connect.storage.StringConverter ❹

config.storage.topic=ksql-connect-configs ❺
offset.storage.topic=ksql-connect-offsets
status.storage.topic=ksql-connect-statuses

errors.tolerance=all ❻

plugin.path=/opt/confluent/share/java/ ❼
```

❶ Список пар хост/порт брокеров Kafka, которые следует использовать для подключения к кластеру Kafka.

❷ Строковый идентификатор кластера Connect, которому принадлежит этот рабочий процесс. Рабочие процессы, настроенные с одним и тем же идентификатором `group.id`, принадлежат одному кластеру и могут совместно использовать рабочую нагрузку для выполнения коннекторов.

❸ «Класс конвертера для преобразования между форматом Kafka Connect и сериализованной формой. Управляет форматом *ключей* в сообщениях, записываемых в Kafka или извлекаемых из него, а поскольку класс не зависит от коннекторов, это позволяет любому коннектору работать с любым форматом сериализации. Примерами распространенных форматов могут служить JSON и Avro». (Документация Connect; <https://oreil.ly/08AW5>.)

❹ «Класс конвертера для преобразования между форматом Kafka Connect и сериализованной формой. Управляет форматом *значений* в сообщениях, записываемых в Kafka или извлекаемых из него, а поскольку класс не зависит от коннекторов, это позволяет любому коннектору работать с любым форматом сериализации. Примерами распространенных форматов могут служить JSON и Avro». (Документация Connect.)

❺ Kafka Connect использует несколько дополнительных тем для хранения информации с настройками коннекторов и задач. Здесь мы просто используем стандартные имена этих тем с префиксом `ksql-`, потому что будем работать

в режиме встроенной интеграции (то есть рабочие процессы будут выполняться под управлением той же JVM, что и экземпляры серверов ksqlDB).

❻ Свойство `errors.tolerance` позволяет настроить политику обработки ошибок по умолчанию в Kafka Connect. Допустимые значения: `none` (немедленный отказ при возникновении ошибки) и `all` (полное игнорирование ошибок или, при использовании со свойством `errors.deadletterqueue.topic.name`, пересылка всех ошибок в тему Kafka по вашему выбору).

❼ Список путей в файловой системе, перечисленных через запятую, где находятся плагины (коннекторов, конвертеров, преобразователей). Как устанавливать коннекторы, вы увидите далее в этой главе.

Как видите, основная масса конфигурационных параметров рабочих процессов довольно проста. Тем не менее некоторые настройки стоит изучить подробнее, потому что они связаны с решением важной задачи сериализации данных — это свойства конвертеров (`key.converter` и `value.converter`). В следующем разделе мы детально рассмотрим конвертеры и форматы сериализации.

Конвертеры и форматы сериализации

Классы конвертеров, используемых в Kafka Connect, играют важную роль в сериализации и десериализации данных. В нашем учебном проекте Hello, world, представленном в предыдущей главе (см. раздел «Учебный проект» главы 8), мы использовали инструкцию из примера 9.1, чтобы создать поток в ksqlDB.

Пример 9.1. Создание потока, читающего данные из темы users

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

Эта инструкция сообщает ksqlDB, что тема `users` (`KAFKA_TOPIC='users'`) содержит записи со значениями, сериализованными в формат JSON (`VALUE_FORMAT='JSON'`). Если есть свой производитель, записывающий в тему данные в формате JSON, то довольно легко рассуждать о формате. Но что, если Kafka Connect используется, например, для потоковой передачи в Kafka данных из PostgreSQL? В какой формат сериализуются данные из PostgreSQL, когда они записываются в Kafka?

Здесь в игру вступают настройки конвертеров. Для управления форматами сериализации ключей и значений записей, которые обрабатывает Kafka Connect,

можно настроить свойства `key.converter` и `value.converter`, определив в них соответствующие классы конвертеров. В табл. 9.1 перечислены наиболее часто используемые классы конвертеров и соответствующие им форматы сериализации ksqlDB (то есть значение, которое указывается в свойстве `VALUE_FORMAT` при создании потока или таблицы, как было показано в примере 9.1).

В табл. 9.1 также отмечено, какие конвертеры опираются на Confluent Schema Registry для хранения схем записей, что может пригодиться, если потребуется более компактный формат сообщений. Schema Registry позволяет хранить схемы записей, то есть имена и типы полей, вне самих сообщений.

Таблица 9.1. Наиболее часто используемые классы конвертеров Kafka Connect и соответствующие им форматы сериализации в ksqlDB

Тип	Класс конвертера	Требуется Schema Registry?	Тип сериализации в ksqlDB
Avro	<code>io.confluent.connect.avro.AvroConverter</code>	Да	AVRO
Protobuf	<code>io.confluent.connect.protobuf.ProtobufConverter</code>	Да	PROTOBUF
JSON (с Schema Registry)	<code>io.confluent.connect.json.JsonSchemaConverter</code>	Да	JSON_SR
JSON	<code>org.apache.kafka.connect.json.JsonConverter¹</code>	Нет	JSON
String	<code>org.apache.kafka.connect.storage.StringConverter</code>	Нет	KAFKA ²
DoubleConverter	<code>org.apache.kafka.connect.converters.DoubleConverter</code>		KAFKA
IntegerConverter	<code>org.apache.kafka.connect.converters.IntegerConverter</code>	Нет	KAFKA
LongConverter	<code>org.apache.kafka.connect.converters.LongConverter</code>	Нет	KAFKA

¹ При использовании `JsonConverter`, входящего в состав Kafka Connect, в конфигурации рабочих процессов Connect можно также настроить параметр `value.converter.schemas.enable`. Если присвоить ему значение `true`, то Connect будет встраивать собственную схему в запись JSON. При этом реестр схем не будет использоваться, но из-за включения схемы в каждую запись размеры сообщений могут получиться довольно большими. Если этому параметру присвоить значение `false`, то ksqlDB будет определять тип поля, используя подсказки, указанные вами при создании потока или таблицы. Мы рассмотрим этот подход позже.

² Формат KAFKA сообщает, что ключ записи или значение были сериализованы с использованием одного из встроенных экземпляров `Serde` в Kafka (см. табл. 3.1).

Для каждого конвертера в табл. 9.1, требующего реестра схем, нужно добавить дополнительное конфигурационное свойство: { key | value }.converter.schema.registry.url. Например, в этой книге мы будем работать в основном с данными Avro, поэтому, чтобы коннекторы записывали значения в этом формате, можно обновить конфигурацию рабочего процесса, как показано в примере 9.2.

Пример 9.2. Конфигурация рабочего процесса, использующего AvroConverter для преобразования значений записей

```
bootstrap.servers=localhost:9092
group.id=ksql-connect-cluster
```

```
key.converter=org.apache.kafka.connect.storage.StringConverter
value.converter=io.confluent.connect.avro.AvroConverter ❶
value.converter.schema.registry.url=http://localhost:8081 ❷
```

```
config.storage.topic=ksql-connect-configs
offset.storage.topic=ksql-connect-offsets
```

```
status.storage.topic=ksql-connect-statuses
```

```
plugin.path=/opt/confluent/share/java/
```

❶ Использовать AvroConverter для сериализации значений в формат Avro.

❷ Конвертеру Avro требуется Confluent Schema Registry для хранения схем записей, поэтому нужно указать URL этого реестра схем, определив свойство value.converter.schema.registry.url¹.

Сейчас, узнав, как задать формат сериализации данных в Kafka Connect, и подготовив конфигурацию для рабочих процессов в Kafka Connect (см. пример 9.2), перейдем к учебному проекту и на практике установим и используем некоторые коннекторы.

Учебный проект

В этом учебном проекте мы используем коннектор-источник JDBC для потоковой передачи данных из PostgreSQL в Kafka. Затем создадим коннектор-приемник Elasticsearch для записи данных из Kafka в Elasticsearch. Полный код этого

¹ Можно также использовать конвертеры, применяющие Schema Registry для ключей записей. В этом случае нужно в конфигурации рабочих процессов установить свойство key.converter.schema.registry.url.

проекта и инструкции по настройке окружения (включая экземпляр PostgreSQL и Elasticsearch) можно найти в репозитории на GitHub (<https://oreil.ly/7ImWJ>).

Начнем с установки коннекторов.

Установка коннекторов

Существует два основных способа установки коннекторов источников и приемников:

- вручную;
- автоматически, через Confluent Hub.

Ручная установка может отличаться для разных реализаций коннекторов и зависит от того, как разработчики коннектора решат распространять артефакт (артефакт коннектора обычно включает один или несколько файлов JAR). Однако обычно процедура установки предполагает загрузку артефакта непосредственно с веб-сайта или из репозитория артефактов, такого как Maven Central или Artifactory. После загрузки файлы JAR помещаются в место, указанное в конфигурационном свойстве `plugin.path`.

Более простой метод загрузки коннекторов, который будет использоваться в этой книге, позволяет устанавливать коннекторы с помощью инструмента командной строки, разработанного в Confluent. Этот инструмент с названием `confluent-hub` можно установить, следуя инструкциям в документации Confluent (<https://oreil.ly/31Sd9>). После установки Confluent Hub установка самих коннекторов не вызывает никаких сложностей. Вот синтаксис команды установки коннектора:

```
confluent-hub install <владелец>/<компонент>:<версия> [параметры]
```

Например, следующая команда установит коннектор-приемник Elasticsearch:

```
confluent-hub install confluentinc/kafka-connect-elasticsearch:10.0.2 \
  --component-dir /home/appuser \ ❶
  --worker-configs /etc/ksqldb-server/connect.properties \ ❷
  --no-prompt ❸
```

❶ Каталог, куда должен быть установлен коннектор.

❷ Местоположение конфигурационных файлов рабочих процессов. Место установки (определяется параметром `--component-dir`) будет добавлено в `plugin.path`, если это еще не было сделано.

❸ Чтобы обойти стороной интерактивные шаги (например, подтверждение установки, принятие лицензионного соглашения и т. д.), можно разрешить интерфейсу командной строки работать с рекомендуемыми значениями/значениями по умолчанию. Это полезно для установки из сценария.

Точно так же можно установить коннектор-источник PostgreSQL:

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

Обратите внимание, что в режиме встроенной интеграции потребуется перезапустить сервер ksqlDB, если коннекторы устанавливались после запуска экземпляра сервера ksqlDB. Выполнив установку коннекторов, необходимых приложению, можно создавать их экземпляры и управлять ими в ksqlDB. Мы обсудим этот вопрос в следующем разделе.

Создание экземпляров коннекторов в ksqlDB

Вот как выглядит синтаксис создания коннектора:

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

Предположим, что у нас уже есть экземпляр PostgreSQL, доступный по адресу `postgres:5432`, в этом случае можно установить коннектор-источник для чтения из таблицы `titles`, выполнив следующую команду в ksqlDB:

```
CREATE SOURCE CONNECTOR `postgres-source` WITH( ❶
  "connector.class"='io.confluent.connect.jdbc.JdbcSourceConnector', ❷
  "connection.url"=
    'jdbc:postgresql://postgres:5432/root?user=root&password=secret', ❸
  "mode"='incrementing', ❹
  "incrementing.column.name"='id', ❺
  "topic.prefix"='', ❻
  "table.whitelist"='titles', ❼
  "key"='id'); ❽
```

❶ Оператор `WITH` используется для передачи конфигурации коннектора (зависит от конкретного коннектора, поэтому необходимо заглянуть в документацию, чтобы узнать список доступных конфигурационных свойств).

❷ Класс Java коннектора.

- ❸ Коннектору-источнику JDBC требуется URL для подключения к хранилищу данных (в данном случае к базе данных PostgreSQL).
- ❹ Коннектор-источник ОВИС поддерживает несколько режимов запуска. Поскольку мы предполагаем передавать любые новые записи, добавляемые в таблицу `titles` и имеющие столбец с автоматическим приращением значения, можно установить режим `incrementing`. Этот и другие режимы, поддерживаемые данным коннектором, подробно описаны в документации (<https://oreil.ly/w8Grb>).
- ❺ Имя столбца с автоматическим приращением, который коннектор-источник будет использовать для определения новых строк.
- ❻ Каждая таблица передается в отдельную тему (например, таблица `titles` будет передаваться в тему `titles`). При желании можно задать префикс для имени темы (например, если настроить префикс `ksql-`, данные будут передаваться в тему `ksql-titles`). В этом проекте мы не будем использовать префикс.
- ❼ Список таблиц для потоковой передачи в Kafka.
- ❽ Значение, используемое в роли ключа записи.

После выполнения инструкции `CREATE SOURCE CONNECTOR` в консоли должно появиться сообщение, подобное следующему:

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

Теперь создадим коннектор-приемник для вывода записей из приложения в Elasticsearch. Эта инструкция очень похожа на инструкцию создания коннектора-источника:

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

Как видите, конфигурации разных коннекторов различаются. Большинство имен конфигурационных параметров говорят сами за себя, а определение назначения остальных я оставляю вам в качестве самостоятельного упражнения. Соответствующие описания конфигурационных параметров `ElasticsearchSinkConnector` можно найти в справочнике по настройке Elasticsearch Sink Connector (<https://oreil.ly/o8h7j>). И снова после выполнения инструкции `CREATE SINK CONNECTOR` в консоли должно появиться сообщение:

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

После создания экземпляров коннекторов в ksqlDB с ними можно взаимодействовать разными способами. В следующих разделах мы рассмотрим некоторые из доступных вариантов взаимодействия.

Вывод списка коннекторов

В режиме интерактивной интеграции иногда полезно получить список всех работающих коннекторов и их состояние. Инструкция получения списка коннекторов имеет следующий синтаксис:

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

Другими словами, можно получить список всех коннекторов, только коннекторов-источников или только коннекторов-приемников. К настоящему моменту мы создали только два коннектора, источник и приемник, поэтому воспользуемся следующим вариантом, чтобы вывести информацию об обоих:

```
SHOW CONNECTORS;
```

В консоли должен появиться такой вывод:

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

Команда `SHOW CONNECTORS` выводит некоторую полезную информацию об активных коннекторах, включая их состояние. В данном случае оба коннектора имеют по одной задаче в состоянии `RUNNING`. Другие состояния, которые можно увидеть, включают: `UNASSIGNED`, `PAUSED`, `FAILED` и `DESTROYED`. Увидев такое состояние, как

FAILED, вы наверняка захотите выяснить причину. Например, если коннектор `postgres-source` потеряет соединение с базой данных PostgreSQL (это можно симитировать, просто остановив экземпляр PostgreSQL), то появится такой вывод:

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

Но как получить дополнительную информацию о коннекторе, например, чтобы выяснить причину неудачной отработки задач? В этом вам поможет возможность получения описаний коннекторов в ksqlDB. Рассмотрим ее ниже.

Получение описаний коннекторов

ksqlDB упрощает получение состояния коннекторов, предлагая инструкцию `DESCRIBE CONNECTOR`. Например, если коннектор `postgres-source` потеряет соединение с хранилищем данных, как обсуждалось в предыдущем разделе, можно попробовать запросить его описание, чтобы получить дополнительную информацию. Например:

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

Если имеет место ошибка, то в консоли появится вывод с трассировкой этой ошибки, как показано ниже:

```
Name           : postgres-source
Class          : io.confluent.connect.jdbc.JdbcSourceConnector
Type           : source
State          : FAILED
WorkerId       : 192.168.65.3:8083
Trace          : org.apache.kafka.connect.errors.ConnectException ❶
```

Task ID	State	Error Trace
0	FAILED	org.apache.kafka.connect.errors.ConnectException ❷

❶ Трассировка стека в этом примере приводится неполностью, но в случае фактического сбоя вы должны увидеть полную трассировку стека исключения.

❷ Разбивка по задачам. Задачи могут находиться в разных состояниях (например, одни могут находиться в состоянии `RUNNING`, а другие — в состоянии `UNASSIGNED`, `FAILED` и т. д.).

Однако чаще вы будете видеть задачи в работоспособном состоянии. Вот пример вывода инструкции `DESCRIBE CONNECTOR`:

```
Name           : postgres-source
Class          : io.confluent.connect.jdbc.JdbcSourceConnector
Type          : source
State         : RUNNING
WorkerId      : 192.168.65.3:8083
```

```
Task ID | State | Error Trace
-----
0       | RUNNING |
```

Теперь, научившись создавать коннекторы и получать их описания, давайте узнаем, как их удалять.

Удаление коннекторов

Удаление коннекторов может понадобиться для их перенастройки или безвозвратного удаления. Синтаксис удаления коннектора:

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

Например, чтобы удалить коннектор PostgreSQL, можно выполнить следующую инструкцию:

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

После удаления коннектора в консоли должно появиться подтверждение, что коннектор действительно удален. Например:

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

Проверка коннектора-источника

Один из быстрых способов проверить работоспособность коннектора-источника PostgreSQL — записать некоторые данные в базу данных, а затем вывести содержимое темы. Например, создадим таблицу `titles` в экземпляре PostgreSQL и заполним ее некоторыми данными:

```
CREATE TABLE titles (
  id          SERIAL PRIMARY KEY,
  title       VARCHAR(120)
);
```

```
INSERT INTO titles (title) values ('Stranger Things');
INSERT INTO titles (title) values ('Black Mirror');
INSERT INTO titles (title) values ('The Office');
```



Это инструкция PostgreSQL, а не ksqlDB.

Наш коннектор-источник PostgreSQL должен автоматически извлечь данные из этой таблицы в тему `titles`. Чтобы убедиться в этом, воспользуемся инструкцией `PRINT`:

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

ksqlDB должен вывести:

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

Обратите внимание, что ksqlDB, как сообщается в первых двух строках вывода, пытается определить формат ключей и значений записей в теме `titles`. Поскольку для ключей у нас используется `StringConverter`, а для значений — `AvroConverter` (см. пример 9.2), этот результат вполне ожидаем.

Точно так же для проверки коннектора-приемника нужно создать принимающую тему, а затем запросить данные из нижестоящего хранилища. Мы оставим это читателю в качестве самостоятельного упражнения (можете заглянуть в репозиторий [<https://oreil.ly/gsl8X>], и вы увидите, как это сделать).

Пришло время посмотреть, как напрямую взаимодействовать с кластером Kafka Connect, и перечислить случаи, когда это может понадобиться.

Взаимодействие с кластером Kafka Connect напрямую

Иногда может потребоваться взаимодействовать с кластером Kafka Connect напрямую, без участия ksqlDB. Например, некоторые конечные точки Kafka Connect предоставляют информацию, недоступную в ksqlDB, и позволяют выполнять важные действия, такие как повторный запуск задач, потерпевших неудачу. Я не собираюсь давать здесь исчерпывающие инструкции по работе с Connect API, а просто приведу несколько примеров запросов, которые вы

можете выполнить в своем кластере Connect. Они перечислены в следующей таблице.

Случай использования	Пример запроса
Получение списка коннекторов	<code>curl -XGET localhost:8083/connectors</code>
Получение описаний коннекторов	<code>curl -XGET localhost:8083/connectors/ elasticsearch-sink</code>
Получение списка задач	<code>curl -XGET -s localhost:8083/connectors/ elasticsearch-sink/tasks</code>
Получение состояний задач	<code>curl -XGET -s localhost:8083/connectors/ elasticsearch-sink/tasks/0/status</code>
Перезапуск задачи	<code>curl -XPOST -s localhost:8083/connectors/ elasticsearch-sink/tasks/0/restart</code>

Наконец, посмотрим, как проверить схемы при использовании форматов сериализации, применяющих Confluent Schema Registry.

Анализ управляемых схем

Некоторые форматы сериализации из перечисленных в табл. 9.1 требуют Confluent Schema Registry для хранения схем записей. При их использовании Kafka Connect будет автоматически сохранять схемы в реестре Confluent Schema Registry. В табл. 9.2 показаны примеры запросов к конечной точке Schema Registry, которые помогут проанализировать управляемые схемы.

Таблица 9.2. Пример запросов к конечной точке Schema Registry

Случай использования	Пример запроса
Получение схемы типов	<code>curl -XGET localhost:8081/subjects/</code>
Получение списка версий схемы	<code>curl -XGET localhost:8081/subjects/titles-value/ versions</code>
Получение схемы с указанной версией	<code>curl -XGET localhost:8081/subjects/titles-value/ versions/1</code>
Получение схемы последней версии	<code>curl -XGET localhost:8081/subjects/titles-value/ versions/latest</code>

Полную справку по API можно найти в справочнике по Schema Registry API (<https://oreil.ly/Q26Si>).

Заключение

Теперь вы знаете, как интегрировать внешние системы с ksqlDB. Получив базовые знания о Kafka Connect и о различных операторах ksqlDB для управления коннекторами, вы готовы начать учиться обрабатывать, преобразовывать и обогащать данные с помощью ksqlDB. В следующей главе мы рассмотрим пример использования потоковой обработки в Netflix, вы познакомитесь с некоторыми дополнительными операторами ksqlDB для создания потоковых приложений с использованием SQL.

Основы потоковой обработки с ksqlDB

В этой главе вы узнаете, как использовать ksqlDB для решения распространенных задач потоковой обработки. Вот примерный список затрагиваемых тем:

- создание потоков и таблиц;
- использование типов данных ksqlDB;
- фильтрация с применением простых логических условий, подстановочных знаков и фильтров диапазона;
- изменение формы данных разворачиванием сложных или вложенных структур;
- выборка подмножества доступных полей с применением проекций;
- использование условных выражений для обработки значений NULL;
- создание производных потоков и таблиц и запись результатов обратно в Kafka.

К концу этой главы вы будете готовы к решению основных задач предварительной обработки и преобразования данных с использованием диалекта ksqlDB SQL. Конструкции SQL, представленные в предыдущей главе, как вы помните, использовали интеграцию ksqlDB с Kafka Connect. Аналогично все конструкции, которые будут представлены в этой главе, используют интеграцию ksqlDB с Kafka Streams. Исследуя их, вы убедитесь, насколько мощной является ksqlDB, позволяя всего несколькими строками SQL создавать полноценные приложения потоковой обработки. Как обычно, изучение тем этой главы будет происходить на примере создания приложения, использующего соответствующие языковые конструкции. Итак, перейдем сразу к делу.

Учебный проект: мониторинг изменений в Netflix

Компания Netflix ежегодно инвестирует миллиарды долларов в видеоконтент. Для ее бесперебойной работы совершенно необходима передача обновлений в различные системы (например, изменение даты выпуска, финансовых поступлений, управление талантами и т. д.). И все это при одновременном производстве большого количества фильмов и телесериалов. Исторически компания Netflix использовала Apache Flink для своих сценариев потоковой обработки. Но мы в сотрудничестве с Нитином Шармой, инженером из Netflix, решили реализовать эту задачу, используя ksqlDB¹.

Цель рассматриваемого приложения проста: получить поток изменений, отфильтровать и преобразовать данные для обработки, обогатить и агрегировать данные для отчетов и, наконец, сделать обработанные данные доступными для нижестоящих систем. На первый взгляд кажется, что для этого придется изрядно потрудиться, но с ksqlDB реализация получится очень простой.

Тип изменений, на котором мы сосредоточимся в этом проекте, — изменение продолжительности сезона сериала. Например, в четвертом сезоне сериала «Очень странные дела» изначально планировалось выпустить 12 серий, но планы можно изменить и сократить сезон до восьми серий, что произведет волновой эффект в различных системах, включая управление талантами, расчет финансовых поступлений и т. д. Этот пример выбран потому, что он не только моделирует реальную задачу, но и затрагивает наиболее распространенные смежные и наведенные задачи, которые вам придется решать в своих приложениях ksqlDB.

На рис. 10.1 показана архитектурная схема нашего будущего приложения мониторинга изменений, а ниже описывается каждый ее шаг.

¹ Имейте в виду, что это не промышленное приложение для Netflix, а лишь демонстрация, реализующая один из вариантов использования потоковой обработки для Netflix с помощью ksqlDB.

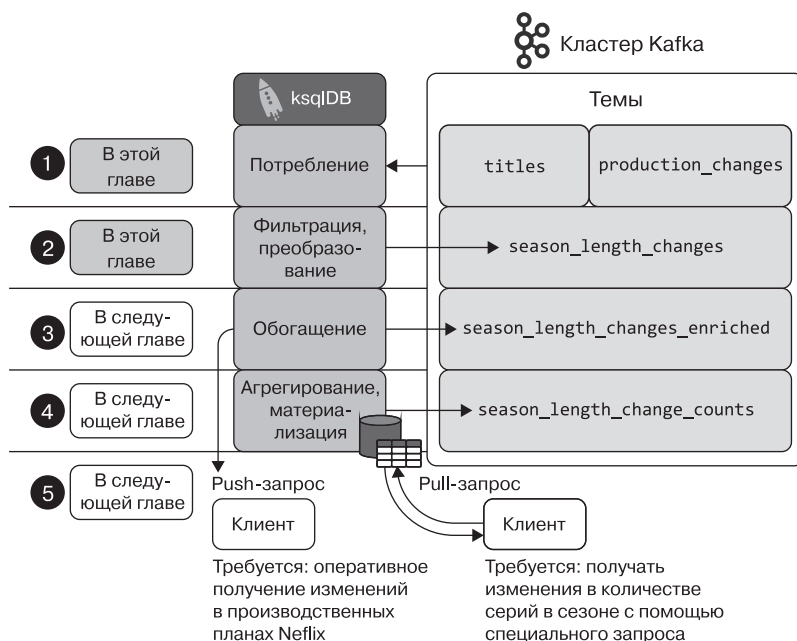


Рис. 10.1. Архитектурная схема приложения мониторинга изменений для Netflix

❶ Наше приложение будет читать данные из двух тем.

- Тема `titles` — это сжатая тема с метаданными (название, дата выпуска и т. д.) для фильмов и телесериалов (именуемых в этом проекте *произведениями*), размещенных в службе Netflix.
- Данные в тему `production_changes` записываются всякий раз, когда происходят изменения в расписании управления талантами, в бюджете, дате выпуска или в продолжительности сезона сериала, находящегося в производстве.

❷ К данным, полученным из исходных тем, необходимо применить некоторую простую обработку (например, фильтрацию и преобразование), чтобы подготовить данные из `production_changes` для обогащения. Предварительно обработанный поток, содержащий после фильтрации только изменения количества серий в сезоне, будет записан в тему Kafka с именем `season_length_changes`.

❸ Затем предварительно обработанные данные подвергаются некоторому обогащению. В частности, мы соединим поток `season_length_changes` с данными из `titles`, чтобы получить комбинированную запись из нескольких источников.

❹ Далее выполняется несколько оконных и неоконных операций агрегирования, чтобы подсчитать количество изменений за пятиминутный период. Полученная в результате таблица материализуется и становится доступной для запросов на получение.

❺ Наконец, мы сделаем обогащенные и агрегированные данные доступными для клиентов двух разных типов. Первый будет получать обновления непрерывно через push-запросы, используя долгоживущее соединение с ksqlDB, а второй будет выполнять точечный поиск с помощью короткоживущих pull-запросов, больше похожих на традиционные запросы `SELECT` в базе данных.

Учитывая огромное количество понятий, которые будут представлены в этом проекте, мы разделим его разработку на две главы. В этой главе основное внимание будет уделено созданию потоков и таблиц, а также основным этапам предварительной обработки и преобразования данных (то есть шагам 1 и 2). В следующей главе мы займемся обогащением данных, агрегированием и запросами push/pull (впрочем, мы уделим некоторое внимание push-запросам и в этой главе, потому что они необходимы для шага 2).

Настройка проекта

Код для примеров этой главы можно найти по адресу <https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>.

Если вы решите отслеживать процесс разработки проекта, то клонируйте репозиторий и перейдите в каталог, содержащий проект этой главы. Для этого выполните следующие команды:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-10/
```

В этом проекте мы используем Docker Compose для запуска каждого компонента, необходимого приложению, в том числе:

- Kafka;
- Schema Registry;
- сервер ksqlDB;
- клиент командной строки ksqlDB CLI.

Просто выполните следующую команду после клонирования репозитория, чтобы запустить каждый компонент:

```
docker-compose up
```

Операторы SQL, обсуждаемые в этой главе, будут выполняться в интерфейсе командной строки ksqlDB. Запустить его можно следующей командой:

```
docker-compose exec ksqldb-cli \ ❶  
    ksql http://ksqldb-server:8088 ❷
```

❶ Применение `docker-compose exec` для запуска команды в контейнере `ksqldb-cli`. Запускаемая команда показана в следующей строке.

❷ `ksql` — это имя выполняемого файла клиента командной строки. Аргумент `http://ksqldb-server:8088` — это URL сервера ksqlDB.

Теперь, после запуска клиента командной строки, начнем работу над проектом.

Исходные темы

Итак, у нас есть данные в теме Kafka, и их нужно обработать с помощью ksqlDB. Отлично! С чего начнем?

Логичнее начать с обзора данных в исходных темах, а затем определить, как моделировать данные в ksqlDB. В этом проекте две основные исходные темы: `titles` и `production_changes`. В табл. 10.1 показаны примеры записей в каждой из этих тем.

Таблица 10.1. Примеры записей в исходных темах

Тема	Пример записи
titles	<pre>{ "id": 1, "title": "Stranger Things", "on_schedule": false }</pre>
production_change	<pre>{ "uuid": 1, "title_id": 1, "change_type": "season_length", "before": { "season_id": 1, "episode_count": 12 }, "after": { "season_id": 1, "episode_count": 8 }, "created_at": "2021-02-08 11:30:00" }</pre>

В предыдущей главе мы обсудили форматы сериализации *на уровне записей*, поддерживаемые в ksqlDB. В их число входят такие популярные форматы, как AVRO, JSON, PROTOBUF, DELIMITED и примитивы (например, типы String, Double, Integer и Long, которые все вместе входят в формат KAFKA). Данные в нашей теме `titles` хранятся в формате JSON, соответственно, мы можем использовать JSON как формат сериализации записей.

Еще мы не рассмотрели *типы данных* на уровне поля. Например, запись `titles` в табл. 10.1 содержит три поля (`id`, `title`, `on_schedule`), хранящие значения разных типов (целочисленные, строковые и логические значения соответственно). Прежде чем создавать какие-либо потоки или таблицы для моделирования данных, необходимо определить тип данных каждого поля. ksqlDB имеет встроенную поддержку множества типов данных, использование которой демонстрируется в следующем разделе.

Типы данных

В табл. 10.2 перечислены типы данных, поддерживаемые в ksqlDB.

Таблица 10.2. Встроенные типы данных

Тип	Описание
ARRAY<тип-элементов>	Коллекция элементов одного типа (например, ARRAY<STRING>)
BOOLEAN	Логическое значение
INT	32-битное целое со знаком
BIGINT	64-битное целое со знаком
DOUBLE	Число с плавающей точкой двойной точности (64-битное) в формате IEEE 754
DECIMAL(precision, scale)	Число с плавающей точкой и с настраиваемыми общим числом разрядов (precision) и числом разрядов после десятичной запятой (scale)
MAP<тип-ключа, тип-элемента>	Объект, содержащий ключи и значения, соответствующие определению объекта (например, MAP<STRING, INT>)
STRUCT<field-name field-type [, ...]>	Структурированная коллекция полей (например, STRUCT<FOO INT, BAR BOOLEAN>)
VARCHAR или STRING	Последовательность символов Юникода (UTF8)

Одна из интересных особенностей типов данных ksqlDB та, что для некоторых форматов сериализации они являются необязательными. К таким форматам относятся все, основанные на Confluent Schema Registry, включая AVRO, PROTOBUF и JSON_SR. Это объясняется тем, что Schema Registry уже хранит имена и типы полей, поэтому нет смысла повторно определять типы данных в операторе CREATE (ksqlDB может получить информацию о схеме из Schema Registry).

Исключением является *ключевой столбец*. В ksqlDB можно использовать идентификатор PRIMARY KEY (для таблиц) или KEY (для потоков), чтобы сообщить, какой столбец будет играть роль ключа сообщения.

Поэтому мы должны указать *частичную схему*, содержащую столбец PRIMARY KEY, чтобы сообщить ksqlDB, что этот столбец служит ключом сообщения. Однако типы столбцов значений (например, title) могут быть получены из схемы. Например, если бы данные в теме titles хранились в формате AVRO и в Schema Registry имела соответствующая схема Avro, то мы могли бы использовать любой из следующих операторов CREATE для создания таблицы titles.

С явным определением типов	С автоматическим определением типов ¹
<pre>CREATE TABLE titles (id INT PRIMARY KEY, title VARCHAR) WITH (KAFKA_TOPIC='titles', VALUE_FORMAT='AVRO', PARTITIONS=4);</pre>	<pre>CREATE TABLE titles (id INT PRIMARY KEY) WITH (KAFKA_TOPIC='titles', VALUE_FORMAT='AVRO', PARTITIONS=4);</pre>

В этой книге используется версия с явным определением типов, чтобы сделать примеры более ясными. Далее мы рассмотрим синтаксис CREATE {STREAM | TABLE} подробнее, а пока продолжим обсуждение типов данных и вы познакомитесь с возможностью определения пользовательских типов.

Пользовательские типы

Пользовательские типы аналогичны составным типам в PostgreSQL, они позволяют определить набор имен полей и их типов данных, а затем ссылаться на этот набор, используя выбранное имя. Пользовательские типы особенно полезны для повторного применения определений сложных типов. Например, нашему приложению необходимо фиксировать изменения протяженности сезона, которые могут иметь структурно идентичные состояния «до» и «после»:

¹ Такой способ возможен только для форматов, использующих Schema Registry.

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

❶ Поля `before` (до) и `after` (после) имеют идентичную структуру. Это хороший пример применения пользовательского типа.

Для определения полей `before` и `after` можно использовать отдельные объявления `STRUCT` или создать пользовательский тип, который можно применять многократно. В этом проекте мы выберем последний вариант, потому что он улучшит читаемость инструкций SQL. В следующей таблице перечислены различные операции для работы с пользовательскими типами и соответствующий им синтаксис SQL.

Операция	Синтаксис
Создание пользовательского типа	<code>CREATE TYPE <имя_типа> AS <определение_типа>;</code>
Вывод всех зарегистрированных пользовательских типов	<code>{ LIST SHOW } TYPES</code>
Удаление пользовательского типа	<code>DROP TYPE <имя_типа></code>

Создадим свой тип с именем `season_length`:

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

После создания типа можно выполнить запрос `SHOW TYPES`, чтобы убедиться, что он зарегистрирован:

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

Удалить зарегистрированный тип можно, выполнив следующую инструкцию:

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

В этом проекте используется тип `season_length`, поэтому, если вы удалили его, последовав за примером выше, то обязательно воссоздайте его снова. Теперь, познакоившись с различными типами данных в ksqlDB, приступим к созданию необходимых потоков и таблиц.

Коллекции

Потоки и таблицы — это две основные абстракции, лежащие в основе Kafka Streams и ksqlDB. В ksqlDB они называются *коллекциями*. Впервые мы познакомились с потоками и таблицами в разделе «Потоки данных и таблицы» главы 2, но мы еще раз кратко рассмотрим различия между ними, а затем перейдем к синтаксису их создания в ksqlDB.

Таблицы можно рассматривать как моментальные снимки постоянно обновляемого набора данных, когда последнее состояние или результаты вычислений (в случае агрегирования) каждого уникального ключа в теме Kafka сохраняются в базовой коллекции¹. Они поддерживаются сжатыми темами и используют хранилища состояний Kafka Streams.

Таблицы часто используются для обогащения данных путем создания соединений, когда можно ссылаться на так называемую таблицу поиска для получения дополнительного контекста о событиях, поступающих через поток. Они также играют особую роль в агрегатах, о которых мы поговорим в следующей главе.

Потоки, напротив, моделируются как неизменяемая последовательность событий. В отличие от таблиц, имеющих изменяемые характеристики, каждое событие в потоке считается независимым от всех других. Потоки не имеют состояния, то есть каждое событие потребляется, обрабатывается и впоследствии забывается.

Чтобы нагляднее показать разницу, рассмотрим такую последовательность событий (ключи и значения отображаются как <ключ, значение>):

Последовательность событий

```
<K1, V1>
<K1, V2>
<K1, V3>
<K2, V1>
```

¹ Коллекции поддерживаются хранилищами состояний RocksDB, как обсуждалось в разделе «Хранилища состояний» главы 4.

Поток моделирует полную историю событий, а таблица фиксирует последнее состояние каждого уникального ключа. Ниже показаны потоковое и табличное представления предыдущей последовательности:

Поток	Таблица
<K1, V1>	<K1, V3>
<K1, V2>	<K2, V1>
<K1, V3>	
<K2, V1>	

Есть несколько способов создания потоков и таблиц в ksqlDB: либо непосредственно из тем Kafka (в этой книге мы называем их *исходными коллекциями*), либо из других потоков и таблиц (мы называем их *производными коллекциями*). В этой книге мы рассмотрим оба способа и начнем с исходных коллекций, потому что они являются отправной точкой для всех приложений потоковой обработки в ksqlDB.

Создание исходных коллекций

Прежде чем начать работать с данными в ksqlDB, нужно создать исходные коллекции на базе тем Kafka. Эта необходимость обусловлена тем, что в ksqlDB запрашиваются не темы Kafka напрямую, а коллекции (то есть потоки и таблицы). Вот как выглядит синтаксис создания потоков и таблиц:

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

В этом проекте нам нужно создать поток и таблицу. Тему **titles** с метаданными о фильмах и телесериалах (далее мы будем называть обе эти сущности *произведениями*) мы смоделируем в виде таблицы, потому что нас интересуют только текущие метаданные, связанные с каждым произведением. Вот как можно создать такую таблицу:

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

❶ PRIMARY KEY определяет ключевой столбец таблицы, он порождается из ключа записи. Не забывайте, что таблицы имеют изменяемую семантику обновления, поэтому если несколько записей получают один и тот же первичный ключ, то в таблице сохранится только самая последняя. Исключением является случай, когда ключ записи имеет значение NULL. В этом случае запись считается *надгробием* (маркером удаления) и служит признаком необходимости удаления соответствующего ключа. Обратите внимание, что при создании таблиц ksqlDB игнорирует все записи с ключами, имеющими значение NULL (это не относится к потокам).

❷ Так как в операторе WITH указано свойство PARTITIONS, ksqlDB создаст базовую тему, если она еще не существует (в этом случае тема будет создана с четырьмя разделами). Здесь также можно определить коэффициент репликации темы, задав свойство REPLICAS. Более подробно мы рассмотрим оператор WITH в следующем разделе.

Теперь смоделируем тему production_changes как поток. Это идеальный тип коллекции для подобных тем, потому что не требует отслеживать последнее состояние каждого события изменения: значения просто потребляются и обрабатываются. Вот как можно создать исходный поток:

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

❶ Потоки, в отличие от таблиц, не имеют столбца первичного ключа и поддерживают неизменяемую семантику в стиле вставки, поэтому однозначная идентификация записей невозможна. Однако идентификатор KEY можно использовать для определения ключа записи, соответствующего ключу записи в Kafka.

❷ Здесь в определениях полей before и after, представляющих сложные объекты¹ с идентичной структурой, используется наш пользовательский тип season_length.

¹ Здесь под словом «сложные» подразумевается «непримитивные». В данном случае поля before и after представлены структурой STRUCT, которую мы определили как собственный тип. MAP и ARRAY тоже являются сложными типами.

❸ Это свойство сообщает, что столбец `created_at` содержит отметку времени, которая должна использоваться в операциях, основанных на времени, включая оконное агрегирование и соединение (рассматриваются в главе 11). Свойство `TIMESTAMP_FORMAT` в последующей строке определяет формат отметок времени. Дополнительные сведения о свойствах `TIMESTAMP` и `TIMESTAMP_FORMAT` приводятся в следующем разделе.

Оператор `WITH` в инструкции `CREATE { STREAM | TABLE }` поддерживает несколько свойств, которые мы еще не исследовали. Давайте рассмотрим их, прежде чем продолжить обсуждение потоков и таблиц.

Оператор WITH

При создании потока или таблицы можно настроить ряд свойств, используя оператор `WITH`. Некоторые из наиболее важных свойств, которые вы наверняка будете использовать, перечислены в табл. 10.3.

Таблица 10.3. Свойства, поддерживаемые оператором `WITH`

Имя свойства	Описание	Обязательное?
<code>KAFFKA_TOPIC</code>	Тема Kafka с данными для потребления	Да
<code>VALUE_FORMAT</code>	Формат сериализации данных в исходной теме (например, <code>AVRO</code> , <code>PROTOBUF</code> , <code>JSON</code> , <code>JSON_SR</code> , <code>KAFFKA</code>)	Да
<code>PARTITIONS</code>	Это свойство определяет количество разделов для исходной темы, если ее создает ksqlDB	Нет
<code>REPLICAS</code>	Это свойство определяет количество реплик для исходной темы, если ее создает ksqlDB	Нет
<code>TIMESTAMP</code>	Имя столбца с отметкой времени, который ksqlDB должна использовать в операциях, основанных на времени, включая оконные операции. Если не установить это свойство, ksqlDB будет использовать отметку времени из метаданных записи. Если столбец имеет тип <code>BIGINT</code> , то ksqlDB сможет использовать его значение напрямую и никаких дополнительных настроек не нужно. Если столбец имеет тип <code>VARCHAR</code> , то необходимо определить свойство <code>TIMESTAMP_FORMAT</code> , которое описано ниже	Нет
<code>TIMESTAMP_FORMAT</code>	Формат отметки времени. Допускается использовать любые форматы, поддерживаемые <code>java.time.format.DateTimeFormatter</code> (например, <code>yyyy-MM-dd HH:mm:ss</code>)	Нет
<code>VALUE_DELIMITER</code>	Символ, играющий роль разделителя полей, когда <code>VALUE_FORMAT='DELIMITED'</code> . По умолчанию в роли разделителя используется запятая (`,`). Другими допустимыми значениями являются <code>'SPACE'</code> и <code>'TAB'</code>	Нет

Продолжим обсуждение потоков и таблиц и посмотрим, как исследовать их содержимое после создания.

Работа с потоками и таблицами

В интерактивном режиме часто бывает полезно просмотреть содержимое или получить описание созданных коллекций. Иногда даже бывает желательно удалить коллекцию, потому что она стала ненужной или было решено создать другой поток или таблицу с тем же именем. В этом разделе мы обсудим все эти варианты использования и покажем примеры вывода соответствующих инструкций SQL. Для начала посмотрим, как вывести список активных потоков и таблиц, определенных в приложении.

Вывод списка потоков и таблиц

Иногда может понадобиться просмотреть некоторую информацию обо *всех* зарегистрированных потоках и таблицах. ksqlDB предлагает пару операторов, которые помогут в этом. Вот как выглядит их синтаксис:

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

Операторы `LIST` и `SHOW` взаимозаменяемы, далее в этой книге мы будем использовать вариант `SHOW`.

Итак, мы создали исходные поток и таблицу в нашем проекте. Давайте теперь попробуем получить некоторую информацию о них с помощью инструкции `SHOW`. Далее вывод каждой инструкции приводится непосредственно после команды:

```
ksql> SHOW TABLES ;
```

Table Name	Kafka Topic	Format	Windowed
TITLES	titles	AVRO	false

```
ksql> SHOW STREAMS ;
```

Stream Name	Kafka Topic	Format
PRODUCTION_CHANGES	production_changes	JSON

Как видите, инструкция вывода списка потоков и таблиц отображает самый минимум информации. Чтобы получить дополнительные сведения, можно добавить ключевое слово `EXTENDED`:

```
ksql> SHOW TABLES EXTENDED; ❶
```

```
Name           : TITLES
Type            : TABLE
Timestamp field : Not set - using <ROWTIME>
Key format      : KAFKA
Value format    : AVRO
Kafka topic     : titles (partitions: 4, replication: 1)
Statement       : CREATE TABLE titles (...) ❷
```

```
Field | Type
```

```
-----
ID    | INTEGER          (primary key)
TITLE | VARCHAR(STRING)
-----
```

```
Local runtime statistics ❸
```

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

(Statistics of the local KSQL server interaction with the Kafka topic titles)

❶ Поддерживается также аналогичная инструкция `SHOW STREAMS EXTENDED`, но мы опустили пример с потоками, потому что он не дает ничего нового.

❷ Полную инструкцию DDL вы увидите, повторив пример у себя; здесь мы сократили ее для краткости.

❸ Статистика времени выполнения включает дополнительные поля, не показанные здесь, в том числе `consumer-total-message-bytes`, `failed-messages-per-sec`, `last-failed` и т. д. Эта информация помогает получить общее представление о пропускной способности и частоте ошибок (например, ошибок десериализации) потоков и таблиц. Обратите внимание, что статистика не выводится, если она недоступна, например, если в потоке или таблице нет активности.

Инструкции `SHOW` используются для просмотра данных обо всех потоках/таблицах в кластере ksqlDB, поэтому предыдущий вывод будет повторяться для каждого зарегистрированного потока/таблицы.

Теперь, узнав, как вывести список всех потоков и таблиц в кластере, посмотрим, как получить описание для отдельных потоков и таблиц.

Получение описаний потоков и таблиц

Инструкция получения описания коллекций похожа на `SHOW`, но дополнительно принимает один аргумент — экземпляр потока или таблицы. Вот ее синтаксис:

```
DESCRIBE [EXTENDED] <идентификатор> ❶
```

❶ <идентификатор> — это имя потока или таблицы. Обратите внимание на отсутствие в этой инструкции ключевого слова **STREAM** или **TABLE**. Поскольку ksqlDB не позволяет создавать потоки и таблицы с одинаковыми именами, то и нет необходимости различать тип коллекции в этой инструкции; достаточно уникального идентификатора.

Например, вот как можно получить описание таблицы **titles**:

```
ksql> DESCRIBE titles ;
```

```
Name           : TITLES
Field | Type
-----|-----
ID    | INTEGER      (primary key)
TITLE | VARCHAR(STRING)
```

Чтобы получить больше информации, скажем статистику времени выполнения, можно воспользоваться вариантом **DESCRIBE EXTENDED**, как показано ниже. Мы опустили вывод инструкции **DESCRIBE EXTENDED**, потому что он идентичен выводу **SHOW { STREAMS | TABLES } EXTENDED**, отличаясь лишь тем, что содержит информацию только для указанного потока/таблицы:

```
ksql> DESCRIBE EXTENDED titles ;
```

Другая распространенная задача в традиционных базах данных — удаление объектов базы данных, ставших ненужными. ksqlDB тоже позволяет удалять потоки и таблицы, как будет показано в следующем разделе.

Изменение потоков и таблиц

Иногда может понадобиться изменить существующую коллекцию. Сделать это можно с помощью инструкции **ALTER**:

```
ALTER { STREAM | TABLE } <идентификатор> параметры [, ...]
```

Начиная с версии ksqlDB 0.14.0, с помощью инструкции **ALTER** можно только добавить столбцы, однако в будущем ее возможности могут быть расширены. Вот пример добавления столбца с помощью **ALTER**:

```
ksql> ALTER TABLE titles ADD COLUMN genre VARCHAR; ❶
```

```
Message
```

```
-----
Table TITLES altered.
-----
```

❶ Добавление столбца типа **VARCHAR** с именем **genre** в таблицу **titles**.

Удаление потоков и таблиц

Мои родители часто говорили: «Мы тебя сюда привели, мы тебя отсюда уведем». Точно так же, если вы добавили поток или таблицу и хотите забрать его, можете воспользоваться командой `DROP { STREAM | TABLE }`. Вот как выглядит ее полный синтаксис:

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

Например, если потребуется удалить поток `production_changes` и соответствующую тему, то можно выполнить следующую инструкцию¹:

```
ksql> DROP STREAM IF EXISTS production_changes DELETE TOPIC ;
```

Message

```
-----  
Source `PRODUCTION_CHANGES` (topic: production_changes) was dropped.  
-----
```

Будьте осторожны с дополнительным оператором `DELETE TOPIC`, так как ksqlDB сразу же удалит соответствующую тему. Оператор `DELETE TOPIC` можно не включать в инструкцию, если требуется удалить только таблицу.

К данному моменту вы многое узнали о работе с коллекциями. Например, вы знаете, как проверить их наличие и получить метаданные с помощью инструкций `SHOW` и `DESCRIBE`, как их создавать и удалять с помощью инструкций `CREATE` и `DROP`. Теперь рассмотрим дополнительные способы работы с потоками и таблицами, исследовав несколько основных шаблонов работы с потоками и связанные с ними операторы SQL.

Простые запросы

В этом разделе мы рассмотрим некоторые основные способы фильтрации и преобразования данных. На данный момент наше приложение мониторинга изменений использует данные из двух тем Kafka: `titles` и `production_changes`. Мы уже выполнили шаг 1 (см. рис. 10.1), создав исходные коллекции для каждой из этих тем (таблицу `titles` и поток `production_changes`). Теперь перейдем к шагу 2 — фильтрации потока `production_changes`, чтобы остались только изменения, касающиеся продолжительности сезона, к последующему преобразованию данных в более простой формат и затем к записи отфильтрованного и преобразованного потока в новую тему `season_length_changes`.

Начнем с инструкции, особенно полезной для разработки: `INSERT VALUES`.

¹ Если вы следите за нашими примерами и решили выполнить эту инструкцию, то обязательно создайте поток `production_changes` заново, прежде чем продолжать.

Вставка значений

Вставка значений в поток или таблицу чрезвычайно полезна, особенно когда есть необходимость заранее заполнить коллекцию данными. Семантика вставки в ksqlDB немного отличается от той, к которой наверняка привыкли пользователи традиционных баз данных. И для потоков, и для таблиц запись добавляется в конец базовой темы Kafka, но в таблицах хранится только последнее представление каждого ключа, поэтому для таблиц вставка больше похожа на операцию *upsert* (обновить или вставить; поэтому в ksqlDB нет отдельной инструкции UPDATE).

В этом проекте мы заранее заполним таблицу `titles` и поток `production_changes` некоторыми тестовыми данными, чтобы облегчить эксперименты с разными типами операторов SQL. Вот как выглядит синтаксис вставки значений в коллекцию:

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

Наше приложение интересуют только изменения в продолжительности сезона произведения, поэтому вставим следующую запись:

```
INSERT INTO production_changes (
    uuid,
    title_id,
    change_type,
    before,
    after,
    created_at
) VALUES (
    1,
    1,
    'season_length',
    STRUCT(season_id := 1, episode_count := 12),
    STRUCT(season_id := 1, episode_count := 8),
    '2021-02-08 10:00:00'
);
```

Поскольку приложение должно отфильтровывать все другие типы изменений, давайте для проверки фильтра вставим изменение даты выпуска. На этот раз используем несколько иной вариант инструкции `INSERT`, указав также значения для `ROWKEY` и `ROWTIME` — специальных псевдостолбцов, автоматически создаваемых в ksqlDB:

```
INSERT INTO production_changes (
    ROWKEY,
    ROWTIME,
```

```

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

```

Наконец, на всякий случай вставим также некоторые данные в таблицу `titles`. Для этого используем третий вариант `INSERT INTO VALUES`, без имен столбцов. Мы просто передадим значения в порядке определения столбцов (например, первое значение будет соответствовать полю `id`, второе — полю `title`):

```

INSERT INTO titles VALUES (1, 'Stranger Things');
INSERT INTO titles VALUES (2, 'Black Mirror');
INSERT INTO titles VALUES (3, 'Bojack Horseman');

```

Теперь, заполнив таблицу и поток тестовыми данными, перейдем к более захватывающей части: выполнению запросов к нашим коллекциям.

Простая выборка (временные push-запросы)

Запросы в простейшей своей форме называются временными (то есть непостоянными) push-запросами. Это обычные запросы `SELECT` с оператором `EMIT CHANGES` в конце:

```

SELECT select_expr [, ...]
FROM from_item
[ LEFT JOIN коллекция_для_соединения ON критерий_соединения ]
[ WINDOW оконное_выражение ]
[ WHERE условие ]
[ GROUP BY выражение_группировки ]
[ PARTITION BY выражение_разделения ]
[ HAVING условие_для_агрегированных_значений ]
EMIT CHANGES
[ LIMIT количество ];

```

Для начала выберем все записи из потока `production_changes`:

```

ksql> SET 'auto.offset.reset' = 'earliest'; ❶
ksql> SELECT * FROM production_changes EMIT CHANGES ; ❷

```

❶ Это позволит читать темы с самого начала, что особенно полезно в нашем случае, так как мы уже добавили некоторые тестовые данные.

❷ Запуск временного запроса.

В отличие от постоянных запросов, которые рассматриваются в конце главы, временные push-запросы не сохраняются после перезапуска сервера ksqlDB.

После запуска приведенный выше запрос выведет на консоль начальные данные и продолжит выполнение, ожидая поступления новых данных. Результат обработки запроса показан в примере 10.1.

Пример 10.1. Вывод временного push-запроса

ROWKEY	UUID	TITLE_ID	CHANGE_TYPE	BEFORE	AFTER	CREATED_AT
2	2	2	release_date	{SEASON_ID=1, EPISODE_COUNT=null}	{SEASON_ID=1, EPISODE_COUNT=null}	2021-02-08...
null	1	1	season_length	{SEASON_ID=1, EPISODE_COUNT=12}	{SEASON_ID=1, EPISODE_COUNT=8}	2021-02-08...

Если открыть еще одно окно терминала, запустить в нем клиент командной строки и выполнить инструкцию `INSERT VALUES`, чтобы вставить новые данные в поток `production_changes`, то вывод результатов обновится автоматически.

Временные push-запросы к таблицам работают точно так же. Убедиться в этом можно, запустив запрос `SELECT` к таблице `titles`:

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

И снова сразу после запуска запроса в консоли должно появиться начальное содержимое таблицы, как показано ниже, а по мере поступления новых данных вывод должен изменяться соответственно:

ID	TITLE
2	Black Mirror
3	Bojack Horseman
1	Stranger Things

Простые операторы `SELECT`, подобные приведенным выше, являются полезной отправной точкой в разработке приложений потоковой обработки, но обработка данных часто требует их преобразования. В следующих двух разделах мы рассмотрим некоторые основные подходы к преобразованию данных.

Проекция

Самая простая, пожалуй, форма преобразования данных включает выборку подмножества столбцов из доступных в потоке или таблице и, соответственно, упрощение модели данных для последующих действий. Эта операция называется *проекцией*, и для ее выполнения требуется заменить синтаксис `SELECT *` явным перечислением столбцов. Например, в этом проекте мы будем работать со столбцами `title_id`, `before`, `after` и `created_at` в потоке `production_changes`, поэтому напомним следующую инструкцию, чтобы получить проекцию этих столбцов для нового упрощенного потока:

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

В результате выполнения этого запроса получается упрощенный поток:

TITLE_ID	BEFORE	AFTER	CREATED_AT
2	{SEASON_ID=1, EPISODE_COUNT=null}	{SEASON_ID=1, EPISODE_COUNT=null}	2021-02-08...
1	{SEASON_ID=1, EPISODE_COUNT=12}	{SEASON_ID=1, EPISODE_COUNT=8}	2021-02-08...

Как видите, поток по-прежнему содержит записи, которые на самом деле не нужны нашему приложению. Нас интересуют только изменения `season_length`, а запись с `TITLE_ID=2` — это изменение `release_date`, как было показано в примере 10.1. Для решения задачи нам нужно выполнить фильтрацию потока. Давайте посмотрим, как это сделать с помощью ksqlDB.

Фильтрация

Диалект ksqlDB SQL включает вездесущий оператор `WHERE`, который можно использовать для фильтрации потоков и таблиц. Поскольку для нашего приложения интерес представляет только определенный вид изменений (изменения продолжительности сезона), то для фильтрации записей можно использовать инструкцию, показанную в примере 10.2.

Пример 10.2. Инструкция с оператором `WHERE` для фильтрации записей

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

Вывод этого запроса показан в примере 10.3.

Пример. 10.3. Результат фильтрации

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

Теперь, прежде чем двинуться дальше, исследуем некоторые особенности оператора `WHERE`.



Помимо вариантов оператора `WHERE`, которые мы обсудим ниже, существует также предикат `IN`, который в настоящее время поддерживается для pull-запросов (мы рассмотрим его в следующей главе). Предполагается, что в будущем предикат `IN` будет поддерживаться и для push-запросов. Он предназначен для сопоставления с несколькими значениями (например, `WHERE id IN (1, 2, 3)`). Если вы используете версию `ksqlDB` выше 0.14.0, то загляните в журнал изменений `ksqlDB` (<https://oreil.ly/QkAJm>), чтобы узнать, поддерживает ли ваша версия `ksqlDB` предикат `IN` для push-запросов.

Подстановочные знаки

`ksqlDB` поддерживает также фильтрацию с подстановочными знаками, которая может пригодиться для сопоставления только части значения столбца. Для более мощного условия фильтрации можно использовать оператор `LIKE` с символом `%`, который представляет ноль или более символов, как показано ниже:

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

❶ Соответствует любой записи, в которой значение столбца `change_type` начинается со слова *season*.

Логические операторы

С помощью логических операторов `AND/OR` можно объединить сразу несколько условий фильтрации. Также для группировки условий можно использовать круглые скобки. Условия можно инвертировать с помощью оператора `NOT`. Следующая инструкция SQL демонстрирует все эти идеи:

```
SELECT title_id, before, after, created_at
FROM production_changes
```

```
WHERE NOT change_type = 'release_date' ❶  
AND ( after->episode_count >= 8 OR after->episode_count <=20 ) ❷  
EMIT CHANGES ;
```

❶ Включит в набор результатов только записи, в которых столбец `change_type` имеет любое значение, отличное от `release_date`.

❷ Два условия в скобках оцениваются вместе. В этом случае в набор результатов входят записи, в которых количество новых серий находится в диапазоне от 8 до 20 включительно.

Наконец, второе условие, включающее в набор результатов записи с количеством серий в диапазоне от 8 до 20, можно выразить более наглядно и компактно. Это будет показано в следующем разделе.

Фильтрация по диапазону (BETWEEN)

Для фильтрации записей со значениями, попадающими в определенный числовой или буквенно-цифровой диапазон, можно использовать оператор `BETWEEN`. Диапазон подразумевает *включение* граничных значений, то есть оператор `BETWEEN 8 AND 20` оставит значения от 8 до 20, включая 8 и 20, например:

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

Для нашего проекта достаточно условия фильтрации, показанного в примере 10.2, но теперь вы должны хорошо понимать, как использовать другие виды условий.

Итак, наш поток отфильтрован, но, как показано в примере 10.3, два столбца в выводе (`before` и `after`) содержат сложные многомерные структуры (например, `{SEASON_ID=1, EPISODE_COUNT=12}`). Это подчеркивает необходимость еще одного распространенного преобразования на этапе предварительной обработки данных: развертывание/упрощение сложных структур в более простые структуры. Это преобразование мы рассмотрим далее.

Развертывание/упрощение сложных вложенных структур

Развертывание значений включает перенос вложенных полей в сложной структуре (например, `STRUCT`) в столбцы верхнего уровня, имеющие единственное значение. Так же как проекция, это помогает упростить модель данных и подготовить ее для последующих операций.

Например, в предыдущем разделе мы видели вот такое сложное значение в столбце `after` для одной записи:

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

Получить доступ к полям вложенной структуры и вывести их значения в отдельные столбцы можно с помощью оператора `->`:

```
SELECT
  title_id,
  after->season_id,
  after->episode_count,
  created_at
FROM production_changes
WHERE change_type = 'season_length'
EMIT CHANGES ;
```

Как видите, эта инструкция производит эффект развертывания, когда многомерная точка данных (`{SEASON_ID=1, EPISODE_COUNT=8}`) преобразуется в два отдельных столбца верхнего уровня:

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

Мы встали на путь создания преобразованной версии потока `production_changes`, которую позже можно использовать для более сложных операций. Но прежде, чем записать результаты обратно в Kafka, рассмотрим еще один тип выражений, который пригодится для базовой потоковой обработки: условные выражения.

Условные выражения

ksqlDB поддерживает несколько видов условных выражений. Их можно использовать по-разному, но чаще они применяются для решения потенциальных проблем с целостностью данных в потоке или таблице путем подстановки альтернативных значений вместо `NULL`.

В качестве примера предположим, что нужно решить потенциальную проблему целостности данных в потоке `production_changes`, где `season_id` может иметь значение `NULL` в одном из столбцов, `before` или `after`, но не в двух сразу. Для этого можно использовать один из трех видов условных выражений и при необходимости подставить в столбец `season_id` альтернативное значение. Начнем с функции `COALESCE`.

COALESCE

Функция `COALESCE` возвращает первое значение, отличное от `NULL`, из заданного списка значений. Вот как выглядит сигнатура функции `COALESCE`:

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

Например, следующим образом можно реализовать инструкцию `SELECT`, замещающую значение `NULL` в столбце `season_id`:

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

В этом примере, если `after->season_id` содержит `NULL`, возвращается значение `before->season_id`. Если `before->season_id` тоже содержит `NULL`, то возвращается значение по умолчанию `0`.

IFNULL

Функция `IFNULL` подобна функции `COALESCE`, но имеет только одно значение для подстановки. Вот сигнатура функции `IFNULL`:

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

Чтобы вернуть значение `before->season_id`, если `after->season_id` содержит `NULL`, можно изменить нашу инструкцию так:

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

Оператор CASE

Из всех условных выражений в `ksqlDB` самым мощным является оператор `CASE`. Он позволяет оценивать любое количество логических условий и возвращать первое значение, при котором условие дает истинное значение. В отличие от `COALESCE` и `IFNULL` условия в операторе `CASE` могут проверять совпадение не только со значением `NULL`.

Синтаксис оператора `CASE`:

```
CASE выражение
  WHEN условие THEN результат [, ...]
  [ELSE результат]
END
```

Например, ниже показан оператор CASE с несколькими условиями. Он возвращает 0, если оба столбца, `after->season_id` и `before->season_id`, имеют значение NULL:

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

Если требуется просто проверить на равенство значению NULL, как в нашем случае, то достаточно использовать COALESCE или IFNULL. В этом проекте мы будем использовать IFNULL.

Теперь наконец можно записать отфильтрованный и преобразованный поток `production_changes` обратно в Kafka.

Запись результатов обратно в Kafka (постоянные запросы)

До настоящего момента мы работали с временными запросами. Эти запросы начинаются с SELECT и возвращают результат клиенту, но не записывают его обратно в Kafka. Кроме того, временные запросы не перезапускаются после повторного запуска сервера.

ksqlDB позволяет также создавать так называемые постоянные запросы, которые записывают результаты в Kafka и вновь стартуют после повторного запуска сервера. Это удобно, когда нужно сделать доступным другим клиентам отфильтрованный, преобразованный и/или обогащенный поток. Для записи результатов запроса обратно в Kafka можно создать производные коллекции. Что это такое, мы обсудим в следующем разделе.

Создание производных коллекций

Производные коллекции — это потоки и таблицы, созданные на основе других потоков и таблиц. Синтаксис создания производных коллекций немного отличается от синтаксиса создания исходных коллекций: в нем отсутствуют схемы столбцов и добавлен оператор AS SELECT. Полный синтаксис создания производных коллекций выглядит следующим образом:

```

CREATE { STREAM | TABLE } [ IF NOT EXISTS ] <идентификатор>
WITH (
    свойство=значение [, ... ]
)
AS SELECT select_expr [, ...]
FROM исходная_коллекция
[ LEFT JOIN коллекция_для_соединения ON критерий_соединения ]
[ WINDOW оконное_выражение ]
[ WHERE условие ]
[ GROUP BY выражение_группировки ]
[ PARTITION BY выражение_разделения ]
[ HAVING условие_для_агрегированных_значений ]
EMIT CHANGES
[ LIMIT количество ];

```

Запросы для создания производных потоков часто обозначаются одной из двух аббревиатур:

- запросы *CSAS* (произносится как *си-кас* и обозначает `CREATE STREAM AS SELECT`) используются для создания производных потоков;
- запросы *CTAS* (произносится как *си-мас* и обозначает `CREATE TABLE AS SELECT`) используются для создания производных таблиц.

Давайте применим фильтр, преобразование данных и условные выражения, о которых вы узнали к данному моменту, и создадим производный поток с именем `season_length_changes`, содержащий только нужные нам типы изменений:

```

CREATE STREAM season_length_changes ❶
WITH ( ❷
    KAFKA_TOPIC = 'season_length_changes', ❸
    VALUE_FORMAT = 'AVRO',
    PARTITIONS = 4,
    REPLICAS = 1
) AS SELECT ❹
    ROWKEY, ❺
    title_id, ❻
    IFNULL(after->season_id, before->season_id) AS season_id, ❼
    before->episode_count AS old_episode_count, ❽
    after->episode_count AS new_episode_count,
    created_at
FROM production_changes
WHERE change_type = 'season_length' ❾
EMIT CHANGES ;

```

❶ Синтаксически создание производных коллекций похоже на создание исходных коллекций, например, `CREATE STREAM` и `CREATE TABLE`.

❷ Оператор `WITH` тоже поддерживается при создании производных коллекций.

❸ Производный поток будет записываться в тему `season_length_changes`.

- ❹ В операторе `AS SELECT` определяется запрос для заполнения производного потока или таблицы.
- ❺ В проекцию обязательно должен включаться столбец ключа. Он сообщает ksqlDB, какое значение следует использовать для ключа в записях Kafka.
- ❻ Указывая отдельные столбцы, мы используем *проекцию* для изменения формы исходного потока или таблицы.
- ❼ Условное выражение `IFNULL` позволяет избежать некоторых проблем с целостностью данных. Здесь и в следующих двух строках мы также используем оператор `AS`, чтобы явно указать имя столбца.
- ❸ Развертывание вложенных или сложных значений упростит работу с коллекцией в нижестоящих обработчиках и на стороне клиентов.
- ❾ Фильтрация позволяет оставить только определенные записи, что является еще одним распространенным вариантом использования потоковой обработки.

После создания производной коллекции в консоли должно появиться подтверждение, что запрос создан:

Message

Created query with ID CSAS_SEASON_LENGTH_CHANGES_0

Как видите, после выполнения предыдущей инструкции CSAS ksqlDB создала непрерывно выполняющийся/постоянный запрос. Запросы, которые создает ksqlDB (например, `CSAS_SEASON_LENGTH_CHANGES_0`), по сути, являются приложениями Kafka Streams. Их динамически создает ksqlDB для реализации обработки предоставленных операторов SQL.

Прежде чем закрыть эту главу, давайте познакомимся еще с некоторыми операторами, которые могут пригодиться при работе с простыми запросами.

Вывод списка запросов

Независимо от вида запросов — временных, просто возвращающих данные, или постоянных, создающих производную коллекцию, — иногда бывает полезно просмотреть список активных запросов в кластере ksqlDB и их текущее состояние.

Инструкция получения списка активных запросов имеет следующий синтаксис:

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


Создав производный поток `season_length` с отфильтрованными и преобразованными изменениями, можно просмотреть некоторую информацию о базовом запросе, выполнив следующую инструкцию:

```
ksql> SHOW QUERIES;
```

Query ID	Query Type	Status ❶
CSAS_SEASON_LENGTH_CHANGES_0	PERSISTENT	RUNNING:1 ❷

❶ Некоторые столбцы в выводе, возвращаемом инструкцией `SHOW QUERIES`, были нами усечены для краткости. Но вообще в выводе содержится больше информации, включая имя темы Kafka, куда записывается постоянный запрос, исходную строку запроса и многое другое.

❷ Идентификатор запроса (`CSAS_SEASON_LENGTH_CHANGES_0`) необходим для определенных операций (например, для завершения и получения описания запроса). Тип запроса отображается как `PERSISTENT` (постоянный) или `PUSH`. `PERSISTENT` — если выполняется инструкция `CSAS` или `CTAS`, `PUSH` — если запрос временный (например, начинается с `SELECT` вместо `CREATE`). Наконец, статус запроса указывает, что он находится в состоянии `RUNNING` (выполняется). Другие допустимые состояния запроса: `ERROR` (ошибка) и `UNRESPONSIVE` (не отвечает).

Получение описаний запросов

Когда нужно получить больше информации об активном запросе, чем дает оператор `SHOW QUERIES`, можно запросить дополнительное описание. Синтаксис получения описания:

```
EXPLAIN { идентификатор_запроса | инструкция_запроса }
```

Например, следующая инструкция вернет описание запроса, созданного в предыдущем разделе:

```
ksql> EXPLAIN CSAS_SEASON_LENGTH_CHANGES_0 ;
```

Аналогично можно получить описание гипотетического запроса, который на самом деле не выполняется. Например:

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

В любом случае вывод получится очень подробным, поэтому оставляю вам его получение как самостоятельное упражнение. В описаниях вы увидите имена и типы полей, состояние серверов ksqlDB, выполняющих запрос (если описание получено не для гипотетического запроса), и описание базовой топологии Kafka Streams, которая используется для фактического выполнения запроса.

Завершение запросов

Выше в этой главе вы узнали, как удалять запущенные на выполнение потоки и таблицы. Однако если применить описанный способ для удаления только что созданного потока `season_length_changes`, то вы увидите ошибку:

```
ksql> DROP STREAM season_length_changes ;
```

```
Cannot drop SEASON_LENGTH_CHANGES.
```

```
The following queries read from this source: [].
```

```
The following queries write into this source: [CSAS_SEASON_LENGTH_CHANGES_0].
```

```
You need to terminate them before dropping SEASON_LENGTH_CHANGES.
```

Текст описания ошибки¹ говорит сам за себя: вы не можете удалить коллекцию, если к ней в данный момент обращается запрос (например, читает из нее или пишет в нее). Поэтому сначала нужно завершить базовый запрос.

Синтаксис завершения запроса:

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

Если запросов несколько, их можно завершить все сразу, выполнив инструкцию `TERMINATE ALL`. Завершить только один запрос, а именно созданный выше (с идентификатором `CSAS_SEASON_LENGTH_CHANGES_0`), можно инструкцией:

```
TERMINATE CSAS_SEASON_LENGTH_CHANGES_0 ;
```

После завершения запроса и при отсутствии других запросов, обращающихся к потоку или таблице, можно удалить коллекцию командой `DROP { STREAM | TABLE }`, как обсуждалось в подразделе «Удаление потоков и таблиц» выше.

Все вместе

В этой главе мы рассмотрели множество вариантов потоковой обработки и операторов SQL, но фактический код, необходимый для реализации шагов 1 и 2 в нашем приложении, довольно лаконичен. Полный набор запросов, необходимый для выполнения этой части проекта, показан в примере 10.4.

¹ Нельзя удалить `SEASON_LENGTH_CHANGES`.

Из этого источника читают следующие запросы: [].

В этот источник записывают следующие запросы: [CSAS_SEASON_LENGTH_CHANGES_0].

Завершите их, прежде чем удалять `SEASON_LENGTH_CHANGES`. — *Примеч. пер.*

Пример 10.4. Набор запросов для реализации шагов 1 и 2 в приложении мониторинга изменений по образцу и подобию Netflix

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

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

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

В следующей главе мы продолжим работу над проектом и посмотрим, как обогащать, агрегировать и материализовывать данные с помощью ksqlDB (см. шаги 3–5 на рис. 10.1).

Заключение

Теперь вы знаете, как решать многие важные задачи обработки потоков с помощью ksqlDB, включая фильтрацию данных, развертывание сложных структур, применение условных выражений и многое другое. Вы также познакомились с некоторыми операторами SQL, которые могут пригодиться для анализа состояния потоков, таблиц и запросов, и, конечно же, с инструкциями создания и удаления каждой из этих сущностей.

Эта глава была посвящена предварительной обработке и преобразованию данных, однако мир ksqlDB намного шире и вам еще предстоит открыть многое, в том числе приемы обогащения и агрегирования данных. В следующей главе мы подробно рассмотрим эти темы и вы познакомитесь с более мощным набором операторов и конструкций SQL, доступных в ksqlDB.

Продвинутая обработка потоков с ksqlDB

В предыдущей главе вы узнали, как выполнять предварительную обработку и преобразование данных с помощью ksqlDB. Обсуждавшиеся в ней операторы SQL не имели состояния и позволяли фильтровать данные, развертывать сложные или вложенные структуры, использовать проекцию для изменения формы данных и многое другое. В этой главе мы продолжим исследовать ksqlDB и обсудим некоторые варианты обогащения и агрегирования данных. Большинство операторов, представленных здесь, имеют состояние (например, оперируют несколькими записями, что требуется для вычисления соединений и агрегирования) и основаны на времени (например, оконные операции), что делает их более сложными, но и более мощными.

Вот некоторые из тем, рассматриваемых в этой главе:

- использование соединений для объединения и обогащения данных;
- агрегирование;
- выполнение pull-запросов (точечный поиск) к материализованным представлениям с использованием интерфейса командной строки;
- работа со встроенными функциями ksqlDB (скалярными, агрегатными и табличными);
- создание пользовательских функций на Java.

Мы продолжим развивать учебный проект мониторинга изменений в Netflix, начатый в предыдущей главе, и на его примере представим многие из этих концепций (загляните в раздел «Учебный проект: мониторинг изменений в Netflix» главы 10, если вам нужно освежить память). Однако некоторые темы, в том

числе функции ksqlDB, будут представлены в виде отдельных обсуждений ближе к концу главы.

Начнем с настройки проекта.

Настройка проекта

Если вы решите следовать примерам в этой главе, клонируйте репозиторий (<https://github.com/mitch-seymour/mastering-kafka-streams-and-ksqldb.git>) и перейдите в каталог с учебным проектом для этой главы. Для этого выполните следующие команды:

```
$ git clone git@github.com:mitch-seymour/mastering-kafka-streams-and-ksqldb.git
$ cd mastering-kafka-streams-and-ksqldb/chapter-11/
```

Как обсуждалось в предыдущей главе, для запуска каждого компонента этого проекта (например, Kafka, сервера ksqlDB, CLI и т. д.) мы используем Docker Compose. Просто выполните следующую команду после клонирования репозитория, чтобы запустить все компоненты:

```
docker-compose up
```

Если не указано иное, операторы SQL, обсуждаемые в этой главе, будут выполняться в интерфейсе командной строки ksqlDB CLI. Выполнить вход в ksqlDB CLI можно командой:

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

До сих пор настройка ничем не отличалась от настройки в главе 10. Однако теперь мы продолжим развивать проект из предыдущей главы, поэтому нужно дополнительно настроить окружение ksqlDB, чтобы привести ее в состояние, в котором она была в предыдущей главе. Эта необходимость подводит нас к очень важному оператору ksqlDB, который мы обсудим в следующем разделе.

Инициализация окружения из файла SQL

Мы уже продвинулись вперед в разработке проекта приложения мониторинга изменений в Netflix, и теперь у нас есть набор запросов, которые нужно запустить вновь, чтобы продолжить с того места, где мы остановились. Сейчас это может показаться уникальным требованием данного конкретного проекта, но вообще выполнение набора запросов для настройки окружения является обычной

рутинной процедурой разработки, и ksqlDB включает специальный оператор, упрощающий необходимые действия.

Он имеет следующий синтаксис:

```
RUN SCRIPT <файл_sql> ❶
```

❶ Файл SQL, содержащий запросы для выполнения.

Мы можем поместить все запросы из предыдущей главы (см. пример 10.4) в файл с именем `/etc/sql/init.sql` и при необходимости выполнить следующую команду, чтобы воссоздать коллекции и запросы, разработанные ранее:

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

После запуска `RUN SCRIPT` вывод каждой инструкции, включенной в файл SQL, будет возвращен клиенту. В интерфейсе командной строки вывод просто появится на экране. Ниже показан усеченный вывод для запросов из главы 10:

```
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  
-----
```

Как видите, `RUN SCRIPT` — очень полезный механизм, помогающий экономить время на этапе настройки окружения при разработке приложений ksqlDB. Мы можем поэкспериментировать с инструкциями в интерфейсе командной строки и добавить вновь разработанные запросы в файл, который можно выполнить позже. Когда приложение будет готово к развертыванию в рабочей среде, мы

сможем использовать тот же файл SQL, чтобы запустить ksqlDB в автономном режиме (см. рис. 8.9).

На рис. 11.1 наглядно показаны шаги, выполненные инструкцией `RUN SCRIPT`, а также шаги, которые предстоит выполнить в этой главе.

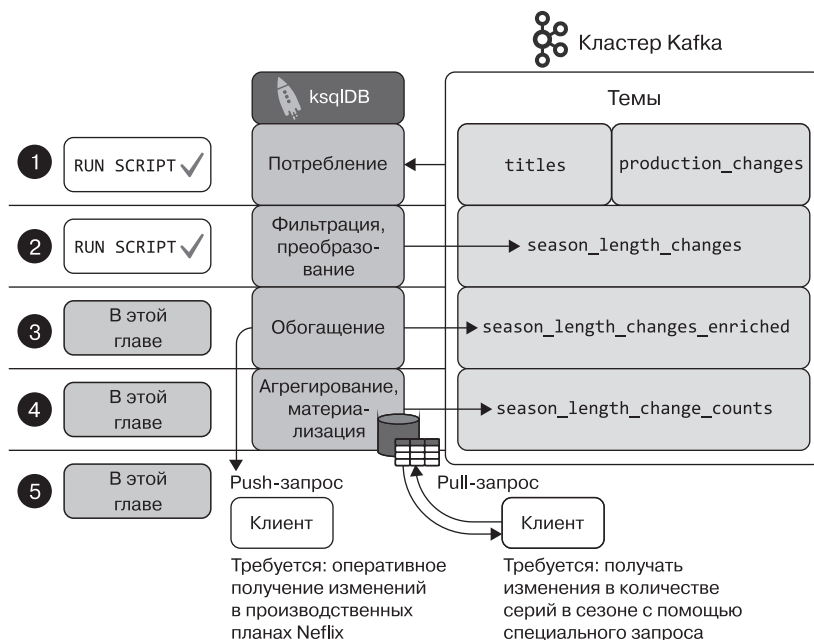


Рис. 11.1. Архитектурная схема приложения мониторинга изменений для Netflix; шаги 1 и 2 были выполнены изначально в предыдущей главе и повторно, с использованием оператора `RUN SCRIPT`, — в этой

Наш следующий шаг — шаг 3, на котором необходимо обогатить отфильтрованные и преобразованные данные в потоке `season_length_changes`. Он будет рассмотрен в следующем разделе.

Обогащение данных

Под обогащением данных подразумевается процесс улучшения или дополнения исходных данных. Это уже не просто преобразование, обычно направленное на изменение формата или структуры данных. Обогащение предполагает *добавление информации* к данным, и одним из наиболее широко используемых методов обогащения в мире баз данных является *соединение*.

На шаге 3 в нашем приложении мы обогатим поток `season_length_changes` информацией из таблицы `titles`.

Соединения

Соединение — это объединение связанных записей из нескольких источников с использованием *предиката соединения* (логического выражения, принимающего истинное значение, если связанная запись найдена, и ложное — в противном случае). Соединения распространены в реляционном мире и в мире потоков, потому что данные часто разбросаны по многим источникам и их требуется объединить для обработки и анализа.

ksqlDB поддерживает множество видов соединений, однако все их варианты можно выразить с помощью двух компонентов:

- выражения соединения (`INNER JOIN`, `LEFT JOIN` и `FULL JOIN`);
- типа объединяемых коллекций (`STREAM`, `TABLE`).

Начнем с первого пункта. В табл. 11.1 приведены описания всех доступных видов соединений в ksqlDB.

Таблица 11.1. Выражения соединения, доступные в ksqlDB

SQL-выражение	Описание
<code>INNER JOIN</code>	Внутреннее соединение. Применяется, когда с обеих сторон соединения присутствуют записи с одним и тем же значением ключа
<code>LEFT JOIN</code>	Левое соединение. Применяется к каждой записи с <i>левой стороны</i> соединения. Если с правой стороны нет соответствующей записи с тем же ключом, то взамен значения справа используется <code>NULL</code>
<code>FULL JOIN</code>	Полное соединение. Применяется к каждой записи с <i>любой стороны</i> соединения. Если на противоположной стороне соединения нет соответствующей записи, то взамен отсутствующего значения используется <code>NULL</code>

Имеющим опыт работы с традиционными базами данных приведенные выше выражения соединений должны показаться знакомыми. Но, в отличие от традиционных баз данных, выполняющих соединение таблиц, ksqlDB поддерживает соединения коллекций двух типов: таблиц и потоков. Кроме того, тип соединения, который можно использовать (`INNER JOIN`, `LEFT JOIN` или `FULL JOIN`), зависит от типов коллекций (`STREAM` или `TABLE`). В табл. 11.2 перечислены доступные комбинации с дополнительным столбцом (`В окне`), указывающим необходимость ограничения соединения по времени.

Таблица 11.2. Типы соединений в ksqlDB

Вид соединения	Поддерживаемое выражение	В окне
Поток — поток	INNER JOIN	Да
	LEFT JOIN	
	FULL JOIN	
Поток — таблица	INNER JOIN	Нет
	LEFT JOIN	
Таблица — таблица	INNER JOIN	Нет
	LEFT JOIN	
	FULL JOIN	

Как видите, соединения «поток — поток» выполняются в окне. Поскольку потоки не ограничены, приходится ограничивать поиск связанных записей некоторым временным интервалом, указанным пользователем, потому что задача поиска связанных записей в двух или более неограниченных непрерывных потоках просто невыполнима. Именно поэтому ksqlDB ограничивает вычисление этого типа соединения окнами.

ПРЕДВАРИТЕЛЬНЫЕ УСЛОВИЯ ДЛЯ СОЕДИНЕНИЯ КОЛЛЕКЦИЙ

Прежде чем мы начнем писать запросы с соединениями, нужно отметить некоторые предварительные условия, а именно:

- все столбцы, участвующие в соединении, должны иметь один и тот же тип данных (STRING, INT, LONG и т. д.);
- количество разделов с каждой стороны соединения должно быть одинаковым¹;
- данные в базовых темах должны быть записаны с использованием одной и той же стратегии секционирования (обычно это означает, что производители используют механизм секционирования по умолчанию, который создает хеш на основе ключа входной записи).

Теперь, учитывая все эти требования, напомним запрос с соединением. Предварительно обработанный поток данных `season_length_changes` в нашем проекте включает столбец с именем `title_id`. Его значение можно использовать для

¹ Это условие проверяется при выполнении инструкции SQL, содержащей соединение. Если количество разделов не совпадает, вы увидите ошибку, например такую: Can't join S with T since the number of partitions don't match (Нельзя вычислить соединение между S и T, потому что количество разделов не совпадает).

поиска дополнительной информации о произведении (включая название, например *Stranger Things* или *Black Mirror*, хранящееся в таблице `titles`). Чтобы выразить это как внутреннее соединение, можно выполнить инструкцию SQL, показанную в примере 11.1.

Пример 11.1. Инструкция SQL, вычисляющая соединение двух коллекций

```
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 ❶
ON s.title_id = t.id ❷
EMIT CHANGES ;
```

❶ Оператор `INNER JOIN` используется потому, что соединение должно вычисляться, только если в таблице `titles` имеется соответствующая запись.

❷ В отношении потока можно указать любой столбец, который будет использоваться для соединения. Для таблиц должен указываться столбец, обозначенный как `PRIMARY KEY`. Это требование легко объяснить, если учесть, что таблицы являются хранилищами ключей и значений. По мере поступления новых записей из потока требуется выполнить точечный поиск в таблице, чтобы определить наличие соответствующей записи. Использование `PRIMARY KEY` — наиболее эффективный способ сделать это, потому что записи уже имеют это значение в базовом хранилище состояний. Это также гарантирует, что записи из потока будут соответствовать не более одной записи из таблицы, потому что первичный ключ является ограничением уникальности для таблиц.

Предыдущая инструкция выведет новую расширенную запись, содержащую название произведения (столбец `TITLE`):

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

Предыдущее соединение очень простое отчасти потому, что все предварительные условия уже были выполнены в этом проекте. Однако в реальных приложениях эти условия могут создавать проблемы. Давайте кратко рассмотрим несколько более сложных ситуаций, требующих дополнительных шагов для вычисления соединения.

Приведение типов столбцов

В некоторых случаях столбцы данных, участвующие в соединении, могут иметь разные типы. Например, представьте, что `s.title_id` на самом деле имеет тип `VARCHAR` и его нужно соединить со столбцом `ROWKEY` типа `INT`. В такой ситуации можно использовать выражение `CAST`, преобразовать `s.title_id` в тип `INT` и таким образом соблюсти первое условие для соединения данных, как описано во врезке «Предварительные условия для соединения коллекций» выше:

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

❶ Имена столбцов опущены для краткости.

❷ Если бы столбец `s.title_id` в этом гипотетическом примере имел тип `VARCHAR`, нам пришлось бы привести `s.title_id` к одному типу со столбцом `t.id` (`INT`), чтобы выполнить первое предварительное условие соединения.

Секционирование данных

Теперь рассмотрим ситуацию, которая требует повторного секционирования данных перед вычислением соединения. Такое может случиться, если не выполняется одно из двух последних условий из врезки «Предварительные условия для соединения коллекций» (см. выше). Например, представьте, что таблица `titles` состоит из восьми разделов, а таблица `season_length_changes` — из четырех.

В этом случае потребуется перераспределить одну из коллекций, чтобы выполнить соединение. Это можно сделать, создав новую коллекцию со свойством `PARTITIONS`, как показано в примере 11.2.

Пример 11.2. Пример повторного секционирования данных с помощью свойства `PARTITIONS`

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

❶ По умолчанию будет создана новая тема с именем, совпадающим с именем коллекции (`TITLES_REPARTITION`). Дополнительно можно указать свойство `KAFKA_TOPIC`, если понадобится дать новой теме осмысленное индивидуальное имя.

После секционирования данных, обеспечивающего одинаковое количество разделов с обеих сторон соединения, можно выполнить соединение с секционированной коллекцией (`titles_repartition`).

Постоянные соединения

Теперь, зная, как удовлетворить предварительные условия для соединения, продолжим реализацию шага 3 нашего проекта. Отправной точкой для реализации соединения нам послужит пример 11.1, потому что предварительные условия уже выполняются.

Как обсуждалось в предыдущей главе, запросы, начинающиеся с `SELECT`, — это временные запросы, то есть они не возобновляются после перезапуска сервера, а результаты не записываются обратно в Kafka. Поэтому, чтобы фактически сохранить результаты соединения, а не просто вывести их в консоль, нужно использовать инструкцию `CREATE STREAM AS SELECT` (также известную как `CSAS`).

Следующая инструкция определяет постоянную версию запроса с соединением, которая записывает полученные записи в новую тему с именем `season_length_changes_enriched` (см. шаг 3 на рис. 11.1):

```
CREATE STREAM season_length_changes_enriched ❶
WITH (
  KAFKA_TOPIC = 'season_length_changes_enriched',
  VALUE_FORMAT = 'AVRO',
  PARTITIONS = 4,
  TIMESTAMP='created_at', ❷
  TIMESTAMP_FORMAT='yyyy-MM-dd HH:mm:ss'
) AS
SELECT ❸
  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 ;
```

❶ Инструкцию `CREATE STREAM` используем, чтобы сделать запрос постоянным.

❷ Это выражение указывает ksqlDB, что столбец `created_at` содержит отметку времени, которую ksqlDB должен использовать для операций, основанных на времени, включая оконные агрегирование и соединение.

❸ В этой строке начинается исходный оператор `SELECT`.

Теперь вы знаете, как с помощью ksqlDB вычислять соединения, и уже реализовали шаг 3 в нашем проекте. Прежде чем завершим обсуждение соединений, поговорим еще об одном типе соединений, который часто используется в приложениях потоковой обработки.

Оконные соединения

В рассматриваемом проекте нам не нужны оконные соединения. Но было бы упущением хотя бы просто не упомянуть их в этой книге, потому что они необходимы для соединений «поток — поток». Оконные соединения отличаются от обычных соединений (таких как в предыдущем разделе) дополнительным атрибутом, определяющим соединение *временем*.

Окнам и времени посвящена целая глава (см. главу 5) в этой книге, поэтому для более глубокого изучения этого вопроса вы можете обратиться к ней. Как отмечалось выше в текущей главе, оконные соединения используют так называемое *скользящее окно* для группировки записей, попадающих в заданные границы времени. Это показано на рис. 11.2.

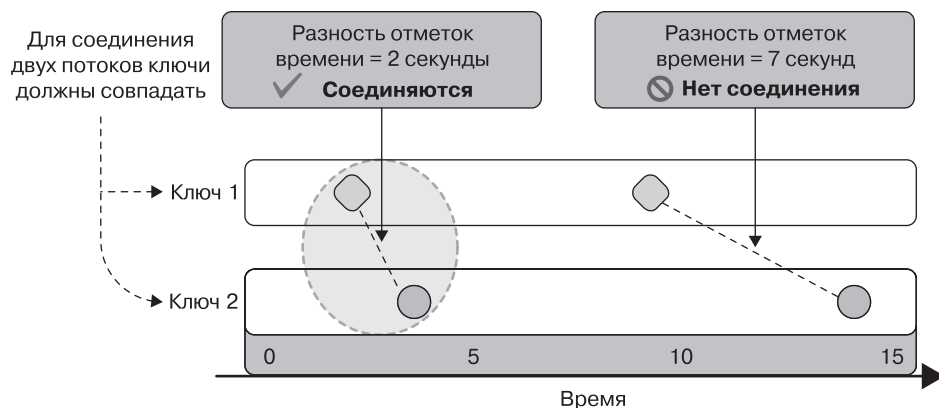


Рис. 11.2. Скользящее окно

Чтобы создать оконное соединение, нужно добавить выражение `WITHIN` в оператор соединения:

```
WITHIN <число> <единицы_измерения_времени>
```

Поддерживаются следующие единицы измерения времени, включая формы в единственном и множественном числе:

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

Ненадолго отвлечемся от нашего проекта (он не требует оконного соединения) и рассмотрим другой вариант использования. Вы работаете в Netflix и решили получить список всех сериалов или фильмов с продолжительностью просмотра менее двух минут до того, как пользователь завершит просмотр. В этой вымышленной ситуации события начала и окончания просмотра записываются в отдельные темы Kafka. В таких условиях показательно демонстрируется использование оконного соединения, потому что для достижения поставленной цели нужно выполнить соединение записей по идентификатору сеанса `session_id` (по обозначению, однозначно идентифицирующему сеанс просмотра) и времени события. Переведем этот пример на язык SQL и создадим для начала два исходных потока, как показано в примере 11.3.

Пример 11.3. Создание двух потоков для событий начала и окончания просмотра

```
CREATE STREAM start_watching_events ( ❶
    session_id STRING, ❷
    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' ❸
);

CREATE STREAM stop_watching_events ( ❹
    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'
);
```

❶ Создать поток для событий начала просмотра. В этот поток будет помещаться новая запись каждый раз, когда пользователь начинает смотреть сериал или фильм.

❷ `session_id` — один из атрибутов соединения. Он представляет один сеанс просмотра и присутствует в обоих потоках.

❸ Необязательный параметр `TIMESTAMP_FORMAT` позволяет задать формат временных отметок в этом разделе Kafka. Он может быть любым значением, поддерживаемым классом `DateTimeFormatter` в Java.

❹ Создать поток для событий окончания просмотра. В этот поток будет помещаться новая запись каждый раз, когда пользователь прекратит смотреть сериал или фильм.

Теперь добавим пару событий начала и окончания просмотра для двух разных сеансов. Они могут соотноситься с двумя разными зрителями, просматривающими два разных сериала. Первый сеанс, `session_123`, имеет общее время просмотра 90 секунд, а второй сеанс, `session_456`, — 25 минут:

```
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');
```

Наконец, извлечем сеансы продолжительностью менее двух минут, используя оконное соединение. Для этого просто выполним такой запрос: узнаем, просмотр каких произведений прекратился (на это указывает отметка времени окончания просмотра) менее чем через две минуты после начала (на это указывает отметка времени начала просмотра). Следующая инструкция SQL демонстрирует, как задать этот вопрос в ksqlDB:

```
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 ❶
ON A.session_id = B.session_id
EMIT CHANGES ;
```


❶ Соединение записей в потоках `start_watch_events` и `stop_watching_events` с отметками времени, отстоящими друг от друга менее чем на две минуты. Отметки имеют одинаковые значения `session_id` (последнее условие выражается в следующей строке). Это позволяет отобразить все сеансы просмотра с продолжительностью менее двух минут.

На экране должен появиться следующий вывод, так как сеанс `session_123` длился всего 90 секунд:

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

Теперь, разобравшись с оконными и обычными соединениями, продолжим работу над нашим учебным проектом (приложением мониторинга изменений в Netflix) и посмотрим, как агрегировать данные в ksqlDB.

Агрегирование

Работа с записями по одной используется для решения многих задач, включая фильтрацию, преобразование структуры данных и даже обогащение по одному событию за раз. Но некоторые из самых важных случаев применения, помогающих извлечь дополнительную информацию из данных, требуют группировки и агрегирования связанных записей. Например, подсчет количества изменений продолжительности сезона в Netflix за определенный период времени может помочь улучшить процессы планирования новых сериалов или фильмов или выявить проекты, в которых возникают операционные сбои.

Операции агрегирования можно применять и к потокам, и к таблицам, но все они возвращают таблицу. Это связано с тем, что агрегатные функции применяются к *группе связанных записей*, а результат агрегатной функции (например, `COUNT`) должен сохраняться в некоторой изменяемой структуре, которую можно легко получить и обновить при появлении новых записей¹.

Существует две широкие категории агрегатных функций: оконные и неоконные. В этом проекте мы будем использовать оконное агрегирование для вычисления таблицы `season_length_change_counts` (см. шаг 4 на рис. 11.1). Но сначала рассмотрим основы агрегирования данных в следующем разделе.

¹ В случае операции `COUNT` значение данного ключа будет инициализироваться нулем (0), а затем увеличиваться на 1 всякий раз при поступлении новой записи с тем же ключом. Как упоминалось выше, эти виды изменяемой семантики требуют использования внутренней таблицы с состоянием, а не потока.

Основы агрегирования

Агрегирование данных в ksqlDB выполняется в два этапа¹:

- создать выражение `SELECT`, использующее некоторую агрегатную функцию;
- сгруппировать связанные записи с помощью оператора `GROUP BY`. Агрегатная функция будет применена к каждой группе.

Нужный нам запрос с агрегированием приводится в примере 11.4.

Пример 11.4. Пример использования агрегатной функции (`COUNT`) и оператора `GROUP BY`

```
SELECT
  title_id,
  COUNT(*) AS change_count, ❶
  LATEST_BY_OFFSET(new_episode_count) AS latest_episode_count ❷
FROM season_length_changes_enriched
GROUP BY title_id ❸
EMIT CHANGES ;
```

❶ Использовать встроенную агрегатную функцию `COUNT`, чтобы подсчитать количество записей для каждого значения `title_id`.

❷ Использовать встроенную агрегатную функцию `LATEST_BY_OFFSET`, чтобы получить количество последних выпусков для каждого `title_id`.

❸ Сгруппировать записи в потоке по столбцу `title_id`.

Запрос в примере 11.4 использует встроенную функцию `COUNT`, но вообще в ksqlDB имеется множество агрегатных функций, включая `AVG`, `COUNT_DISTINCT`, `MAX`, `MIN`, `SUM` и др. Агрегатные функции даже не всегда математические. Яркими примерами могут служить функции `EARLIEST_BY_OFFSET` и `LATEST_BY_OFFSET`, возвращающие определяемое по смещению самое старое и самое новое значение столбца соответственно. В конце этой главы вы увидите, как получить список всех доступных агрегатных функций в ksqlDB, и узнаете, как писать свои функции, если это понадобится. А пока просто помните, что в вашем распоряжении имеется множество различных встроенных функций.

Второе требование — группировка записей с помощью оператора `GROUP BY`. Он помещает записи в отдельные сегменты с учетом значений указанных

¹ Как будет показано в следующем разделе, существует необязательный третий этап, включающий определение временного окна, в рамках которого должно выполняться агрегирование.

столбцов. Например, запрос в примере 11.4 объединит все записи с одинаковым значением `title_id` в одну группу.

Наконец, выражение `SELECT` может также включать дополнительные столбцы. Например, добавим в запрос столбцы `title_id` и `season_id`:

```
SELECT
  title_id,
  season_id, ❶
  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 ❷
EMIT CHANGES ;
```

❶ Добавить в запрос новый столбец (`season_id`), не участвующий в агрегировании.

❷ Добавить столбец `season_id` в оператор `GROUP BY`.

Обратите внимание, что для включения в выражение `SELECT` столбца, не участвующего в агрегировании, его также необходимо включить в предложение `GROUP BY`. Например, предположим, что мы решили выполнить следующий запрос:

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

❶ Добавили `season_id` в выражение `SELECT` (то есть в проекцию)...

❷ ...Но не добавили его в оператор `GROUP BY`.

В этом случае возникнет ошибка:

```
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 SELECT1
```

В противовес сообщению об ошибке есть третий вариант решения проблемы: применить функцию агрегирования к столбцу, не включенному в оператор `GROUP BY`.

¹ Неагрегированные выражения в `SELECT`, не являющиеся частью `GROUP BY`: `SEASON_ID`
Добавьте столбец в `GROUP BY` или удалите его из `SELECT`. — *Примеч. пер.*



Каждое поле, включенное в оператор GROUP BY, становится частью ключа базовой таблицы. Например, GROUP BY title_id создаст таблицу с ключом для каждого уникального значения title_id. Если группировка выполняется по нескольким столбцам, например GROUP BY title_id, season_id, то базовая таблица получит составной ключ, в котором значения столбцов отделены друг от друга комбинацией символов |+| (например, 1|+|2).

Об этом важно помнить, выполняя pull-запросы к материализованной таблице, потому что, выполняя запрос к таблице с составными ключами, в него необходимо включить разделитель (например, SELECT * FROM T WHERE COL='1|+|2'). В будущих версиях ksqlDB способ использования составных ключей может усовершенствоваться, но задокументированное здесь поведение присутствовало по крайней мере в версии 0.14.0.

Все предыдущие примеры агрегирования представляли простое (не оконное) агрегирование, потому что записи группируются только по полям в предложении GROUP BY без использования отдельного условия, определяющего временное окно. Однако ksqlDB поддерживает и оконное агрегирование, которое мы рассмотрим в следующем разделе.

Оконное агрегирование

Иногда может понадобиться выполнить агрегирование за определенный период времени, например, чтобы узнать, сколько изменений в продолжительности сезона произошло за сутки. Это добавляет в агрегирование еще одно измерение: время. К счастью, ksqlDB поддерживает оконное агрегирование, разработанное для подобных вариантов использования.

В главе 5 мы обсудили три типа окон в Kafka Streams (загляните в подраздел «Типы окон» главы 5, чтобы освежить память). Те же типы окон доступны в ksqlDB. Все они, а также соответствующие выражения ksqlDB показаны в табл. 11.3.

Таблица 11.3. Типы окон в ksqlDB

Тип окна	Пример
Переворачивающиеся окна	WINDOW TUMBLING (SIZE 30 SECONDS)
Шагающие окна	WINDOW HOPPING (SIZE 30 SECONDS, ADVANCE BY 10 SECONDS)
Окна сеансов	WINDOW SESSION (60 SECONDS)

Чтобы включить определение окна в запрос, достаточно добавить выражение `WINDOW` перед оператором `GROUP BY`:

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

❶ Сгруппировать записи во временные интервалы длительностью один час.

❷ Сгруппировать записи по столбцам `title_id` и `season_id`.

Этот запрос вернет следующий результат:

TITLE_ID	SEASON_ID	CHANGE_COUNT	LATEST_EPISODE_COUNT
1	1	1	8

Несмотря на простоту использования оконных выражений в ksqlDB, дополнительное измерение времени требует учитывать дополнительные особенности, перечислим их.

- Когда передавать результаты нижестоящим узлам-обработчикам?
- Как обрабатывать данные, поступившие с задержкой/не по порядку?
- Как долго сохранять каждое окно?

Каждую из этих особенностей мы рассмотрим в следующих разделах.

Задержавшиеся данные

Говоря о задержке поступления данных в ksqlDB или Kafka Streams, мы имеем в виду не только задержку события согласно системным часам. Например, если событие произошло в 10:02, но было обработано приложением в 10:15, то его можно считать задержавшимся в традиционном смысле этого слова.

Однако, как рассказывалось в главе 5, Kafka Streams (и ksqlDB) поддерживает также внутренние часы, называемые *временем потока*, значение которого может только увеличиваться и передается в отметке времени потребляемых записей. Отметки времени можно извлечь из метаданных записи или из свойств `TIMESTAMP` и `TIMESTAMP_FORMAT` в самой записи, как было показано в примере 11.3.

Когда записи потребляются не в порядке отметок времени, любая запись, поступающая с отметкой времени меньше текущего времени потока, считается задержавшейся. В оконном агрегировании задержавшиеся записи требуют особого внимания, потому что вы можете допустить их в окно (и учитывать при вычислении агрегатных функций) или просто игнорировать по истечении определенного времени (с этого момента они считаются *опоздавшими* и не допускаются в окно). Допустимая задержка называется *льготным периодом* (grace period) и определяется в ksqlDB так:

```
WINDOW <тип_окна> ( ❶
    <свойства_окна>, ❷
    GRACE PERIOD <число> <единицы_измерения_времени> ❸
)
```

❶ Допустимые типы окна: HOPPING, TUMBLING или SESSION.

❷ Свойства для каждого типа окна перечислены в табл. 11.3.

❸ Определение льготного периода. Обратите внимание, что в определениях льготных периодов можно использовать те же единицы измерения времени, что и в определениях скользящих окон (см. раздел «Оконные соединения» выше в этой главе).

Например, чтобы выполнить оконное агрегирование, допускающее задержку до 10 минут, и игнорировать все записи, поступающие после этого времени, можно использовать следующий запрос:

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

❶ Сгруппировать записи во временные интервалы продолжительностью один час и разрешить задержку до 10 минут.

Указав льготный период 10 MINUTES, можно попробовать добавить новые записи в другом окне терминала (во время выполнения предыдущего запроса), чтобы наглядно увидеть, как это влияет на время потока и на допуск записей в открытое окно. В примере 11.5 показана инструкция для создания этих записей.

Пример 11.5. Добавление записей с различными отметками времени в другом окне терминала с целью наглядно показать влияние на время потока

```
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' ❶  
);
```

```
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' ❷  
);
```

```
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' ❸  
);
```

```
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' ❹  
);
```

```
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 10:59:00' ❺  
);
```

❶ Время потока устанавливается равным 10:00, и эта запись будет добавлена в окно (10:00–11:00).

❷ Время потока устанавливается равным 11:00, и эта запись будет добавлена в окно (11:00–12:00).

❸ Время потока не изменяется, потому что отметка времени этой записи находится до текущего времени потока. Запись будет добавлена в окно (10:00–11:00) благодаря действию льготного периода.

❷ Время потока устанавливается равным 11:10, и эта запись будет добавлена в окно (11:00–12:00).

❸ Эта запись не будет добавляться в окно, потому что опоздала на 11 минут. Другими словами, $\text{stream_time (11:10)} - \text{current_time (10:59)} = 11$ минут, что больше льготного периода, равного 10 минутам.

Если запись игнорируется из-за опоздания (например, последняя запись в примере 11.5), в журнале сервера ksqlDB должна появиться дополнительная полезная информация:

```
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
```

❶ Текущая отметка времени в этой записи: 2021-02-24 10:59:00.

❷ Окно, куда должна попасть эта запись, имеет диапазон от 2021-02-24 10:00:00 до 2021-02-24 11:00:00.

❸ Время истечения 2021-02-24 11:00:00. Вычисляется вычитанием льготного периода (10 минут) из времени передачи (см. следующий пункт).

❹ Время потока — отметка времени с самым большим значением, наблюдавшаяся в ksqlDB: 2021-02-24 11:10:00.

Льготный период можно не устанавливать (хотя настоятельно рекомендуется), все зависит от того, как вы решите обрабатывать неупорядоченные/задержавшиеся данные. Если не установить льготный период, окно будет оставаться открытым, пока не истечет срок его хранения и оно не закроется.

Срок хранения окна тоже настраивается, но играет другую роль в ksqlDB, ограничивая время, в течение которого можно запрашивать оконные данные. Как управлять хранением окон, будет показано в следующем разделе.

Хранение окна

Планируя запрашивать результаты оконного агрегирования, как это делаем мы, можно контролировать срок хранения старых окон в ksqlDB. После удаления окна вы не сможете запросить его. Еще одна причина настройки периода хране-

ния окон — ограничение объема хранилища состояний. Чем больше хранящихся окон, тем больший объем будут иметь хранилища состояний. Как рассказывалось в главе 6, ограничение объема хранилища состояний может уменьшить влияние механизма перебалансировки на производительность и объем потребляемых ресурсов приложения¹.

Чтобы ограничить срок хранения окна, достаточно указать свойство `RETENTION` в выражении `WINDOW`:

```
WINDOW { HOPPING | TUMBLING | SESSION } ( ❶
    <свойства_окна>, ❶
    RETENTION <число> <единицы_измерения_времени> ❷
)
```

❶ Свойства для каждого типа окна перечислены в табл. 11.3.

❷ Определение периода хранения окна. Обратите внимание, что в определениях периодов хранения окон можно использовать те же единицы измерения времени, что и в определениях скользящих окон (см. раздел «Оконные соединения» выше в этой главе).



Период хранения окна должен быть больше или равен его размеру плюс льготный период. Кроме того, период хранения — это нижняя граница срока поддержки окон. Скорее всего, окно не будет удалено точно в момент истечения срока хранения.

Например, давайте реорганизуем наш запрос так, чтобы установить срок хранения равным двум дням. Инструкция SQL в примере 11.6 показывает, как это сделать.

Пример 11.6. Инструкция, явно устанавливающая период хранения окна

```
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 (
```

¹ По умолчанию в роли хранилища состояний используется встроенное хранилище ключей и значений RocksDB, сохраняющее данные в памяти. Оно при необходимости может копироваться на диск, если пространство ключей окажется слишком большим. Поэтому при обслуживании больших хранилищ состояний следует учитывать потребление памяти и дискового пространства.

```
    SIZE 1 HOUR,  
    RETENTION 2 DAYS, ❶  
    GRACE PERIOD 10 MINUTES  
)  
GROUP BY title_id, season_id  
EMIT CHANGES ;
```

❶ Устанавливает период хранения окна равным двум дням.

Важно отметить, что период хранения управляется внутренними часами Kafka Streams и ksqlDB — временем потока. Он не связан с системными часами.

Пример 11.6 подвел нас практически к самому завершению этапа 4 в нашем проекте (см. рис. 11.1). Нам осталось лишь создать из этого запроса материализованное представление. Как это сделать, рассказывается в следующем разделе.

Материализованные представления

Материализованные представления давно существуют в мире баз данных и используются для хранения результатов запроса (это называется *материализацией*). Сохраненные результаты запроса затем становятся доступными для запрашивающей стороны и способствуют повышению производительности ресурсоемких запросов в традиционных базах данных. Такие запросы работают со многими записями сразу в пакетном режиме.

ksqlDB тоже поддерживает идею материализованных представлений, обладающих некоторыми свойствами своих традиционных аналогов:

- получают в результате выполнения запроса к другой коллекции;
- их можно получить, выполняя поисковые запросы (в ksqlDB они называются pull-запросами).

Однако материализованные представления в ksqlDB имеют и несколько важных отличий:

- на момент написания этих строк материализованные представления в ksqlDB поддерживались только для агрегированных запросов;
- автоматическое обновление по мере поступления новых данных; сравните эту черту с традиционными системами, где обновления могут планироваться, выполняться по требованию или вообще отсутствовать.

Говоря о материализованных представлениях в ksqlDB, мы имеем в виду определенный тип таблиц, к которым можно выполнять pull-запросы. Как вы уже

знаете, таблицы можно создавать непосредственно поверх тем Kafka (см. подраздел «Создание исходных коллекций» в главе 10) или неагрегированных запросов (см. пример 11.2). На момент написания этих строк в ksqlDB не поддерживался поиск по ключу (pull-запросы) для таких таблиц (хотя это ограничение может исчезнуть в будущих версиях ksqlDB).

Однако мы можем выполнять pull-запросы к материализованным представлениям — объектам **TABLE**, созданным на основе агрегированного запроса¹. Для этого нужно лишь создать производную коллекцию-таблицу (см. подраздел «Создание производных коллекций» в главе 10), используя агрегированный запрос.

Давайте для примера создадим материализованное представление из оконного агрегированного запроса в примере 11.6. В примере 11.7 приводится инструкция SQL, показывающая, как это сделать.

Пример 11.7. Создание материализованного представления на основе агрегированного запроса

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

Теперь у нас есть материализованное представление с именем `season_length_change_counts`. На этом завершается этап 4 нашего проекта (см. рис. 11.1), и мы готовы перейти к последнему этапу: выполнению запросов с использованием различных клиентов.

¹ Термин «представление» был заимствован из традиционных систем, но в ksqlDB нет отдельного объекта «представления». Поэтому, услышав слово «представление» в связи с ksqlDB, понимайте под ним набор таблиц, который можно использовать в pull-запросах.

Клиенты

Последний этап в нашем проекте — убедиться в доступности для различных клиентов данных, обработанных с помощью ksqlDB. Рассмотрим процесс выполнения pull-запросов к нашему материализованному представлению (`season_length_change_counts`) и push-запросов к обогащенному потоку (`season_length_changes_enriched`). Мы опробуем оба типа запросов, используя CLI и `curl`.

Утилита `curl` выбрана нами не в расчете на то, что вы будете создавать клиенты командной строки, использующие ее, а потому, что она послужит хорошей демонстрацией возможностей для реализации RESTful-клиента на выбранном вами языке. Не последнее влияние на выбор оказала простота синтаксиса `curl`, который позволяет передать в запросе тип HTTP-метода, заголовки HTTP-клиента и полезную нагрузку. Кроме того, такой подход позволяет проанализировать необработанный ответ, возвращаемый `curl`, и получить четкое понимание формата ответа, возвращаемого ksqlDB.



На момент написания этих строк вышла начальная версия клиента на Java. В примерах для этой главы вы найдете исходный код, демонстрирующий применение разработанного клиента. Мы не привели его здесь из-за неуверенности в том, как может измениться интерфейс клиента на Java в ближайшем будущем. Кроме того, работая над этой главой, мы обсуждали предложение по улучшению ksqlDB (ksqlDB Improvement Proposal, KLIP; <https://oreil.ly/TOurV>), в котором упоминались потенциальные изменения в клиенте на Java, ksqlDB REST API и даже возможное введение новых официальных клиентов на Python или Go.

Сначала рассмотрим pull-запросы.

Pull-запросы

Имея материализованное представление, можно выполнять pull-запросы к нему. Вот как выглядит синтаксис pull-запроса:

```
SELECT выражение_выбора [, ...]
FROM объект_откуда_производится_выборка
WHERE условие
```

Как видите, pull-запросы очень просты. На момент написания этих слов pull-запросы не поддерживали ни соединений, ни агрегирования (хотя обе эти операции можно выполнить при создании материализованного представления, как

показано в примере 11.7). Поэтому pull-запросы можно рассматривать как простые поисковые запросы, ссылающиеся на ключевой столбец, а в случае оконных представлений они могут также ссылаться на псевдостолбец `WINDOWSTART`, содержащий нижнюю границу временного диапазона окна (верхняя граница хранится в другом псевдостолбце с именем `WINDOWEND`, но он недоступен для запросов).

Как называется ключевой столбец, по которому можно выполнить поиск? В разных случаях по-разному. Если в операторе `GROUP BY` указан один столбец, то имя ключевого столбца будет совпадать с его именем. Например, предположим, что запрос, на основе которого создается материализованное представление, включает следующее выражение:

```
GROUP BY title_id
```

Мы могли бы реализовать pull-запрос, выполняющий поиск по столбцу `title_id`. Однако в нашем случае в операторе `GROUP BY` указаны два столбца:

```
GROUP BY title_id, season_id
```

В такой ситуации ksqlDB сгенерирует ключевой столбец `KSQL_COL_?` (это еще один артефакт текущей реализации ksqlDB, который может измениться в будущем). Получить имя столбца можно из вывода `DESCRIBE`:

```
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

В этом случае выполнить поиск в представлении `season_length_change_counts` можно с помощью pull-запроса из примера 11.8.

Пример 11.8. Пример pull-запроса

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

❶ При группировке по нескольким полям ключ содержит составное значение, в котором значения исходных столбцов разделены последовательностью символов `|+|`. Если бы группировка выполнялась только по одному полю (например, `title_id`), то можно было бы использовать выражение `WHERE title_id=1`.



В pull-запросах также можно использовать предикат **IN** для сопоставления с несколькими возможными ключами (например, `WHERE KSQL_COL_0 IN ('1|+|1', '1|+|2')`).

Если выполнить предыдущий запрос в интерфейсе командной строки, то он выведет:

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

Как видите, в вывод включены два псевдостолбца, `WINDOWSTART` и `WINDOWEND`. В условии `WHERE` также можно использовать столбец `WINDOWSTART`, сравнивая его значение с отметкой времени в стиле Unix или с более удобочитаемой строкой даты и времени. Ниже показаны оба типа таких запросов:

```
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';
```

Эти два запроса выведут одни и те же данные:

KSQL_COL_0	WINDOWSTART	WINDOWEND	CHANGE_COUNT	EPISODE_COUNT	
1 + 1	1614160800000	1614164400000	2	8	

Когда придет время запрашивать данные из материализованных представлений, то, вполне вероятно, вы решите делать это с помощью клиента, отличного от клиента командной строки. Посмотрим, как выполнить pull-запрос с помощью `curl`, потому что пример с `curl` легко переложить на язык Python, Go или любой другой.

curl

Самый простой, пожалуй, способ выполнить запрос к серверу `ksqlDB` без использования клиента командной строки — использовать широко известную

утилиту `curl`. Следующая команда выполняет тот же pull-запрос, который был показан в примере 11.8:

```
curl -X POST "http://localhost:8088/query" \ ❶
  -H "Content-Type: application/vnd.ksql.v1+json; charset=utf-8" \ ❷
  --data '${ ❸
    "ksql": "SELECT * FROM season_length_change_counts WHERE KSQL_
COL_0='\`1|+|1\`";",
    "streamsProperties": {}
  }'
```

❶ Чтобы выполнить pull-запрос, нужно отправить HTTP POST-запрос в конечную точку `/query`.

❷ Значение заголовка `Content-Type` включает версию API `v1` и формат сериализации (`json`).

❸ Pull-запрос передается в теле HTTP POST-запроса, как значение ключа `ksql` в объекте JSON.

Ниже показан вывод предыдущей команды. Здесь можно видеть имена столбцов (`header.schema`) и одну или несколько строк:

```
[
  {
    "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
]
```

❶ Остальные строки опущены для краткости.

Как видите, послать запрос серверу `ksqlDB` без использования клиента командной строки очень просто.

Теперь перейдем к push-запросам.

Push-запросы

Мы уже видели множество примеров push-запросов. Например, мы легко можем выполнить запрос к потоку `season_length_change_counts` из клиента командной строки, введя следующую инструкцию SQL:

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

Однако основное внимание в этом разделе будет уделено выполнению push-запросов из `curl`.

Push-запросы из curl

Push-запросы выполняются с помощью `curl` точно так же, как pull-запросы. Вот простой пример выполнения push-запроса:

```
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": {}
  }'
```

Ниже показан полученный результат:

```
[{"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"]}},
]
```

Если в ответ ничего не появилось на экране, то, скорее всего, причина в том, что смещение потребителя оказалось установлено на самую последнюю запись, поэтому может потребоваться повторно выполнить инструкции из примера 11.5.

В отличие от pull-запросов, сразу после выполнения push-запроса соединение не разрывается. Выходные данные продолжают передаваться клиенту порциями по долгоживущему соединению по мере появления новых данных. Эта возможность поддерживается благодаря способности протокола HTTP передавать данные по частям.

Теперь, узнав, как запрашивать обогащенные и агрегированные данные, мы завершили последний этап нашего проекта мониторинга изменений в Netflix. Но, прежде чем завершить эту главу, рассмотрим еще одну тему: функции и операторы.

Функции и операторы

ksqlDB поддерживает богатый набор функций и операторов для работы с данными. Для начала кратко рассмотрим некоторые операторы.

Операторы

ksqlDB поддерживает несколько операторов, которые можно включить в инструкции SQL:

- арифметические операторы (+, -, /, *, %);
- операторы конкатенации строк (+, ||);
- операторы индексирования для доступа к элементам массива или ключам карты ([]);
- операторы разыменования структур (->).

Мы уже встречали некоторые из них (см. пример 8.1). Но давайте не будем тратить много времени на рассмотрение операторов в этой книге; их полный список, описания и примеры использования можно найти в официальной документации ksqlDB (<https://docs.ksqldb.io/>). Обратимся к более интересной части нашего обсуждения — функциям.

Вывод списка доступных функций

Одной из самых интересных особенностей ksqlDB является библиотека встроенных функций. Мы уже видели одну такую функцию `COUNT`, но их гораздо больше, и библиотека функций постоянно растет. Получить список доступных функций можно с помощью инструкции `SHOW FUNCTIONS`.

Например:

```
ksql> SHOW FUNCTIONS ;
```

Инструкция `SHOW FUNCTIONS` выведет довольно длинный список, ниже показана только его часть:

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
...	

Видно, что функции подразделяются на три типа (см. столбец `TYPE`). В следующей таблице приводятся описания всех этих типов.

Тип функций	Описание
SCALAR	Функции без состояния, оперирующие одной записью и возвращающие одно значение
AGGREGATE	Функции с состоянием, осуществляющие агрегирование данных. Эти функции тоже возвращают одно значение
TABLE	Функции без состояния, принимающие одно входное значение и возвращающие 0 или более выходных значений. Своим характером напоминают функцию <code>flatMap</code> в <code>Kafka Streams</code>

При просмотре библиотеки функций вам может понадобиться получить дополнительную информацию о конкретной функции. Как это сделать, мы покажем в следующем разделе.

Получение описаний функций

Получить дополнительную информацию о конкретной функции всегда можно в официальной документации на веб-сайте `ksqlDB` (<https://ksqldb.io/>). Однако краткое описание функции доступно прямо в интерфейсе командной строки,

потому что ksqlDB предлагает для этого специальную инструкцию. Она имеет следующий синтаксис:

```
DESCRIBE FUNCTION <идентификатор>
```

Для демонстрации попробуем получить дополнительную информацию о встроенной функции `EARLIEST_BY_OFFSET`, как показано в примере 11.9.

Пример 11.9. Пример получения описания функции в ksqlDB

```
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

...
```

❶ Описание функции.

❷ Тип функции (список типов доступных функций приводится в табл. 11.3).

❸ В описаниях встроенных функций свойство `Jar` будет иметь значение `internal` (внутренняя). Как создавать свои функции, мы покажем в следующем разделе. Для них в свойстве `Jar` будет отображаться путь к фактическому файлу JAR на диске.

❹ Раздел **Variations** содержит список всех допустимых сигнатур для функции. Этот раздел был усечен для краткости, но вообще для этой функции вы увидите не менее двух вариантов: один вариант принимает аргумент `BOOLEAN` и возвращает значение `BOOLEAN`, а другой принимает и возвращает `INT`.

ksqlDB имеет внушительное количество встроенных функций, но также дает разработчику возможность реализовать свои в случае необходимости. Так что, если после просмотра списка, возвращаемого инструкцией `SHOW FUNCTIONS`, вы не найдете ничего подходящего, не расстраивайтесь. В следующем разделе мы расскажем, как создать свою функцию.

Создание своих функций

Иногда возникает необходимость создать нестандартную функцию для использования в своих запросах, например, чтобы применить к столбцу специализированную математическую функцию или вызвать модель машинного обучения и передать ей входные данные из потока. Независимо от сложности варианта использования ksqlDB поддерживает интерфейс Java и дает возможность дополнить встроенную библиотеку набором пользовательских функций.

В ksqlDB поддерживается три типа пользовательских функций, соответствующих типам встроенных функций, обсуждавшимся в предыдущем разделе (скалярные, агрегатные и табличные функции). Список типов пользовательских функций приводится в следующей таблице.

Тип	Описание
Определяемые пользователем функции (User-Defined Functions, UDF)	Пользовательские скалярные функции, соответствующие типу SCALAR. Не имеют состояния и возвращают точно одно значение
Определяемые пользователем агрегатные функции (User-Defined Aggregate Functions, UDAF)	Пользовательские агрегатные функции, соответствующие типу AGGREGATE. Имеют состояние и возвращают точно одно значение
Определяемые пользователем табличные функции (User-Defined Table Functions, UDTF)	Пользовательские табличные функции, соответствующие типу TABLE. Не имеют состояния и возвращают ноль или более значений

Для лучшего понимания мы реализуем пользовательскую функцию UDF, удаляющую стоп-слова из текстовой строки. В исходном коде для этой главы вы найдете также примеры UDAF и UDTF, но в тексте они не приводятся, потому что процесс разработки и развертывания пользовательских функций практически не зависит от их типа. Конечно, существуют некоторые тонкие различия в реализации UDF, UDAF и UDTF, и мы обозначили их в исходном коде для этой главы.

Теперь посмотрим, как создать пользовательскую функцию UDF.

Функция UDF для удаления стоп-слов

Удаление так называемых стоп-слов из текстовой строки — распространенная задача предварительной обработки данных. Стоп-слова — это общеупотребительные слова (такие как «а», «и», «да», «или», «тот» и т. д.), не добавляющие большого значения к основному тексту. При работе с моделями машинного обучения или обработки естественного языка, извлекающими информацию из

текста, сначала из входных данных обычно удаляют стоп-слова. Поэтому мы создадим UDF с именем `REMOVE_STOP_WORDS`. Процесс создания пользовательской функции любого типа прост и состоит из следующих шагов.

1. Создать проект Java, который будет содержать код функции.
2. Добавить в проект зависимость `io.confluent.ksql:ksql-udf`, содержащую аннотации для нашего класса.
3. Добавить любые другие зависимости, необходимые для реализации логики функции. Например, если есть сторонняя зависимость Maven, которая будет использоваться в коде, то добавьте ее в файл сборки.
4. Написать логику функции, используя соответствующие аннотации (мы вскоре рассмотрим их).
5. Создать и упаковать код в архив uber-JAR, объединяющий исходный код функции и все его зависимости в один файл JAR.
6. Скопировать файл uber-JAR в каталог расширений ksqlDB. Путь к этому каталогу настраивается на сервере ksqlDB и определяется в свойстве `ksql.extension.dir`.
7. Перезапустить сервер ksqlDB. Когда сервер ksqlDB вернется в рабочий режим, он загрузит новую функцию, сделав ее доступной для использования в инструкциях SQL.

Теперь реализуем нашу функцию, используя это пошаговое руководство, чтобы понять, как это работает. Сначала создадим новый проект Java командой `init` Gradle:

```
mkdir udf && cd udf

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

Добавим зависимость `ksql-udf` в файл сборки (`build.gradle`). Эта зависимость находится в репозитории Confluent Maven, поэтому дополним также блок `repositories`:

```
repositories {
    // ...
    maven {
        url = uri('http://packages.confluent.io/maven/') ❶
    }
}
```

```
dependencies {
    // ...
    implementation 'io.confluent.ksql:ksqldb-udf:6.0.0' ❷
}
```

❶ Добавление репозитория Confluent Maven, где находится зависимость `ksql-udf`.

❷ Добавление зависимости `ksql-udf` в проект. Этот артефакт содержит все аннотации, которые мы должны будем добавить в наш код.

Наша функция не имеет других зависимостей, но если бы они были, мы могли бы добавить в блок `dependencies` любые другие артефакты, необходимые нашей UDF¹. Теперь пришло время реализовать логику функции. Для этого создадим файл `RemoveStopWordsUdf.java` и добавим в него следующий код:

```
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) { ❶
        return Stream.of(source.toLowerCase().split(" "))
            .collect(Collectors.toCollection(ArrayList<String>::new));
    }

    private String wordsToString(ArrayList<String> words) { ❷
        return words.stream().collect(Collectors.joining(" "));
    }

    public String apply(final String source) { ❸
        ArrayList<String> words = stringToWords(source);
        words.removeAll(stopWords);
        return wordsToString(words);
    }
}
```

❶ Этот метод преобразует текстовую строку в список слов.

❷ Этот метод преобразует список слов обратно в строку.

❸ Этот метод содержит логику нашей UDF, которая удаляет из исходной строки стоп-слова, перечисленные в списке `stopWords`. Методу можно дать любое имя, единственное требование — он должен быть нестатическим и иметь общедоступную область видимости `public`.

Покончив с логикой, добавим аннотации из зависимости `ksql-udf`. Они выявляются сервером `ksqlDB` при загрузке JAR и предоставляют дополнительную

¹ Зависимости нужно будет упаковать в `uber-JAR` с помощью плагина, такого как `com.github.johnrengelman.shadow`. Дополнительную информацию ищите в документации по UDF (<https://oreil.ly/1g847>).

информацию о функции, включая ее имя, описание и параметры. Как вы вскоре увидите, значения, указанные в аннотациях, можно просмотреть в выводе `DESCRIBE FUNCTION`:

```
@UdfDescription( ❶
    name = "remove_stop_words", ❷
    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") ❸
    public String apply(
        @UdfParameter(value = "source", description = "the raw source string") ❹
        final String source
    ) { ... }
}
```

❶ Аннотация `UdfDescription` сообщает серверу `ksqlDB`, что этот класс содержит функцию, которую следует загрузить в библиотеку `ksqlDB` при запуске. При реализации UDAF или UDTF вместо этой аннотации нужно использовать `UdafDescription/UdtfDescription`.

❷ Имя нашей функции, под которым она будет доступна в библиотеке. Остальные свойства (`description`, `version` и `author`) будут отображаться в выводе `DESCRIBE FUNCTION`.

❸ Аннотацию `Udf` следует применять к общедоступному методу, который будет вызываться сервером `ksqlDB`. В классе пользовательской функции UDF может быть несколько методов с аннотацией `Udf`, и каждый из них будет считаться самостоятельным вариантом функции, как обсуждалось, когда мы знакомились с функцией `EARLIEST_BY_OFFSET` (см. пример 11.9). Обратите внимание, что для UDTF и UDAF используются разные аннотации (`Udtf` и `UdafFactory`). Подробности смотрите в исходном коде примеров для этой главы (<https://oreil.ly/rJN5s>).

❹ `UdfParameter` можно использовать для описания каждого параметра в пользовательской функции UDF. Эти описания тоже отображаются в выводе `DESCRIBE FUNCTION` в разделе `Variations`.

После добавления аннотаций к классу и методам можно создать архив uber-JAR. В Gradle это делается следующей командой:

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

Эта команда создаст архив JAR в следующем пути относительно корневого каталога проекта Java:

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

Чтобы сообщить серверу ksqlDB о новой пользовательской функции, нужно поместить этот архив JAR в каталог, определяемый конфигурационным свойством `ksql.extension.dir`. Это свойство можно определить в файле конфигурации сервера ksqlDB. Например:

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

Определив и создав каталог для расширений, можно скопировать в него архив JAR и перезапустить сервер ksqlDB:

```
cp build/libs/udf.jar /etc/ksqldb/extensions
```

```
ksql-server-stop
ksql-server-start
```

После перезапуска функция UDF должна появиться в выводе инструкции `SHOW FUNCTIONS`:

```
ksql> SHOW FUNCTIONS ;

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

❶ Имя функции, как определено аннотацией `UdfDescription`, появится в списке функций, упорядоченном по алфавиту.

Можно получить описание нашей функции, используя ту же инструкцию SQL, что и для получения описания встроенных функций:

```
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
```

❶ Обратите внимание, что свойство `Jar` теперь ссылается на физическое местоположение архива JAR с функцией UDF на диске.

Теперь функцию можно использовать точно так же, как и любую встроенную функцию ksqlDB. Давайте создадим поток с именем `model_inputs` и добавим в него некоторые тестовые данные, на которых опробуем нашу функцию:

```
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');
```

Теперь применим функцию:

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

Как показывает следующий вывод, наша функция выполнила ожидаемое действие:

ORIGINAL	NO_STOP_WORDS
The quick brown fox jumps over the lazy dog	quick brown fox jumps lazy dog

Дополнительная информация о пользовательских функциях ksqlDB

Создание пользовательских функций для ksqlDB — обширная тема, и мы легко могли бы посвятить ей несколько глав. Однако я уже много говорил и даже писал о функциях ksqlDB в других местах, да и официальная документация содержит массу полезной информации по этой теме. Желающие могут ознакомиться со следующими ресурсами, чтобы получить дополнительную информацию о пользовательских функциях ksqlDB:

- *The Exciting Frontier of Custom KSQL Functions* (<https://oreil.ly/HTb-F>; Митч Сеймур (Mitch Seymour), Kafka Summit 2019);
- *ksqlDB UDFs and UDAFs Made Easy* (<https://oreil.ly/HMU9F>; Митч Сеймур, блог Confluent);
- Официальная документация ksqlDB (<https://oreil.ly/qEafS>).

Заключение

Несмотря на простой интерфейс, ksqlDB поддерживает множество вариантов потоковой обработки от простых до очень сложных, включая соединение данных в разные коллекции, агрегирование, создание материализованных представлений, которые можно запрашивать с помощью поиска по ключу, и многое другое. Кроме того, ksqlDB предоставляет обширную библиотеку встроенных функций, которые можно использовать для решения широкого круга задач обработки и обогащения данных. В библиотеке доступны общие математические функции (AVG, COUNT_DISTINCT, MAX, MIN, SUM и т. д.), строковые функции (LPAD, REGEXP_EXTRACT, REPLACE, TRIM, UCASE и т. д.) и даже геопространственные (GEO_DISTANCE), с помощью которых ksqlDB может удовлетворить самые разнообразные потребности.

Но особенно ярко ksqlDB блистает своим высокоуровневым интерфейсом, предоставляя разработчикам возможность расширять встроенную библиотеку своими функциями на Java. Это невероятно важно, так как позволяет создавать приложения потоковой обработки с использованием простого диалекта SQL, даже если приложение требует нестандартной бизнес-логики. Это несколько усложняет использование ksqlDB, поскольку требует знания Java, но, как показано в этой главе, ksqlDB упрощает даже этот процесс.

Теперь, когда вы знаете, как решать простые и сложные задачи потоковой обработки с помощью ksqlDB, перейдем к последней главе этой книги, где вы узнаете, как тестировать, развертывать и осуществлять мониторинг приложений Kafka Streams и ksqlDB.

Часть IV

Путь к промышленной эксплуатации

Тестирование, мониторинг и развертывание

В предыдущих главах вы узнали, как создавать различные приложения потоковой обработки с помощью Kafka Streams и ksqlDB. В этой последней главе я расскажу о некоторых дополнительных шагах, которые необходимо предпринять, чтобы передать приложение в промышленное окружение. Возможно, вам интересно, почему я объединил обсуждение вопросов развертывания приложений Kafka Streams и ksqlDB в промышленном окружении в одну главу. Дело в том, что, несмотря на некоторые различия, особенно в отношении тестирования, процесс во многом одинаков, и упрощение ментальной модели поможет вам наладить сопровождение ваших приложений в будущем (особенно в гибридных окружениях, где используются и Kafka Streams, и ksqlDB).

Вот список некоторых тем, рассматриваемых в этой главе.

- Как тестировать приложения Kafka Streams и запросы ksqlDB.
- Как тестировать топологии Kafka Streams.
- Какие виды мониторинга следует использовать.
- Как организовать доступ к встроенным метрикам JMX в Kafka Streams и ksqlDB.
- Как помещать в контейнеры и развертывать приложения Kafka Streams и ksqlDB.
- С какими задачами, скорее всего, придется столкнуться в период эксплуатации.

Начнем с изучения особенностей тестирования приложения потоковой обработки.

Тестирование

В большинстве проектов после завершения первоначальной разработки запросов ksqlDB или приложения Kafka Streams продолжается выпуск обновлений. Например, изменение бизнес-требований, обнаруженная ошибка или проблема с производительностью могут потребовать обновить код программного обеспечения.

Однако каждый раз, когда вносятся изменения, необходимо приложить все силы, чтобы случайно не внести ошибки, способные повлиять на правильность или производительность приложения. Помочь в этом может применение передовых методов тестирования. В этом разделе мы обсудим несколько стратегий тестирования, которые помогут вам обеспечить бесперебойное развитие приложений потоковой обработки.

Тестирование запросов ksqlDB

Тестировать запросы ksqlDB очень просто. Разработчики ksqlDB создали инструмент под названием `ksql-test-runner`, принимающий три аргумента:

- файл с одним или несколькими инструкциями SQL, которые нужно протестировать;
- файл с входными данными для одной или нескольких тем-источников;
- файл с ожидаемыми результатами для одной или нескольких тем-приемников.

Давайте посмотрим на примере, как это работает. Сначала создадим набор инструкций SQL, читающих данные из темы `users` и преобразующих каждую запись в приветствие. Инструкции, которые будут тестироваться, показаны в следующем листинге, который мы сохраним в файле `statement.sql`:

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

```
CREATE STREAM greetings  
WITH (KAFKA_TOPIC = 'greetings') AS  
SELECT ROWKEY, 'Hello, ' + USERNAME AS "greeting"  
FROM users  
EMIT CHANGES;
```

Следующий шаг после создания запросов — определение входных данных для тестов. В этом примере используется единственная тема-источник с именем `users`. Поэтому сохраним следующий код в файл с именем `input.json`, требующий от `ksql-test-runner` добавить две записи в тему `users` при запуске теста:

```
{
  "inputs": [
    {
      "topic": "users",
      "timestamp": 0,
      "value": {"USERNAME": "Isabelle"},
      "key": 0
    },
    {
      "topic": "users",
      "timestamp": 0,
      "value": {"USERNAME": "Elyse"},
      "key": 0
    }
  ]
}
```

Наконец, опишем ожидаемые результаты запросов. Поскольку наши запросы записывают результаты в тему с именем `greetings`, мы определим, что должно содержаться в этой выходной теме (после того как запросы обработают тестовые данные), сохранив следующие строки в файл с именем `output.json`:

```
{
  "outputs": [
    {
      "topic": "greetings",
      "timestamp": 0,
      "value": {
        "greeting": "Hello, Isabelle"
      },
      "key": 0
    },
    {
      "topic": "greetings",
      "timestamp": 0,
      "value": {
        "greeting": "Hello, Elyse"
      },
      "key": 0
    }
  ]
}
```

Теперь, имея три необходимых файла, можно запустить тестирование следующей командой:

```
docker run \
  -v "$(pwd)":/ksqldb/ \
  -w /ksqldb \
  -ti confluentinc/ksqldb-server:0.14.0 \
  ksql-test-runner -s statements.sql -i input.json -o output.json
```

На момент написания этих строк инструмент тестирования производил довольно подробный вывод, но где-то в выводе вы должны увидеть следующий текст:

```
>>> Test passed!
```

Если в будущем вы случайно внесете критическое изменение, тест потерпит неудачу. Например, изменим текст генерируемого приветствия. Вместо простого Hello (Привет) пожелаем пользователю Good morning (Доброе утро):

```
CREATE STREAM greetings
WITH (KAFKA_TOPIC = 'greetings') AS
SELECT ROWKEY, 'Good morning, ' + USERNAME AS "greeting"
FROM users
EMIT CHANGES;
```

Если теперь вновь запустить тестирование, то появится сообщение об ошибке. Это вполне объяснимо, потому что мы изменили запрос, но не изменили ожидаемый результат в `output.json`:

```
>>>>> Test failed: Topic 'greetings', message 0:
  Expected <0, {"greeting":"Hello, Isabelle"}> with timestamp=0
  but was <0, {greeting=Good morning, Isabelle}> with timestamp=0
```

Пример довольно тривиальный, но важна сама идея. Тестирование запросов ksqlDB таким способом помогает предотвратить случайные регрессии и настоятельно рекомендуется перед отправкой изменений в рабочее окружение.



При разработке пользовательских функций ksqlDB (UDF, UDAF и/или UDTF) желательно также организовать модульное тестирование кода на Java. Этот процесс очень похож на стратегию модульного тестирования, которую мы рассмотрим в примере 12.1, а за дополнительной информацией я предлагаю обращаться к моей статье *KsqlDB UDFs and UDAFs Made Easy* в блоге Confluent (<https://oreil.ly/PUmD7>).

Теперь посмотрим, как тестировать приложения Kafka Streams. Это сложнее, но ненамного.

Тестирование приложений Kafka Streams

Для тестирования приложений Kafka Streams вам понадобится фреймворк автоматизированного тестирования. Выбор фреймворка в конечном счете зависит от вас, но в примерах в этом разделе будут использоваться: JUnit для запуска тестов и AssertJ для улучшения читаемости утверждений.

Помимо фреймворка тестирования, к проекту также желательно подключить библиотеку `kafka-streams-test-utils`, которая входит в состав официального проекта Kafka. Эта библиотека включает:

- среду времени выполнения для запуска топологий Kafka Streams;
- вспомогательные методы для чтения и записи данных в тестовых темах Kafka;
- фиктивные объекты (имитации), которые можно использовать для модульного тестирования узлов-обработчиков и преобразований.

Как обычно, чтобы добавить в проект Kafka Streams сторонние пакеты, достаточно отредактировать файл сборки (`build.gradle`), вписав в него соответствующие зависимости. Поэтому, чтобы добавить вышеупомянутые элементы в проект Kafka Streams (среда тестирования и вспомогательные библиотеки), изменим файл сборки, как показано ниже:

```
dependencies {
    testImplementation "org.apache.kafka:kafka-streams-test-utils:${kafkaVersion}"
    testImplementation 'org.assertj:assertj-core:3.15.0'
    testImplementation 'org.junit.jupiter:junit-jupiter:5.6.2'
}

test {
    useJUnitPlatform()
}
```

После добавления зависимостей в проект можно начинать писать тесты. Существует множество способов тестирования приложений Kafka Streams, поэтому мы разобьем этот раздел на подразделы, в каждом из которых рассмотрим отдельную стратегию тестирования. Если явно не указано иное, то будет предполагаться, что создаваемые тесты сохраняются в каталоге `src/test/java`¹ в папке проекта и могут быть запущены командой:

```
./gradlew test --info
```

¹ Например, тест топологии можно определить в файле с именем `src/test/java/com/magicalpipelines/GreeterTopologyTest.java`.

Модульные тесты

Модульное тестирование предполагает тестирование отдельных фрагментов кода — модулей. В отношении топологий Kafka Streams такими *модулями* чаще всего являются отдельные узлы-обработчики, составляющие топологию. Поскольку обработчики могут определяться по-разному — с использованием DSL или Processor API, — тесты будут отличаться применяемым API. Для начала рассмотрим модульное тестирование узлов-обработчиков, написанных на DSL.

DSL. При использовании DSL обычной практикой является передача лямбда-выражения одному из встроенных операторов Kafka Streams. Например, в следующей топологии логика оператора `selectKey` определяется с помощью встроенного лямбда-выражения:

```
public class MyTopology {
    public Topology build() {
        StreamsBuilder builder = new StreamsBuilder();
        builder
            .stream("events", Consumed.with(Serdes.String(), Serdes.ByteArray()))
            .selectKey(
                (key, value) -> { ❶
                    // ... ❷
                    return newKey;
                })
            .to("events-repartitioned");

        return builder.build();
    }
}
```

❶ Логика `selectKey` определена внутри лямбда-выражения.

❷ Сама логика опущена для краткости.

Простую и лаконичную логику внутри лямбда-выражения легко понять. Но если реализация логики длинная и сложная, то для простоты тестирования ее лучше выделить в отдельный метод. Например, предположим, что логика для нашей операции `selectKey` определяется большим количеством строк кода и хотелось бы протестировать ее отдельно от общей логики топологии. В этом случае можно заменить лямбда-выражение ссылкой на метод, как показано ниже:

```
public class MyTopology {
    public Topology build() {
        StreamsBuilder builder = new StreamsBuilder();
        builder
            .stream("events", Consumed.with(Serdes.String(), Serdes.ByteArray()))
```

```

        .selectKey(MyTopology::decodeKey) ❶
        .to("events-repartitioned");

    return builder.build();
}

public static String decodeKey(String key, byte[] payload) {
    // ... ❷
    return newKey;
}
}

```

❶ Взамен лямбда-выражения используется ссылка на метод.

❷ Логика, изначально реализованная в лямбда-выражении, перемещена в отдельный метод. Это значительно упрощает тестирование. Здесь также фактическая логика была опущена для краткости.

После переноса логики в отдельный метод ее стало намного проще тестировать. Для этого даже не нужен пакет `kafka-streams-test-utils`, потому что для проверки этого метода можно просто использовать средства фреймворка модульного тестирования, как показано в примере 12.1.

Пример 12.1. Простой модульный тест для оператора `selectKey`, основанного на отдельном методе `MyTopology.decodeKey`

```

class MyTopologyTest {
    @Test
    public void testDecodeId() {
        String key = "1XRZTUW3";
        byte[] value = new byte[] {};
        String actualValue = MyTopology.decodeKey(key, value); ❶
        String expectedValue = "decoded-1XRZTUW3"; ❷
        assertEquals(actualValue, expectedValue); ❸
    }
}

```

❶ Тест вызывает тот же метод, что используется в обработчике `selectKey`. Здесь ему передаются предопределенные ключ и значение. Также можно было бы использовать параметризованные тесты Junit (<https://oreil.ly/0FYD9>), чтобы протестировать метод с разными парами «ключ — значение».

❷ Определяется ожидаемый результат метода.

❸ Используется библиотека `AssertJ` для проверки значения, возвращаемого методом `MyTopology.decodeKey`, на соответствие ожиданиям.

При таком способе тестирования логики обработки можно проверить самые разные тестовые случаи, которые могут помочь выявить и предотвратить регрессии

в коде. Кроме того, приложения Kafka Streams часто включают не только узлы-обработчики, но также различные вспомогательные методы, служебные классы или пользовательские реализации Serdes, обеспечивающие дополнительную поддержку топологии. В этом случае важно протестировать и эти модули кода, используя описываемый здесь подход.

Эта форма тестирования хорошо подходит для проверки приложений, использующих DSL. Однако, когда дело доходит до Processor API, необходимо принять во внимание некоторые дополнительные нюансы, касающиеся структурирования и выполнения тестов. Мы рассмотрим это в следующем разделе.

Processor API. При использовании Processor API для логики узлов-обработчиков вместо лямбда-выражений и ссылок на методы обычно используются классы, реализующие интерфейс `Processor` или `Transformer`. Соответственно, стратегия тестирования узлов немного отличается, потому что для этого нужно симитировать базовый экземпляр `ProcessorContext`, который Kafka Streams передает узлам-обработчикам низкого уровня при их первой инициализации.



Говоря об узлах-обработчиках, мы не имеем в виду интерфейс `Processor`, а используем этот термин в более широком смысле для обозначения любого кода, отвечающего за применение логики обработки/преобразования данных к входному потоку. При использовании Processor API узел-обработчик может реализовать интерфейс `Processor` или `Transformer`. Чтобы освежить в памяти различия между этими интерфейсами, вернитесь к разделу «Обработчики и преобразователи» главы 7.

Для демонстрации тестирования узла-обработчика низкого уровня реализуем преобразователь с состоянием, запоминающий количество записей для каждого уникального ключа. Реализация такого преобразователя показана ниже:

```
public class CountTransformer
    implements ValueTransformerWithKey<String, String, Long> {

    private KeyValueStore<String, Long> store;

    @Override
    public void init(ProcessorContext context) { ❶
        this.store =
            (KeyValueStore<String, Long>) context.getStateStore("my-store"); ❷
    }

    @Override
    public Long transform(String key, String value) { ❸
```

```
// обработать маркеры удаления (надгробия)
if (value == null) {
    store.delete(key);
    return null;
}

// получить предыдущее значение счетчика для этого ключа
// или 0, если ключ встречен впервые
Long previousCount = store.get(key);
if (previousCount == null) {
    previousCount = 0L;
}

// увеличить счетчик
Long newCount = previousCount + 1;
store.put(key, newCount);
return newCount;
}

@Override
public void close() {}
}
```

❶ Реализации интерфейсов `Processor` и `Transformer` должны иметь функцию `init`, которая получает контекст `ProcessorContext`, используемый для решения различных задач, таких как получение хранилищ состояния и планирование периодических функций.

❷ Этот преобразователь имеет хранимое состояние — количество значений для каждого уникального ключа, встреченного в потоке, поэтому мы сохраняем ссылку на хранилище состояний.

❸ Метод `transform` содержит логику этого узла-обработчика. Комментарии в этом блоке кода описывают детали реализации.

Создав преобразователь, можно переходить к определению модульного теста для него. В отличие от простых тестов для отдельных методов, показанных в предыдущем разделе, этот тест требует использовать некоторые вспомогательные функции из пакета `kafka-streams-test-utils`. Кроме всего прочего, этот пакет позволяет создавать объекты `MockProcessorContext`, помогающие получить доступ к состоянию и планировать периодические функции без фактического запуска топологии.

Типичный подход — создать `MockProcessorContext` в функции `setup`, которая запускается до выполнения модульных тестов. Если обработчик имеет хранимое состояние, то также нужно инициализировать и зарегистрировать хранилище

состояний. Несмотря на то что в топологии может использоваться постоянное хранилище состояний, для модульных тестов рекомендуется использовать хранилище в памяти. Ниже показан пример, как настроить `MockProcessorContext` и зарегистрировать хранилище состояний:

```
public class CountTransformerTest {
    MockProcessorContext processorContext; ❶

    @BeforeEach ❷
    public void setup() {
        Properties props = new Properties(); ❸
        props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
        props.put(StreamsConfig.BootstrapServersConfig, "dummy:1234");
        processorContext = new MockProcessorContext(props); ❹

        KeyValueStore<String, Long> store = ❺
            Stores.keyValueStoreBuilder(
                Stores.inMemoryKeyValueStore("my-store"),
                Serdes.String(), Serdes.Long())
                .withLoggingDisabled()
                .build();

        store.init(processorContext, store); ❻
        processorContext.register(store, null);
    }
}
```

❶ Объект `MockProcessorContext` сохраняется в переменной экземпляра, чтобы на него можно было сослаться позже в тестах.

❷ Метод `setup` отмечен аннотацией `JUnit BeforeEach`, заставляющей фреймворк выполнять эту функцию перед каждым тестом.

❸ Чтобы создать `MockProcessorContext`, нужно определить некоторые свойства `Kafka Streams`. Здесь определяются только два обязательных свойства.

❹ Создание экземпляра `MockProcessorContext`.

❺ Наш преобразователь имеет хранимое состояние, поэтому для него создается хранилище в памяти.

❻ Инициализация и регистрация хранилища состояний для его использования в тестах.

После реализации `MockProcessorContext` можно просто создать и инициализировать экземпляр класса `Processor` или `Transformer`, используя этот объект контекста, а затем выполнить модульные тесты. Ниже показано, как

протестировать наш преобразователь `CountTransformer`, применяя этот подход. Тест проверяет базовое поведение счетчика, а также обработку маркеров удаления (надгробий):

```
public class CountTransformerTest {
    MockProcessorContext processorContext;

    @BeforeEach
    public void setup() {
        // см. предыдущий раздел
    }

    @Test ❶
    public void testTransformer() {
        String key = "123";
        String value = "some value";

        CountTransformer transformer = new CountTransformer(); ❷
        transformer.init(processorContext); ❸

        assertThat(transformer.transform(key, value)).isEqualTo(1L); ❹
        assertThat(transformer.transform(key, value)).isEqualTo(2L);
        assertThat(transformer.transform(key, value)).isEqualTo(3L);
        assertThat(transformer.transform(key, null)).isNull(); ❺
        assertThat(transformer.transform(key, value)).isEqualTo(1L);
    }
}
```

❶ Аннотация `Test` сообщает фреймворку JUnit, что этот метод выполняет тест.

❷ Создание экземпляра `CountTransformer`.

❸ Инициализация экземпляра `CountTransformer` с помощью `MockProcessorContext`, созданного в методе `setup`.

❹ Проверка поведения преобразователя. Поскольку `CountTransformer` подсчитывает, сколько раз встречен каждый ключ, здесь выполняется серия тестов, чтобы убедиться, что счетчик увеличивается, как ожидалось.

❺ Преобразователь содержит также логику обработки маркеров удаления (надгробий — записей со значением `Null`). В `Kafka Streams` надгробия сигнализируют о том, что запись должна быть удалена из хранилища состояний. Эта и следующая строки обеспечивают правильную обработку надгробий.

`MockProcessorContext` также предоставляет дополнительные возможности тестирования, например, проверку обработчиков, полагающихся на периодические функции для передачи записей нижестоящим обработчикам. Дополнительные примеры модульного тестирования низкоуровневых обработчиков вы найдете в официальной документации (<https://oreil.ly/3YAbM>).

Поведенческие тесты

Модульное тестирование приносит определенную пользу, но топология Kafka Streams нередко включает целый набор обработчиков, функционирующих вместе. Можно ли протестировать поведение всей топологии? Да, можно, и в этом нам опять может помочь пакет `kafka-streams-test-utils`. На этот раз мы используем смоделированную среду выполнения, включенную в эту библиотеку.

Для демонстрации создадим очень простую топологию и протестируем ее, написав несколько тестов.

Ниже показана простая топология Kafka Streams, извлекающая имена пользователей из темы `users`, генерирующая приветствие для каждого пользователя и помещающая его в тему `greetings`. Однако здесь есть одно исключение: если пользователя зовут Randy, то приветствие не генерируется. Это условие фильтрации даст нам более интересный тестовый сценарий (и совсем не с целью обидеть кого-либо с именем Randy):

```
class GreeterTopology {

    public static String generateGreeting(String user) {
        return String.format("Hello %s", user);
    }

    public static Topology build() {
        StreamsBuilder builder = new StreamsBuilder();

        builder
            .stream("users", Consumed.with(Serdes.Void(), Serdes.String()))
            .filterNot((key, value) -> value.toLowerCase().equals("randy"))
            .mapValues(GreeterTopology::generateGreeting)
            .to("greetings", Produced.with(Serdes.Void(), Serdes.String()));

        return builder.build();
    }
}
```

Теперь напишем модульный тест, использующий тестовый драйвер из библиотеки Kafka Streams, который называется *тестовым драйвером топологии*. Этот тестовый драйвер позволяет направлять данные в топологию Kafka Streams (в данном случае в определенную нами топологию `GreeterTopology`) и анализировать данные, добавляемые в выходные темы. Начнем с определения методов `setup` и `teardown`, создающих и уничтожающих экземпляр тестового драйвера:

```
class GreeterTopologyTest {
    private TopologyTestDriver testDriver;
    private TestInputTopic<Void, String> inputTopic;
    private TestOutputTopic<Void, String> outputTopic;
```

```

@BeforeEach
void setup() {
    Topology topology = GreeterTopology.build(); ❶

    Properties props = new Properties(); ❷
    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "dummy:1234"); ❸

    testDriver = new TopologyTestDriver(topology, props); ❹

    inputTopic =
        testDriver.createInputTopic( ❺
            "users",
            Serdes.Void().serializer(),
            Serdes.String().serializer());

    outputTopic =
        testDriver.createOutputTopic( ❻
            "greetings",
            Serdes.Void().deserializer(),
            Serdes.String().deserializer());
}

@AfterEach
void teardown() {
    testDriver.close(); ❼
}
}

```

❶ Создание экземпляра топологии Kafka Streams.

❷ Настройки Kafka Streams с двумя обязательными конфигурационными параметрами.

❸ Топология будет работать в смоделированной среде выполнения, поэтому загрузочные серверы не обязательно должны иметь действительные имена.

❹ Создание тестового драйвера топологии. Тестовые драйверы имеют вспомогательные методы, упрощающие тестирование топологий.

❺ Тестовый драйвер имеет вспомогательный метод для создания входной темы. Здесь создается тема `users`.

❻ Тестовый драйвер имеет также вспомогательный метод для создания выходной темы. Здесь создается тема `greetings`.

❼ Освобождение ресурсов после каждого теста вызовом метода `TopologyTestDriver.close()`.

Теперь, организовав создание экземпляра `TopologyTestDriver` для каждого теста, можно начинать писать тесты для топологии. Наши тесты просто передают со-

общения во входную тему и проверяют наличие ожидаемых записей в выходной теме. Пример реализации этой стратегии тестирования показан ниже:

```
class GreeterTopologyTest {
    // ...

    @Test
    void testUsersGreeted() {
        String value = "Izzy"; ❶

        inputTopic.pipeInput(value); ❷

        assertThat(outputTopic.isEmpty()).isFalse(); ❸

        List<TestRecord<Void, String>> outRecords =
            outputTopic.readRecordsToList(); ❹
        assertThat(outRecords).hasSize(1); ❺

        String greeting = outRecords.get(0).getValue(); ❻
        assertThat(greeting).isEqualTo("Hello Izzy");
    }
}
```

❶ Создание записи для передачи во входную тему.

❷ Передача тестовой записи во входную тему. Метод `pipeInput` имеет еще одну перегруженную версию, которая принимает ключ и значение.

❸ Вызов метода `isEmpty()`, чтобы убедиться, что выходная тема содержит хотя бы одну запись.

❹ Прочитать записи из выходной темы можно несколькими способами. Здесь все записи считываются в виде списка с помощью метода `readRecordsToList()`. Для чтения также можно использовать методы: `readValue()`, `readKeyValue()`, `readRecord()` и `readKeyValuesToMap()`.

❺ Проверка присутствия в выходной теме только одной записи (топологии, использующие операторы `flatMap`, могут для одной входной записи создавать несколько выходных).

❻ Чтение значения выходной записи. Для доступа к дополнительным данным в записях можно использовать методы: `getKey()`, `getRecordTime()` и `getHeaders()`.

Метод `TestOutputTopic.readRecordsToList()` удобен для тестирования потоков, потому что возвращает всю последовательность выходных событий. С другой стороны, для тестирования таблиц удобно использовать `TestOutputTopic.readKeyValuesToMap()`, потому что он возвращает только самое последнее представление каждого ключа. Мы уже обсуждали связи между потоками

и списками (оба используют семантику вставки) и таблицами и массивами ключей-значений (оба используют семантику обновления) в подразделе «Потоково-табличный дуализм» главы 2, а также связи, которые моделируют эти методы.

Оценка производительности

В отличие от `ksqlDB`, которая берет на себя все хлопоты по созданию базовой топологии, `Kafka Streams` дает нам свободу выбора между DSL и Processor API, что увеличивает вероятность по неосторожности ухудшить производительность. Например, если случайно добавить медленный вычислительный шаг или реорганизовать топологию так, что она станет менее эффективной, то эта проблема окажется менее болезненной в случае, когда удастся обнаружить регрессию до того, как она попадет в рабочее окружение.

Чтобы защититься от проблем снижения производительности, необходимо проводить хронометраж после каждого изменения приложения `Kafka Streams`. К счастью, для этого можно объединить смоделированную среду выполнения из пакета `kafka-streams-test-utils` с фреймворком тестирования производительности, таким как `JMH`.

Первое, что нужно сделать, — просто добавить плагин `me.champeau.gradle.jmh` в файл `build.gradle` и настроить задачу `jmh`, которую создает этот плагин. Как все это сделать, показано ниже:

```
plugins {  
    id 'me.champeau.gradle.jmh' version '0.5.2'  
}  
  
jmh { ❶  
    iterations = 4  
    benchmarkMode = ['thrpt']  
    threads = 1  
    fork = 1  
    timeOnIteration = '3s'  
    resultFormat = 'TEXT'  
    profilers = []  
  
    warmupIterations = 3  
    warmup = '1s'  
}
```

❶ Плагин `jmh` поддерживает несколько конфигурационных параметров. Полный список этих параметров с их описанием вы найдете в документации плагина (<https://oreil.ly/vBYCb>).

Теперь определим класс для выполнения хронометража. В отличие от других тестов Kafka Streams, созданных нами выше, ожидается, что код, осуществляющий хронометраж, будет находиться в каталоге `src/jmh/java`. Вот как выглядит определение нашего класса:

```
public class TopologyBench {
    @State(org.openjdk.jmh.annotations.Scope.Thread)
    public static class MyState {
        public TestInputTopic<Void, String> inputTopic;

        @Setup(Level.Trial) ❶
        public void setupState() {
            Properties props = new Properties();
            props.put(StreamsConfig.APPLICATION_ID_CONFIG, "test");
            props.put(StreamsConfig.BootstrapServers_CONFIG, "dummy:1234");
            props.put(StreamsConfig.CacheMaxBytesBuffering_CONFIG, 0);

            // создать топологию
            Topology topology = GreeterTopology.build();

            // создать экземпляр тестового драйвера для передачи данных в топологию
            TopologyTestDriver testDriver = new TopologyTestDriver(topology, props);

            testDriver = new TopologyTestDriver(topology, props);

            // создать входную тему
            inputTopic =
                testDriver.createInputTopic(
                    "users", Serdes.Void().serializer(), Serdes.String().serializer());
        }
    }

    @Benchmark ❷
    @BenchmarkMode(Mode.Throughput) ❸
    @OutputTimeUnit(TimeUnit.SECONDS)
    public void benchmarkTopology(MyState state) {
        state.inputTopic.pipeInput("Izzy"); ❹
    }
}
```

❶ Аннотация `Setup` сообщает фреймворку, что этот метод должен выполняться перед тестом производительности. Здесь этот метод используется для настройки экземпляра `TopologyTestDriver` и входной темы. Аргумент `Level.Trial` сообщает фреймворку JMH, что этот метод следует вызывать перед каждым запуском теста (это *не* означает, что `Setup` будет вызываться перед каждой итерацией).

❷ Аннотация `Benchmark` сообщает фреймворку JMH, что это метод тестирования производительности.

❸ Здесь устанавливается режим тестирования `Mode.Throughput`. В этом режиме подсчитывается, сколько раз в секунду может выполняться этот метод.

❹ Передача тестовой записи во входную тему.

Теперь, когда тест оценки производительности готов, можно выполнить его с помощью Gradle, запустив следующую команду:

```
$ ./gradlew jmh
```

В результате должен появиться примерно такой вывод:

Benchmark	Mode	Cnt	Score	Error	Units
TopologyBench.benchmarkTopology	thrpt	4	264794.572 ±	39462.097	ops/s

Означает ли это, что приложение сможет обрабатывать более 264 тысяч сообщений в секунду? Скорее всего, нет. Не забывайте, что тест оценки производительности, как и тесты топологии в предыдущем разделе, выполняются в моделируемой среде выполнения. Кроме того, приложение не участвует в сетевых взаимодействиях с внешним кластером Kafka (а именно это является обычным делом в рабочих окружениях) и тестирование производительности часто выполняется на другом оборудовании (например, на сервере непрерывной интеграции), а не на рабочих серверах. Поэтому цифры, полученные при тестировании, следует использовать только для относительной оценки производительности топологии и для сравнения с результатами тестирования в будущем.

Оценка производительности кластера Kafka

Независимо от того, работаете ли вы с Kafka Streams или ksqlDB, вы также можете оценить производительность на уровне кластера Kafka. В составе Kafka есть несколько сценариев командной строки, которые могут помочь в этом. Они позволяют измерить пропускную способность чтения/записи, выполнить нагрузочное тестирование кластера, а также определить влияние на производительность кластера различных параметров клиентов (размера пакета, буферной памяти, подтверждений производителя, количества потребителей) и характеристик входных данных (размера записи, объема сообщений).

Проанализировать производительность передачи данных в тему можно с помощью команды `kafka-producer-perf-test`. Она имеет множество параметров, и часть из них показана в следующем примере:

```
kafka-producer-perf-test \
  --topic users \
  --num-records 1000000 \
  --record-size 100 \
  --throughput -1 \
```

```
--producer-props acks=1 \
bootstrap.servers=kafka:9092 \
buffer.memory=67108864 \
batch.size=8196
```

Другая полезная версия этой команды предполагает передачу файла с тестовыми данными. Записи, перечисленные в этом файле, будут последовательно добавляться в указанную тему Kafka. Следующий код добавит три тестовые записи в файл с именем `input.json`:

```
cat <<EOF >./input.json
{"username": "Mitch", "user_id": 1}
{"username": "Isabelle", "user_id": 2}
{"username": "Elyse", "user_id": 3}
EOF
```

Теперь можно заменить флаг `--record-size` аргументом `--payload-file input.json`, чтобы запустить оценку производительности с использованием записей из этого файла:

```
kafka-producer-perf-test \
--topic users \
--num-records 1000000 \
--payload-file input.json \
--throughput -1 \
--producer-props acks=1 \
bootstrap.servers=kafka:9092 \
buffer.memory=67108864 \
batch.size=8196
```

Вот пример отчета с результатами оценки производительности:

```
1000000 records sent, 22166.559528 records/sec (0.76 MB/sec),
58.45 ms avg latency, 465.00 ms max latency,
65 ms 50th, 165 ms 95th, 285 ms 99th, 380 ms 99.9th.
```

Есть также сценарий командной строки для оценки производительности потребителей. Вот пример его вызова:

```
kafka-consumer-perf-test \
--bootstrap-server kafka:9092 \
--messages 100000 \
--topic users \
--threads 1
```

```
# пример вывода (переформатирован, чтобы уместить по ширине страницы)
start.time          end.time          data.consumed.in.MB
2020-09-17 01:23:41:932  2020-09-17 01:23:42:817  9.5747
MB.sec    data.consumed.in.nMsg  nMsg.sec
10.8189   100398
```

Заключительные замечания о тестировании

Обязательно подумайте об автоматизации запуска тестов после каждого изменения кода. Вот пример сценария рабочего процесса с такой автоматизацией:

- все изменения кода переносятся в отдельную ветвь в системе управления версиями (например, GitHub, Bitbucket);
- чтобы добавить свои изменения, разработчик должен открыть запрос на прием;
- при открытии запроса автоматически выполняются тесты с использованием таких систем, как Jenkins, Travis, GitHub Actions и т. д.;
- если один или несколько тестов потерпят неудачу, добавление изменений блокируется, иначе код объявляется доступным для рецензирования.

Подобные рабочие процессы могут быть обыденным делом для многих читателей. И все же я хочу еще раз подчеркнуть, что автоматизация тестирования играет важную роль в предотвращении появления регрессий в коде, поэтому стоит регулярно пересматривать применяемые практики тестирования и рабочие процессы.

Теперь, узнав, как тестировать приложения `ksqlDB` и `Kafka Streams`, перейдем к другому немаловажному требованию, которое должно быть удовлетворено перед передачей программного обеспечения в эксплуатацию: к поддержке мониторинга.

Мониторинг

Мониторинг — обширная сфера, в которой используется множество различных технологий. В этом разделе мы не будем пытаться подробно описать все подходы, а дадим лишь краткий список различных видов мониторинга, которые вы сможете внедрить у себя, и приведем некоторые примеры технологий.

Однако мониторинг `Kafka Streams` и `ksqlDB` имеет одну особенность, которую мы рассмотрим подробно. И `Kafka Streams`, и `ksqlDB` имеют набор встроенных метрик JMX¹ и позволяют извлекать эти метрики, чем значительно повышают

¹ Аббревиатура JMX расшифровывается как Java Management Extensions (управляющие расширения Java) и обозначает механизм, используемый для мониторинга и управления ресурсами. Библиотека `Kafka Streams` регистрирует и обновляет набор метрик JMX, которые дают представление о работе приложений. Поскольку `ksqlDB` основана на `Kafka Streams`, она автоматически наследует эти метрики JMX.

уровень наблюдаемости приложений и запросов. Чуть ниже мы рассмотрим технические детали извлечения метрик, но прежде перечислим возможные виды мониторинга.

Виды мониторинга

В следующей таблице перечислены некоторые стратегии мониторинга, которые можно использовать для наблюдения за приложениями Kafka Streams и ksqlDB:

Стратегия мониторинга	Что находится под наблюдением	Пример технологии
Мониторинг кластера	<ul style="list-style-type: none"> • Недостаточно реплицированные разделы. • Отставание потребителей. • Смещение продвижения. • Пропускная способность темы 	kafka_exporter ¹
Мониторинг журналов	<ul style="list-style-type: none"> • Общая частота операций записи в журнал. • Частота операций записи/ошибок 	ELK, ElastAlert, Cloud Logging
Мониторинг метрик	<ul style="list-style-type: none"> • Скорость потребления. • Скорость производства. • Задержка обработки. • Время опроса 	Prometheus
Нестандартное инструментирование ²	<ul style="list-style-type: none"> • Пользовательские метрики 	OpenCensus
Профилирование	<ul style="list-style-type: none"> • Взаимоблокировки. • Горячие точки 	YourKit
Визуализация	<ul style="list-style-type: none"> • Все перечисленное выше 	Grafana
Предупреждения	<ul style="list-style-type: none"> • SLO (service-level objectives — цели уровня обслуживания) 	Alertmanager

Извлечение метрик JMX

Kafka Streams и ksqlDB предоставляют доступ к метрикам через Java Management Extensions (JMX). Получить доступ к этим метрикам после запуска приложения Kafka Streams или сервера ksqlDB можно с помощью, например, JConsole, и этот способ описывается в большинстве руководств. Если JConsole имеется

¹ См. https://github.com/danielqsj/kafka_exporter.

² Преимущественно используется для Kafka Streams.

на том же компьютере, где выполняется Kafka Streams и/или ksqlDB, то можно просто запустить команду `jconsole` без всяких аргументов и приступить к исследованию метрик JMX.

Например, ниже показано, как настроить различные свойства для удаленного мониторинга JMX и запустить ksqlDB с включенной поддержкой JMX:

`docker-compose up` ❶

`MY_IP=$(ipconfig getifaddr en0);` ❷

`docker run \` ❸

`--net=chapter-12_default \` ❹

`-p 1099:1099 \`

`-v "$(pwd)/ksqldb":/ksqldb \`

`-e KSQL_JMX_OPTS="\` ❺

`-Dcom.sun.management.jmxremote \`

`-Djava.rmi.server.hostname=$MY_IP \`

`-Dcom.sun.management.jmxremote.port=1099 \`

`-Dcom.sun.management.jmxremote.rmi.port=1099 \`

`-Dcom.sun.management.jmxremote.authenticate=false \`

`-Dcom.sun.management.jmxremote.ssl=false" \`

`-ti confluentinc/ksqldb-server:0.14.0 \`

`ksql-server-start /ksqldb/config/server.properties` ❻

❶ Установка Docker Compose для запуска кластера Kafka. Экземпляр сервера ksqlDB, который запускается парой строк ниже, будет общаться с этим кластером Kafka.

❷ Сохранение IP-адреса в переменной окружения. Она будет использоваться в свойствах JMX ниже для доступа к метрикам JMX по этому IP-адресу.

❸ Запуск экземпляра сервера ksqlDB с включенной поддержкой JMX.

❹ Сервер ksqlDB запускается вручную вне Docker Compose, поэтому для подключения к сети Docker используется флаг `--net`.

❺ Определение системных свойств, необходимых для доступа к метрикам JMX.

❻ Запуск экземпляра сервера ksqlDB.

Открыть JConsole можно с помощью следующей команды:

`jconsole $MY_IP:1099`

На вкладке MBeans можно увидеть все доступные метрики, экспортируемые приложением Kafka Streams или ksqlDB. Например, на рис. 12.1 показан спи-

сок метрик сервера ksqlDB. Так как ksqlDB основана на Kafka Streams, а Kafka Streams — на клиентах Kafka Consumer и Kafka Producer более низкого уровня, то в списке можно увидеть метрики, экспортируемые всеми базовыми библиотеками.

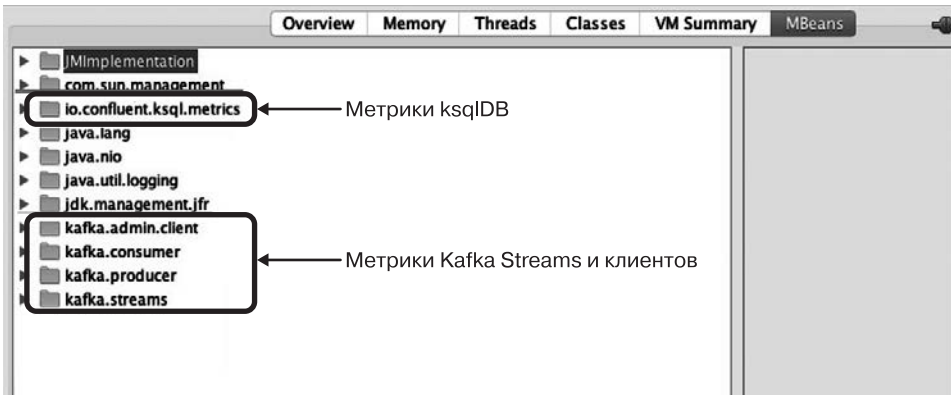


Рис. 12.1. Группы метрик Kafka Streams в окне JConsole

Каждую группу можно раскрыть и просмотреть метаданные и значения каждой метрики, как показано на рис. 12.2.

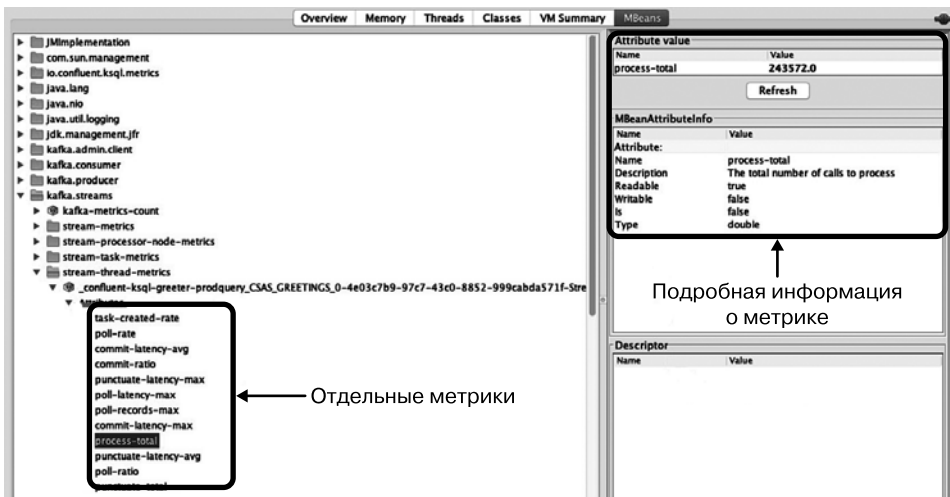


Рис. 12.2. Просмотр более подробной информации на примере некоторых метрик Kafka Streams

JSON Console отлично подходит для динамической проверки метрик в среде тестирования или разработки, но после развертывания программного обеспечения в промышленном окружении необходимо более надежное решение, способное хранить исторические наборы данных, выполнять запросы к данным и интегрироваться с системами оповещения.

На момент написания этих строк технология, которую я рекомендую для работы с Kafka Streams и метриками JMX ksqlDB, — это Prometheus. Чтобы получить метрики в Prometheus, нужен экспортер, поставляющий метрики через конечную точку HTTP. Настройка экспорта для экземпляров Kafka Streams и ksqlDB аналогична. Для этого нужно:

- загрузить файл JAR экспортера Prometheus JMX (<https://oreil.ly/t-7Lg>);
- запустить Kafka Streams или ksqlDB с флагом:

```
-javaagent:./jmx_prometheus_javaagent-0.14.0.jar=8080:config.yaml
```
- настроить Prometheus для доступа к конечной точке HTTP: определить IP-адрес или имя хоста, где запущено приложение Kafka Streams или ksqlDB, и номер порта, настроенный для экспорта метрик JMX.

Исходный код для этой главы включает полный пример использования Prometheus с Kafka Streams и ksqlDB. Мы опустили технические особенности Prometheus из этого текста, поскольку это детали реализации. Но главный вывод заключается в том, что, независимо от того, используете ли вы Prometheus или другую систему мониторинга, вы должны экспортировать встроенные метрики JMX, предоставляемые Kafka Streams и ksqlDB во внешнюю систему для улучшения наблюдаемости вашего приложения.

Развертывание

Kafka Streams и ksqlDB могут работать на компьютерах без операционной системы, внутри виртуальных машин или в контейнерах. Наиболее предпочтительный путь — контейнеры Docker. Вот некоторые из преимуществ этого подхода:

- контейнеры предлагают облегченную изолированную среду выполнения для приложений;
- реестры контейнеров, как частные, так и общедоступные, упрощают хранение, управление, совместное использование и поддержку нескольких версий контейнеров¹;

¹ В числе наиболее популярных реестров контейнеров можно назвать Docker Hub, Artifactory и Google Container Registry.

- контейнеры позволяют отделить базовую инфраструктуру от прикладного кода, то есть один и тот же контейнер можно запустить на нескольких серверах в одном вычислительном центре, на множестве облачных платформ или даже на локальном компьютере.

Посмотрим, как работать с обеими технологиями, используя контейнеры.

Контейнеры ksqlDB

Компания Confluent уже опубликовала официальные образы контейнеров для сервера ksqlDB и клиента командной строки. Когда наступит момент развернуть запросы в рабочей среде, вы сможете использовать официальный образ контейнера сервера ksqlDB (`confluentinc/ksqldb-server`) или производный от него, если вам понадобится внести какие-то коррективы¹.

Официальный образ сервера ksqlDB позволяет задать практически любые конфигурационные свойства ksqlDB с помощью переменных окружения. Однако этот подход имеет несколько недостатков:

- конфигурация должна храниться в системе управления версиями; настройка всех необходимых свойств ksqlDB с помощью переменных окружения усложняет эту задачу;
- команды запуска образа контейнера с приложением, имеющим множество настроек, могут стать слишком громоздкими;
- установка переменных окружения — это более быстрая замена монтированию фактического конфигурационного файла в контейнер, но, если ksqlDB запускается в автономном режиме или используются некоторые средства интеграции данных, все равно придется монтировать файлы в контейнер.

Поэтому я рекомендую не полагаться исключительно на переменные окружения для настройки экземпляров сервера ksqlDB, а монтировать конфигурационный файл из системы управления версиями в контейнер. Например, давайте сохраним конфигурацию сервера ksqlDB в файле с именем `config/server.properties` на локальном компьютере. Мы упростим содержимое файла ради демонстрации, но вы можете включить в него любые свойства ksqlDB (см. приложение Б):

```
bootstrap.servers=kafka:9092
ksql.service.id=greeter-prod
```

¹ Производный образ просто использует официальный образ ksqlDB в качестве базового, ссылаясь на него в инструкции FROM в верхней части Dockerfile. Любые дополнительные настройки указываются далее в файле Dockerfile.

Этот файл можно смонтировать в официальный образ контейнера и вызвать команду `ksql-server-start`:

```
docker run \
  --net=chapter-12_default \ ❶
  -v "$(pwd)/config":/config \ ❷
  -ti confluentinc/ksqldb-server:0.14.0 \
  ksql-server-start /config/server.properties ❸
```

❶ Этот флаг характерен для нашего учебного проекта и позволяет подключиться к кластеру Kafka, работающему в окружении Docker Compose.

❷ Монтирует конфигурацию из хост-системы в работающий контейнер.

❸ Запуск экземпляра сервера ksqlDB со смонтированным конфигурационным файлом.

Напомню, что существуют другие конфигурационные файлы ksqlDB, определяемые следующими параметрами.

`ksql.connect.worker.config`

Необязательный конфигурационный файл, определяющий местоположение конфигурации рабочего процесса Kafka Connect при запуске Kafka Connect в режиме встроенной интеграции (см. рис. 9.3).

`queries.file`

При запуске сервера ksqlDB в автономном режиме этот параметр указывает местоположение файла для выполняемых запросов.

Если конфигурация включает какое-либо из этих свойств, необходимо также смонтировать в контейнер файлы, на которые они ссылаются.

Теперь давайте посмотрим, как запускать приложения Kafka Streams внутри контейнеров.

Контейнеры Kafka Streams

Запустить Kafka Streams внутри контейнера немного сложнее, но все равно довольно просто. Для этого можете создать свой образ Docker, используя базовый образ более низкого уровня¹. Однако самый простой вариант — применить популярный конструктор образов под названием Jib (<https://oreil.ly/xjvpR>), разработанный в Google и упрощающий контейнеризацию и распространение приложений на Java.

¹ Пример подходящего базового образа: `openjdk:8-jdk-slim`.

Чтобы использовать Jib, нужно добавить зависимость в файл сборки. В этой книге мы используем систему сборки Gradle, поэтому добавим в файл `build.gradle` следующий код:

```
plugins {  
    id 'com.google.cloud.tools.jib' version '2.1.0'  
}  
  
jib {  
    to {  
        image = 'magicalpipelines/myapp:0.1.0'  
    }  
    container {  
        jvmFlags = []  
        mainClass = application.mainClassName  
        format = 'OCI'  
    }  
}
```

Для именования образа можно по желанию использовать самые разные схемы, чтобы Jib автоматически помещал наш образ Docker в тот или иной реестр контейнеров. Официальная документация Jib (<https://oreil.ly/3C-09>) — лучший ресурс, где можно найти перечень доступных реестров контейнеров и схем именования. На момент написания этих строк поддерживались следующие реестры:

- Docker Hub;
- Google Container Registry (GCR);
- Amazon Elastic Container Registry (ECR);
- Azure Container Registry (ACR).

После обновления файла сборки можно выполнить одну из следующих задач.

`./gradlew jib`

Создаст образ контейнера и отправит его в реестр. Для ее успешного выполнения необходимо пройти аутентификацию в выбранном реестре контейнеров.

`./gradlew jibDockerBuild`

Создаст образ контейнера с помощью локального демона Docker, но *не* отправит его в реестр. Это самый простой способ проверить, можно ли создать свой образ.

Для демонстрации давайте создадим образ Docker локально, выполнив следующую команду:

`./gradlew jibDockerBuild`

Она должна вывести следующий текст:

```
Built image to Docker daemon as magicalpipelines/myapp:0.1.0
```

Тег образа Docker в выводе содержит имя нашего приложения Kafka Streams, и сам образ может выполняться на любом компьютере, способном запускать контейнеры. Теперь мы оказываемся на одном уровне с ksqlDB, поскольку имеем возможность запускать рабочие нагрузки ksqlDB или Kafka Streams с использованием контейнеров. Но как запускать эти контейнеры? Один из способов описывается в следующем разделе.

Оркестрация контейнеров

Мы твердо убеждены, что лучший способ запустить приложение Kafka Streams или ksqlDB — использовать систему оркестрации контейнеров, такую как Kubernetes. Этот подход имеет следующие преимущества:

- простоту масштабирования рабочих нагрузок: чтобы масштабировать, достаточно увеличить количество реплик контейнера;
- абстрактную базовую инфраструктуру; однако в случаях, когда нужен некоторый контроль над местом, где работают контейнеры, можно использовать *селекторы узлов* или *правила сходства*;
- простоту развертывания так называемых sidecar-контейнеров (контейнеров-прицепов), работающих вместе с вашими контейнерами Kafka Streams или ksqlDB и предоставляющих такие услуги, как:
 - сбор и экспорт метрик в системы мониторинга;
 - журналирование;
 - проксирование взаимодействий (например, если перед приложением Kafka Streams, где включена поддержка интерактивных запросов, требуется добавить уровень аутентификации; этот подход намного проще, чем создание уровня аутентификации и авторизации внутри приложения Kafka Streams);
- простоту координации перезапуска приложения после обновления кода;
- поддержку ресурсов StatefulSet, обеспечивающую постоянное и стабильное хранилище для топологий с состоянием.

Кроме того, если узел, где работает контейнер, выйдет из строя, система оркестрации автоматически переместит контейнер на исправный узел.

Подробное изучение Kubernetes выходит за рамки этой книги, но есть много отличных ресурсов по этой теме, в том числе «Kubernetes. Лучшие практики» Брендана Бернса (Brendan Burns) и др. (Питер, 2021).

Операции

В этом последнем разделе мы рассмотрим несколько практических задач, с которыми вы наверняка столкнетесь при поддержке приложения Kafka Streams или ksqlDB.

Повторная обработка данных в приложении Kafka Streams

Иногда может потребоваться сбросить или запустить с начала обработку данных в приложении Kafka Streams. Такая необходимость может возникнуть, например, после устранения ошибки в системе, чтобы повторно обработать все или часть данных в темах Kafka. Для упрощения этой задачи разработчики Kafka создали инструмент, который распространяется вместе с исходным кодом Kafka (<https://oreil.ly/zNwrl>).



Теоретически этот инструмент можно использовать и с приложениями ksqlDB, но для этого необходимо знать группу потребителей. В Kafka Streams существует прямое соответствие между конфигурацией `application.id` и идентификатором группы потребителей. В ksqlDB нет такого же прямого соответствия между `service.id` и группой потребителей.

Поэтому, решив применить эти инструкции в отношении приложения ksqlDB, будьте особенно осторожны и убедитесь, что правильно указали группу потребителей своего приложения. В этом может помочь сценарий командной строки `kafka-consumer-groups`.

Этот инструмент выполняет несколько операций:

- обновляет смещение потребителя согласно указанной позиции в *темах-источниках*;
- переходит в конец *промежуточных тем*;
- удаляет *внутренний журнал изменений* и *темы повторного секционирования*.



Будьте особенно осторожны, применяя этот инструмент к приложениям с состоянием, потому что он не сбрасывает состояние приложения. Обязательно прочитайте следующие инструкции полностью, чтобы не пропустить важный шаг.

Используя инструмент сброса приложения, выполните следующие действия.

1. Остановите все экземпляры приложения. Не переходите к следующему шагу, пока группа потребителей остается активной. Определить активность

или неактивность группы можно командой, которая дана ниже. В ее выводе неактивные группы потребителей отображаются как пустые:

```
kafka-consumer-groups \
  --bootstrap-server kafka:9092 \ ❶
  --describe \
  --group dev \ ❷
  --state
```

❶ Требуется обновить брокер, на который ссылается пара «хост/порт».

❷ В Kafka Streams группа определяется конфигурационным параметром `application.id`. В приложениях `ksqlDB`, к сожалению, нет прямой зависимости между параметром `service.id` и именем группы потребителей, поэтому группу следует указать явно.

2. Запустите инструмент сброса приложения с соответствующими параметрами. Полный список параметров и подробную информацию о них можно получить, выполнив следующую команду:

```
kafka-streams-application-reset --help
```

Например, чтобы сбросить созданное в этой главе приложение приветствия, можно выполнить такую команду:

```
kafka-streams-application-reset \
  --application-id dev \
  --bootstrap-servers kafka:9092 \
  --input-topics users \
  --to-earliest
```

Она должна вывести примерно следующее:

```
Reset-offsets for input topics [users]
Following input topics offsets will be reset to (for consumer group dev)
Topic: users Partition: 3 Offset: 0
Topic: users Partition: 2 Offset: 0
Topic: users Partition: 1 Offset: 0
Topic: users Partition: 0 Offset: 0
Done.
Deleting all internal/auto-created topics for application dev
Done.
```

3. Если приложение не имеет состояния, его экземпляры можно перезапустить после выполнения команды `kafka-streams-application-reset`. Иначе нужно сбросить состояние приложения перед перезапуском. Это можно сделать двумя способами:

- вручную удалить все каталоги с хранимым состоянием;
- вызвать в коде метод `KafkaStreams.cleanUp` (только для Kafka Streams). Это следует делать только при первом запуске приложения после применения инструмента сброса.

После сброса состояния приложения одним из предыдущих способов можно запустить его заново.

Ограничение скорости вывода приложением

Kafka Streams и ksqlDB могут обеспечить высокую пропускную способность, но иногда возможна ситуация, когда нижестоящие системы, обрабатывающие выходные темы, не справляются с передаваемым объемом. В этом случае можно ограничить скорость вывода данных приложением с помощью *кэша записей*.

В Kafka Streams DSL или ksqlDB кэш записей помогает сократить количество выходных записей, записываемых в хранилища состояний, а также количество записей, пересылаемых нижестоящим узлам. В Processor API кэш записей может уменьшить только количество записей, записываемых в хранилища состояний.

Всем потокам выполнения, генерирующим потоки данных (их количество управляется параметром `num.stream.threads`), выделяются равные доли от общего кэша. То есть если кэш имеет размер 10 485 760 (10 Мбайт) и приложение выполняет 10 потоков, то каждому потоку выполнения будет выделен примерно 1 Мбайт памяти для кэша записей.

Этот параметр влияет на производительность не только вашего приложения, но и последующих систем, извлекающих данные, которые ваше приложение передает в Kafka. Для большей ясности рассмотрим топологию, подсчитывающую количество сообщений для каждого встреченного ключа. Пусть имеет место такая последовательность событий:

```
<key1, value1>;  
<key1, value2>;  
<key1, value3>.
```

Топология, выполняющая агрегирование счетчика с отключенным кэшем записей (то есть с `cache.max.bytes.buffering`, равным 0), выведет такую последовательность агрегированных значений:

```
<key1, 1>;  
<key1, 2>;  
<key1, 3>.
```

Однако если обновления следуют слишком быстро (например, в течение нескольких миллисекунд друг за другом), то нужно определиться: действительно ли необходимо получать все промежуточные состояния (`<key1, 1>` и `<key1, 2>`)? На этот вопрос нет правильного или неправильного ответа, это зависит от желаемых логики и семантики использования. Если вы решите

ограничить скорость вывода данных приложением и уменьшить количество видимых промежуточных состояний, то можно использовать кэш записей.

Если выделить немного памяти для кэша записей с помощью этого конфигурационного параметра, то Kafka Streams и `ksqlDB` потенциально могут сократить объем выводимых агрегатных значений до лаконичного:

```
<key1, 3>.
```

Кэш записей не всегда может уменьшить объем выходных записей, потому что обслуживание кэша осуществляется несколькими компонентами, в том числе выталкивающими содержимое кэша при поступлении новых сообщений. И все же кэш — это важный рычаг, который можно использовать для уменьшения объема данных, производимых приложением. Когда приложение передает меньше данных нижестоящим узлам в топологии и системам, работающим вне Kafka Streams и `ksqlDB`, им приходится обрабатывать меньше данных.

За дополнительной информацией об этом важном параметре обращайтесь к официальной документации (<https://oreil.ly/ddP7x>).

Обновление Kafka Streams

На момент написания этих строк существовал план выпуска новых версий Apache Kafka (включая Kafka Streams) с привязкой ко времени. Поэтому можно ожидать, что новая версия Kafka Streams будет выпускаться каждые четыре месяца. Стратегия управления версиями приводится здесь:

```
major.minor.bug-fix
```

В большинстве случаев четырехмесячный график выпуска применяется к младшим версиям продукта. Например, версия 2.5.0 была выпущена в апреле 2020 года, а версия 2.6.0 — четыре месяца спустя, в августе 2020 года. В редких случаях можно увидеть увеличение старшего номера версии, когда вносятся особенно важные изменения (например, существенные изменения в формате сообщений Kafka или в общедоступном API) или когда проект достигает важной вехи. Исправления ошибок выходят чаще и без привязки к конкретной дате.

Всякий раз, обновляя Kafka Streams, важно следовать руководству по обновлению, которое публикуется на официальном сайте Kafka (<https://oreil.ly/DohxP>) для каждого выпуска. В некоторых случаях может потребоваться установить в параметре `upgrade.from` более старую версию, а затем выполнить серию повторных запусков, чтобы убедиться в безопасном выполнении обновления. Это особенно верно, если вы выжидаете какое-то время перед обновлением (например, при обновлении с версии 2.3.0 до версии 2.6.0).

Есть несколько способов оставаться в курсе появления новых версий:

- подписаться на один или несколько официальных списков рассылки (<https://oreil.ly/LkcYH>), чтобы своевременно получать объявления о выходе новой версии;
- посетить официальный репозиторий GitHub для Apache Kafka и включить «наблюдение» за проектом, а чтобы избавиться от лишнего шума, можно включить флажок **Releases Only** (Только релизы). Также можно дать звезду проекту, чтобы поддержать людей, работающих над Kafka Streams и более широкой экосистемой Kafka;
- подписаться на @apachekafka в «Твиттере» (<https://oreil.ly/3cbly>).

Обновление ksqlDB

ksqlDB быстро развивается, и, вероятно, время от времени в нее вносятся критические изменения, делающие обновление чуть более коварным. Вот выдержка из официальной документации ksqlDB.

«До версии ksqlDB 1.0 каждый второстепенный выпуск потенциально может содержать критические изменения. Это означает, что нельзя просто обновить двоичные файлы ksqlDB и перезапустить сервер (-ы).

Модели данных и двоичные форматы, используемые в ksqlDB, постоянно меняются. Это означает, что данные, локальные для каждого узла ksqlDB и централизованно хранящиеся во внутренних темах Kafka, могут оказаться несовместимыми с новой версией, которую вы пытаетесь развернуть».

ksqldb.io

На самом деле составители документации вообще не рекомендуют устанавливать обновления после развертывания кластера в промышленном окружении, если разработчиками *не обещана* обратная совместимость¹. Однако это *не означает*, что не следует обновляться, а просто подчеркивает неопределенность в отношении последствий, которые влекут некоторые обновления, необходимость строго придерживаться порядка применения обновлений и осознавать, какие потенциальные критические изменения (если таковые имеются) принесет обновление. Перед обновлением кластера ksqlDB всегда следует прочитать официальную документацию ksqlDB (<https://docs.ksqldb.io/>).

¹ Вот точная цитата из документации: «Если вы используете ksqlDB в промышленном окружении и вам не нужны функции или исправления, которые предлагает новая версия, то рекомендуется отложить обновление до тех пор, пока не выйдет версия с действительно нужными вам функциями или исправлениями или пока ksqlDB не достигнет версии 1.0 и не будет обещана обратная совместимость».

Заключение

Kafka Streams и ksqlDB включают утилиты тестирования, помогающие укрепить уверенность в правильности нашего кода перед отправкой в промышленную эксплуатацию и избежать непреднамеренных регрессий, которые могут проникнуть в процессе изменения и улучшения приложений. Кроме того, независимо от используемой технологии поддержка встроенных метрик JMX позволяет наблюдать за работой приложения. Запуская приложения в контейнерной среде, можно сделать их легко переносимыми путем создания собственных образов Docker для приложений Kafka Streams. Или же для запуска сервера ksqlDB и экземпляров клиента командной строки можно использовать образы, поддерживаемые компанией Confluent.

Дополнительные советы по мониторингу, тестированию производительности, ограничению скорости генерирования данных и сбросу приложений в исходное состояние, приведенные в этой главе, помогут вам создавать и поддерживать приложения потоковой обработки. Я уверен, что эти знания наряду с остальной информацией, предлагаемой в этой книге, помогут вам успешно решать широкий круг бизнес-задач, используя две лучшие технологии в области потоковой обработки.

Поздравляю, вы практически дочитали эту книгу!

Настройка Kafka Streams

Kafka Streams имеет множество настроек, а набор доступных параметров и их значения по умолчанию постоянно меняются. Поэтому конфигурационные свойства, перечисленные в этом приложении, следует использовать лишь как отправную точку для знакомства с различными параметрами, а за самой актуальной информацией лучше обращаться к официальной документации (<https://oreil.ly/-dIwr>).

Управление конфигурацией

В книге мы настраивали приложения Kafka Streams, создавая экземпляры `Properties` и вручную определяя различные параметры. Пример этой стратегии показан в следующем коде:

```
class App {  
  
    public static void main(String[] args) {  
        Topology topology = GreeterTopology.build();  
  
        Properties config = new Properties();  
        config.put(StreamsConfig.APPLICATION_ID_CONFIG, "dev-consumer");  
        config.put(StreamsConfig.BootstrapServers_CONFIG, "kafka:9092");  
  
        KafkaStreams streams = new KafkaStreams(topology, config);  
  
        // ....  
    }  
}
```

Однако, когда придет время развернуть приложение в промышленном окружении, следует подумать о загрузке конфигурации из файла вместо определения значений непосредственно в коде приложения. Внесение изменений

в конфигурацию, не затрагивая код, провоцирует меньше ошибок. А если файл конфигурации можно переопределять прямо во время выполнения (например, с помощью системного флага), то вы, считайте, имеете возможность управлять несколькими развертываниями вашего приложения¹.

Полное обсуждение стратегий управления конфигурацией приложений на Java (включая Kafka Streams) выходит за рамки этой книги, но в примерах к этому приложению наглядно показано, как управлять несколькими конфигурациями для одного приложения Kafka Streams с помощью библиотеки под названием Typesafe Config (<https://oreil.ly/i42AO>).

Независимо от способа управления конфигурациями значения конфигурационных параметров, указанные вами, могут повлиять на производительность приложения Kafka Streams или ksqlDB. Поэтому большая часть этого раздела будет посвящена самим параметрам конфигурации.

Начнем с параметров, характерных для Kafka Streams.

Конфигурационные свойства

Прежде всего, для настройки приложений Kafka Streams необходимыми являются два параметра.

`application.id`

Это уникальный идентификатор приложения Kafka Streams. Есть возможность запустить несколько экземпляров приложения с одним и тем же `application.id`, чтобы распределить рабочую нагрузку между несколькими машинами/процессами. За кулисами этот идентификатор используется для нескольких целей, в том числе:

- как идентификатор группы потребителей;
- как префикс имен для внутренних тем (то есть для тем повторного секционирования и журнала изменений, созданных Kafka Streams);
- как префикс `client.id` по умолчанию для производителей и потребителей, используемых библиотекой Kafka Streams;
- как подкаталог для постоянных хранилищ состояний (см. пример 6.1).

¹ У вас может быть несколько развертываний, если приложение должно обрабатывать данные в нескольких кластерах Kafka или если его необходимо развернуть в разных окружениях (например, в промежуточном, тестовом, промышленном).

`bootstrap.servers`

Список пар «хост/порт», соответствующих одному или нескольким брокерам Kafka. Может использоваться для соединения с кластером Kafka.

Также есть несколько необязательных параметров.

`acceptable.recovery.lag`

Максимальное количество задержек (то есть количество непрочитанных записей), которое может иметь задача Kafka Streams во входном разделе, чтобы считаться «теплой» и готовой к назначению. Когда экземпляру приложения с состоянием необходимо восстановить часть или все свое состояние, может быть нежелательно, чтобы Kafka Streams назначала ему работу во время восстановления. В таком случае можно установить этот параметр, чтобы разрешить экземплярам приложения получать назначение задачи, только когда задержка падает ниже этого порога.

`cache.max.bytes.buffering`

Этот параметр управляет размером кэша записей в Kafka Streams. Мы подробно обсуждали кэш записей в подразделе «Ограничение скорости вывода приложением» главы 12. Если присвоить этому параметру значение больше 0, то активируется кэш записей, что может привести к ограничению скорости вывода данных приложением.

`default.deserialization.exception.handler`

Класс для обработки ошибок десериализации. Мы познакомились с ним во врезке «Устранение ошибок десериализации в Kafka Streams» в главе 3. Встроенные варианты: `LogAndContinueExceptionHandler` и `LogAndFailExceptionHandler`. Первый — более либеральный, он позволяет Kafka Streams продолжать обработку записей после появления ошибки десериализации. Второй генерирует исключение и вызывает прекращение обработки. Можно также подставить свою реализацию интерфейса `DeserializationExceptionHandler`, который определен в библиотеке Kafka Streams.

`default.production.exception.handler`

Класс для обработки ошибок, возникающих при создании данных для Kafka. Например, если запись слишком велика, то базовый производитель стенирует исключение, которое необходимо обработать. По умолчанию используется встроенный класс `DefaultProductionExceptionHandler`, который вызывает аварийное завершение Kafka Streams. Другой вариант — класс `AlwaysContinueProductionExceptionHandler`, позволяющий Kafka Streams продолжать обработку данных. Можно также подставить свою реализацию

интерфейса `ProductionExceptionHandler`, если у вас появится желание реализовать свою логику.

`default.timestamp.extractor`

Класс, используемый для связывания записей с отметками времени. Этот параметр подробно обсуждается в разделе «Экстракторы отметок времени» главы 5.

`default.key.serde, default.value.serde`

Класс по умолчанию для сериализации и десериализации ключей (`default.key.serde`) и значений (`default.value.serde`) записей. В этой книге мы в основном определяли используемые классы `Serdes` прямо в коде. Например:

```
KStream<byte[], Tweet> stream =  
    builder.stream(  
        "tweets",  
        Consumed.with(Serdes.ByteArray(), JsonSerdes.Tweet())); ❶
```

❶ В этом примере классы `Serdes` для десериализации ключей (`Serdes.ByteArray()`) и значений (`JsonSerdes.Tweet()`) определены в коде.

Вместо этого можно было бы установить классы `Serdes` по умолчанию, используя эти два параметра, и опустить строку `Consumed.with(...)`, что вынудило бы Kafka Streams использовать классы по умолчанию, указанные в этих параметрах.

`max.task.idle.ms`

Максимальное время, в течение которого задача потоковой обработки должна ждать, пока данные появятся в буферах всех ее разделов. Большие значения могут увеличить задержку, но позволяют предотвратить обработку данных не по порядку, когда задача читает из нескольких входных разделов (например, выполняя соединение).

`max.warmup.replicas`

Количество реплик (в дополнение к `num.standbys`), которые можно использовать для разогрева задачи.

`metrics.recording.level`

Уровень детализации метрик, собираемых Kafka Streams. По умолчанию используется уровень `INFO`, но также можно задать уровень `DEBUG`, чтобы получить больше информации о приложении. Метрики передаются через механизм Java Management Extensions (JMX), это обсуждалось в подразделе «Извлечение метрик JMX» главы 12.

`num.standby.replicas`

Количество реплик для каждого хранилища состояний. Дополнительные реплики помогают сократить время простоя, потому что при отключении задачи Kafka Streams может переназначить работу другому экземпляру приложения, передав ему состояние из реплики. Это позволяет избежать дорогостоящей операции восстановления состояния задачи с нуля. О резервных репликах подробно рассказывается в подразделе «Резервные реплики» главы 6.

`num.stream.threads`

Как рассказывалось в подразделе «Задачи и потоки выполнения» главы 2, потоки выполнения — это то, что выполняет задачи Kafka Streams. Увеличение количества потоков выполнения может помочь более полно использовать доступные вычислительные ресурсы и повысить производительность. Например, приложение Kafka Streams, читающее данные из одной темы, содержащей восемь разделов, создаст восемь задач. При желании можно организовать выполнение этих задач в одном, двух или даже четырех потоках выполнения. Количество зависит от вашего решения, главное, чтобы оно было не больше количества задач. Но если данные обрабатываются на машине с восьмиядерным процессором, то увеличение количества потоков выполнения до восьми позволит распределить работу между всеми доступными ядрами.

Этот параметр особенно важен для максимизации пропускной способности приложения.

`processing.guarantee`

Рассмотрим поддерживаемые значения.

`at_least_once`

При некоторых сбоях (например, проблемы с сетью или сбой брокера, из-за которых производитель повторно отправляет сообщение в исходную тему) записи могут доставляться повторно и никогда не теряются. Это значение по умолчанию в Kafka Streams гарантирует обработку и идеально подходит для приложений, чувствительных к задержкам, но не подверженных влиянию повторной обработки одних и тех же записей.

`exactly_once`

Записи обрабатываются точно один раз с использованием производителя, поддерживающего транзакции, и встроенного потребителя с уровнем изоляции `read_committed`¹. С этим значением могут возникать некоторые

¹ Этот уровень изоляции гарантирует, что потребитель не увидит никаких сообщений, отправленных в ходе неудачных/прерванных транзакций.

задержки в потребителях (поддержка транзакций вызывает относительно небольшое снижение производительности), зато гарантируется однократная обработка каждой записи, что требуют некоторые приложения. Этот вариант поддерживают только брокеры версии 0.11.0 или выше.

`exactly_once_beta`

Этот параметр дает гарантии однократной обработки (по аналогии с предыдущим) с улучшенной масштабируемостью и сокращением накладных расходов для приложений Streams¹. Поддерживается только брокерами версии 2.5 или выше.

`replication.factor`

Фактор репликации для внутренних журналов изменений или тем повторного секционирования, созданных Kafka Streams. Имейте в виду, что не все топологии приводят к созданию внутренней темы, но если топология повторно вводит какие-либо записи или выполняет агрегирование с состоянием (с явным отключением журнала изменений), то она создаст внутреннюю тему (имя темы будет начинаться с префикса, определяемого параметром `application.id`). Рекомендуемое значение — 3, оно допускает до двух отказов брокера.

`rocksdb.config.setter`

Пользовательский класс для настройки хранилищ состояний RocksDB. Этот параметр может быть интересен тем, кто занимается тонкой настройкой приложений с состоянием. Обычно нет необходимости настраивать RocksDB с применением пользовательского класса, но те, кому это интересно, найдут пример в официальной документации (<https://oreil.ly/AdQah>). Как правило, это не первая рукоятка, за которую следует дергать при попытках оптимизировать производительность, и я привел этот параметр здесь, потому что вы должны знать, что есть такой параметр, который можно попробовать настроить, когда нужно выжать максимум производительности из приложения, узким местом которого является RocksDB.

`state.dir`

Имя каталога (в виде абсолютного пути, например, `/tmp/kafka-streams`), в котором должны создаваться хранилища состояний. Этот параметр об-

¹ `exactly_once` создает производителя с поддержкой транзакций для каждого входного раздела. `exactly_once_beta` создает производителя с поддержкой транзакций для каждого потока выполнения. Возможность достижения семантики «точно один раз» с меньшим количеством производителей (параметр `exactly_once_beta`) помогает уменьшить объем памяти, занимаемой приложением, и количество сетевых подключений к брокерам.

суждается в разделе «Организация хранилища состояний на диске» главы 6. Важно помнить, что не должно существовать нескольких экземпляров приложения с одним и тем же `application.id`, пишущих в один и тот же каталог хранилища состояний.

`topology.optimization`

Присвойте этому параметру значение `all` (то есть `StreamsConfig.OPTIMIZE`), если хотите, чтобы Kafka Streams использовала некоторые внутренние оптимизации (например, уменьшала количество тем повторного секционирования, что может сэкономить на передаче данных по сети, или предотвращала создание ненужных разделов журнала изменений при создании `KTable` из исходной темы). На момент написания этих строк оптимизация топологии по умолчанию отключена.

`upgrade.from`

Этот параметр используется при обновлении Kafka Streams. Вопрос обновлений подробно обсуждается в подразделе «Обновление Kafka Streams» главы 12.

Конфигурационные свойства потребителей

Используя следующие префиксы, можно также настроить различных потребителей, применяемых Kafka Streams.

`main.consumer.`

Применяйте этот префикс для настройки *потребителя по умолчанию*, используемого для источников потока.

`restore.consumer.`

Применяйте этот префикс для настройки *потребителя восстановления*, используемого для восстановления хранилищ состояний из разделов журнала изменений.

`global.consumer.`

Этот префикс служит для настройки *глобального потребителя*, используемого для заполнения `GlobalKTable`.

ПРИЛОЖЕНИЕ Б

Настройка ksqlDB

ksqlDB поддерживает большинство конфигурационных параметров Kafka Streams и Kafka Client (то есть параметры производителей и потребителей). Рекомендуется при настройке параметров ksqlDB добавлять префикс `ksql.streams` к именам параметров, которые используются в Kafka Streams и Kafka Client. Например, чтобы настроить кэш записей, по аналогии с параметром `cache.max.bytes.buffering` в Kafka Streams, следует присвоить требуемое значение параметру `ksql.streams.cache.max.bytes.buffering` в файле `server.properties`. Аналогично для настройки параметра, подобного параметру `auto.offset.reset` в Kafka Consumer, следует использовать имя параметра `ksql.streams.auto.offset.reset`. Технически префикс необязателен, но рекомендуется создателями ksqlDB (Confluent).

В дополнение к стандартным параметрам Kafka Streams и Kafka Client ksqlDB поддерживает также параметры Kafka Connect, если используются функции интеграции данных ksqlDB (например, всякий раз, когда выполняется инструкция `CREATE {SOURCE|SINK} CONNECTOR`). Мы уже обсуждали это в разделе «Настройка рабочих процессов Connect» главы 9, поэтому более подробную информацию ищите там.

Наконец, есть несколько параметров, характерных для ksqlDB. Мы сгруппировали некоторые из наиболее важных параметров в две категории: параметры запросов и параметры сервера. Эту страницу следует использовать в качестве отправной точки для настройки ksqlDB. Используйте этот список лишь как старт для знакомства с различными параметрами, а за самой актуальной информацией, пожалуйста, обращайтесь к официальной документации ksqlDB¹.

¹ Официальная документация доступна по адресу <https://docs.ksqldb.io>. Описания параметров, представленные в этом приложении, были взяты непосредственно из документации.

Параметры запросов

Следующие конфигурационные параметры управляют различными аспектами выполнения запросов в ksqlDB.

`ksql.service.id`

Это *обязательное* свойство служит той же цели, что и параметр `application.id` в Kafka Streams: идентифицирует группу взаимодействующих приложений и позволяет распределять работу между экземплярами с одинаковыми идентификаторами. В отличие от `application.id` в Kafka Streams параметр `ksql.service.id` не имеет прямой корреляции с идентификатором группы потребителей.

Значение по умолчанию: нет.

`ksql.streams.bootstrap.servers`

Это *обязательное* свойство определяет список пар «хост/порт» брокеров Kafka, которые следует использовать для начального подключения к кластеру Kafka.

Значение по умолчанию: нет.

`ksql.fail.on.deserialization.error`

Если присвоить значение `true`, то ksqlDB будет прекращать обработку данных при невозможности десериализовать запись. Значение по умолчанию: `false`, то есть ksqlDB будет регистрировать ошибку десериализации и продолжать обработку. Если этот параметр кажется вам похожим на `default.deserialization.exception.handler` в Kafka Streams, то вы не ошиблись, так и есть. Одноименный параметр в ksqlDB используется для назначения соответствующего класса обработчика исключений.

Значение по умолчанию: `false`.

`ksql.fail.on.production.error`

В редких случаях можно столкнуться с проблемой при создании данных для темы Kafka. Например, временная проблема с сетью может вызвать исключение на стороне производителя. Если желательно, чтобы при возникновении исключения на стороне производителя обработка данных продолжилась (не рекомендуется в большинстве случаев), присвойте этому параметру значение `false`. Иначе оставьте значение по умолчанию.

Значение по умолчанию: `true`.

ksql.schema.registry.url

Если в запросах используется формат сериализации данных, для которого требуется Confluent Schema Registry (перечень таких форматов сериализации приводится в табл. 9.1), то присвойте этому параметру URL с адресом экземпляра Schema Registry.

Значение по умолчанию: нет.

ksql.internal.topic.replicas

Фактор репликации для внутренних тем, используемых сервером ksqlDB. Для производственных окружений следует присваивать значение ≥ 2 .

Значение по умолчанию: 1.

ksql.query.pull.enable.standby.reads

Когда количество резервных реплик $\geq 1^1$, этот параметр определяет возможность использования резервной реплики для обслуживания запросов на чтение, когда активная задача (которая обычно обрабатывает трафик чтения) не работает. Значение `true` может обеспечить высокую доступность pull-запросов.

Значение по умолчанию: `false`.

ksql.query.pull.max.allowed.offset.lag

Управляет максимальной задержкой (то есть количеством смещений в определенном разделе, которые не были прочитаны), допустимой для pull-запроса к таблице. Относится как к активным, так и к резервным задачам. Этот параметр активируется, только если в `ksql.lag.reporting.enable` установлено значение `true`.

Значение по умолчанию: `Long.MAX_VALUE`.

Параметры сервера

Следующие свойства используются для настройки экземпляров сервера ksqlDB.

ksql.query.persistent.active.limit

Максимальное количество постоянных запросов, выполняемых одновременно в интерактивном режиме. По достижении указанного предела пользова-

¹ Управляется параметром Kafka Streams и может быть установлено параметром `ksql.streams.num.standby.replicas`.

тели должны будут завершить один из запущенных постоянных запросов, чтобы запустить новый.

Значение по умолчанию: `Integer.MAX_VALUE`.

`ksql.queries.file`

Чтобы запустить ksqlDB в автономном режиме, укажите в этом параметре абсолютный путь к файлу с запросами для выполнения на сервере ksqlDB. Дополнительные сведения вы найдете в разделе «Режимы развертывания» главы 8.

Значение по умолчанию: нет.

`listeners`

Конечная точка REST API, на которой сервер ksqlDB будет принимать запросы.

Значение по умолчанию: по умолчанию используется URL `http://0.0.0.0:8088`, который соответствует всем интерфейсам IPv4. Чтобы организовать прием запросов на всех интерфейсах IPv6, присвойте параметру значение `http://[::]:8088`. Чтобы привязать сервер к конкретному интерфейсу, конкретизируйте значение.

`ksql.internal.listener`

Адрес взаимодействий между узлами. Этот параметр может пригодиться в случаях, когда требуется привязать отдельные адреса для внутренних и внешних конечных точек (например, чтобы защитить разные конечные точки отдельно на сетевом уровне).

Значение по умолчанию: первый адрес, определенный в свойстве `listeners`.

`ksql.advertised.listener`

Конечная точка, которая будет использоваться совместно с другими узлами ksqlDB в кластере для внутренней связи. В окружении IaaS может потребоваться, чтобы она отличалась от интерфейса, к которому привязан сервер.

Значение по умолчанию: значение `ksql.internal.listener` или, если этот параметр не определен, первый адрес, определенный в свойстве `listeners`.

`ksql.metrics.tags.custom`

Список тегов, которые должны включаться в метрики JMX в форме пар «ключ — значение»; обычно теги перечислены через запятую. Например, `key1:value1,key2:value2`.

Значение по умолчанию: нет.

Настройки безопасности

ksqlDB быстро развивается, а в области безопасности важно ссылаться на самую последнюю доступную информацию, поэтому мы не будем пытаться охватить всю широту параметров безопасности, а просто посоветуем обратиться к официальной документации ksqlDB (<https://oreil.ly/7RVCJ>).

Об авторе

Митч Сеймур — инженер и технический руководитель группы Data Services в Mailchimp. Используя Kafka Streams и ksqlDB, он создал несколько приложений для потоковой обработки, которые каждый день обрабатывают миллиарды событий с задержкой менее секунды. Активный участник сообщества пользователей и разработчиков программного обеспечения с открытым исходным кодом, пропагандирует технологии потоковой обработки на международных конференциях (Kafka Summit London, 2019), рассказывает о Kafka Streams и ksqlDB на местных встречах разработчиков и публикует свои статьи в блоге Confluent.

Иллюстрация на обложке

На обложке изображена райская рыба (*Macropodus opercularis*). В дикой природе райскую рыбу можно встретить в большинстве пресных водоемов Восточной Азии. Она также пользуется большой популярностью у аквариумистов.

У этих маленьких и агрессивных рыбок раздвоенный хвост. Тело окрашено в яркие полосы красного или оранжевого цвета, перемежающиеся синими или зелеными полосами. Они могут менять цвет, например, когда рыба вступает в бой с противником. Брюшные плавники всегда оранжевые. Взрослые особи могут достигать в длину 6,7 см, но чаще размер не превышает 5,5 см. Эти рыбы неприхотливы и выживают в разнообразной водной среде, но чаще предпочитают мелководье с густой растительностью.

Райские рыбы — хищники, питаются насекомыми, беспозвоночными и мальками рыб. Они воинственны, готовы преследовать друг друга и нападать на сородичей, а также на других мелких рыб. Подмечено, что их агрессивность усиливается по мере удаления от привычных мест обитания. Будучи распространенной аквариумной рыбой, райская рыба имеет природоохранный статус «Вызывающий наименьшие опасения», хотя загрязнение рек угрожает некоторым ее региональным популяциям.

Многие животные, изображаемые на обложках книг издательства O'Reilly, находятся под угрозой исчезновения, но они все важны для нашего мира.

Иллюстрацию для обложки нарисовала Карен Монтгомери (Karen Montgomery) на основе черно-белой гравюры из книги *Encyclopedie D'Histoire Naturelle*.

Митч Сеймур

Kafka Streams и ksqlDB: данные в реальном времени

Перевели с английского А. Киселев, И. Рузмайкина

Руководитель дивизиона	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>А. Пителимов</i>
Ведущий редактор	<i>Н. Гринчик</i>
Литературный редактор	<i>Н. Куликова</i>
Художественный редактор	<i>В. Мостипан</i>
Корректоры	<i>О. Андреевич, Е. Рафалюк, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 02.2023. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортёр в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 07.12.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 34,830. Тираж 1000. Заказ 0000.

Гвен Шапира, Тодд Палино, Раджини Сиварам, Крит Петти

АРАСНЕ КАФКА. ПОТОКОВАЯ ОБРАБОТКА И АНАЛИЗ ДАННЫХ

2-е издание



При работе любого enterprise-приложения образуются данные: это файлы логов, метрики, информация об активности пользователей, исходящие сообщения и т. п. Правильные манипуляции над всеми этими данными не менее важны, чем сами данные. Если вы архитектор, разработчик или выпускающий инженер, желающий решать подобные проблемы, но пока не знакомы с Apache Kafka, то именно отсюда вы узнаете, как работать с этой свободной потоковой платформой, позволяющей обрабатывать очереди данных в реальном времени.

КУПИТЬ