Kafka Streams в действии

Приложения и микросервисы в реальном времени с API Kafka Streams

Билл Беджек

Предисловие Ния Нархид

MANNING

Kafka Streams in Action

Real-time apps and microservices with the Kafka Streams API

William P. Bejeck Jr.
Foreword by Neha Narkhede

MANNING
Shelter Island

Билл Беджек

Предисловие Ния Нархид

Kafka Streams в действии

Приложения и микросервисы в реальном времени с API Kafka Streams

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону • Самара • Минск

2019

ББК 32.973.23
УДК 004.3
Б38

Беджек Билл

Б38 Kafka Streams в действии. Приложения и микросервисы для работы в реальном времени. — СПб.: Питер, 2019. — 304 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1201-2

Узнайте, как реализовать потоковую обработку на платформе Kafka! В этой книге рассмотрены реальные примеры сбора, преобразования и агрегации данных. Показана работа со множественными процессорами, обработка событий в режиме реального времени. Вы узнаете даже о потоковом SQL с KSQL! Эксплуатация и тестирование, мониторинг и отладка современных распределенных систем — вы получите всю необходимую информацию о самых сложных аспектах потоковой обработки. Kafka Streams API — ключ к эффективному применению Kafka на практике.

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

ББК 32.973.23
УДК 004.3

ISBN 978-1617294471 англ.
ISBN 978-5-4461-1201-2

Краткое содержание

Оглавление

ЧАСТЬ II. РАЗРАБОТКА С ПОМОЩЬЮ KAFKA STREAMS

ЧАСТЬ III. АДМИНИСТРИРОВАНИЕ KAFKA STREAMS

ЧАСТЬ IV. ПЕРЕДОВЫЕ ВОЗМОЖНОСТИ KAFKA STREAMS

ПРИЛОЖЕНИЯ

Предисловие

Я считаю, что в будущем архитектуры, ориентированные на потоки событий и обработку их в режиме реального времени, будут встречаться повсеместно. Технически продвинутые компании, например Netflix, Uber, Goldman Sachs, Bloomberg и др., уже настроили масштабную работу подобных больших платформ потоковой обработки событий. Как бы громко это ни прозвучало, но я считаю, что появление потоковой обработки и событийно-управляемых архитектур повлияет на модель использования компаниями данных так же сильно, как повлияли в свое время реляционные базы данных.

Событийно-ориентированный образ мыслей и создание событийно-управляемых приложений, ориентированных на потоковую обработку, требует определенной смены мировоззрения от тех, кто привык к приложениям в стиле «запрос/ответ» и к реляционным базам данных. Именно тут и пригодится книга *Kafka Streams*.

Потоковая обработка влечет фундаментальный переход от командно-ориентированного мышления к мышлению событийно-ориентированному — это перемена, позволяющая создавать быстро реагирующие, событийно-управляемые, расширяемые, гибкие приложения для работы в режиме реального времени. С точки зрения бизнеса событийно-ориентированное мышление открывает организациям дорогу к принятию решений и операциям в реальном времени с учетом контекста. С точки зрения технологии событийно-ориентированное мышление дает возможность создания более автономных и расцепленных приложений и, следовательно, адаптивно масштабируемых и расширяемых систем.

В обоих случаях конечной целью является увеличение скорости адаптации (agility) — как для бизнеса, так и для облегчающих его задачи технологий. В основе событийно-ориентированной архитектуры лежит внедрение событийно-ориентированного мышления в масштабах всей организации. А технологией, делающей возможным этот переход, выступает потоковая обработка.

Kafka Streams — нативная библиотека потоковой обработки Apache Kafka, предназначенная для создания событийно-управляемых приложений на языке Java. Kafka Streams позволяет приложениям выполнять сложные преобразования потоков данных, причем автоматически обеспечивает их отказоустойчивость и прозрачность, а также адаптивное распределение по экземплярам приложения. С момента

ее появления в 2016 году, в версии 0.10 Apache Kafka, множество компаний начало промышленную эксплуатацию Kafka Streams, включая Pinterest, «Нью-Йорк Таймс», Rabobank, LINE.

Наша цель относительно Kafka Streams и KSQL — упростить потоковую обработку до такой степени, чтобы создание событийно-управляемых приложений, реагирующих на события, стало естественным и не пришлось использовать тяжеловесные фреймворки для обработки больших данных. В нашей модели основной сущностью является не код, выполняющий обработку, а потоки данных в Kafka.

Руководство *Kafka Streams* — прекрасный способ изучить библиотеку Kafka Streams, а также понять, почему она представляет собой ключевое средство создания событийно-ориентированных приложений. Надеюсь, вы получите от прочтения этой книги не меньшее удовольствие, чем я!

Ния Нархид (Neha Narkhede),
соучредитель и технический директор
компании Confluent, одна из создателей
Apache Kafka

Введение

В бытность разработчиком программного обеспечения мне посчастливилось работать с современным ПО над интересными проектами. Поначалу я работал как над клиентскими, так и над серверными приложениями, но обнаружил, что мне больше нравится взаимодействовать исключительно с прикладной частью, так что этим я и занялся. Со временем я перешел к распределенным системам, начиная с Hadoop (тогда еще версии до 1.0). В очередном новом проекте я столкнулся с платформой Kafka. Сначала меня поразила простота работы с ней, а также ее возможности и гибкость. С каждым разом я находил все новые способы интеграции Kafka в процесс доставки данных проекта. Написание генераторов и потребителей оказалось элементарной задачей, а качество системы благодаря Kafka значительно улучшилось.

Затем я узнал о Kafka Streams. Я сразу сообразил: «Зачем мне лишний кластер для чтения данных из Kafka и немедленной их записи обратно?» Внимательно посмотрев на доступные API, я нашел все, что только нужно было для потоковой обработки: соединения, ассоциативные массивы, операции свертки и группировки. Но что важнее, подход Kafka Streams к добавлению состояния превосходил все решения, с которыми мне только приходилось работать.

Мне всегда нравилось объяснять что-то другим просто и понятно. Так что, когда появилась возможность написать о Kafka Streams, я знал, что эта сложная работа стоит того. Я надеюсь, что мой труд принесет свои плоды и в данной книге мне удастся продемонстрировать, что Kafka Streams — простой и в то же время изящный и эффективный способ осуществления потоковой обработки.

Благодарности

Прежде всего мне хотелось бы поблагодарить свою жену Бет и выразить ей признательность за всю поддержку, которую она мне оказывала в процессе написания книги. Написание книги — задача, занимающая немало времени, и без ее содействия мне никогда бы это не удалось. Бет, ты потрясающая, и я очень рад, что ты — моя жена. Я хотел бы также поблагодарить моих детей, которые терпеливо переносили сидение папы в офисе все выходные напролет, удовлетворяясь туманным ответом «скоро» на вопрос о том, когда же я закончу.

Далее я благодарю Гочжена Вана (Guozhang Wang), Маттиаса Сакса (Matthias Sax), Дэмиана Гая (Damian Guy) и Ино Терезку (Eno Thereska) — основных разработчиков библиотеки Kafka Streams. Без их гениальных озарений и упорного труда библиотека Kafka Streams просто не появилась бы и у меня не было бы возможности написать об этом революционном инструменте.

Я благодарю моего редактора из издательства «Мэннинг», Фрэнсис Лефковиц (Frances Lefkowitz), чьи советы эксперта и бесконечное терпение превратили написание данной книги *почти* в развлечение. Я также благодарен Джону Хайдаку (John Hyaduck) за меткие технические замечания и Валентину Креттазу (Valentin Crettaz), техническому корректору, за великолепную работу по анализу исходного кода. Кроме того, я говорю спасибо рецензентам, благодаря которым читатели этой книги получили продукт намного лучшего качества, за их непростую работу и бесценные отзывы — это Александер Кутмос (Alexander Koutmos), Боян Джуркович (Bojan Djurkovic), Дилан Скотт (Dylan Scott), Hamish Dickson (Хэмиш Диксон), Джеймс Фроннхофер (James Frohnhofer), Джим Мэнтили (Jim Manthely), Хосе Сан Лиандро (Jose San Leandro), Кэрри Коич (Kerry Koitzsch), Ласло Хегедюш (László Hegedüs), Мэтт Беланже (Matt Belanger), Мишель Аддучи (Michele Adduci), Николас Уайтхед (Nicholas Whitehead), Рикардо Хорхе Перейра Мано (Ricardo Jorge Pereira Mano), Робин Коу (Robin Coe), Сумант Тамбе (Sumant Tambe) и Венката Марапу (Venkata Marrapu).

Наконец, я хотел бы поблагодарить всех разработчиков фреймворка Kafka за создание столь превосходного программного продукта, особенно Джея Крепса (Jay Kreps), Нию Нархид (Neha Narkhede) и Дзюна Рао (Jun Rao), не только за саму идею Kafka в первую очередь, но и за основание компании Confluent — потрясающего и вдохновляющего места для работы.

Об этой книге

Я написал книгу *Kafka Streams*, чтобы познакомить вас с Kafka Streams и, в меньшей степени, вообще научить применять потоковую обработку. Я писал эту книгу с точки зрения парного программирования, представляя, что сижу рядом с вами, пока вы пишете код и изучаете API. Мы начнем с создания простого приложения и будем добавлять в него новый функционал по мере погружения в Kafka Streams. Вы узнаете, как выполнять тестирование и мониторинг, и, наконец, мы завершим книгу созданием продвинутого приложения Kafka Streams.

Кому стоит прочитать эту книгу

Эта книга подойдет для любого разработчика, который хочет разобраться в потоковой обработке. Понимание распределенного программирования поможет лучше изучить Kafka и Kafka Streams. Было бы неплохо знать и сам фреймворк Kafka, но это не обязательно: я расскажу вам все, что нужно. Опытные разработчики Kafka, как и новички, благодаря этой книге освоят создание интересных приложений для потоковой обработки с помощью библиотеки Kafka Streams. Java-разработчики среднего и высокого уровня, уже привычные к таким понятиям, как сериализация, научатся применять свои навыки для создания приложений Kafka Streams. Исходный код книги написан на Java 8 и существенно использует синтаксис лямбда-выражений Java 8, так что умение работать с лямбда-функциями (даже на другом языке программирования) вам пригодится.

Структура издания

Книга состоит из четырех частей, разбитых на девять глав. Часть I познакомит вас с ментальной моделью библиотеки Kafka Streams, чтобы дать комплексное представление о ее функционировании. В этих главах также приводятся основы Kafka для тех, кто ничего о ней не знает или хотел бы освежить свои знания.

- Глава 1 описывает историю вопроса: как и почему потоковая обработка стала необходимым элементом широкомасштабной обработки данных в режиме

реального времени. В ней также приводится ментальная модель Kafka Streams. Вместо демонстрации кода я просто опишу в ней, как работает Kafka Streams.

- Глава 2 — руководство для разработчиков, еще не имевших дела с Kafka. Те, у кого уже есть опыт работы с ней, могут эту главу пропустить и перейти непосредственно к Kafka Streams.

В части II я расскажу о Kafka Streams: начну с основ API и постепенно перейду к более продвинутым возможностям.

- В главе 3 вы найдете приложение типа Hello World, а далее и более реалистичный пример: разработку приложения для вымышленного розничного торговца, включая продвинутые возможности.
- В главе 4 обсуждается понятие состояния и объясняется, почему оно иногда необходимо для потоковых приложений. Вы узнаете из нее про реализации хранилищ состояния, а также выполнение соединений в Kafka Streams.
- Глава 5 посвящена дуализму таблиц и потоков, в ней вы познакомитесь с новым понятием: интерфейсом `KTable`. Если `KStream` представляет собой поток событий, то `KTable` — это поток взаимосвязанных событий или поток, предназначенный для обновления записей.
- В главе 6 мы углубимся в низкоуровневый API узлов-обработчиков (Processor API). До сих пор вы имели дело с высокоуровневым предметно-ориентированным языком (Domain Specific Language, DSL), а здесь научитесь использовать API узлов обработки на случай, если вам понадобится писать адаптированные к конкретной задаче части приложения.

В части III мы перейдем от разработки приложений Kafka Streams к управлению библиотекой Kafka Streams.

- Глава 7 рассказывает о тестировании приложений Kafka Streams. Вы научитесь тестировать топологию приложения в целом, осуществлять модульное тестирование отдельного узла обработки, а также использовать встроенный брокер для комплексных тестов.
- Глава 8 охватывает вопросы мониторинга приложений Kafka Streams с целью как оценки времени обработки записей, так и обнаружения потенциальных узких мест.

Часть IV завершает эту книгу, в ней мы займемся разработкой продвинутого приложения на основе Kafka Streams.

- Глава 9 рассказывает об интеграции существующих источников данных с Kafka Streams с помощью фреймворка Kafka Connect. Вы узнаете, как включать таблицы базы данных в потоковое приложение, а затем и как использовать интерактивные запросы для создания визуализаций и приложений — информационных панелей на основе проходящих через Kafka Streams данных без необходимости применения реляционных баз данных. В этой главе вы также познакомитесь с механизмом KSQL, благодаря которому с помощью Kafka можно выполнять непрерывные запросы без написания кода, используя один SQL.

О коде

Эта книга содержит множество примеров исходного кода как в пронумерованных листингах, так и внутри обычного текста. В обоих случаях исходный код отформатирован с помощью `такого моноширинного шрифта`, чтобы можно было отличить его от обычного текста.

Во многих случаях исходный код был переформатирован: добавлены разрывы строк и изменены отступы, чтобы оптимально использовать место на странице. В редких случаях даже этого было недостаточно и листинги содержат символы продолжения строки (➥). Кроме того, из исходного кода часто убраны комментарии, если он описывается в тексте. Многие листинги сопровождают пояснения к коду, подчеркивающие важные понятия.

Наконец, важно отметить, что многие из примеров кода несамодостаточны и содержат лишь фрагменты кода, наиболее важные для обсуждаемого вопроса. Все примеры из книги в их изначальном виде можно найти в прилагаемом к данной книге исходном коде. Он находится на GitHub по адресу https://github.com/bbejeck/kafka-streams-in-action или на сайте издательства: http://www.manning.com/books/kafka-streams-in-action.

Исходный код для книги представляет собой комплексный проект, использующий утилиту сборки Gradle (https://gradle.org). Этот проект можно импортировать в среды разработки IntelliJ или Eclipse с помощью соответствующих команд. Полные инструкции по применению исходного кода и навигации по нему можно найти в прилагаемом файле `README.md`.

Другие онлайн-ресурсы

- Документация по фреймворку Apache Kafka: https://kafka.apache.org/.
- Документация по платформе Confluent: https://docs.confluent.io/current.
- Документация по библиотеке Kafka Streams: https://docs.confluent.io/current/streams/index.html#kafka-streams.
- Документация по движку KSQL: https://docs.confluent.io/current/ksql.html#ksql.

Об авторе

Билл Беджек (Bill Bejeck) — участник проекта Kafka, работает в компании Confluent, в команде разработчиков Kafka Streams. Разработкой программного обеспечения он занимается более 13 лет, шесть из которых посвятил исключительно прикладной части, а именно обработке больших объемов данных. Билл также работал в командах, занимающихся вводом и обработкой данных, где использовал Kafka для повышения качества информационных потоков, отправляемых конечным потребителям. Билл — автор книги *Getting Started with Google Guava* (издательство Packt). Кроме того, он регулярно ведет блог «Беспорядочные размышления о написании кода» (http://codingjunkie.net/).

Часть I
Знакомство с Kafka Streams

В части I этой книги мы поговорим об эре больших данных: как она началась, когда появилась необходимость обрабатывать большие объемы данных и постепенно выросла до принципа потоковой обработки — обработки данных по мере их доступности. Мы также обсудим, что представляет собой библиотека Kafka Streams, и я покажу вам ментальную модель ее функционирования, без какого-либо кода, чтобы вы могли сконцентрироваться на общей картине происходящего. Мы также вкратце рассмотрим платформу Kafka, чтобы ввести вас в курс работы с ней.

1 Добро пожаловать в Kafka Streams

В этой главе:

- как большие данные повлияли на программирование;
- как работает потоковая обработка и зачем она нужна;
- знакомство с Kafka Streams;
- задачи, решаемые Kafka Streams.

С помощью этой книги вы научитесь использовать Kafka Streams для решения насущных проблем ваших потоковых приложений. Мы рассмотрим в ней компоненты Kafka Streams, начиная от простейших операций извлечения, преобразования и загрузки (extract, transform and load, ETL) и до сложных преобразований и соединений записей. После ее прочтения вы сможете легко решать подобные сложные задачи в своих потоковых приложениях.

Прежде чем углубиться в Kafka Streams, мы коротко рассмотрим историю обработки больших данных. По мере описания нами задач и их решений вы наглядно увидите, как возникла необходимость в Kafka, а затем и Kafka Streams. Давайте посмотрим, как началась эра больших данных и что привело к появлению Kafka Streams.

1.1. Движение больших данных, и как оно повлияло на программирование

Современное программирование начало бурно расти с появлением фреймворков и технологий больших данных. Конечно, разработка приложений на стороне клиента тоже подверглась изменениям, и число мобильных приложений также резко выросло. Но вне зависимости от размеров рынка мобильных устройств или развития технологий клиентских приложений неизменным остается одно: с каждым днем приходится обрабатывать все больше и больше данных. А по мере роста объемов данных с такой же скоростью растут потребности в их анализе и использовании в своих интересах.

Однако далеко не всегда достаточно возможности обрабатывать большие объемы данных крупными фрагментами *(пакетной обработки)*. У организаций все чаще возникает необходимость обрабатывать данные по мере поступления *(потоковой обработки)*. Библиотека Kafka Streams дает возможность обработки записей по событиям — это самый передовой подход к потоковой обработке. *Обработка по событиям* (per-event processing) означает, что каждая запись обрабатывается сразу же, как только оказывается доступна, никакой группировки данных в небольшие пакеты *(микропакетирования)* не нужно.

ПРИМЕЧАНИЕ

Когда потребности обработки данных стали совершенно явственными, была разработана новая стратегия — микропакетирование (microbatching). Как понятно из названия, оно представляет собой просто пакетную обработку, но с меньшими объемами данных. Благодаря снижению размера пакета микропакетирование позволяет иногда получать результаты быстрее, но это все равно пакетная обработка, хотя и с более короткими промежутками времени. Это не настоящая обработка по событиям.

1.1.1. Возникновение больших данных

Интернет начал всерьез влиять на нашу повседневную жизнь в середине 1990-х. С тех пор благодаря сетевой связи у нас появился беспрецедентный доступ к информации и возможность общения с кем угодно, в любой точке мира. Но побочным продуктом этой всей сетевой связи оказалась генерация колоссальных объемов данных.

Для наших целей можно считать, что эра больших данных началась в 1998 году, когда Сергей Брин (Sergey Brin) и Ларри Пейдж (Larry Page) создали компанию Google. Брин и Пейдж разработали новый способ ранжирования веб-страниц для поиска — алгоритм PageRank. Если не вдаваться в подробности, алгоритм PageRank оценивает сайт по количеству и качеству указывающих на него ссылок. При этом предполагается, что чем важнее или значимее веб-страница, тем больше сайтов будет на нее ссылаться.

Рисунок 1.1 наглядно иллюстрирует алгоритм PageRank.

- Сайт А — наиболее важный, поскольку на него указывает большинство ссылок.
- Сайт Б тоже довольно важен. Хотя на него указывает меньше ссылок, но среди них — важный сайт (А).
- Сайт В менее важен, чем А или Б. На сайт В указывает больше ссылок, чем на В, но их качество ниже.
- На сайты внизу рисунка (от Г до И) никакие ссылки не указывают, что делает их наименее ценными.

Алгоритм PageRank на рисунке сильно упрощен, но этого достаточно, чтобы получить основное представление о его работе.

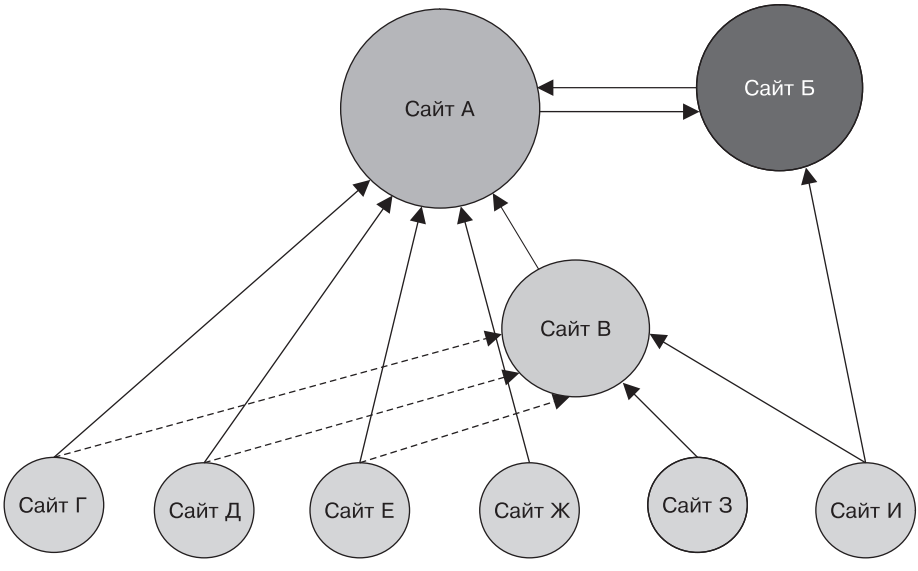


Рис. 1.1. Алгоритм PageRank в действии. Круги означают сайты, самые большие — те, на которые указывает больше ссылок с других сайтов

На то время алгоритм PageRank был поистине революционным подходом. До той поры поиски в Интернете чаще всего возвращали результаты на основе булевой логики. Если сайт содержал все искомые ключевые слова или большинство из них, то оказывался в результатах поиска вне зависимости от качества контента. Но для работы алгоритма PageRank в масштабах всего контента Интернета необходим был другой подход — традиционные подходы к работе с данными были слишком медленными. Чтобы выжить и развиваться, Google необходимо было быстро (где «быстро» — понятие относительное) индексировать весь этот контент и предоставить аудитории качественные результаты.

И компания Google разработала еще один революционный подход для обработки всех этих данных — парадигму MapReduce («отображение — свертка»). Парадигма MapReduce не только дала Google возможность выполнить всю необходимую работу, но и нечаянно породила целую новую отрасль вычислений.

1.1.2. Важнейшие понятия парадигмы MapReduce

Функции отображения (map) и свертки (reduce) не были чем-то новым на момент разработки компанией Google MapReduce. Уникальным в подходе Google было применение этих простых принципов в колоссальных масштабах на множестве машин.

По сути, MapReduce берет начало в функциональном программировании. Функция отображения получает на входе данные и отображает их во что-то без изменения исходных данных. Вот простой пример на языке Java 8, в котором объект `LocalDate`

отображается в сообщение типа `String`, причем исходный объект `LocalDate` остается неизменным:

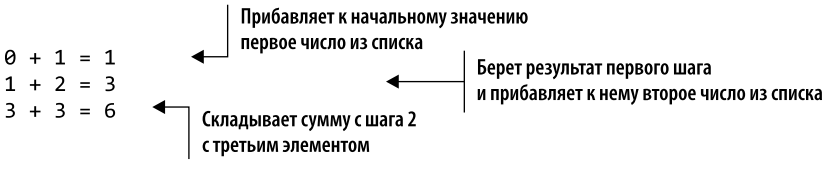
```
Function<LocalDate, String> addDate =
       (date) -> "The Day of the week is " + date.getDayOfWeek();
```

Хотя и очень простой, этот короткий пример достаточен для демонстрации функции отображения.

С другой стороны, функция свертки принимает на входе несколько параметров и свертывает их в единое (или по крайней мере меньшее по размеру) значение. Хороший пример свертки — сложение всех значений из набора чисел.

Для выполнения свертки набора чисел сначала необходимо задать начальное значение. В данном случае мы возьмем `0` (нейтральный по отношению к операции сложения элемент). Следующий шаг состоит в сложении этого начального значения с первым элементом списка. Функция свертки повторяет данный процесс до тех пор, пока не достигнет последнего значения, в результате чего получается одно число.

Ниже приведены шаги свертки объекта `List<Integer>`, содержащего значения `1`, `2` и `3`:



Как вы можете видеть, функция свертки сворачивает результаты для большей компактности. Как и в функции отображения, исходный список чисел остается неизменным.

Следующий пример демонстрирует реализацию простой функции свертки с помощью лямбда-функций Java 8:

```
List<Integer> numbers = Arrays.asList(1, 2, 3);

int sum = numbers.reduce(0, (i, j) -> i + j );
```

Наша книга посвящена не MapReduce, так что на этом мы завершим обсуждение данного побочного вопроса. Но какие-то из основных концепций, появившихся в парадигме MapReduce (и позднее реализованные в Hadoop, исходная версия которого, с открытым исходным кодом, была основана на официальном описании MapReduce от Google), применяются в Kafka Streams:

- распределение данных по кластеру так, чтобы процесс обработки хорошо масштабировался;
- использование пар «ключ/значение» и секций для группировки распределенных данных;
- вместо избегания сбоев — подготовка к ним с помощью репликации.

В следующих разделах мы обсудим эти концепции в общих словах. Обратите на них внимание, поскольку они еще не раз будут встречаться в данной книге.

Распределение данных по кластеру для лучшей масштабируемости

Работа с объемами данных порядка 5 Тбайт (5000 Гбайт) может оказаться для отдельной машины непосильной задачей. Но такую проблему можно минимизировать, разбив данные и воспользовавшись несколькими дополнительными машинами, чтобы каждая обрабатывала посильный ей объем данных. Это ясно видно из табл. 1.1.

Таблица 1.1. Как разбиение 5 Тбайт на части повышает пропускную способность обработки

Количество машин	Объем обрабатываемых каждым сервером данных, Гбайт
10	500
100	50
1000	5
5000	1

Как можно видеть из приведенной таблицы, вероятно, сначала объемы данных, которые нужно обработать, будут совершенно «неподъемны», но за счет распределения нагрузки по дополнительным серверам эти сложности можно устранить. Один гигабайт данных в последней строке таблицы — вполне посильный для обработки на обычном ноутбуке объем данных.

Это первая ключевая концепция MapReduce: за счет распределения нагрузки по кластеру машин можно превратить непомерные объемы данных во вполне доступные для обработки.

Использование пар «ключ/значение» и секций для группировки распределенных данных

Пара «ключ/значение» — простая структура данных с большими возможностями. В предыдущем разделе вы видели, какое значение может иметь распределение большого объема данных по кластеру машин. Распределение данных решает задачу обработки, но при этом возникает задача сбора распределенных данных обратно.

Для перегруппировки распределенных данных можно воспользоваться ключами из пар «ключ/значение» для секционирования данных. Термин *«секционирование»* подразумевает группировку, но я имею в виду группировку не по одинаковым ключам, а по ключам с одним хеш-кодом. Для разбиения данных на секции по ключу можно применить следующую формулу:

```
int partition = key.hashCode() % numberOfPartitions;
```

Рисунок 1.2 демонстрирует использование функции хеширования для получения данных по различным олимпийским видам спорта, хранящихся на отдельных серверах, и группировки их по секциям для различных видов спорта.

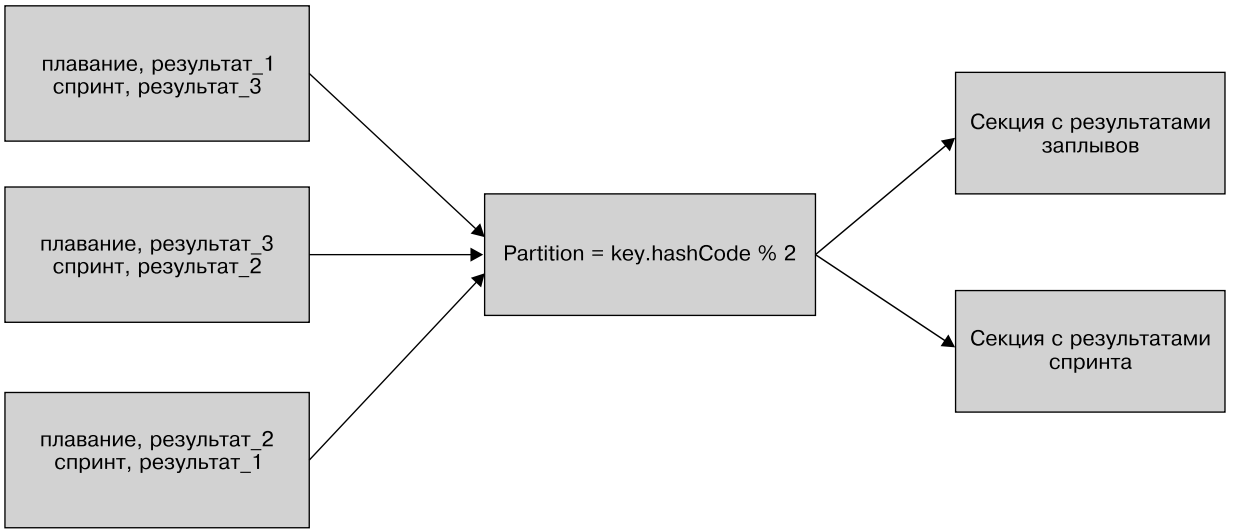


Рис. 1.2. Группировка записей по ключу по секциям. Хотя записи берутся с различных серверов, они все равно оказываются в соответствующих секциях

Все данные хранятся в виде пар «ключ/значение». На рисунке под ключами находятся названия событий, а значение представляет собой результат отдельного спортсмена.

Секционирование — важная концепция, в следующих главах вы увидите ее подробные примеры.

Готовимся к сбоям с помощью репликации

Еще один ключевой компонент MapReduce — файловая система Google (Google File System, GFS). Подобно тому как Hadoop — реализация MapReduce с открытым исходным кодом, так и файловая система Hadoop (Hadoop File System, HDFS) — реализация GFS с открытым исходным кодом.

Если не вдаваться в подробности, то как GFS, так и HDFS разбивают данные на блоки и распределяют эти блоки по кластеру. Но одна из неотъемлемых частей GFS/HDFS — их подход к обработке отказов серверов и дисков. Вместо того чтобы пытаться предотвращать сбои, фреймворк учитывает их возможность с помощью репликации блоков данных по кластеру (по умолчанию коэффициент репликации равен 3).

Благодаря репликации блоков данных на различных серверах больше не нужно бояться, что отказ дисков или даже сервера в целом приведет к простою в работе. Репликация данных критически важна для обеспечения отказоустойчивости распределенных приложений, которая, в свою очередь, необходима для их успешной работы. Далее мы увидим, как работает репликация в Kafka Streams.

1.1.3. Одной пакетной обработки недостаточно

Мода на Hadoop охватила мир обработки данных подобно лесному пожару. Он дал возможность обрабатывать огромные объемы данных отказоустойчивым образом с помощью стандартного аппаратного обеспечения (а значит, экономя средства). Но использование Hadoop/MapReduce, по сути, пакетный процесс, так что приходится собирать большие объемы данных, обрабатывать их, а затем сохранять результаты для дальнейшего применения. Пакетная обработка идеально подходит для алгоритмов, подобных PageRank, поскольку все равно нельзя принимать решения о ценности ресурсов в масштабах всего Интернета на основе наблюдения за переходами пользователей по ссылкам в режиме реального времени.

Но в коммерческой сфере все сильнее требуются как можно более оперативные ответы на такие важные вопросы, как:

- какие тенденции существуют в настоящий момент;
- сколько неудачных попыток входа в систему было сделано за последние 10 минут;
- насколько востребована пользователями только что выпущенная возможность?

Потребность в ином решении задачи была очевидна, и этим решением стала потоковая обработка.

1.2. Знакомство с потоковой обработкой

Существует много различных определений потоковой обработки. В этой книге я называю *потоковой обработкой* (stream processing) работу с данными по мере поступления их в систему. Определение можно уточнить, сказав, что потоковая обработка — способность оперировать бесконечным потоком данных по мере их движения посредством непрерывных вычислений без необходимости их сбора или сохранения для того, чтобы на них отреагировать.

На рис. 1.3 показан поток данных, где каждый круг на прямой отражает состояние данных в какой-либо момент времени. Информационный поток непрерывен, поскольку данные при потоковой обработке ничем не ограничены.

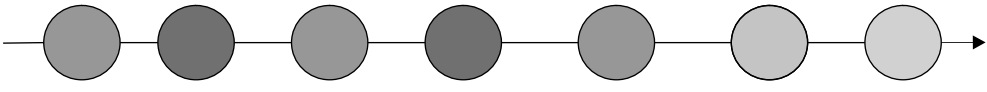


Рис. 1.3. Эта «жемчужная» диаграмма — упрощенное представление потоковой обработки. Каждый круг соответствует какой-либо информации или происходящему в конкретный момент времени событию. Количество событий ничем не ограничено, и они перетекают слева направо

Кому требуется потоковая обработка? Всем, кому нужна своевременная реакция на наблюдаемое событие. Рассмотрим несколько примеров.

Когда имеет смысл использовать потоковую обработку, а когда — нет. Как и любое техническое решение, потоковая обработка не подходит на все случаи жизни. Оптимальный сценарий использования для потоковой обработки — тот, где необходимо быстро реагировать на поступающие данные или уведомить об их поступлении. Вот несколько примеров.

- *Мошенничество с платежными картами.* Владелец платежной карты может не заметить ее кражи, но путем анализа покупок и сравнения их с устоявшимися паттернами (местоположение, общий характер потребительских расходов) можно заметить кражу платежной карты и оповестить ее владельца.
- *Обнаружение атак.* Благодаря анализу файлов журналов приложения после проникновения в систему можно предотвратить будущие атаки и повысить уровень безопасности, но это не отменяет критической важности мониторинга аномального поведения в реальном времени.
- *Финансовый сектор.* Для принятия удачных решений о покупке или продаже брокерам и потребителям необходимо иметь возможность отслеживать рыночные цены и тенденции в режиме реального времени.

С другой стороны, потоковая обработка подходит не для всех предметных областей. Для удачных прогнозов будущего поведения, например, необходимо обработать большое количество данных по длительному промежутку времени, чтобы исключить аномалии и выявить паттерны и тенденции. В следующих областях

разумнее анализировать данные за длительный промежуток времени, а не только текущие.

- *Экономическое прогнозирование*. Для точного прогноза, например, тенденций процентной ставки на рынке недвижимости необходимо собрать данные о множестве переменных за длительный период времени.
- *Изменения школьных программ обучения*. Только через один или два семестра администрация школы сможет оценить, достигли ли своей цели изменения программы обучения.

Важно помнить следующее: потоковая обработка — оптимальный подход для случая, когда выдать уведомление или предпринять какие-либо действия необходимо сразу же при поступлении данных. Если же требуется углубленный анализ или сбор большого архива данных для дальнейшего анализа, возможно, потоковая обработка не то, что вам требуется. Теперь посмотрим на конкретный пример потоковой обработки.

1.3. Обработка транзакции покупки товара

Начнем с применения общего подхода потоковой обработки к примеру с продажей товаров в розницу. А затем рассмотрим использование Kafka Streams для реализации потокового приложения.

Допустим, Джейн Доу едет с работы домой и вспоминает, что забыла купить зубную пасту. Она останавливается у магазина ZMart и заходит внутрь, берет зубную пасту и направляется к кассе для оплаты. Кассир спрашивает ее, член ли она ZClub, и сканирует ее карту участника. Теперь информация о том, что Джейн — член клуба постоянных покупателей, включена в транзакцию покупки товара.

После отображения суммы покупки Джейн передает кассиру свою дебетовую карту. Кассир проводит картой по терминалу и отдает Джейн чек. Выйдя из магазина, Джейн проверяет электронный почтовый ящик и обнаруживает там сообщение от ZMart с благодарностью за приверженность их магазину и различные купоны на скидки, которые можно использовать при следующем визите.

Эта транзакция — обычное событие, о котором покупатель не задумывается, но вы наверняка видите, какое изобилие информации в нем содержится, и благодаря ей ZMart может работать более эффективно и лучше обслуживать покупателей. Перенесемся в прошлое и посмотрим, как данная транзакция стала возможной.

1.3.1. Рассматриваем вариант с потоковой обработкой

Допустим, вы ведущий разработчик отдела ZMart, занимающегося потоковыми данными. ZMart — сеть розничных гипермаркетов, разбросанных по стране. Дела у ZMart идут отлично, ежегодные доходы превышают $1 млрд. Вы собираетесь провести интеллектуальный анализ данных о транзакциях компании, чтобы повы-

сить эффективность бизнеса. Вы знаете, что объемы данных продаж колоссальны, так что реализуемая технология должна работать быстро и хорошо масштабироваться, чтобы справиться с ними.

Вы решаете воспользоваться потоковой обработкой, поскольку некоторые коммерческие решения лучше принимать по горячим следам, сразу после транзакции, чтобы не упустить возможностей. Если данные собраны, ждать многие часы для принятия решений нет смысла. Вы проводите совещание с руководством и командой разработчиков и вырабатываете следующие четыре основных требования, необходимых для успеха проекта потоковой обработки.

- *Защита персональной информации*. Прежде всего ZMart ценит свои отношения с покупателями. При всей нынешней шумихе вокруг соблюдения конфиденциальности информации первоочередная задача — защитить персональную информацию покупателей, а главное — номера их платежных карт. Как бы ни использовалась информация о транзакциях, рисковать раскрытием информации о платежных картах покупателей недопустимо.
- *Поощрение покупателей*. При новой программе поощрения покупателей зарабатываемые покупателями бонусы зависят от количества потраченных на определенные товары денег. Необходимо быстро оповещать покупателей о получении бонусов, ведь мы хотим, чтобы они вернулись за следующими покупками! Опять же при этом требуется соответствующий мониторинг действий. Помните, как Джейн получила сообщение по электронной почте сразу же после выхода из магазина? Именно так все и должно работать.
- *Данные о продажах*. Компания ZMart не отказалась бы усовершенствовать свою стратегию рекламы и продаж. В частности, ее руководителям хотелось бы отслеживать продажи по областям, чтобы выяснить, какие товары популярнее в определенных частях страны. Цель — таргетинг продаж и организация рекламных акций по наиболее продаваемым в конкретных областях страны товарам.
- *Хранилище*. Все записи о продажах должны храниться во внешнем центре хранения для ретроспективного и ситуативного анализа.

Эти требования сами по себе довольно просты, но как реализовать их применительно к отдельной транзакции покупки, такой как у Джейн Доу?

1.3.2. Представление требований в виде графа

Предыдущие требования можно легко представить в виде *ориентированного ациклического графа* (directed acyclic graph, DAG). Момент завершения покупателем транзакции на кассе является узлом-источником всего графа. Требования ZMart становятся узлами-потомками главного узла-источника (рис. 1.4).

Далее необходимо разобраться, как отобразить транзакцию покупки на граф требований.

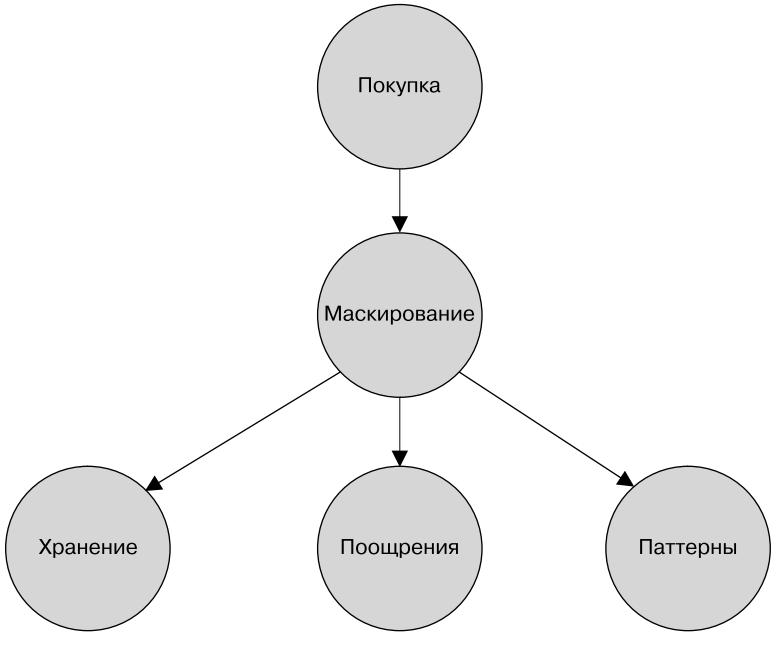


Рис. 1.4. Бизнес-требования к потоковому приложению в виде ориентированного ациклического графа. Каждая вершина соответствует требованию, а ребра отражают движение данных по графу

1.4. Транзакция покупки с другой точки зрения

В этом разделе мы пройдемся по этапам покупки и посмотрим, как они соотносятся, без особых подробностей, с графом требований с рис. 1.4. В следующем разделе мы выясним, как применить для данного процесса библиотеку Kafka Streams.

1.4.1. Узел-источник

Узел-источник графа (рис. 1.5) — место, в котором приложение потребляет транзакцию покупки. Этот узел служит источником информации о транзакциях покупок, проходящей по графу.

Рис. 1.5. Простое начало графа транзакций покупок. Данный узел является источником первичной информации о транзакциях покупок, которая будет перемещаться по графу

1.4.2. Узел маскирования номеров платежных карт

В дочернем узле источника графа происходит маскирование номеров платежных карт (рис. 1.6). Это первая вершина (узел) графа, отражающая бизнес-требования, и единственная получающая первичные данные о продажах из узла-источника, что делает ее, по сути, источником для остальных соединенных с ней узлов.

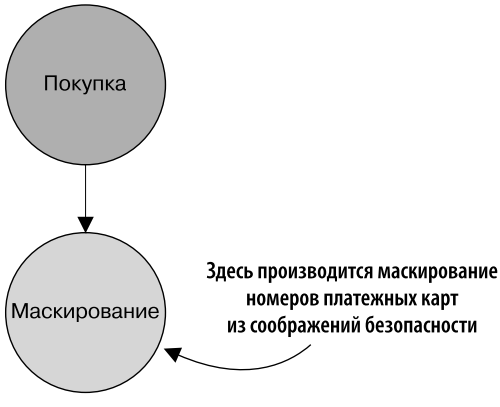


Рис. 1.6. Первый из узлов графа, отражающих бизнес-требования. Данный узел отвечает за маскирование номеров платежных карт, он единственный, кто получает первичные данные о продажах из узла-источника, что делает его, по существу, источником для остальных соединенных с ним узлов

При операции маскирования платежных карт делается копия данных с последующим преобразованием всех цифр номера в *x*, за исключением четырех последних. В данных во всех остальных частях графа поле номера платежной карты будет иметь вид *xxxx-xxxx-xxxx-1122*.

1.4.3. Узел паттернов

Узел «Паттерны» (рис. 1.7) извлекает соответствующую информацию с целью выяснения, в каких местах страны покупатели приобретают товары. Вместо копирования данных узел паттернов извлекает товар, дату и почтовый индекс (ZIP) продажи и создает новый объект с этими полями.

1.4.4. Узел поощрений

Следующий дочерний узел в процессе — сумматор бонусов (рис. 1.8). У компании ZMart есть программа поощрения покупателей, в соответствии с которой за приобретенные в ее магазинах товары покупателям начисляются бонусы. Роль данного узла состоит в извлечении потраченной клиентом суммы (в долларах) и его идентификатора и создании затем нового объекта, содержащего эти два поля.

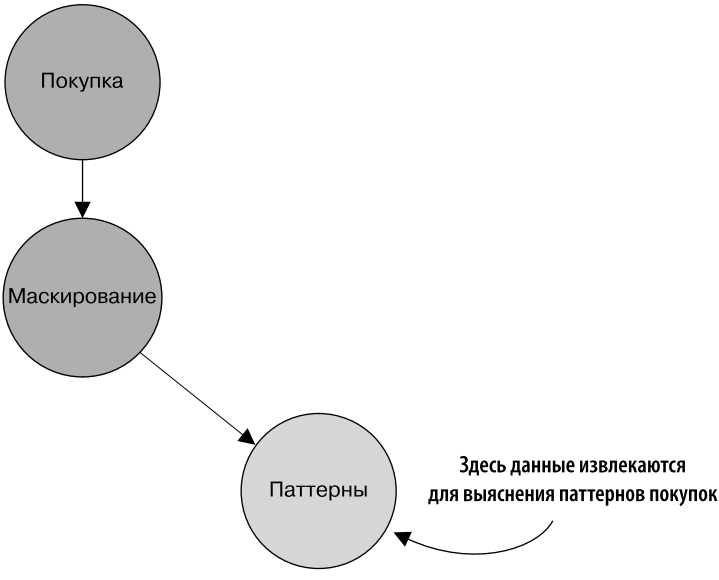


Рис. 1.7. Узел паттернов потребляет информацию о продажах из узла маскирования и преобразует ее в запись, указывающую, когда покупатель приобрел товар, и содержащую почтовый индекс места, где была завершена транзакция

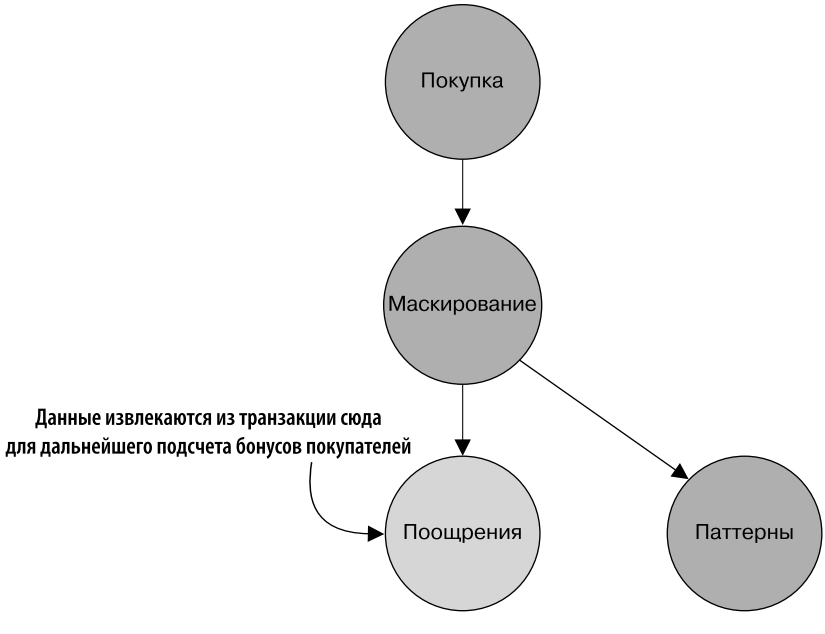


Рис. 1.8. Узел поощрений отвечает за потребление записей о продажах из узла маскирования и преобразование их в записи, содержащие общую сумму покупки и идентификатор покупателя

1.4.5. Узел хранения

Последний из дочерних узлов записывает данные о покупке в NoSQL-хранилище данных для дальнейшего анализа (рис. 1.9).

Мы проследили пример транзакции продажи по всему графу требований ZMart. Посмотрим теперь, как с помощью Kafka Streams преобразовать этот граф в полнофункциональное потоковое приложение.

Рис. 1.9. Узел хранилища также потребляет записи из узла маскирования. Эти записи не преобразуются в другой формат, а сохраняются в NoSQL-хранилище данных для дальнейшего ситуативного анализа

1.5. Kafka Streams как граф узлов обработки

Kafka Streams — библиотека, позволяющая выполнять обработку записей по мере поступления каждого события. С ее помощью можно обрабатывать данные при их поступлении без группировки в микропакеты. Каждая запись обрабатывается сразу же, как только оказывается доступна.

Большинство целей компании ZMart чувствительны ко времени в том смысле, что действия необходимо предпринимать при первой же возможности. В идеале

сбор информации должен производиться при появлении событий. Кроме того, магазинов ZMart в стране несколько, так что все записи о транзакциях следует сосредоточить в едином *потоке* данных для анализа. Именно поэтому библиотека Kafka Streams оптимальна для наших целей. Kafka Streams дает возможность обрабатывать записи по мере их поступления, обеспечивая требуемое низкое значение задержки.

В Kafka Streams необходимо задать *топологию* обрабатывающих *узлов* (я буду попеременно использовать как термин *(узел-)обработчик* (processor), так и термин *(обрабатывающий) узел* (node)). Источником для одного или нескольких узлов будет топик Kafka, причем можно добавлять дополнительные узлы, которые будут считаться дочерними (если вы плохо представляете себе, что такое топик Kafka, не волнуйтесь, я объясню все в главе 2). Любой дочерний узел может задавать свои дочерние узлы. Каждый из узлов обработки выполняет назначенную задачу, после чего переправляет запись каждому из своих дочерних узлов. Процесс выполнения заданий и последующей отправки данных дочерним узлам продолжается до тех пор, пока все дочерние узлы не выполнят свои задачи.

Не напоминает ли вам этот процесс что-либо? Должен напоминать, поскольку он схож с вышеприведенным преобразованием бизнес-требований ZMart в граф узлов обработки. Kafka Streams работает путем обхода графа типа DAG (то есть топологии узлов обработки).

Начнем мы с источника (родительского узла с одним или несколькими дочерними). Данные всегда перемещаются от родительского к дочерним узлам, и никогда — наоборот. У каждого дочернего узла могут быть, в свою очередь, свои дочерние узлы и т. д.

Записи перемещаются «вглубь» графа. Важные следствия такого подхода: каждая запись (пара «ключ/значение») *полностью* обрабатывается всем графом, прежде чем начинается перемещение следующей записи по топологии. А поскольку все записи проходят при обработке «вглубь» всего DAG, нет необходимости встраивать в Kafka Streams контроль обратного потока данных.

ОПРЕДЕЛЕНИЕ

> Существует несколько определений контроля обратного потока данных (backpressure), но тут мы обозначим это понятие как необходимость ограничения потока данных посредством буферизации или механизма блокирования. Контроль обратного потока данных необходим тогда, когда источник (source) генерирует данные быстрее, чем сток (sink) может их получать и обрабатывать.

Благодаря способности связывать или соединять цепочкой несколько узлов обработки можно быстро создавать сложную логику с сохранением относительной простоты отдельных компонентов. Именно при подобном объединении узлов обработки проявляются вся мощь и сложность Kafka Streams.

ОПРЕДЕЛЕНИЕ

Топология — способ организации частей системы и соединения их друг с другом. Говоря о топологии Kafka Streams, я подразумеваю преобразование данных посредством их пропускания через один или несколько узлов обработки.

Kafka Streams и Kafka

Как вы могли догадаться по названию, библиотека Kafka Streams — надстройка над фреймворком Kafka. В этой вводной главе вам не нужны знания Kafka, поскольку я сосредотачиваю ваше внимание на обобщенных принципах функционирования Kafka Streams. Возможно, вам встретится несколько относящихся к Kafka терминов, но в основном я сконцентрирую внимание на Kafka Streams с точки зрения потоковой обработки.

Если вы еще не сталкивались с платформой Kafka или не очень хорошо с ней знакомы, то всю информацию о ней, которая может вам понадобиться, вы найдете в главе 2. Знания Kafka жизненно важны для эффективной работы с Kafka Streams.

1.6. Использование Kafka Streams для потока данных транзакций покупок

Снова сформируем граф обработки, создав теперь программу Kafka Streams. Напомню, что на рис. 1.4 приведен граф требований для бизнес-требований компании ZMart. Напомню также: вершины — узлы обработки данных, а ребра отражают движение данных.

Хотя мы будем создавать программу Kafka Streams по мере создания нашего нового графа, речь все равно будет идти об относительно высокоуровневом подходе. Некоторые подробности мы опустим и вернемся к ним далее в данной книге, когда будем рассматривать реальный код.

Программа Kafka Streams будет потреблять записи, при этом мы станем преобразовывать первичные записи в объекты типа `Purchase`. Объект `Purchase` будет состоять из следующих элементов данных:

- идентификатора покупателя ZMart (полученного путем сканирования карты постоянного покупателя);
- итоговой потраченной суммы в долларах;
- купленных товаров;
- почтового индекса магазина, где была произведена покупка;
- даты и времени транзакции;
- номера дебетовой или кредитной карты.

1.6.1. Задаем источник

Первый этап при создании любой программы Kafka Streams — задание источника для потока данных. Источник может представлять собой что-то из следующего:

- отдельный топик Kafka;
- несколько топиков Kafka в виде разделенного запятыми списка;
- регулярное выражение, соответствующее одному или нескольким топикам.

В данном случае роль источника будет играть отдельный топик под названием `transactions`. Если какие-либо из этих терминов Kafka вам незнакомы, помните — обо всех них будет рассказано в главе 2.

Рис. 1.10. Узел-источник: топик Kafka

Важно отметить, что для фреймворка Kafka программа Kafka Streams выглядит точно так же, как и любое другое сочетание потребителей и генераторов. Вместе с вашей потоковой программой из одного топика может читаться произвольное количество приложений. Рисунок 1.10 демонстрирует узел-источник в топологии.

1.6.2. Первый узел-обработчик: маскирование номеров платежных карт

После задания источника можно приступить к созданию узлов-обработчиков, которые будут работать с данными. Первая наша задача — наложить маски на номера платежных карт из входящих записей о покупках. Первый наш узел-обработчик будет преобразовывать номера платежных карт из вида 1234-5678-9123-2233 в xxxx-xxxx-xxxx-2233.

Представленное на рис. 1.11 маскирование будет осуществляться с помощью метода `KStream.mapValues`. Он будет возвращать новый экземпляр класса `KStream` со значениями, маскированными в соответствии с указанным в `ValueMapper`. Этот экземпляр `KStream` будет родительским узлом-обработчиком для всех остальных создаваемых вами узлов-обработчиков.

Создание топологий узлов-обработчиков. При каждом создании нового экземпляра `KStream` с помощью метода преобразования вы, по существу, создаете новый узел-обработчик, соединенный с другими, уже существующими узлами-обработчиками. Kafka Streams дает возможность путем объединения узлов-обработчиков элегантно создавать сложные информационные потоки.

Важно отметить, что вызов метода, возвращающего новый экземпляр `KStream`, не приводит к приостановке потребления данных исходным экземпляром. Метод преобразования создает новый узел-обработчик и добавляет его к уже существующей топологии узлов-обработчиков. Далее эта дополненная топология используется в качестве параметра при создании нового экземпляра `KStream`, начинающего получать сообщения с момента создания.

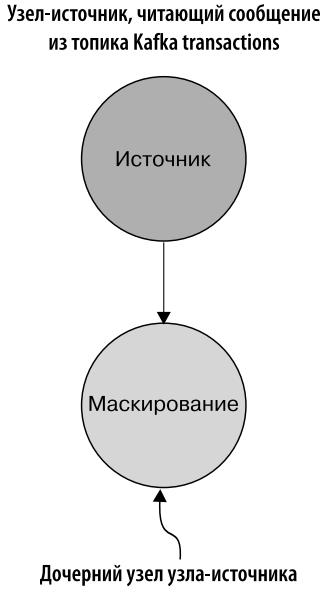


Рис. 1.11. Узел-обработчик маскирования — дочерний узел главного узла-источника. Он получает все первичные данные о транзакциях продаж и выдает новые записи с маскированными номерами платежных карт

Скорее всего, вы будете создавать новые экземпляры `KStream` для выполнения дополнительных преобразований, оставляя исходный поток данных для первоначальной его задачи. Вы увидите пример этого дальше, когда мы будем задавать второй и третий узлы-обработчики.

Интерфейс `ValueMapper` можно использовать и для приведения входящих значений к совершенно новому типу, но в таком случае он будет возвращать обновленную копию объекта `Purchase`. Применение метода отображения для изменения объекта — паттерн, с которым вы часто будете сталкиваться в дальнейшем.

У вас, полагаю, уже должно было сложиться четкое представление о методике создания конвейеров узлов-обработчиков для преобразования и вывода данных.

1.6.3. Второй узел-обработчик: паттерны покупок

Следующий узел-обработчик, который мы создадим, будет предназначен для захвата информации, необходимой для выявления паттернов покупок в различных регионах страны (рис. 1.12). Для этого мы добавим дочерний обрабатывающий узел к первому созданному нами узлу-обработчику. Данный первый узел-обработчик генерирует объекты `Purchase` с замаскированными номерами платежных карт.

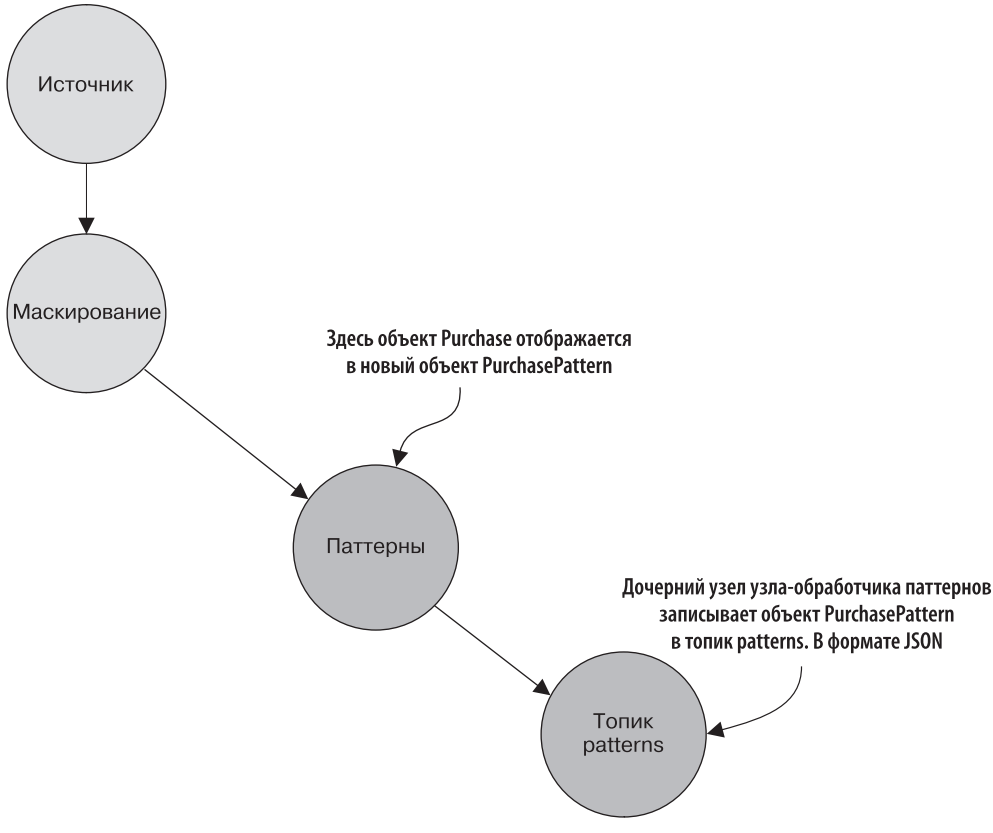


Рис. 1.12. Узел-обработчик паттернов покупок принимает на входе объекты Purchase и преобразует их в объекты PurchasePattern, содержащие информацию о купленных товарах и почтовые индексы магазинов, в которых были осуществлены транзакции. Новый узел-обработчик получает записи от узла-обработчика паттернов и записывает их в топик Kafka

Узел-обработчик паттернов покупок получает на входе от родительского узла объект `Purchase` и отображает его в новый объект `PurchasePattern`. Процесс отображения включает извлечение информации о приобретенном товаре (например, зубной пасте) и почтовом индексе его покупки и создание на основе этой информации объекта `PurchasePattern`. Мы рассмотрим процесс отображения во всех подробностях в главе 3.

Далее узел-обработчик паттернов покупок добавляет в топологию дочерний узел-обработчик, который получает на входе новый объект `PurchasePattern` и записывает его в топик Kafka `patterns`. Объект `PurchasePattern` при записи в тему преобразуется в форму, подходящую для передачи данных, так что потом другие приложения смогут потреблять эту информацию и использовать ее для выяснения уровней запасов на складах, а также потребительских тенденций в заданном регионе.

1.6.4. Третий узел-обработчик: поощрение покупателей

Третий узел-обработчик извлекает информацию, предназначенную для программы поощрения постоянных покупателей (рис. 1.13). Это тоже дочерний узел-обработчик начального узла-обработчика. Он получает на входе объекты `Purchase` и отображает их в объекты другого типа — `RewardAccumulator`.

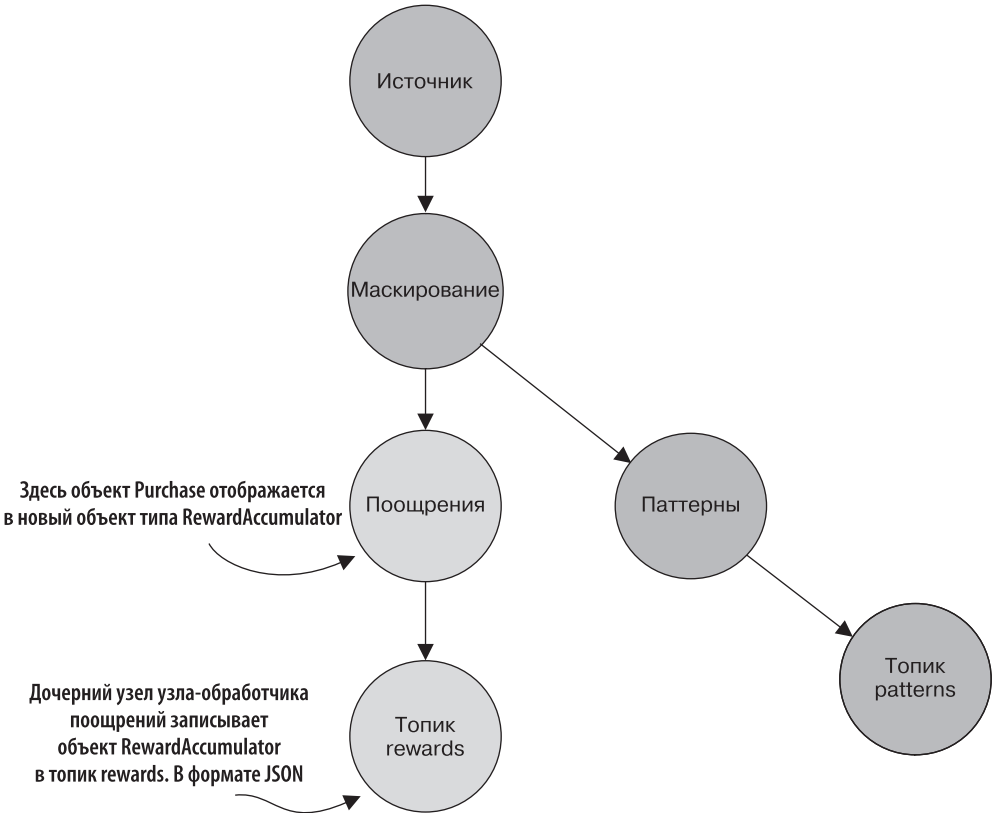


Рис. 1.13. Узел-обработчик поощрений покупателей отвечает за преобразование объектов Purchase в объекты типа RewardAccumulator, содержащие идентификатор покупателя, дату и сумму транзакции в долларах. Дочерний узел-обработчик записывает объекты RewardAccumulator в другой топик Kafka

Узел-обработчик поощрений покупателей также добавляет дочерний узел-обработчик для записи объектов `RewardAccumulator` в топик Kafka `rewards`. Другие приложения могут посредством потребления записей из топика `rewards` определить число бонусов клиентов ZMart и сгенерировать, например, сообщения электронной почты, подобные тому, которое получила Джейн Доу.

1.6.5. Четвертый узел-обработчик: запись данных о покупках

Последний из наших узлов-обработчиков показан на рис. 1.14. Это третий дочерний узел узла-обработчика маскирования, он заносит полные маскированные записи о покупках в тему Kafka под названием `purchases`. В дальнейшем приложение хранения NoSQL будет потреблять записи из данного топика по мере их поступления. Эти записи пригодятся для последующего анализа.

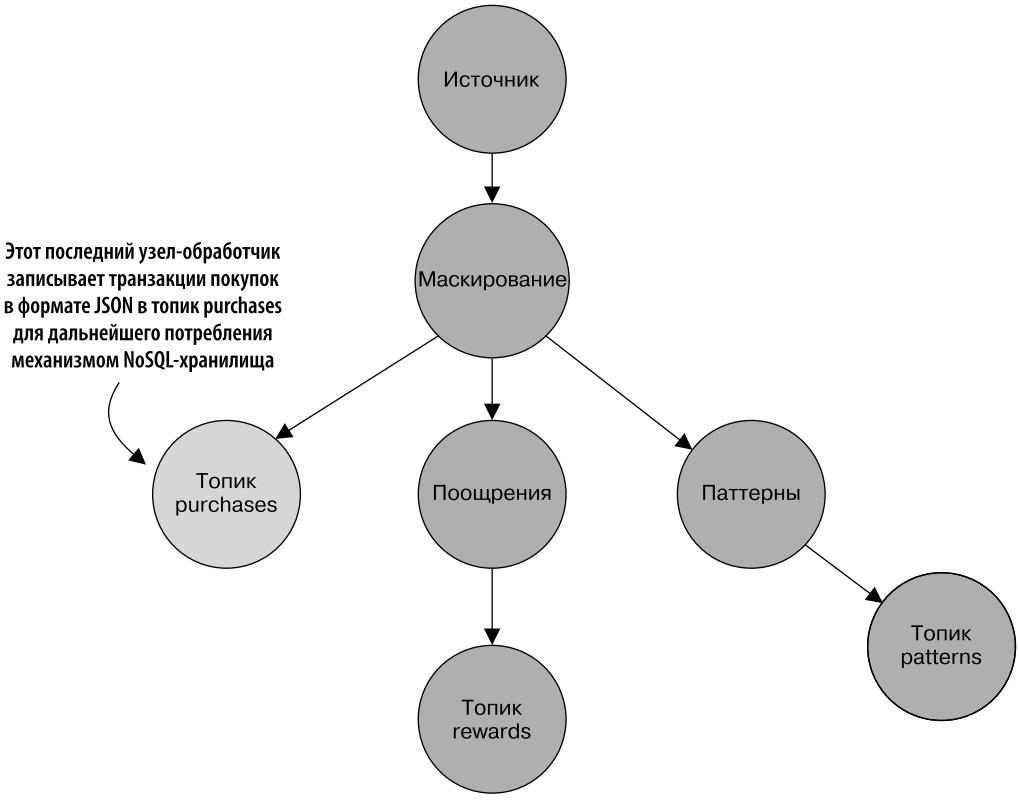


Рис. 1.14. Последний узел-обработчик отвечает за запись объектов Purchase целиком в еще один топик Kafka. Потребитель этого топика будет сохранять результаты в NoSQL-хранилище, например MongoDB[1]

Как вы можете видеть, первый узел-обработчик, маскирующий номера платежных карт, служит источником данных для трех других узлов-обработчиков: двух, которые далее улучшают структуру данных или преобразуют их, и одного, записы-

[1] Автор почему-то не выделяет на этом и последующих рисунках отдельный узел хранения, сливая его воедино с узлом для топика purchases. См. рис. 1.4. — *Примеч. пер.*

вающего маскированные результаты в топик для дальнейшего использования другими потребителями. С помощью Kafka Streams можно создавать графы обработки из связанных узлов, обладающие большими возможностями потоковой обработки входящих данных.

Резюме

- Kafka Streams представляет собой граф узлов обработки, объединенных в единое целое для сложной полнофункциональной потоковой обработки.
- Пакетная обработка обладает большими возможностями, но ее недостаточно для удовлетворения потребностей при работе с данными в режиме реального времени.
- Распределение данных, пары «ключ/значение», секционирование и репликация данных критически важны для распределенных приложений.

Чтобы хорошо разобраться с библиотекой Kafka Streams, вам понадобятся определенные значения платформы Kafka. Для тех, кому Kafka в новинку, мы рассмотрим основные ее принципы в главе 2:

- установку Kafka и отправку сообщений;
- архитектуру Kafka и распределенные журналы;
- топики и их использование в Kafka;
- функционирование генераторов и потребителей и эффективное их написание.

Если вы уже хорошо знакомы с Kafka, то можете спокойно перейти прямо к главе 3, где мы создадим приложение Kafka Streams на базе обсуждавшегося в данной главе примера.

2 Kafka в двух словах

В этой главе:

- архитектура Kafka;
- отправка сообщений с помощью генераторов;
- чтение сообщений с помощью потребителей;
- установка и запуск Kafka.

Хотя данная книга посвящена библиотеке Kafka Streams, она была бы неполной без обсуждения самого фреймворка Kafka. В конце концов, Kafka Streams — надстройка над Kafka.

Библиотека Kafka Streams спроектирована очень удачно, так что для подготовки ее к работе практически не нужно какого-либо опыта работы с Kafka, но неумение выполнить тонкую настройку Kafka может серьезно ограничить ваши успехи в работе с ней. Чтобы извлечь максимальную пользу из Kafka Streams, совершенно необходимы глубокие базовые знания Kafka.

ПРИМЕЧАНИЕ

> Данная глава предназначена для разработчиков, желающих познакомиться с Kafka Streams, у которых — совсем или почти — нет опыта работы с Kafka. Если вы уже достаточно работали с Kafka на практике, то можете спокойно пропустить эту главу и перейти прямо к главе 3.

Kafka — слишком объемная тема, чтобы охватить ее во всей полноте в одной главе. Я расскажу достаточно, чтобы вы хорошо разобрались в работе Kafka и нескольких основных настройках ее конфигурации, которые вам понадобятся. Для более подробного описания Kafka загляните в книгу *Kafka in Action* Дилана Скотта (Dylan Scott) (издательство Manning, 2018).

2.1. Проблема данных

В настоящее время организации практически завалены данными. Интернет-компании, финансовые учреждения и крупные розничные торговцы имеют сейчас больше возможностей для использования этих данных, чем когда бы то ни было, как для

лучшего обслуживания своих клиентов, так и для поиска лучших способов ведения бизнеса. (Мы, как оптимисты, будем предполагать, что изучение данных клиентов происходит лишь с самими лучшими намерениями.)

Вероятно, вам хотелось бы, чтобы программное решение для управления данными ZMart удовлетворяло таким требованиям, как:

- наличие способа быстрой отправки данных в центральное хранилище;
- возможность репликации данных, чтобы неизбежные частые отказы машин не приводили к простою и потерям данных;
- потенциал масштабирования до произвольного количества потребителей данных без необходимости отслеживания различных приложений. Данные должны быть доступны всем работникам компании без необходимости отслеживания того, кто просматривал данные, а кто — нет.

2.2. Использование Kafka для обработки данных

В главе 1 вы познакомились с большой розничной компанией ZMart. На тот момент ZMart требовалась потоковая платформа обработки данных об их продажах, чтобы усовершенствовать свою работу с покупателями и повысить продажи в целом. Но за шесть месяцев до этого ZMart пыталась хоть как-то справиться со своими проблемами с данными. У нее в наличии было созданное специально для них программное решение, сначала работавшее нормально, но затем вышедшее из-под контроля по причинам, которые вам скоро станут понятны.

2.2.1. Первоначальная платформа данных компании ZMart

Первоначально ZMart была маленькой компанией, в систему которой данные по розничным продажам поступали из отдельных приложений. Сначала такая схема работала нормально, но спустя время стало ясно, что нужно что-то новое. Данные по продажам одного отдела интересны не только этому отделу, но и нескольким другим подразделениям компании, у каждого из которых разное представление о том, что важно, и о желаемой структуре данных. На рис. 2.1 показана первоначальная платформа данных компании ZMart.

Со временем ZMart выросла за счет приобретения других компаний и расширения предложений в существующих магазинах. При каждом увеличении ее размера связи между приложениями становились все более сложными. То, что начиналось как горстка взаимодействующих друг с другом приложений, превратилось в настоящую кучу спагетти. Как вы можете видеть на рис. 2.2, всего лишь между тремя приложениями — огромное количество соединений, что создает сильную путаницу. Очевидно, что при добавлении новых приложений эта структура данных быстро выйдет из-под контроля.

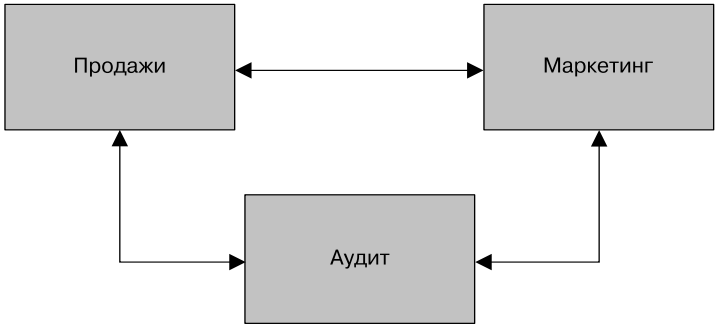


Рис. 2.1. Первоначальная архитектура данных компании ZMart была достаточно проста для того, чтобы поток информации в каждый источник или из него был линейным

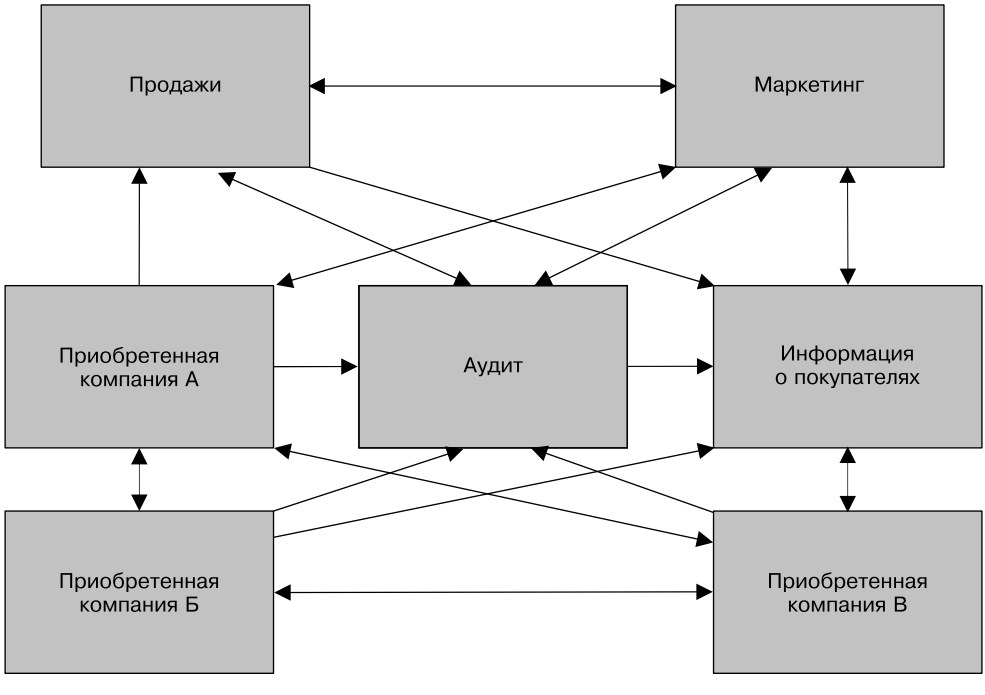


Рис. 2.2. После добавления со временем еще нескольких приложений соединения источников информации становятся более запутанными

2.2.2. Концентратор информации о транзакциях продаж на основе Kafka

Решением проблемы ZMart было бы создание одного процесса-приемника для всех данных о транзакциях — концентратора информации о транзакциях. Этот концентратор информации о транзакциях должен быть без сохранения состояния, в его

задачи входит получение информации о транзакциях и сохранение ее так, чтобы любое приложение-потребитель могло ее извлечь при необходимости. При этом отслеживать, какую информацию они уже видели, должны сами приложения-потребители. Концентратору информации о транзакциях будет известно только, сколько времени в нем хранятся данные о конкретной транзакции и когда их необходимо размножить или удалить.

Если вы еще не поняли, наш пример — идеальный сценарий использования для Kafka. Kafka — это отказо- и ошибкоустойчивая система обмена сообщениями по типу «публикация/подписка». Отдельный узел Kafka называется *брокером* (broker), а несколько серверов Kafka образуют *кластер* (cluster). Записываемые *генераторами* (producers) сообщения Kafka хранит в *топиках* (topics). *Потребители* (consumers) подписываются на топики и обращаются к Kafka, чтобы узнать, появились ли новые сообщения в топиках, на которые они подписаны. Рисунок 2.3 иллюстрирует использование Kafka в качестве концентратора информации о транзакциях.

Это был обзор Kafka с высоты птичьего полета. В следующих разделах мы взглянем на нее поближе.

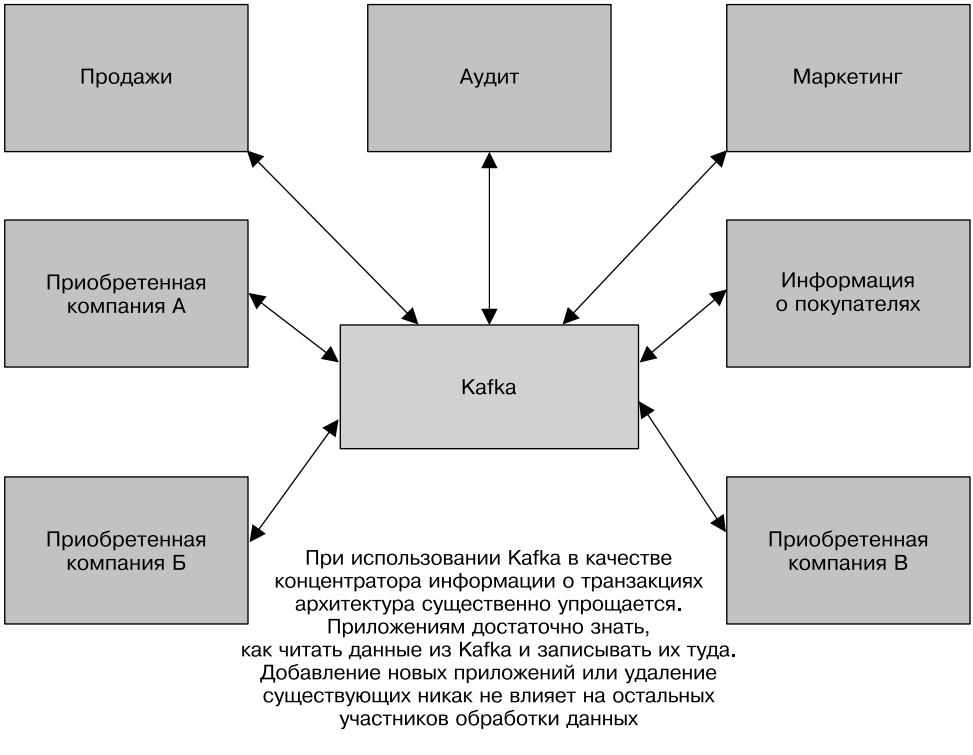


Рис. 2.3. Использование Kafka в качестве концентратора информации о транзакциях существенно упрощает архитектуру ZMart. Теперь каждой машине больше не нужно знать обо всех остальных источниках информации. Достаточно знать, как читать данные из Kafka и записывать их туда

2.3. Архитектура Kafka

В следующих нескольких подразделах мы рассмотрим важнейшие элементы архитектуры Kafka и разберемся, как она функционирует. Если вы хотите поскорее опробовать Kafka, то переходите прямо к разделу 2.6, посвященному установке и запуску Kafka. После установки можете вернуться обратно и продолжить ее изучение.

2.3.1. Kafka — это брокер сообщений

Ранее я говорил, что Kafka — система обмена сообщениями по типу «публикация/подписка», но точнее было бы сказать, что Kafka — это брокер сообщений. *Брокер* (broker) — это посредник, который сводит вместе две, возможно, незнакомые друг с другом стороны для обоюдовыгодного товарообмена или заключения сделки. Рисунок 2.4 показывает дальнейшее развитие архитектуры данных компании ZMart. В него были добавлены генераторы и потребители для демонстрации взаимодействия отдельных частей архитектуры с Kafka. Друг с другом они непосредственно не взаимодействуют.

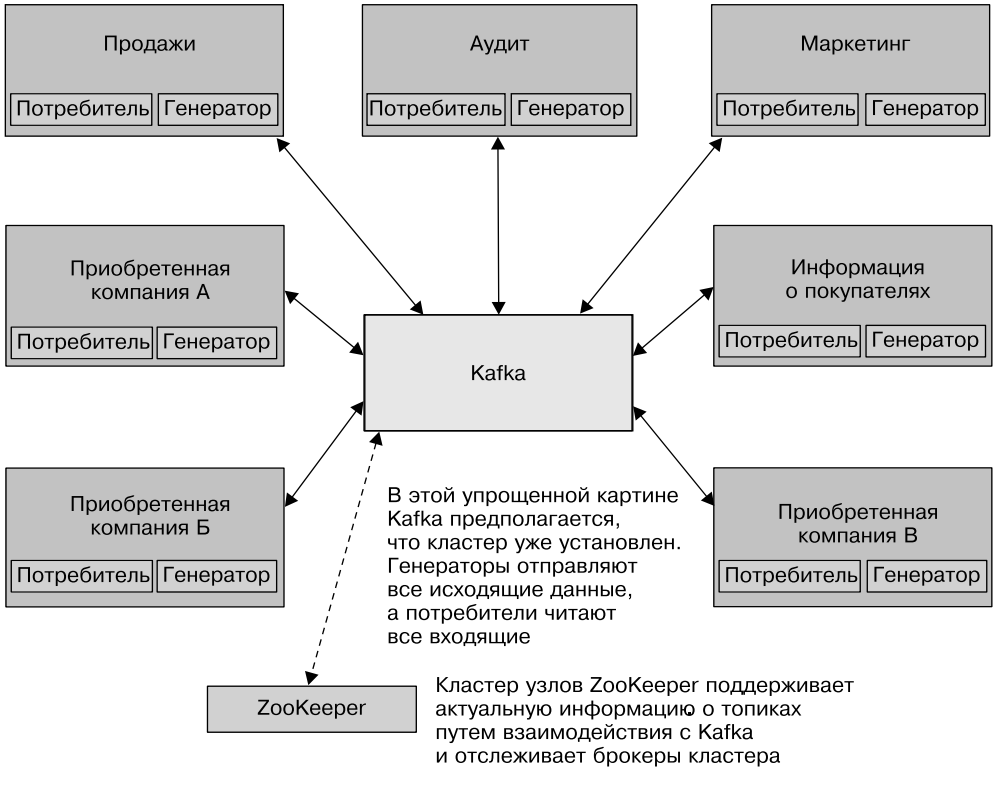


Рис. 2.4. Kafka — брокер сообщений. Генераторы отправляют в Kafka сообщения, которые там сохраняются. Потребители могут получать эти сообщения посредством подписки на топики

Kafka хранит сообщения в топиках и извлекает их оттуда. Генераторы и потребители сообщений никак непосредственно не связаны. Кроме того, Kafka не хранит состояние генераторов и потребителей, а играет роль исключительно координационного центра для сообщений.

По существу, «под капотом» топик Kafka представляет собой *журнал* (log), то есть файл, в который Kafka дописывает входящие записи. Чтобы справиться с поступающим в топик объемом сообщений, Kafka использует секции. Мы уже обсуждали секции в главе 1, и вы, наверное, помните, что они, в частности, применяются для объединения расположенных на различных машинах данных. Вскоре мы поговорим о секциях подробнее.

2.3.2. Kafka — это журнал

В основе Kafka лежит механизм *журналов*. Большинству разработчиков ПО хорошо знакомо отслеживание с помощью журналов действий приложений. В случае проблем с производительностью или ошибками в приложении вы прежде всего заглянете в его журналы. Но это совсем другие журналы. В контексте Kafka (или любой другой распределенной системы) журналом называется «предназначенная только для дописывания в ее конец, полностью упорядоченная по времени последовательность записей»[1].

На рис. 2.5 приведен пример журнала. Приложение добавляет записи в конец журнала по мере их поступления. Записи упорядочиваются по времени, хотя меток даты/времени в них может и не быть, просто слева располагаются записи, поступившие первыми, а справа — последними.

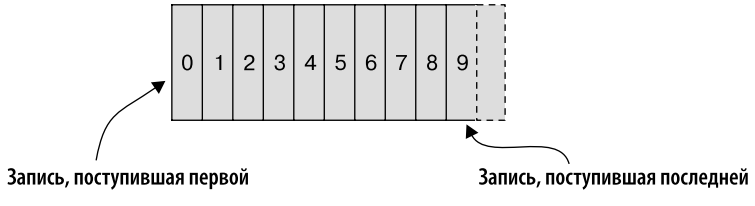


Рис. 2.5. Журнал — это файл, в котором входящие записи добавляются в конец, каждая поступающая запись помещается непосредственно за последней из полученных ранее. Этот процесс приводит к упорядочению записей по времени

Журналы — простая абстракция данных, имеющая очень большое значение. При упорядоченных по времени записях разрешение конфликтов и выбор нужного обновления для различных машин сильно упрощаются: просто выбирается последняя запись.

---

[1] *Kreps J.* The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction (Джей Крепс «Журнал: что каждый разработчик ПО должен знать об абстракции, объединяющей всю обработку поступающих в реальном времени данных»), http://mng.bz/eE3w.

Топики в Kafka представляют собой журналы, разбитые в соответствии с названием топика. Топики можно рассматривать практически как маркированные журналы. Если журнал реплицируется среди кластера машин и одна из этих машин выходит из строя, можно легко восстановить ее состояние, просто снова выполнив действия из файла журнала. Распределенные журналы фиксаций служат именно для восстановления после сбоя.

Мы затронули лишь самый краешек очень сложной темы, связанной с распределенными приложениями и согласованностью данных, но изложенное, надеюсь, дало вам хотя бы общее представление о том, что происходит «под капотом» Kafka.

2.3.3. Функционирование журналов в Kafka

Один из параметров конфигурации, задаваемых при установке Kafka, — `log.dir`, определяющий место хранения журналов Kafka. Каждый топик соответствует подкаталогу в указываемом в этом параметре каталоге. Количество подкаталогов равно числу секций топиков, формат их названий: `имя-секции_номер-секции` (я расскажу о секциях в следующем разделе). В каждом каталоге находится файл журнала, в который добавляются входящие сообщения. После достижения файлом журнала определенного размера (в смысле числа записей или размера на диске) или по достижении заданной (в конфигурации) разницы между метками даты/времени сообщений файл журнала заменяется и Kafka начинает добавлять сообщения в новый журнал (рис. 2.6).

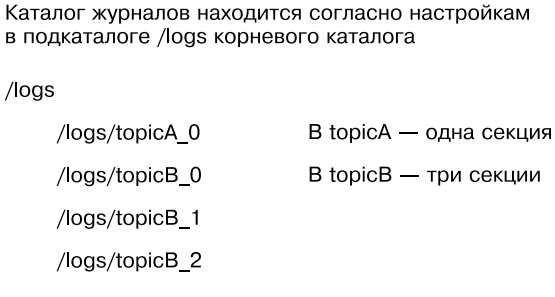


Рис. 2.6. Каталог журналов — основное хранилище сообщений. Каждый подкаталог в каталоге /logs соответствует секции одного из топиков. Имена файлов в таком каталоге начинаются с названия топика, за которым следуют символ подчеркивания и номер секции

Как вы можете видеть, понятия «журнал» и «топик» тесно связаны. Можно сказать, что топик — это журнал или что он символизирует журнал. Название топика показывает, в каком журнале будут сохранены сообщения, отправленные в Kafka посредством генераторов. Теперь, когда мы разобрались с понятием журналов, обсудим другое фундаментальное понятие Kafka — секции.

2.3.4. Kafka и секции

Секции — важнейший элемент архитектуры Kafka. Они необходимы для обеспечения производительности, и они гарантируют, что данные с одинаковыми ключами будут отправлены одному потребителю в правильном порядке. Рисунок 2.7 иллюстрирует работу секций.

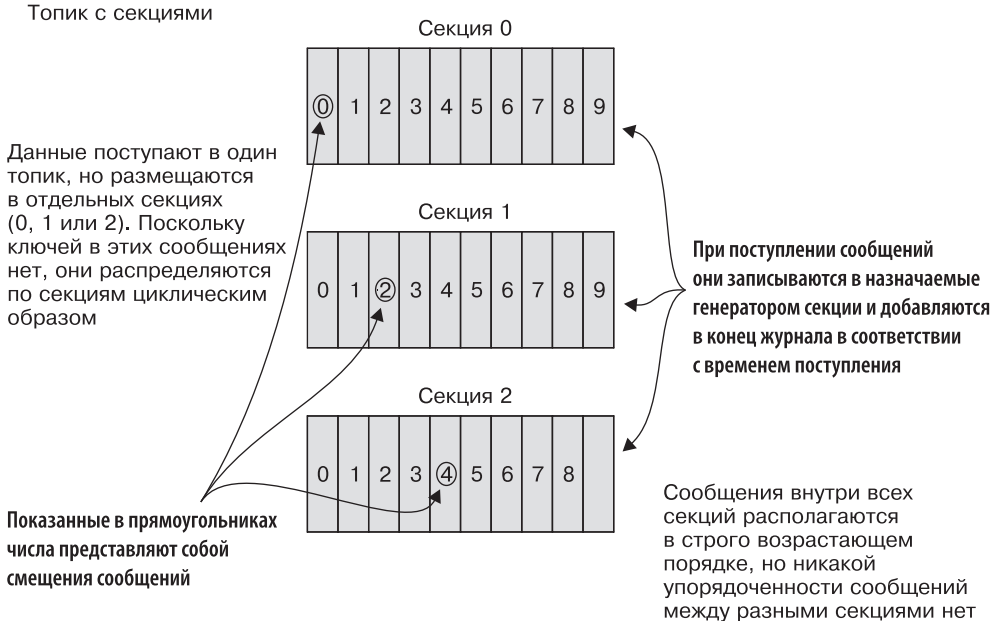


Рис. 2.7. Секции применяются в Kafka для достижения высокой пропускной способности и распределения сообщений отдельного топика по нескольким машинам кластера

При секционировании топика отправляемые в него данные, по существу, разбиваются на параллельные потоки. Именно в этом и заключается секрет потрясающей пропускной способности Kafka. Я уже упоминал, что топик — это распределенный журнал; каждая секция сама по себе тоже является журналом и следует тем же правилам. Kafka добавляет каждое из входящих сообщений в конец журнала, и все эти сообщения оказываются строго упорядоченными по времени. Каждому сообщению присваивается смещение. Упорядоченность гарантируется только внутри секции, но не между различными секциями.

Секционирование служит и другой цели, помимо повышения пропускной способности. Благодаря ему можно распределять сообщения топиков по нескольким машинам, так что вместимость конкретного топика не будет ограничиваться дисковым пространством, доступным на одном сервере.

Секции играют и еще одну важнейшую роль: они обеспечивают совместное расположение сообщений с одинаковыми ключами, о чем мы расскажем в следующем разделе.

2.3.5. Секции группируют данные по ключу

Kafka оперирует данными в виде пар «ключ/значение». Если их ключи пустые, генератор Kafka заносит записи в секции, выбираемые циклическим образом. Рисунок 2.8 демонстрирует работу механизма назначения секции в случае непустых ключей.

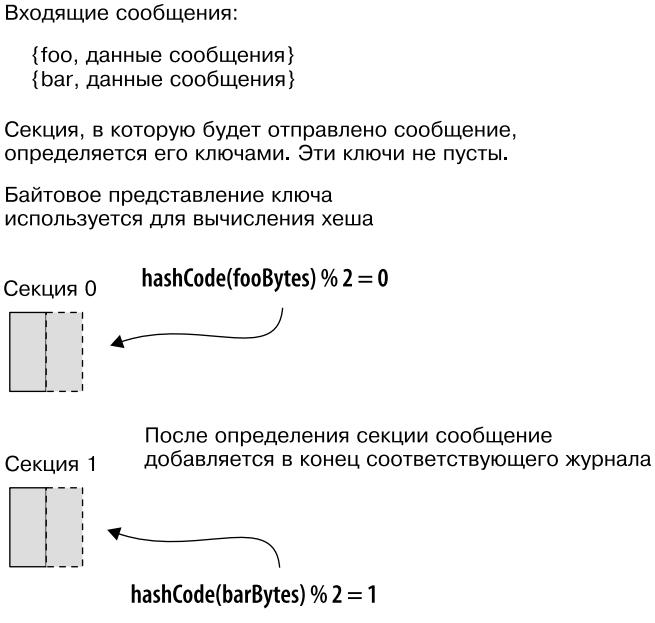


Рис. 2.8. foo отправляется в секцию 0, а bar — в секцию 1. Номер секции определяется путем хеширования байтового представления ключа по модулю числа секций

Если ключи не пусты, Kafka использует следующую формулу (показана в виде псевдокода), чтобы определить, в какую секцию поместить пару «ключ/значение»:

```
хеш.(ключ) % число секций
```

Благодаря детерминистичности подхода к выбору секции записи с одним ключом *всегда* будут попадать в одну секцию, причем в правильном порядке. Такой подход применяется методом секционирования по умолчанию; при необходимости иной стратегии выбора секций можно создать пользовательский метод секционирования.

2.3.6. Написание пользовательского класса секционирования

Зачем может понадобиться пользовательский способ секционирования? По нескольким причинам, из которых мы рассмотрим тут одну — применение составных ключей.

Пускай данные о покупках поступают в Kafka, причем ключи состоят из двух значений: идентификатора покупателя и даты транзакции. При этом вам нужно группировать данные по идентификатору покупателя, так что просто взять хеш идентификатора покупателя и даты покупки нельзя. В таком случае необходимо написать пользовательский метод секционирования, который бы «знал», на основании какой части составного ключа выбирается секция. Например, в листинге 2.1 показан составной ключ из src/main/java/bbejeck/model/PurchaseKey.java (исходный код можно найти по адресу https://manning.com/books/kafka-streams-in-action).

Листинг 2.1. Составной ключ PurchaseKey

```
public class PurchaseKey {

    private String customerId;
    private Date transactionDate;

    public PurchaseKey(String customerId, Date transactionDate) {
        this.customerId = customerId;
        this.transactionDate = transactionDate;
    }

    public String getCustomerId() {
        return customerId;
    }

    public Date getTransactionDate() {
        return transactionDate;
    }
}
```

При секционировании необходимо убедиться, что все транзакции для конкретного покупателя попадают в одну секцию, но сделать это с помощью ключа целиком не получится. Учет при секционировании даты приведет к множеству различных значений ключей для одного покупателя, поскольку покупки происходят в разные дни, в результате чего транзакции будут попадать в секции хаотичным образом. Все транзакции с одним идентификатором покупателя должны попасть в одну секцию. Единственный способ добиться этого — применять при выборе секции только идентификатор покупателя.

Пользовательский метод секционирования из следующего примера именно это и делает. Класс `PurchaseKeyPartitioner` (из src/main/java/bbejeck/chapter_2/partitioner/PurchaseKeyPartitioner.java) извлекает идентификатор покупателя из ключа и определяет на его основе, какую секцию применять (листинг 2.2).

Листинг 2.2. Пользовательский класс секционирования PurchaseKeyPartitioner

```
public class PurchaseKeyPartitioner extends DefaultPartitioner {

    @Override
    public int partition(String topic, Object key,
                         byte[] keyBytes, Object value,
                         byte[] valueBytes, Cluster cluster) {
        Object newKey = null;
        if (key != null) {
            PurchaseKey purchaseKey = (PurchaseKey) key;
            newKey = purchaseKey.getCustomerId();
            keyBytes = ((String) newKey).getBytes();
        }
        return super.partition(topic, newKey, keyBytes, value,
        valueBytes, cluster);
    }
}
```

Если ключ не пуст, извлекаем идентификатор покупателя

Присваиваем переменной keyBytes новое значение

Делегируем вычисление секции для нового ключа методу родительского класса

Этот пользовательский класс секционирования расширяет класс `DefaultPartitioner`. Можно реализовать непосредственно интерфейс `Partitioner`, но в классе `DefaultPartitioner` уже реализована логика, которая нам пригодится.

Конечно, при создании пользовательского метода секционирования вы можете не ограничиваться одним только ключом. Можно применить также только значение или значение в сочетании с ключом.

ПРИМЕЧАНИЕ

API Kafka содержит интерфейс Partitioner, который можно применять при создании пользовательского метода секционирования. Мы не будем рассматривать написание метода секционирования с нуля, но основные принципы не отличаются от описанных в листинге 2.2.

Вы только что посмотрели на создание пользовательского класса секционирования. Теперь подключим его к Kafka.

2.3.7. Настройка пользовательского секционирования

После написания пользовательского класса секционирования нужно сообщить Kafka о необходимости его применять вместо секционирования по умолчанию. Мы пока еще не обсуждали генераторы, вот пример задания другого класса секционирования в конфигурации генератора Kafka:

```
partitioner.class=bbejeck_2.partitioner.PurchaseKeyPartitioner
```

Благодаря настройке, позволяющей задавать отдельный способ секционирования для каждого экземпляра генератора, можно использовать тот класс секционирования для любого генератора, какой вам нужен. Мы обсудим настройки генераторов подробно в посвященном генераторам разделе.

ПРЕДУПРЕЖДЕНИЕ

Следует с осторожностью выбирать применяемые ключи и части пары «ключ/значение», по которым выполняется секционирование. Проверьте, чтобы выбранный вами ключ был распределен равномерно по всем вашим данным. В противном случае вы в конце концов столкнетесь с проблемой асимметрии данных, поскольку большая их часть будет располагаться лишь в небольшой части секций.

2.3.8. Выбор правильного числа секций

Выбор числа секций при создании топика — искусство и наука одновременно. Главное, что нужно учесть, — объем поступающих в этот топик данных. Для повышения пропускной способности должно быть тем больше секций, чем больше данных. Но, как всегда в жизни, приходится идти на компромиссы.

Повышение числа секций приводит к росту количества TCP-соединений и открытых дескрипторов файлов. Кроме того, длительность процесса обработки записи в потребителе также является определяющим фактором для пропускной способности. Если в потребителе производится «тяжеловесная» обработка, то добавление дополнительных секций может решить проблему на время, но в конце концов производительность из-за этого начнет страдать[1].

2.3.9. Распределенный журнал

Мы обсудили основные принципы журналов и секционированных топиков. Соединим эти понятия в виде распределенного журнала.

До сих пор мы сосредотачивали наше внимание на журналах и топиках на одном сервере (брокере) Kafka, но обычно среда реального кластера Kafka включает несколько машин. Я умышленно удерживал обсуждение в рамках одноузловых систем, ведь в случае одного узла разбираться в принципах работы проще. Но на практике в Kafka всегда работают с кластером машин.

При секционировании топика Kafka не размещает его секции на одной машине, а распределяет их по нескольким машинам из кластера. При добавлении записей в журнал Kafka распределяет их по нескольким машинам посредством секционирования. На рис. 2.9 можно наблюдать этот процесс в действии.

Рассмотрим короткий пример с рис. 2.9 в качестве ориентира. В этом примере будет один топик при пустых ключах, так что генератор назначит секции циклическим образом.

Генератор отправляет первое сообщение в секцию 0 на брокере Kafka 1, второе сообщение — в секцию 1 на брокере Kafka 1, а третье — в секцию 2 на брокере Kafka 2. При отправке шестого сообщения генератор переходит с брокера Kafka 5 на брокер Kafka 3, и следующее сообщение начинает цикл заново, с секции 0 на брокере

[1] *Rao J.* How to Choose the Number of Topics/Partitions in a Kafka Cluster? (Дзюн Рао «Как выбрать число топиков/секций в кластере Kafka»), http://mng.bz/4C03.

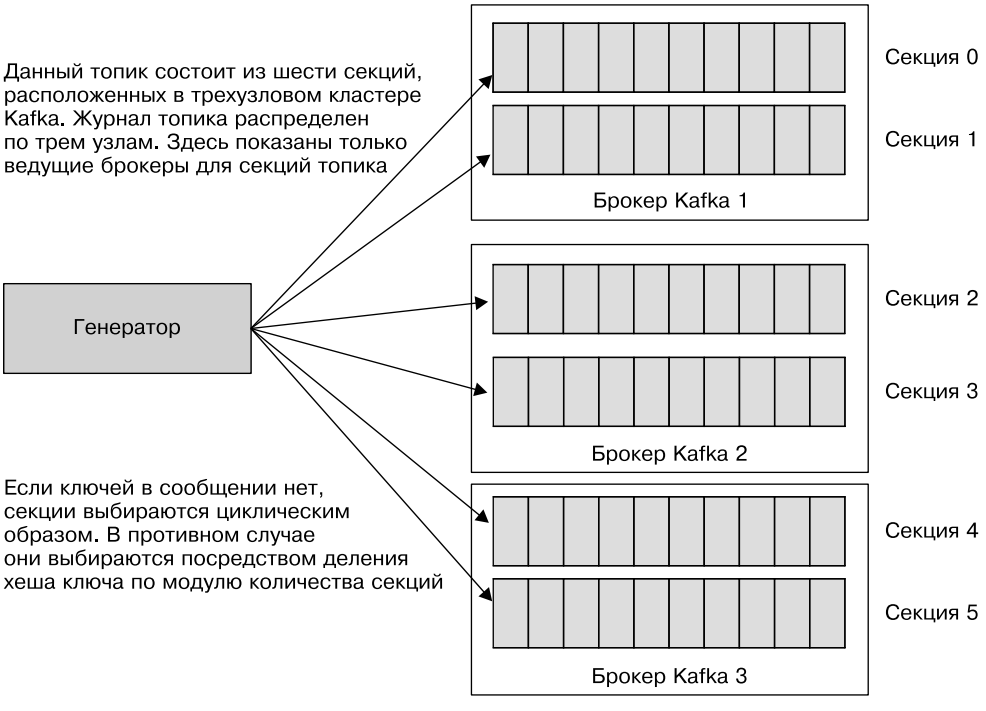


Рис. 2.9. Генератор записывает сообщения в секции топика. Если ключа в сообщении нет, он (генератор) выбирает секции циклическим образом. В противном случае используется деление хеша ключа по модулю числа секций

Kafka 1. Сообщения продолжают размещаться так и дальше, распределяя трафик по всем узлам кластера Kafka.

Хотя может показаться, что хранить данные удаленно довольно рискованно, ведь сервер может выйти из строя, это не так благодаря предоставляемой Kafka избыточности данных. При записи на один из брокеров в Kafka данные реплицируются на одну или несколько машин кластера (репликацию мы рассмотрим в следующем разделе).

2.3.10. ZooKeeper: ведущие/ведомые брокеры и репликация

До сих пор мы обсуждали то, какую роль играют в Kafka топики, а также как и почему их нужно секционировать. Как вы видели, секции не располагаются все на одной машине, а распределяются по брокерам в кластере. Пришло время рассказать, как Kafka обеспечивает доступность данных в случае отказов отдельных машин.

В Kafka существует понятие ведущего и ведомого брокеров: для каждой секции топика один из брокеров выбирается в качестве *ведущего* (leader) для остальных

брокеров — *ведомых* (followers). Одна из главных задач ведущего брокера — назначение ведомых брокеров для *репликации* (replication) секций топика. Аналогично тому как Kafka распределяет секции топика по кластеру, Kafka также реплицирует секции по машинам. Прежде чем углубляться в подробности того, как функционируют ведущие/ведомые брокеры и репликация, необходимо обсудить используемую Kafka для этого технологию.

2.3.11. Apache ZooKeeper

Если вы новичок в Kafka, то, наверное, задаетесь вопросом: «Почему в книге о Kafka вдруг зашла речь об Apache ZooKeeper?» Apache ZooKeeper — неотъемлемая часть архитектуры Kafka, именно благодаря ему у Kafka есть ведущие брокеры и она может, например, отслеживать репликацию топиков (https://zookeeper.apache.org/).

> *ZooKeeper — централизованный сервис, предназначенный для хранения настроек и информации о наименованиях, распределенной синхронизации и предоставления групповых сервисов. Все эти сервисы так или иначе применяются в распределенных приложениях.*

А поскольку Kafka — распределенное приложение, становится понятна роль ZooKeeper в его архитектуре. В данной книге мы будем рассматривать только системы Kafka с двумя и более серверами.

Один из брокеров в любом кластере Kafka «выбирается» *контроллером* (controller). В предыдущем разделе мы обсуждали секции и говорили о том, как Kafka распределяет секции по различным машинам кластера. У секций топика есть ведущий и ведомый (-е) брокер (-ы) (количество брокеров для репликации определяется ее уровнем). При генерации сообщения Kafka отправляет запись ведущему брокеру соответствующей секции.

2.3.12. Выборы контроллера

Для выборов брокера-контроллера Kafka использует ZooKeeper. Обсуждение алгоритмов консенсуса выходит далеко за рамки этой книги, так что мы с высоты птичьего полета только отметим, что ZooKeeper выбирает, какой брокер из кластера будет контроллером.

При отказе брокера-контроллера или его недоступности (по какой-либо причине) ZooKeeper выбирает новый из числа брокеров, не отстающих от ведущего (так называемые *согласованные реплики* (in-sync replica, ISR)). Это множество состоит из нормально функционирующих брокеров, и в качестве претендентов на роль ведущего брокера[1] ZooKeeper рассматривает только его участников.

---

[1] См. документацию Kafka: Replicated Logs: Quorums, ISRs, and State Machines (Oh my!) («Реплицируемые журналы: кворумы, ISR и конечные автоматы (вот это да!)»), http://kafka.apache.org/documentation/#design_replicatedlog.

2.3.13. Репликация

Чтобы обеспечить доступность данных в случае отказа одного из брокеров кластера, Kafka реплицирует записи между брокерами. Можно задавать уровень репликации индивидуально для каждого топика (как вы видели в нашем предыдущем примере с публикацией и потреблением) или для всех топиков в кластере. Рисунок 2.10 иллюстрирует поток репликации между брокерами.

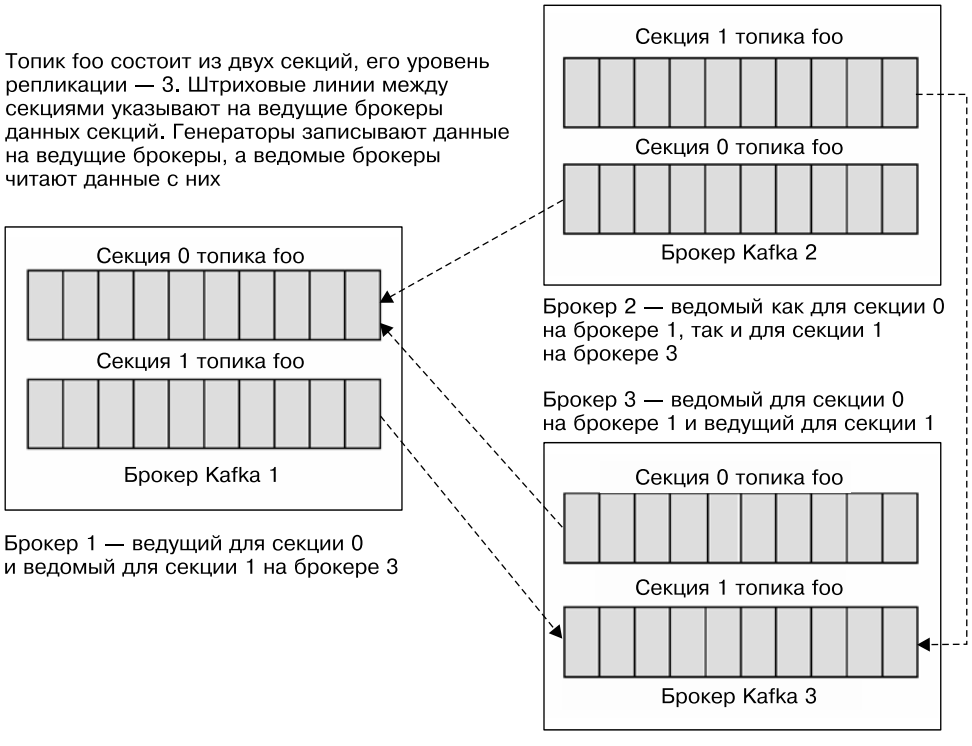


Рис. 2.10. Брокеры 1 и 3 — ведущие для одной секции топика и ведомые для другой, в то время как брокер 2 — лишь ведомый. Ведомые брокеры копируют данные с ведущего

Процесс репликации Kafka достаточно прост. Ведомые для секции топика брокеры потребляют сообщения из соответствующего ведущего брокера и дописывают их в журналы. Как уже обсуждалось в предыдущем разделе, ведомые брокеры, которые не отстают от ведущего, считаются согласованными репликами (ISR). ISR-брокер может быть избран ведущим при сбое или недоступности нынешнего ведущего брокера[1].

[1] См. документацию Kafka: Replication («Репликация»), http://kafka.apache.org/documentation/#replication.

2.3.14. Обязанности контроллера

Брокер-контроллер отвечает за организацию взаимодействия ведущих/ведомых для всех секций топика. Если узел Kafka перестает работать или не реагирует (на периодические контрольные сигналы ZooKeeper), брокер-контроллер переназначает все назначенные тому секции (как те, для которых он является ведущим, так и те, для которых он является ведомым). Рисунок 2.11 демонстрирует работу брокера-контроллера[1].

На следующем рисунке приведен простой сценарий отказа. На шаге 1 брокер-контроллер обнаруживает, что брокер 3 недоступен. На шаге 2 брокер-контроллер делает ведущим для секции 1 брокер 2 вместо брокера 3.

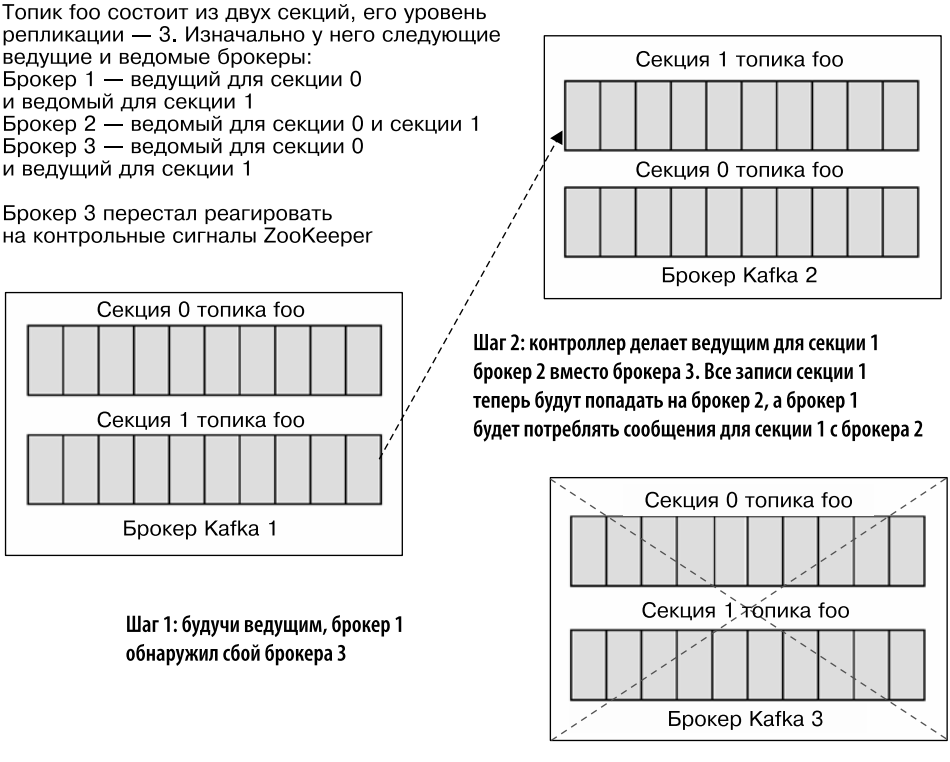


Рис. 2.11. Брокер-контроллер отвечает за назначение других брокеров на роль ведущих для одних топиков/секций и ведомых для других топиков/секций. Когда брокер перестает реагировать, брокер-контроллер переназначает другим брокерам все, что было назначено сбойному брокеру

[1] Часть приводимой в этом разделе информации основана на ответах Гвен Шапиры (Gwen Shapira) на вопросы: «Какую роль на самом деле играет Zookeeper в Kafka? Какие возможности я упущу, если не буду использовать их вместе?» — на сайте Quora: http://mng.bz/25Sy.

ZooKeeper также участвует в таких аспектах работы Kafka, как:

- *членство в кластере* — осуществляет присоединение к кластеру и обслуживание членства в нем. Когда брокер перестает реагировать, ZooKeeper удаляет его из числа членов кластера;
- *конфигурация топика* — отслеживает топики кластера, их ведущие брокеры, число секций в топике, а также любые конкретные переопределения настроек топика по умолчанию;
- *управление доступом* — определяет, кто может читать из конкретных топиков и записывать информацию в них.

Теперь вы видите, почему Apache Kafka зависит от ZooKeeper. Именно благодаря ZooKeeper существуют в Apache Kafka ведущие и ведомые брокеры. Главному брокеру при этом доверяется важнейшая роль: назначение ведомым брокерам секций топика для репликации, а также переназначение их в случае отказа одного из брокеров.

2.3.15. Управление журналами

Мы говорили о дописывании сообщений, но пока еще не обсуждали вопрос управления журналами по мере их роста. Доступное на жестких дисках кластера место — ресурс ограниченный, так что для Kafka важно удалять сообщения через какое-то время. Существует два подхода к удалению старых данных в Kafka: традиционный подход с удалением журналов и их сжатие.

2.3.16. Удаление журналов

Стратегия удаления журналов является двухэтапной: сначала журналы архивируются в сегменты, после чего наиболее старые сегменты удаляются. Kafka *архивирует* (roll) журналы в сегменты, чтобы как-то справиться с их возрастающими размерами. Хронометраж архивирования журналов зависит от включаемых в сообщения меток даты/времени. Kafka архивирует журнал при поступлении нового сообщения с меткой даты/времени, превышающей метку даты/времени первого сообщения в журнале плюс значение параметра конфигурации `log.roll.ms`. В этот момент журнал архивируется и создается новый сегмент, который далее будет играть роль нового активного журнала. Из ранее активного сегмента по-прежнему извлекаются сообщения для потребителей.

При настройке брокера Kafka можно указать одну из двух опций архивирования журналов[1]:

- `log.retention.ms` — основной параметр, без значения по умолчанию;
- `log.roll.hours` — дополнительный параметр, используется только тогда, когда не задано значение параметра `log.retention.ms`. По умолчанию его значение равно 168 часам.

---

[1] См. документацию Kafka: Broker Configs («Конфигурация брокеров»), http://kafka.apache.org/documentation/#brokerconfigs.

Со временем число сегментов будет расти и придется удалять старые сегменты, чтобы освободить место для поступающих данных. Для этого можно задать значение длительности сохранения старых сегментов. Процесс архивирования показан на рис. 2.12.

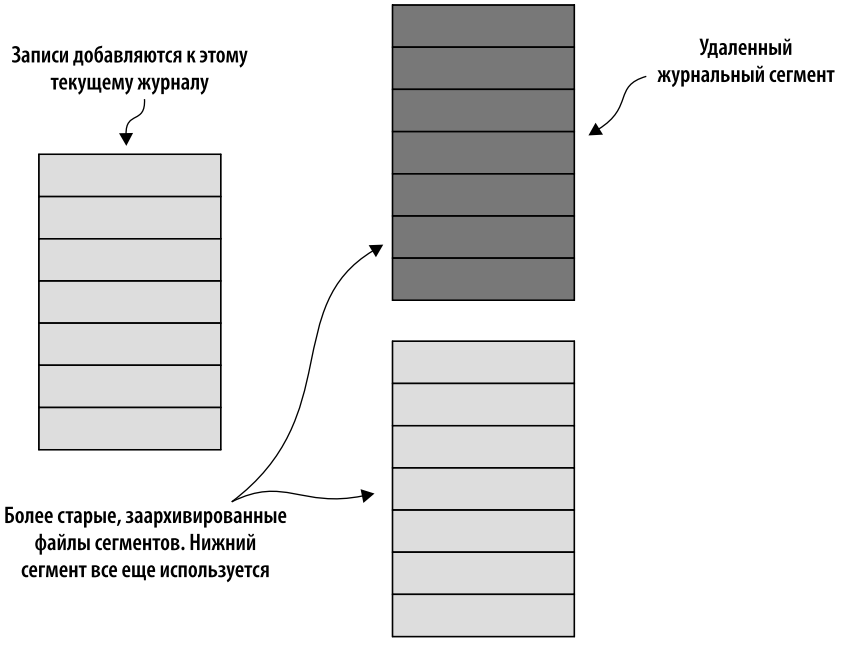


Рис. 2.12. *Слева* показаны текущие журнальные сегменты. *Справа сверху* — удаленный журнальный сегмент, а ниже его — только что заархивированный, но еще используемый сегмент

Как и архивирование журналов, удаление сегментов основывается на метках даты/времени в сообщениях, а не на системном времени или времени последнего изменения файла. Удаляются журнальные сегменты с максимальной меткой даты/времени в журнале. Существует три параметра настройки, перечисленных ниже в порядке убывания их приоритета. Это значит, что расположенный выше в списке параметр имеет большую силу, чем расположенные ниже:

- `log.retention.ms` — длительность хранения файла журнала в миллисекундах;
- `log.retention.minutes` — длительность хранения файла журнала в минутах;
- `log.retention.hours` — длительность хранения файла журнала в часах.

Эти настройки рассчитаны на топики большого объема, где за заданный период времени гарантированно достигается максимальный размер файла. Существует еще одна настройка, `log.retention.bytes`, подходящая для применения в случае большей предельной длительности промежутка архивирования с целью минимизации операций ввода/вывода. Наконец, чтобы обезопасить себя от существенных всплесков

объема сегментов при относительно больших значениях настроек архивирования, можно воспользоваться настройкой `log.segment.bytes`, которая задает максимальный размер отдельного журнального сегмента.

Удаление журналов хорошо подходит для записей без ключей, или независимых записей. Но для ваших потребностей в случае записей с непустыми ключами и возможных обновлений данных лучше подойдет другой метод.

2.3.17. Сжатие журналов

Представьте себе, что ваши данные, снабженные ключами, периодически обновляются: новые записи с теми же ключами заменяют предыдущие значения. Например, ключом может быть биржевой тиккер, а роль периодически обновляемого значения будет играть цена одной акции. Представьте себе, что такая информация используется в вашем приложении для отображения курсов акций и происходит фатальный сбой (или возникает необходимость перезапуска). При этом должна сохраняться возможность возобновить работу с наиболее свежими данными для всех ключей[1].

При применении стратегии удаления удаление сегмента может произойти между последним обновлением и фатальным сбоем/перезапуском приложения, так что при запуске у вас не окажется всех записей. Лучше будет сохранить последнее известное значение для заданного ключа, рассматривая следующую запись с тем же ключом в качестве обновления таблицы базы данных.

Сжатые топики (журналы) производят обновление записей по ключу. Вместо высокоуровневого подхода с удалением целых сегментов в зависимости от интервала времени или размера сжатие позволяет действовать более аккуратно и удалять старые записи в журнале *по ключу*. Если не углубляться в подробности, очиститель журналов (пул потоков) работает в фоновом режиме, перезаписывая файлы журнальных сегментов и удаляя записи, если далее в том же журнале встречается запись с тем же ключом. Рисунок 2.13 иллюстрирует сохранение только последнего сообщения для каждого ключа при сжатии журналов.

Такой подход гарантирует наличие в журнале только последней записи для данного ключа. Можно задавать настройки хранения информации отдельно для каждого топика, то есть некоторые топики будут хранить информацию определенное время, а другие — использовать сжатие.

По умолчанию включена очистка журналов. Чтобы переключиться на сжатие журналов для топика, необходимо при его создании задать свойство `log.cleanup.policy=compact`.

Сжатие применяется в Kafka Streams при хранении состояния, но вам не придется создавать эти журналы/топики вручную — обо всем позаботится фреймворк. Тем не менее важно понимать принцип работы сжатия. Сжатие журналов — обширная тема, и здесь мы коснемся ее лишь вкратце. Дополнительную информацию вы можете найти в документации Kafka: http://kafka.apache.org/documentation/#compaction.

---

[1] См. документацию Kafka: Log Compaction («Сжатие журналов»), http://kafka.apache.org/documentation/#compaction.

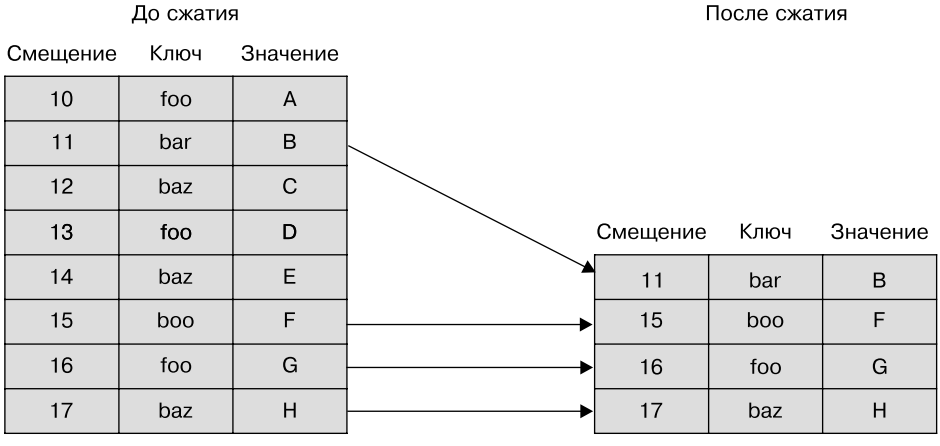


Рис. 2.13. *Слева* показан журнал до сжатия, в котором можно заметить дублирующиеся ключи с различными значениями — обновлениями для данного ключа. *Справа* показан журнал после сжатия — сохранено только последнее значение для каждого ключа, а журнал уменьшился в размере

ПРИМЕЧАНИЕ

У вас может возникнуть вопрос: как при равном compact значении свойства cleanup.policy удалить запись из журнала? В случае сжатого топика удаление устанавливает значение заданного ключа в null, формируя таким образом отметку об удалении. Ключ со значением null гарантирует, что все предыдущие записи с тем же ключом уже удалены, как будет удалена через некоторое время и сама отметка об удалении.

Основной вывод из этого раздела: в случае независимых, автономных событий или сообщений следует использовать удаление журналов. В случае же обновляемых событий или сообщений лучше подойдет сжатие журналов.

Мы потратили немало времени на обсуждение внутренних механизмов работы Kafka с данными. Настало время выйти за рамки Kafka и рассмотреть отправку в нее сообщений с помощью генераторов и чтение сообщений из Kafka с помощью потребителей.

2.4. Отправка сообщений с помощью генераторов

Возвращаясь к ZMart и их потребности в централизованном концентраторе информации о транзакциях продаж, обсудим, как мы будем отправлять эту информацию в Kafka. В Kafka используемый для отправки сообщений клиент называется *генератором* (producer). На рис. 2.14 снова показана архитектура данных ZMart, генераторы в ней выделены, чтобы подчеркнуть их участие в информационном потоке.

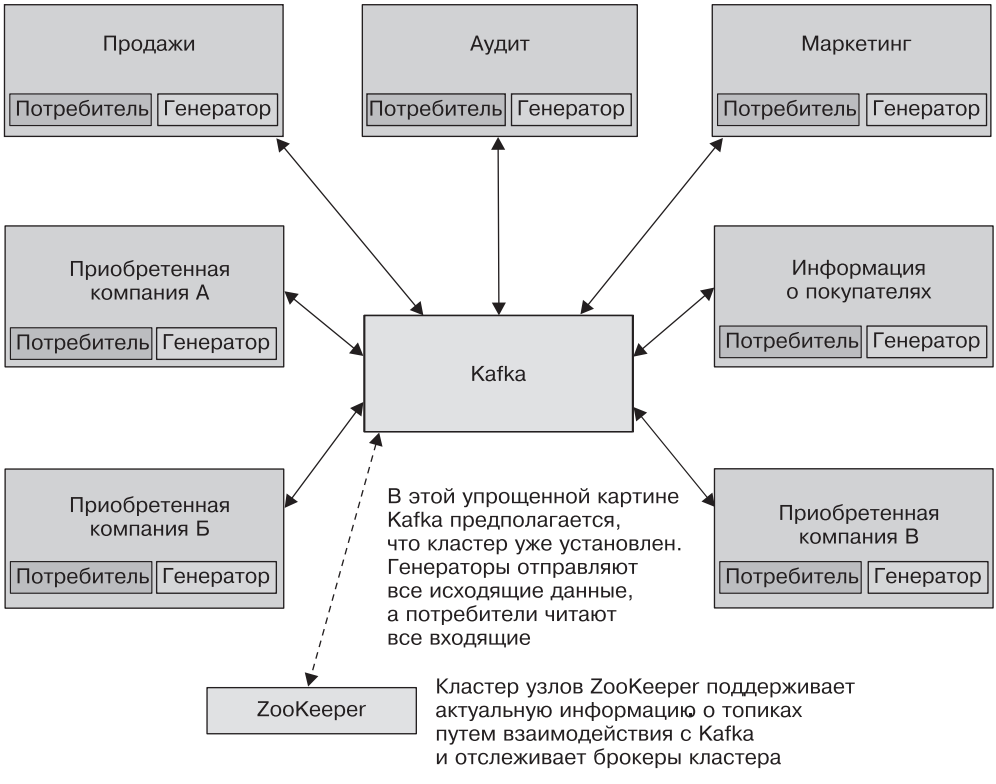


Рис. 2.14. Для отправки сообщений в Kafka используются генераторы. Генераторы «не знают», какой из потребителей и когда прочитает эти сообщения

Хотя у ZMart множество различных транзакций продаж, мы рассмотрим пока покупку отдельного товара — книги стоимостью $10,99. По завершении покупателем транзакции покупки информация о ней преобразуется в пару «ключ/значение» и отправляется генератором в Kafka.

Ключом служит идентификатор покупателя, `123447777`, а значение находится в формате JSON: `"{\"item\":\"book\",\"price\":10.99}"` (я экранировал двойные кавычки, чтобы можно было представить JSON в виде строкового литерала). Данные в этом формате можно отправить с помощью генератора в кластер Kafka. Вы можете найти следующий пример в файле src/main/java/bbejeck.chapter_2/producer/SimpleProducer.java (листинг 2.3).

Генераторы Kafka являются потокобезопасными. Отправка данных в Kafka производится асинхронно — возврат из метода `Producer.send` происходит сразу же после помещения генератором записи во внутренний буфер. Этот буфер отправляет записи пакетами. В зависимости от ваших настроек при отправке сообщения при заполненном буфере генератора вы можете столкнуться с блокировкой.

Листинг 2.3. Пример простого генератора

```
Properties properties = new Properties();
properties.put("bootstrap.servers", "localhost:9092");
properties.put("key.serializer",
➥ "org.apache.kafka.common.serialization.StringSerializer");
properties.put("value.serializer",
➥ "org.apache.kafka.common.serialization.StringSerializer");
properties.put("acks", "1");
properties.put("retries", "3");
properties.put("compression.type", "snappy");
properties.put("partitioner.class",
➥ PurchaseKeyPartitioner.class.getName());

PurchaseKey key = new PurchaseKey("12334568", new Date());

 try(Producer<PurchaseKey, String> producer =
➥ new KafkaProducer<>(properties)) {
    ProducerRecord<PurchaseKey, String>  record =
➥   new ProducerRecord<>("transactions", key,
     "{\"item\":\"book\", \"price\":10.99}");

  Callback callback = (metadata, exception) -> {
            if (exception != null) {
                System.out.println("Encountered exception "
➥               + exception);
            }
        };

  Future<RecordMetadata> sendFuture =
➥ producer.send(record, callback);
}
```

Свойства для настройки генератора

Создание объекта KafkaProducer

Создание экземпляра ProducerRecord

Обратный вызов

Отправка записи и присвоение переменной возвращаемого Future

Вышеупомянутый метод `Producer.send` принимает в качестве параметра экземпляр типа `Callback`. После подтверждения получения записи ведущим брокером генератор инициирует выполнение метода `Callback.onComplete`. Только один из аргументов метода `Callback.onComplete` будет непустым. В данном случае нас интересует только вывод трассы вызовов при ошибке, так что нам важно, чтобы объект исключения был не пуст. Возвращаемый фьючерс после подтверждения получения записи сервером выдает объект типа `RecordMetadata`.

ОПРЕДЕЛЕНИЕ

В листинге 2.3 метод Producer.send возвращает объект Future, который представляет собой результат выполнения асинхронной операции. Что важнее, Future дает возможность извлечь результаты асинхронных операций отложенным образом, вместо того чтобы ждать их завершения. Более подробную информацию о фьючерсах вы можете найти в документации Java, в разделе «Интерфейс Future<V>»: http:// mng.bz/0JK2.

2.4.1. Свойства генераторов

При создании экземпляра класса `KafkaProducer` передается параметр типа `java.util.Properties`, содержащий настройки генератора. Настройки генератора `KafkaProducer` довольно просты, но стоит обратить внимание на некоторые из них. Именно в них можно задать, например, пользовательский класс секционирования. Свойств конфигурации слишком много, чтобы охватить тут их все, поэтому рассмотрим лишь те, что задействовались в листинге 2.3.

- *Серверы для начальной загрузки* — `bootstrap.servers` представляет собой разделенный запятыми список значений вида `хост:порт`. В конце концов генератор использует все брокеры кластера; этот список применяется для первоначального подключения к кластеру.
- *Сериализация* — свойства `key.serializer` и `value.serializer` определяют способ преобразования фреймворком Kafka ключей и значений в байтовые массивы. Внутри Kafka для ключей и значений применяются байтовые массивы, так что нужно предоставить Kafka правильные сериализаторы для преобразования объектов в байтовые массивы перед отправкой.
- `acks` — свойство `acks` задает минимальное число подтверждений от брокера, при получении которого генератор будет считать отправку записи успешной. Допустимые значения этого свойства: `all`, `0` и `1`. При значении `all` генератор будет ждать получения от брокера подтверждения фиксации записи во всех ведомых брокерах. При значении `1` брокер вносит запись в свой журнал, но не ждет подтверждения фиксации записи никакими ведомыми брокерами. Значение `0` означает, что генератор вообще не ждет никаких подтверждений — по сути, это стратегия типа «сделать и забыть».
- *Стратегии повторов отправки* — свойство `retries` задает число попыток повторной отправки на случай сбоя отправки пакета. Если порядок записей важен, имеет смысл установить свойству `max.in.flight.requests.per.connection` значение `1`, чтобы второй пакет не оказался успешно отправленным до повторной отправки сбойного.
- *Тип сжатия* — свойство `compression.type` определяет используемый алгоритм сжатия (если сжатие вообще применяется). Задание этого свойства указывает генератору на необходимость сжатия пакета перед отправкой. Обратите внимание, что сжимается пакет в целом, а не отдельные сообщения.
- *Класс секционирования* — свойство `partitioner.class` задает название класса, который реализует интерфейс `Partitioner`. Свойство `partitioner.class` имеет отношение к нашему обсуждению пользовательских классов секционирования в подразделе 2.3.7.

Более подробную информацию о генераторах вы можете найти в документации Kafka: http://kafka.apache.org/documentation/#producerconfigs.

2.4.2. Указание секции или метки даты/времени

При создании объекта `ProducerRecord` можно указать секцию, метку даты/времени или и то и другое. При создании экземпляра `ProducerRecord` в листинге 2.3 мы воспользовались одним из четырех[1] имеющихся перегруженных конструкторов. Остальные конструкторы этого класса позволяют указать секцию и метку даты/времени или одну только секцию:

```
ProducerRecord(String topic, Integer partition, String key, String value)
ProducerRecord(String topic, Integer partition,
               Long timestamp, String key,
               String value)
```

2.4.3. Указание секции

В подразделе 2.3.4 мы говорили о том, насколько важны в Kafka секции. Мы также обсуждали, как работает класс `DefaultPartitioner` (метод секционирования по умолчанию) и как можно задать пользовательский класс секционирования. Зачем может понадобиться явным образом указывать секцию? Существует множество обоснованных причин для этого. Вот один пример.

Допустим, что к вам поступают снабженные ключами данные, но для вас нет никакой разницы, в какую секцию попадут записи, поскольку логика потребителей позволяет обрабатывать любые возможные значения ключей. Кроме того, распределение ключей может быть неравномерным, так что вам нужно обеспечить приблизительно одинаковый объем данных в каждой из секций. Ниже приведена черновая реализация этого (листинг 2.4).

Листинг 2.4. Указываем секцию вручную

```
AtomicInteger partitionIndex = new AtomicInteger(0);

int currentPartition = Math.abs(partitionIndex.getAndIncrement()) %
➥ numberPartitions;
ProducerRecord<String, String> record =
➥ new ProducerRecord<>("topic", currentPartition, "key", "value");
```

Создание переменной экземпляра класса AtomicInteger

Получаем текущую секцию и используем ее в качестве параметра

Здесь мы задействуем вызов функции `Math.abs`, так что не нужно следить, чтобы значение целочисленной переменной не превысило `Integer.MAX_VALUE`.

[1] В текущей версии Kafka 2.0 в классе ProducerRecord шесть перегруженных конструкторов. См. документацию Kafka: https://kafka.apache.org/20/javadoc/org/apache/kafka/clients/producer/ProducerRecord.html. — *Примеч. пер.*

ОПРЕДЕЛЕНИЕ

Класс AtomicInteger относится к пакету java.util.concurrent.atomic, содержащему классы с поддержкой потокобезопасных операций без блокировок для отдельных переменных. Дополнительную информацию вы можете найти в документации Java по пакету java.util.concurrent.atomic: http://mng.bz/PQ2q.

2.4.4. Метки даты/времени в Kafka

В Kafka версии 0.10 в записи были добавлены метки даты/времени. Метка даты/времени задается при создании объекта `ProducerRecord` с помощью следующего вызова перегруженного конструктора:

```
ProducerRecord(String topic, Integer partition,
➥ Long timestamp, K key, V value)
```

Если не задать метку даты/времени, это сделает генератор (с использованием текущего системного времени) перед отправкой записи брокеру Kafka. На метки даты/времени также влияет параметр конфигурации брокера `log.message.timestamp.type`, который может принимать значение или `CreateTime` (по умолчанию), или `LogAppendTime`. Подобно многим другим настройкам брокеров, задаваемое для брокера значение применяется ко всем топикам по умолчанию, но при создании топика можно задать для него другое значение. Если указать значение `LogAppendTime` и топик не перекрывает настроек брокера, то брокер будет записывать вместо метки даты/времени текущее время при добавлении записи в журнал. В противном случае будет использоваться метка даты/времени из объекта `ProducerRecord`.

Как выбрать одно из этих значений? `LogAppendTime` рассматривается в качестве времени обработки, а `CreateTime` — времени события. Какое выбрать — зависит от ваших бизнес-требований. Нужно решить для себя: вы хотите знать, когда Kafka обработал запись или когда фактически произошло событие? В следующих главах мы увидим, какую важную роль метки даты/времени играют в контроле информационных потоков в Kafka Streams.

2.5. Чтение сообщений с помощью потребителей

Вы уже посмотрели на работу генераторов; время взглянуть на потребители в Kafka. Допустим, что вы создаете предварительную версию приложения для отображения актуальной статистики продаж ZMart. Для данного примера мы прочитаем сообщение, отправленное в предыдущем примере с помощью генератора. А поскольку создание этого приложения еще только начинается, мы пока только прочитаем указанное сообщение и выведем информацию в консоль.

ПРИМЕЧАНИЕ

Мы обсудим тут только новую версию потребителя, включенную в Kafka 0.9, поскольку рассматриваемая в данной книге версия Kafka Streams все равно требует версии Kafka 0.10.2 или более поздней.

Для потребления сообщений из Kafka мы воспользуемся классом `KafkaConsumer`. Класс `KafkaConsumer` прост в применении, но при его эксплуатации следует учесть несколько нюансов. В приведенной на рис. 2.15 архитектуре ZMart выделены места участия потребителей в информационном потоке.

Рис. 2.15. Потребители читают данные из Kafka. Подобно тому как генераторы «не знают» ничего о потребителях, потребители понятия не имеют о том, кто сгенерировал читаемые ими из Kafka сообщения

2.5.1. Управление смещениями

Класс `KafkaConsumer`, в отличие от `KafkaProducer`, частично сохраняет состояние посредством периодической фиксации смещений потребленных из Kafka сообщений. Смещения идентифицируют сообщения уникальным образом и отражают их

начальные позиции в журнале. Потребителям приходится периодически фиксировать смещения полученных ими сообщений.

Фиксация смещений означает для потребителя две вещи:

- подразумевает, что потребитель полностью обработал сообщение;
- соответствует отправной точке для этого потребителя на случай сбоя или перезапуска.

В случае нового экземпляра потребителя или возникшего сбоя может оказаться, что последнее зафиксированное смещение неизвестно, так что место, с которого потребитель начнет работу, зависит от настроек:

- `auto.offset.reset="earliest"` — извлечение сообщений будет производиться, начиная с самого первого из доступных смещений. Будут извлечены все сообщения, которые еще не были удалены в ходе процесса управления журналами;
- `auto.offset.reset="latest"` — извлечение сообщений будет производиться, начиная с последнего из доступных смещений, по существу, сообщения будут потребляться только с момента присоединения потребителя к кластеру;
- `auto.offset.reset="none"` — стратегия сброса смещений не задана. Брокер генерирует исключение и передает его потребителю.

На рис. 2.16 показаны результаты выбора различных значений параметра `auto.offset.reset`. При выборе значения `earliest` вы получите сообщения, начиная со смещения 1 (точнее, 0. — *Примеч. пер.*). Если же выбрать `latest`, то вы получите сообщения, начиная со смещения 11 (точнее, 10. — *Примеч. пер.*).

Рис. 2.16. Наглядное представление работы настройки auto.offset.reset при значении earliest по сравнению со значением latest. В случае выбора earliest вы получите все еще не удаленные сообщения, а latest означает, что вам придется ждать поступления следующего сообщения

Далее мы обсудим опции, связанные с фиксацией смещений, которую можно выполнять вручную или автоматически.

2.5.2. Автоматическая фиксация смещений

Автоматическая фиксация смещения включена по умолчанию, для ее настройки существует свойство `enable.auto.commit`. Есть также дополнительная к этой опция — `auto.commit.interval.ms`, — задающая частоту фиксации смещений потребителем (по умолчанию раз в 5 секунд). Менять данный параметр следует осмотрительно. Слишком маленькое значение приведет к интенсификации сетевого трафика, слишком большое — к получению потребителями больших объемов повторных данных в случае сбоя или перезапуска.

2.5.3. Фиксация смещения вручную

Существует два типа фиксации смещений вручную: асинхронное и синхронное. Вот код для синхронных фиксаций:

```
consumer.commitSync()
consumer.commitSync(Map<TopicPartition, OffsetAndMetadata>)
```

Метод `commitSync()` без аргументов удерживает блокировку до момента успешной фиксации всех смещений, возвращенных при последней операции извлечения (вызова метода `poll`). Этот вызов применяется ко всем подписанным топикам и секциям. Вторая версия метода принимает на входе параметр типа `Map<TopicPartition, OffsetAndMetadata>` и фиксирует только указанные в данном ассоциативном массиве смещения для определенных в нем секций и топиков.

Существуют также аналогичные, полностью асинхронные, методы `consumer.commitAsync()`, возврат из которых происходит сразу же. Один из этих перегруженных методов — без параметров, а двум методам `consumer.commitAsync` можно передать объект `OffsetCommitCallback`, вызываемый по завершении фиксации, неважно, успешной или нет. Передача экземпляра функции обратного вызова дает возможность асинхронной обработки, в том числе ошибок. Преимущество фиксации смещений вручную — в возможности управлять тем, в какой момент запись будет считаться обработанной.

2.5.4. Создание потребителя

Создание потребителя аналогично созданию генератора. Передаются настройки в виде объекта класса `java.util.Properties`, и вы получаете в ответ экземпляр `KafkaConsumer`. Затем этот экземпляр подписывается на топики в соответствии с передаваемым списком их названий или регулярным выражением. Обычно потребители запускаются в цикле с задаваемым в миллисекундах промежутком между опросами.

Результатом опроса становится объект `ConsumerRecords<K, V>`. Класс `ConsumerRecords` реализует интерфейс `Iterable`, так что каждый вызов метода `next()`

возвращает объект `ConsumerRecord`, содержащий метаданные сообщения, помимо самих ключа и значения.

После исчерпания всех объектов `ConsumerRecord`, которые были возвращены в результате последнего вызова `poll`, происходит возврат вверх цикла и новый опрос в течение заданного времени. На практике потребители обычно работают подобным образом в течение произвольного времени, разве что произойдет ошибка или возникнет необходимость в остановке и перезапуске приложения (при этом как раз пригодятся зафиксированные смещения — при перезапуске потребитель сможет начать работу с того места, где ее прекратил).

2.5.5. Потребители и секции

Обычно требуется несколько экземпляров потребителя — по одному для каждой секции топика. Один потребитель может читать из нескольких секций, однако нередки случаи пулов потоков выполнения с числом потоков, равным числу секций, где в каждом потоке потребитель работает с одной назначенной ему секцией.

Паттерн «один потребитель из расчета на одну секцию» позволяет добиться максимальной пропускной способности, но *общее* количество потоков при распределении потребителей по нескольким приложениям или машинам не должно превышать общего числа секций топика. Все лишние (сверх общего количества секций) потоки будут простаивать. В случае сбоя потребителя ведущий брокер переназначает его секции другому нормально функционирующему потребителю.

ПРИМЕЧАНИЕ

Потребитель, подписанный всего на один топик, показан в данном примере исключительно в демонстрационных целях. Потребитель может быть подписан на произвольное число топиков.

Ведущий брокер назначает секции топиков всем доступным потребителям с одинаковым `group.id`. Это параметр конфигурации, определяющий принадлежность потребителя к определенной *группе потребителей* (consumer group). Благодаря этому потребители могут располагаться на различных машинах. На деле лучше, чтобы они были распределены по нескольким машинам. В таком случае при отказе одной из машин ведущий брокер сможет переназначить секции топиков потребителям с нормально функционирующих машин.

2.5.6. Перебалансировка

Процесс назначения топиков/секций потребителям и отмены этих назначений, описанный в предыдущем разделе, называется *перебалансировкой* (rebalancing). Назначения топиков/секций потребителям не статичны, они меняются в процессе работы. При добавлении потребителей с тем же идентификатором группы часть назначений топиков/секций активным потребителям отменяется и эти топики/секции

назначаются новым потребителям. Такой процесс переназначения продолжается до тех пор, пока все секции не окажутся распределены по потребителям.

Все потребители, оставшиеся после достижения этой точки равновесия, будут простаивать. Если потребитель покидает группу по какой-либо причине, назначенные ему топики/секции переназначаются другим потребителям.

2.5.7. Более точное назначение топиков/секций потребителям

В подразделе 2.5.5 я описал использование пула потоков и подписку нескольких потребителей (из одной группы) на одни и те же топики. Хотя Kafka и выравнивает нагрузку по топикам/секциям между всеми потребителями, назначение топиков и секций носит недетерминистский характер: нельзя предугадать, какие пары топиков/секций получит каждый из потребителей.

В классе `KafkaConsumer` имеются методы, с помощью которых можно подписаться на конкретные топик и секцию:

```
TopicPartition fooTopicPartition_0 = new TopicPartition("foo", 0);
TopicPartition barTopicPartition_0 = new TopicPartition("bar", 0);

consumer.assign(Arrays.asList(fooTopicPartition_0, barTopicPartition_0));
```

У назначения топиков/секций вручную есть свои недостатки:

- в случае отказа одной из машин не произойдет автоматического переназначения секций топиков, даже для потребителей с одним идентификатором группы. Для изменения назначений понадобится выполнить еще один вызов метода `consumer.assign`;
- для фиксации используется группа, указанная в настройках потребителя, но, поскольку все потребители будут функционировать сами по себе, имеет смысл задать для каждого из них свой уникальный идентификатор группы.

2.5.8. Пример потребителя

В листинге 2.5 приведен код потребителя для предварительной версии приложения ZMart, которая потребляет информацию о транзакциях и выводит ее в консоль. Вы можете найти этот код в файле src/main/java/bbejeck.chapter_2/consumer/ThreadedConsumerExample.java.

Листинг 2.5. Пример ThreadedConsumerExample

```
public void startConsuming() {
        executorService = Executors.newFixedThreadPool(numberPartitions);
        Properties properties = getConsumerProps();

        for (int i = 0; i < numberPartitions; i++) {
```

```
        Runnable consumerThread = getConsumerThread(properties);
        executorService.submit(consumerThread);
    }
}

private Runnable getConsumerThread(Properties properties) {
    return () -> {
         Consumer<String, String> consumer = null;
         try {
             consumer = new KafkaConsumer<>(properties);
             consumer.subscribe(Collections.singletonList(
           ➥ "test-topic"));
             while (!doneConsuming) {
                 ConsumerRecords<String, String> records =
              ➥ consumer.poll(5000);
                 for (ConsumerRecord<String, String> record : records) {
                     String message = String.format("Consumed: key =
                   ➥ %s value = %s with offset = %d partition = %d",
                             record.key(), record.value(),
                             record.offset(), record.partition());
                     System.out.println(message);
                 }

             }
         } catch (Exception e) {
             e.printStackTrace();
         } finally {
             if (consumer != null) {
                 consumer.close();
             }
         }
    };
}
```

Создаем поток выполнения потребителя

Подписываемся на топик

Опрос длительностью 5 секунд

Выводим отформатированное сообщение

Закрываем потребитель во избежание утечки ресурсов

Этот фрагмент кода несамодостаточен, я для ясности опустил в нем некоторые другие части класса. Полный код примера можно найти в исходном коде для данной главы.

2.6. Установка и запуск Kafka

На момент написания данной книги текущей версией Kafka является 1.0.0. Поскольку Kafka — проект для языка Scala, то каждый его выпуск поставляется в двух версиях: для Scala 2.11 и Scala 2.12. В данной книге мы будем использовать версию для Scala 2.12. Хотя вы можете скачать этот выпуск и сами, исходный код для книги включает в себя дистрибутив Kafka, подходящий для работы с Kafka Streams так, как показано и описано в книге. Для установки Kafka разархивируйте файл `.tgz` из репозитория исходного кода книги (его можно найти на сайте книги, вот здесь: https://manning.com/books/kafka-streams-in-action) куда-нибудь в каталог `libs` вашей машины.

ПРИМЕЧАНИЕ

Этот дистрибутив Kafka включает Apache ZooKeeper, так что устанавливать его дополнительно не надо.

2.6.1. Локальные настройки Kafka

Запуск Kafka на локальной машине требует лишь минимальных настроек, если согласиться со значениями по умолчанию. По умолчанию Kafka использует порт 9092, а ZooKeeper — порт 2181. Если у вас нет уже работающих на этих портах приложений, то все готово.

Kafka записывает журналы в каталог `/tmp/kafka-logs`, а ZooKeeper применяет для хранения журналов каталог `/tmp/zookeeper`. В зависимости от вашей машины может понадобиться изменить права доступа или владельца указанных каталогов или задать другое место для хранения журналов.

Для изменения каталога журналов Kafka перейдите с помощью команды `cd` в каталог `<kafka-install-dir>/config` и откройте файл `server.properties`. Найдите элемент `log.dirs` и измените значение на то, которое хотели бы использовать. В том же каталоге откройте файл `zookeeper.properties` и измените в нем элемент `dataDir`.

Мы обсудим настройку Kafka подробнее далее в этой книге, а пока описанных настроек вполне достаточно для наших целей. Не забывайте, что упомянутые «журналы» представляют собой реальные данные, используемые Kafka и ZooKeeper, а не журналы уровня приложения, служащие для отслеживания его поведения. Журналы приложения располагаются в каталоге `<kafka-install-dir>/logs`.

2.6.2. Запуск Kafka

Запуск Kafka не представляет сложности. Поскольку для должной работы кластера Kafka необходим ZooKeeper (ZooKeeper определяет ведущий брокер, хранит информацию о топиках, осуществляет проверки функционирования членов кластера и т. д.), вам придется запустить его до запуска Kafka.

ПРИМЕЧАНИЕ

Начиная с этого момента, во всех ссылках на каталоги предполагается, что вы работаете в каталоге, куда установлена Kafka. Если ваша машина работает под управлением операционной системы Windows, таким каталогом будет <kafka-install-dir>/bin/windows.

Запуск ZooKeeper

Для запуска ZooKeeper откройте командную строку и введите следующую команду[1]:

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

[1] Для ОС Windows команда имеет следующий вид: bin\windows\zookeeper-server-start.bat config\zookeeper.properties. — *Примеч. пер.*

После запуска на экране появится большое количество информации, и в результате вывод будет выглядеть примерно так, как показано на рис. 2.17.

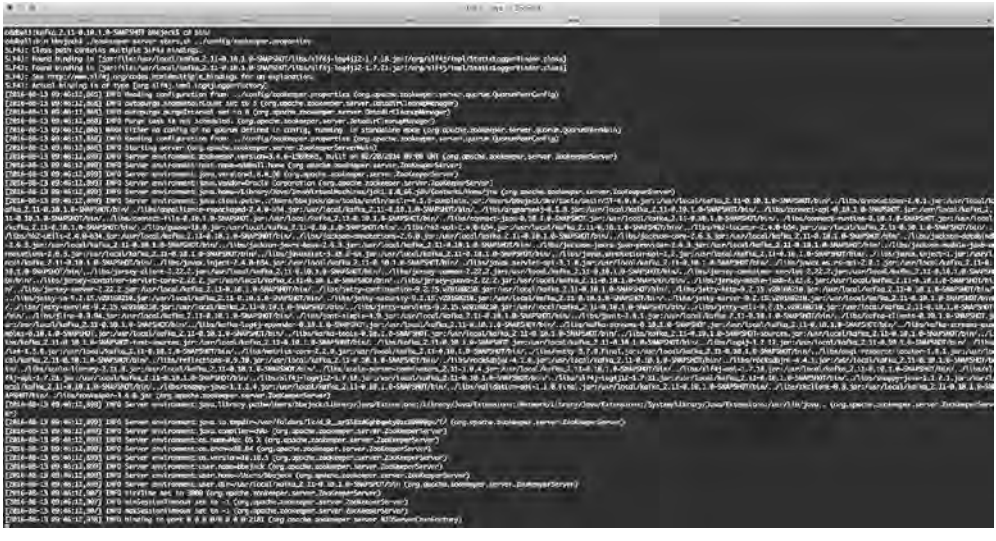


Рис. 2.17. Выводимые на экран результаты при запуске ZooKeeper

Запуск Kafka

Для запуска ZooKeeper откройте еще одну оболочку командной строки и введите вот такую команду[1]:

```
bin/Kafka-server-start.sh config/server.properties
```

Опять же вы увидите пробегающий по экрану текст. После завершения запуска Kafka на экране будет что-то вроде показанного на рис. 2.18.

СОВЕТ

> ZooKeeper необходим для работы Kafka, так что важно производить завершение работы в обратном порядке: сначала завершить работу Kafka, а затем — ZooKeeper. Для завершения работы Kafka можно нажать Ctrl+C в терминале, где запущена Kafka, или выполнить сценарий kafka-server-stop.sh[2] из другого терминала. То же самое относится и к ZooKeeper, за исключением того, что сценарий останова называется zookeeper-server-stop.sh[3].

---

[1] Для ОС Windows команда имеет следующий вид: bin\windows\Kafka-server-start.bat config\server.properties. — *Примеч. пер.*

[2] Для ОС Windows — kafka-server-stop.bat. — *Примеч. пер.*

[3] Для ОС Windows — zookeeper-server-stop.bat. — *Примеч. пер.*

Рис. 2.18. Вывод при запуске Kafka

2.6.3. Отправляем наше первое сообщение

После завершения настройки и запуска Kafka воспользуемся Kafka для ее главной задачи: отправки и получения сообщений. Но перед отправкой сообщения необходимо задать топик, в который генератор мог бы отправить сообщение.

Ваш первый топик

Ничего сложного в создании топика в Kafka нет. Нужно просто запустить сценарий, передав ему несколько параметров конфигурации. Настройки достаточно просты, но серьезно влияют на производительность.

По умолчанию Kafka создает топики автоматически, то есть при попытке отправки сообщения в несуществующий топик или чтения из него брокер Kafka создаст его (на основе настроек по умолчанию из файла `server.properties`). Полагаться в вопросе создания топиков на брокер, даже при разработке, — не самая лучшая идея, первая же попытка генерации/потребления данных завершится неудачей, ведь для распространения информации (метаданных) о существовании топика требуется время. Всегда создавайте топики заранее.

Создание топика

Для создания топика необходимо выполнить сценарий `kafka-topics.sh`[1]. Откройте окно терминала и выполните следующую команду:

```
bin/kafka-topics.sh --create --topic first-topic --replication-factor 1
➥ --partitions 1 --zookeeper localhost:2181
```

[1] Для ОС Windows — kafka-topics.bat. — *Примеч. пер.*

После выполнения сценария вы увидите в терминале что-то вроде показанного на рис. 2.19.

Рис. 2.19. Результаты создания топика. Важно создавать топики заранее, чтобы можно было задать для каждого из них нужные настройки. В противном случае в созданных автоматически топиках будут использованы конфигурация по умолчанию или настройки из файла server.properties

Большинство флагов конфигурации в предыдущей команде говорят сами за себя, остановимся подробнее лишь на двух из них:

- `replication-factor` — этот флаг задает число копий сообщения (считая исходное. — *Примеч. пер.*), распределяемое по кластеру ведущим брокером. В данном случае `replication-factor` равен `1`, так что никаких копий производиться не будет. В Kafka будет храниться только исходное сообщение. Для демонстрации или предварительной версии приложения вполне достаточно коэффициента репликации, равного `1`, но на практике почти всегда используется коэффициент репликации `2` или `3`, чтобы обеспечить доступность данных в случае отказов машин;
- `partitions` — этот флаг задает число применяемых топиком секций. Опять же для данного примера достаточно одной секции, но при более высокой нагрузке наверняка их понадобится больше. Определение правильного числа секций — своего рода искусство, а не точная наука.

Отправка сообщения

Отправка сообщения в Kafka обычно требует написания клиента генератора, но в Kafka есть также удобный сценарий `kafka-console-producer`, с помощью которого можно отправлять сообщение из окна терминала. В этом примере мы воспользуемся консольным генератором, но в подразделе 2.4.1 мы обсудили, как применять `KafkaProducer`.

Для отправки вашего первого сообщения наберите следующую команду (показана также на рис. 2.20):

```
# Предполагается, что эта команда выполняется из каталога bin
./kafka-console-producer.sh --topic first-topic --broker-list localhost:9092
```

Рис. 2.20. Консольный генератор — замечательный инструмент для быстрого тестирования настроек и проверки сквозной функциональности

Существует несколько опций настройки консольного генератора, но пока мы указали только обязательные: топик для отправки сообщения, список брокеров Kafka для подключения (в данном случае список состоит только из одного брокера на локальной машине).

Сценарий запуска консольного генератора — «блокирующий», так что после выполнения вышеприведенной команды введите какой-нибудь текст и нажмите клавишу Enter. Можно отправлять столько сообщений, сколько нужно, но для нашей демонстрации введите одно: `the quick brown fox jumped over the lazy dog`, нажмите клавишу Enter, а затем сочетание Ctrl+C, чтобы выйти из генератора.

Чтение сообщения

В Kafka также есть консольный потребитель для чтения сообщений из командной строки. Он похож на консольный генератор: его достаточно запустить один раз — и он будет читать сообщения из топика вплоть до завершения работы сценария вручную (нажатием Ctrl+C).

Для запуска консольного потребителя выполните следующую команду:

```
bin/kafka-console-consumer.sh --topic first-topic
➥ --bootstrap-server localhost:9092 --from-beginning
```

После запуска консольного потребителя вы увидите в терминале что-то наподобие изображенного на рис. 2.21.

Параметр `--from-beginning` означает, что вы получите все сообщения, не удаленные из данного топика. У консольного потребителя не будет каких-либо зафиксированных смещений, так что без настройки `--from-beginning` вы получите только сообщения, отправленные после его запуска.

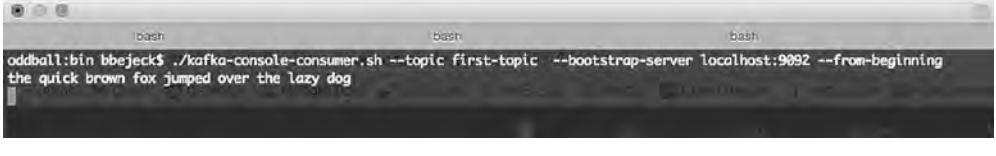


Рис. 2.21. Консольный потребитель — удобная утилита для того, чтобы понять, «текут» ли данные и содержат ли сообщения ожидаемую информацию

Вы только что завершили молниеносный тур по Kafka, а также сгенерировали и прочитали свое первое сообщение. Если вы не читали первую часть этой главы — самое время вернуться в начало и разобраться в нюансах работы Kafka!

Резюме

- Kafka представляет собой брокер сообщений, получающий сообщения и сохраняющий их таким образом, чтобы можно было удобно и быстро реагировать на запросы потребителей. Сообщения никогда не «проталкиваются» к потребителям, а сохранение сообщений в Kafka совершенно не зависит от того, когда и насколько часто они потребляются.

- Для достижения высокой пропускной способности, а также возможности упорядоченной группировки сообщений с одинаковыми ключами в Kafka используются секции.
- Для отправки сообщений в Kafka применяются генераторы.
- При пустых ключах секции назначаются в циклическом порядке; в противном случае генератор задействует для назначения секций хеш ключа по модулю числа секций.
- Для чтения сообщений в Kafka используются потребители.
- Топики/секции назначаются потребителям, состоящим в группе потребителей, так, чтобы попробовать добиться равномерного распределения сообщений.

В следующей главе мы займемся Kafka Streams на конкретном примере из мира розничных продаж. Хотя создание всех экземпляров генераторов и потребителей возьмет на себя Kafka Streams, вы сможете увидеть, как сыграют там свою роль понятия, с которыми я вас познакомил.

Часть II
Разработка с помощью Kafka Streams

Эта часть книги продолжает предыдущую, в ней я приведу ментальную модель Kafka Streams в действие, и мы создадим ваше первое приложение Kafka Streams. А после того как вы распробуете Kafka Streams, мы рассмотрим важнейшие API данной библиотеки.

Вы узнаете об использовании состояния в потоковых приложениях, в частности для выполнения соединений, подобных тем, с которыми вы могли иметь дело в SQL-запросах. Далее мы познакомимся с новой абстракцией из Kafka Streams — API `KTable`. Мы начнем эту часть книги с высокоуровневого предметно-ориентированного языка, а завершим ее обсуждением низкоуровневых API узлов-обработчиков, с помощью которых можно добиться от Kafka Streams практически всего, что вам от нее нужно.

3 Разработка приложений Kafka Streams

В этой главе:

- знакомство с Kafka Streams;
- создание приложения Hello World с помощью Kafka Streams;
- изучаем подробнее основанное на Kafka Streams приложение для ZMart;
- разбиваем входящий поток данных на несколько потоков.

Из главы 1 вы узнали про существование библиотеки Kafka Streams, о топологии узлов обработки — о графе преобразования данных при их потоковом поступлении в Kafka. В этой главе вы научитесь создавать такую топологию обработки с помощью API библиотеки Kafka Streams.

Для создания приложений Kafka Streams используются API Kafka Streams. Вы научитесь создавать приложения Kafka Streams, но, что важнее, станете лучше понимать, как компоненты взаимодействуют друг с другом и как с их помощью выполнить нужную потоковую обработку.

3.1. API потоковых узлов-обработчиков

Предметно-ориентированный язык Kafka Streams представляет собой высокоуровневый API, предназначенный для быстрого создания приложений Kafka Streams. Степень продуманности этого высокоуровневого API очень высока, в нем есть готовые методы для удовлетворения большинства потребностей потоковой обработки, что позволяет создавать сложные программы потоковой обработки без особых трудностей. В центре данного API лежит объект `KStream`, олицетворяющий потоковые записи пар «ключ/значение».

Большинство методов DSL Kafka Streams возвращает ссылку на объект `KStream`, что делает возможным стиль программирования с использованием текучих интерфейсов. Кроме того, немалая доля методов интерфейса `KStream` принимает на входе типы, состоящие из интерфейсов с одним методом, что позволяет применять

лямбда-выражения Java 8. Учитывая эти факторы, легко представить себе простоту и удобство создания приложений Kafka Streams.

Когда-то, в 2005 году, Мартин Фаулер (Martin Fowler) и Эрик Эванс (Eric Evans) придумали понятие *текучего интерфейса* (fluent interface) — интерфейса, в котором при вызове экземпляром класса метода возвращается этот же экземпляр (https://martinfowler.com/bliki/FluentInterface.html). Такой подход удобен при формировании объектов на основе нескольких параметров, например: `Person.builder().firstName("Beth").withLastName("Smith").withOccupation("CEO")`. В Kafka Streams существует одно маленькое, но важное отличие от такого подхода: возвращаемый объект `KStream` представляет собой новый экземпляр того же класса, а не тот, который вызвал метод.

Существует также низкоуровневый API узлов-обработчиков, не столь лаконичный, как DSL Kafka Streams, но предоставляющий больше возможностей контроля. Мы рассмотрим API узлов-обработчиков в главе 6. После этого введения можно приступить к созданию требуемой программы Hello World для Kafka Streams.

3.2. Программа Hello World для Kafka Streams

В нашем первом примере Kafka Streams мы временно оставим обрисованную в главе 1 задачу и возьмемся за более простой сценарий использования. Это позволит вам быстрее разобраться в функционировании Kafka Streams. Мы вернемся к задаче из главы 1 чуть позднее, в подразделе 3.2.1, с более реалистичным и конкретным примером.

Нашей первой программой будет «игрушечное» приложение, которое преобразует входящие сообщения к верхнему регистру — как бы кричит на читателей сообщений. Мы назовем его приложением Yelling («кричащим»).

Прежде чем углубиться в код, взглянем на топологию обработки этого приложения. Мы будем следовать той же схеме, что и в главе 1, где мы создавали топологию графа обработки, в котором у каждого узла была своя задача. Основное отличие состоит в том, что теперь граф будет проще, как можно видеть из рис. 3.1.

Как вы видите, мы создали простой граф обработки — настолько простой, что он напоминает скорее связный список узлов, чем необычную древовидную структуру графа. Но из него вполне понятно, что можно ожидать увидеть в коде: узел-источник, узел обработки, преобразующий входящий текст в верхний регистр, и обрабатывающий узел-сток, который записывает результаты в топик.

Это лишь простейший пример, но его код вполне показателен для программ Kafka Streams. Структура большинства примеров схожа.

1. Описание элементов конфигурации.
2. Создание экземпляров интерфейса Serde.
3. Построение топологии обработки.
4. Создание и запуск объекта KStream.

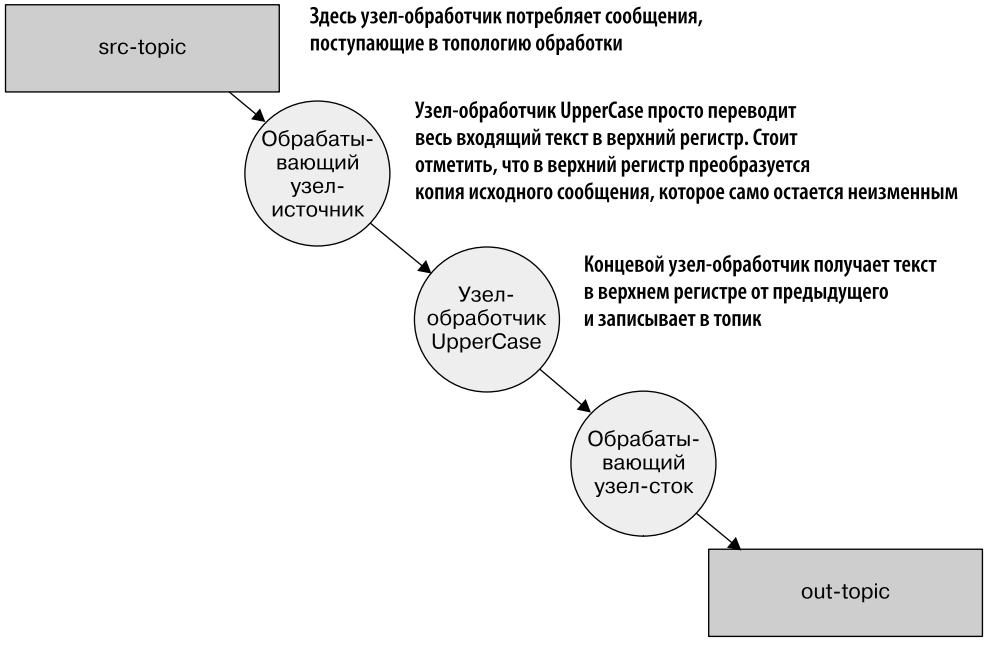


Рис. 3.1. Граф (топология) Yelling App

Как мы увидим, основное отличие в более продвинутых примерах будет заключаться в степени сложности топологии обработки. Приступим с учетом этого к созданию нашего первого приложения.

3.2.1. Создание топологии для Yelling

Первый шаг при создании любого приложения Kafka Streams — создание узла-источника. Узел-источник отвечает за потребление протекающих через приложение записей из топика. На рис. 3.2 в графе выделен узел-источник.

Следующая строка кода создает узел-источник (родительский узел) графа (листинг 3.1).

Листинг 3.1. Описываем источник для потока данных

```
KStream<String, String> simpleFirstStream = builder.stream("src-topic",
➥ Consumed.with(stringSerde, stringSerde));
```

Экземпляр класса `simpleFirstStreamKStream` настроен на потребление записанных в топик `src-topic` сообщений. Помимо указания имени топика, мы передаем также объекты типа `Serde` (через экземпляр класса `Consumed`). Мы будем применять класс `Consumed` для передачи необязательных параметров при каждом создании узла-источника в Kafka Streams.

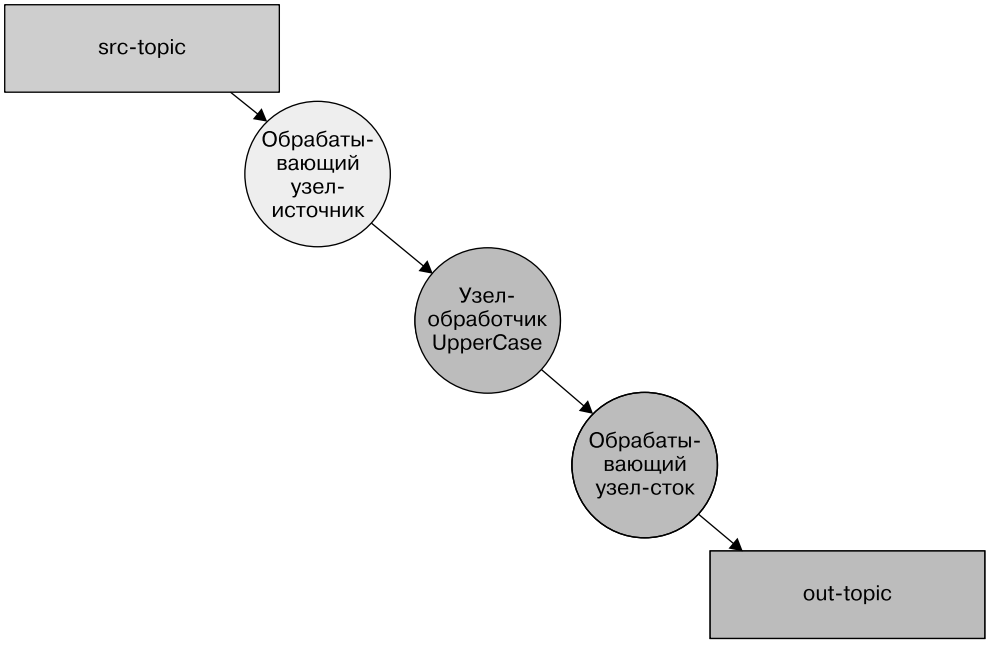


Рис. 3.2. Создание узла-источника приложения Yelling

У нас теперь есть узел-источник для нашего приложения, но нам нужно присоединить к нему обрабатывающий узел, чтобы воспользоваться этими данными, как показано на рис. 3.3. Код для присоединения узла-обработчика (дочернего для узла-источника) показан в следующем листинге. С помощью этой строки кода мы создаем еще один экземпляр `KStream` — дочерний узел первого родительского узла (листинг 3.2).

Листинг 3.2. Преобразование входящего текста в текст в верхнем регистре

```
KStream<String, String> upperCasedStream =
➥ simpleFirstStream.mapValues(String::toUpperCase);
```

Вызов функции `simpleFirstStream.mapValues` соответствует созданию нового обрабатывающего узла, на выходе которого можно видеть результаты прохождения данных через вызов метода mapValues.

Важно не забывать о том, что нельзя модифицировать *исходное* значение передаваемого в метод `mapValues` объекта `ValueMapper`. Экземпляр `upperCasedStream` получает уже преобразованные копии исходного значения, возвращаемые в результате вызова функции `simpleFirstStream.mapValues`. В данном случае это текст в верхнем регистре.

Метод `mapValues()` принимает на входе экземпляр интерфейса `ValueMapper<V, V1>`. В интерфейсе `ValueMapper<V, V1>` описан только один метод, `ValueMapper.apply`, что делает его идеальным кандидатом для применения лямбда-выражения Java 8. Поэтому мы и воспользовались тут ссылкой на метод `String::toUpperCase` — сокращенной формой лямбда-выражения Java 8.

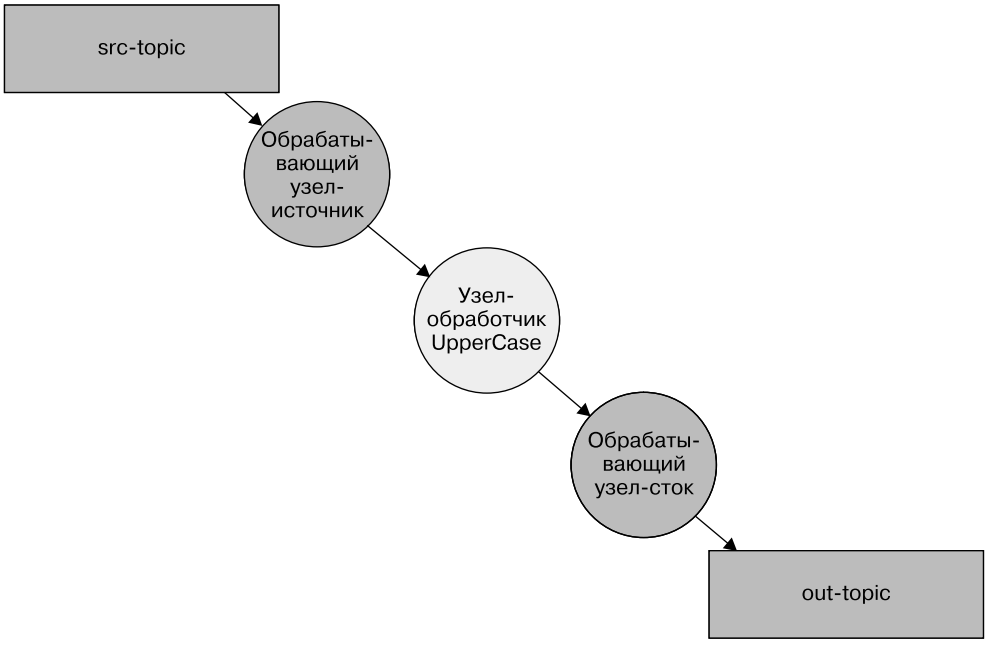


Рис. 3.3. Добавляем узел для преобразования в верхний регистр к приложению Yelling

ПРИМЕЧАНИЕ

Существует множество руководств по лямбда-выражениям Java 8 и ссылкам на методы. Для начала можно обратиться к документации Oracle по языку Java, разделам Lambda Expressions («Лямбда-выражения», http://mng.bz/J0Xm) и Method References («Ссылки на методы», http://mng.bz/BaDW).

Можно было также воспользоваться формой `s → s.toUpperCase()`, но, поскольку `toUpperCase` представляет собой метод экземпляра класса `String`, мы применили ссылку на метод.

Мы часто будем встречаться с паттерном использования лямбда-выражений вместо конкретных реализаций при работе с потоковым API узлов-обработчиков в данной книге. Поскольку большинство методов этих API ожидают на входе типы, представляющие собой интерфейсы с одним методом, почти всегда можно применять лямбда-выражения Java 8.

Пока наше приложение Kafka Streams потребляет записи и преобразовывает их в верхний регистр. Нам осталось только добавить обработчик-сток, который бы записывал результаты в топик. Рисунок 3.4 демонстрирует, до какого места в создании топологии мы добрались.

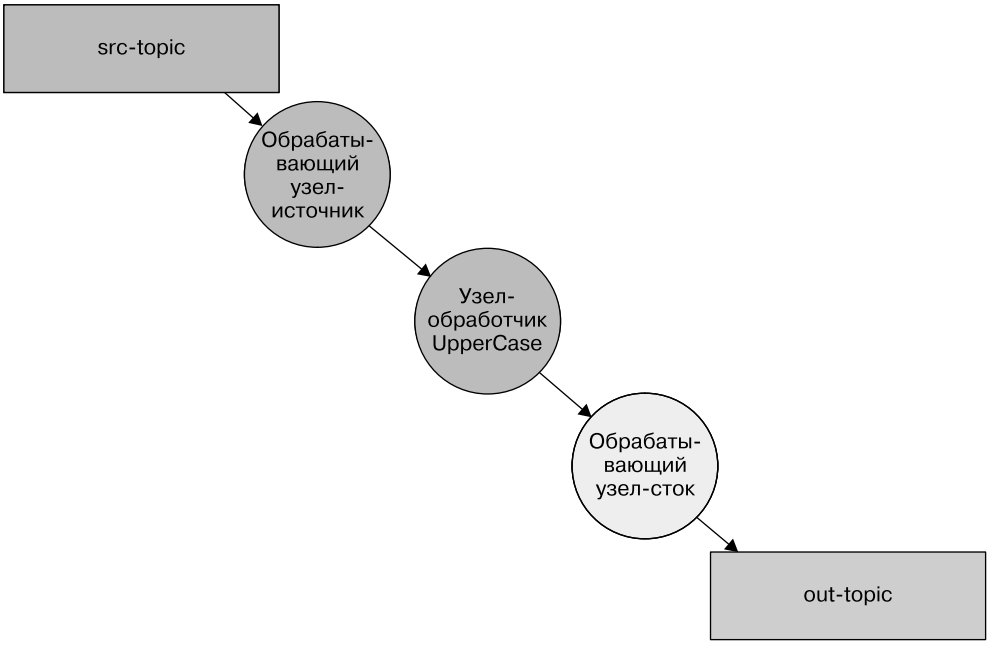


Рис. 3.4. Добавляем узел-обработчик для записи результатов приложения Yelling

Следующая строка кода добавляет в граф последний узел-обработчик (листинг 3.3).

Листинг 3.3. Создание узла-стока

```
upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

Метод `KStream.to` создает в топологии обрабатывающий узел-сток. Обрабатывающие узлы-стоки вносят записи обратно в Kafka. Данный узел-сток получает записи от узла-обработчика `upperCasedStream` и записывает их в топик `out-topic`. Опять же мы передаем в качестве параметров экземпляры `Serde`, на этот раз для сериализации записываемых в топик Kafka записей. Но в данном случае мы задействуем экземпляр класса `Produced`, используемый для передачи необязательных параметров при создании узла-стока в Kafka Streams.

ПРИМЕЧАНИЕ

Передавать объекты Serde объектам Consumed и Produced вовсе не обязательно. Если этого не сделать, приложение будет использовать указанный в конфигурации сериализатор/десериализатор. Кроме того, с помощью классов Consumed и Produced можно задать объект Serde только для ключа или только для значения.

В предыдущем примере топология создается с помощью трех строк кода:

```
KStream<String,String> simpleFirstStream =
➥ builder.stream("src-topic", Consumed.with(stringSerde, stringSerde));
KStream<String, String> upperCasedStream =
➥ simpleFirstStream.mapValues(String::toUpperCase);
   upperCasedStream.to("out-topic", Produced.with(stringSerde, stringSerde));
```

Для демонстрации различных этапов процесса создания топологии мы прибегли к пошаговому созданию, где одна строка соответствует одному шагу. Но все методы API интерфейса `KStream`, кроме создающих концевые узлы, возвращают новый экземпляр `KStream`, благодаря чему можно воспользоваться вышеупомянутым стилем программирования текучих интерфейсов. Для демонстрации этой идеи покажу альтернативный способ создания топологии приложения Yelling:

```
builder.stream("src-topic", Consumed.with(stringSerde, stringSerde))
➥ .mapValues(String::toUpperCase)
➥ .to("out-topic", Produced.with(stringSerde, stringSerde));
```

Мы сократили программу с трех строк до одной без потерь понятности или функциональности. Начиная с данного момента, все примеры в книге будут использовать стиль текучих интерфейсов, за исключением случаев, когда это повлекло бы за собой снижение понятности программы.

Первая ваша топология Kafka Streams готова, но мы обошли стороной важные шаги настройки и создания объектов `Serde`. Давайте теперь рассмотрим их.

3.2.2. Настройка Kafka Streams

Хотя Kafka Streams предоставляет очень широкие возможности настройки и множество свойств, которые можно менять под свои нужды, в нашем первом примере будут использоваться только два параметра конфигурации — `APPLICATION_ID_CONFIG` и `BOOTSTRAP_SERVERS_CONFIG`:

```
props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");
```

Оба параметра обязательны, поскольку у них нет значений по умолчанию. Попытка запуска программы Kafka Streams с неопределенными значениями этих двух свойств приведет к генерации исключения `ConfigException`.

Свойство `StreamsConfig.APPLICATION_ID_CONFIG` идентифицирует приложение Kafka Streams, его значение должно быть уникальным в пределах всего кластера. Оно также служит значением по умолчанию для префикса идентификатора клиента и идентификатора группы, если они не заданы. Префикс идентификатора клиента представляет собой задаваемое пользователем значение, которое однозначно идентифицирует подключающиеся к Kafka клиенты. Идентификатор группы применяется для управления членством в группе потребителей, читающих из одного топика, обеспечивая возможность чтения потребителями топиков, на которые они подписаны.

Свойство `StreamsConfig.BOOTSTRAP_SERVERS_CONFIG` может представлять собой одну или несколько пар `имя_хоста:порт`, разделенных запятыми. Значение этого параметра указывает приложению Kafka Streams на расположение кластера Kafka. Мы обсудим еще несколько элементов настроек по мере их появления в дальнейших примерах в книге.

3.2.3. Создание объектов Serde

Класс `SerDes`[1] в Kafka Streams предоставляет удобные методы для создания экземпляров `Serde`:

```
Serde<String> stringSerde = Serdes.String();
```

В этой строке мы создаем экземпляр `Serde`, необходимый для сериализации/десериализации с помощью класса `Serdes`. В ней мы создаем переменную, которая будет использоваться в топологии неоднократно для ссылки на объект `Serde`. Класс `Serdes` обеспечивает реализации по умолчанию для таких типов данных, как:

- строковый тип;
- байтовый массив;
- длинное целое;
- целое;
- число с двойной точностью.

Польза реализаций интерфейса `Serde` заключается в наличии сериализатора/десериализатора, благодаря чему вам не требуется указывать четыре параметра (сериализатор ключа, сериализатор значения, десериализатор ключа, десериализатор значения) всякий раз, когда требуется передать объект `Serde` в метод класса `KStream`. В следующем примере мы создадим реализацию `Serde`, которая будет отвечать за сериализацию/десериализацию более сложных типов.

Посмотрим на нашу программу целиком. Исходный код можно найти в файле src/main/java/bbejeck/chapter_3/KafkaStreamsYellingApp.java (исходный код для данной книги размещен на ее сайте по адресу https://manning.com/books/kafka-streams-in-action) (листинг 3.4).

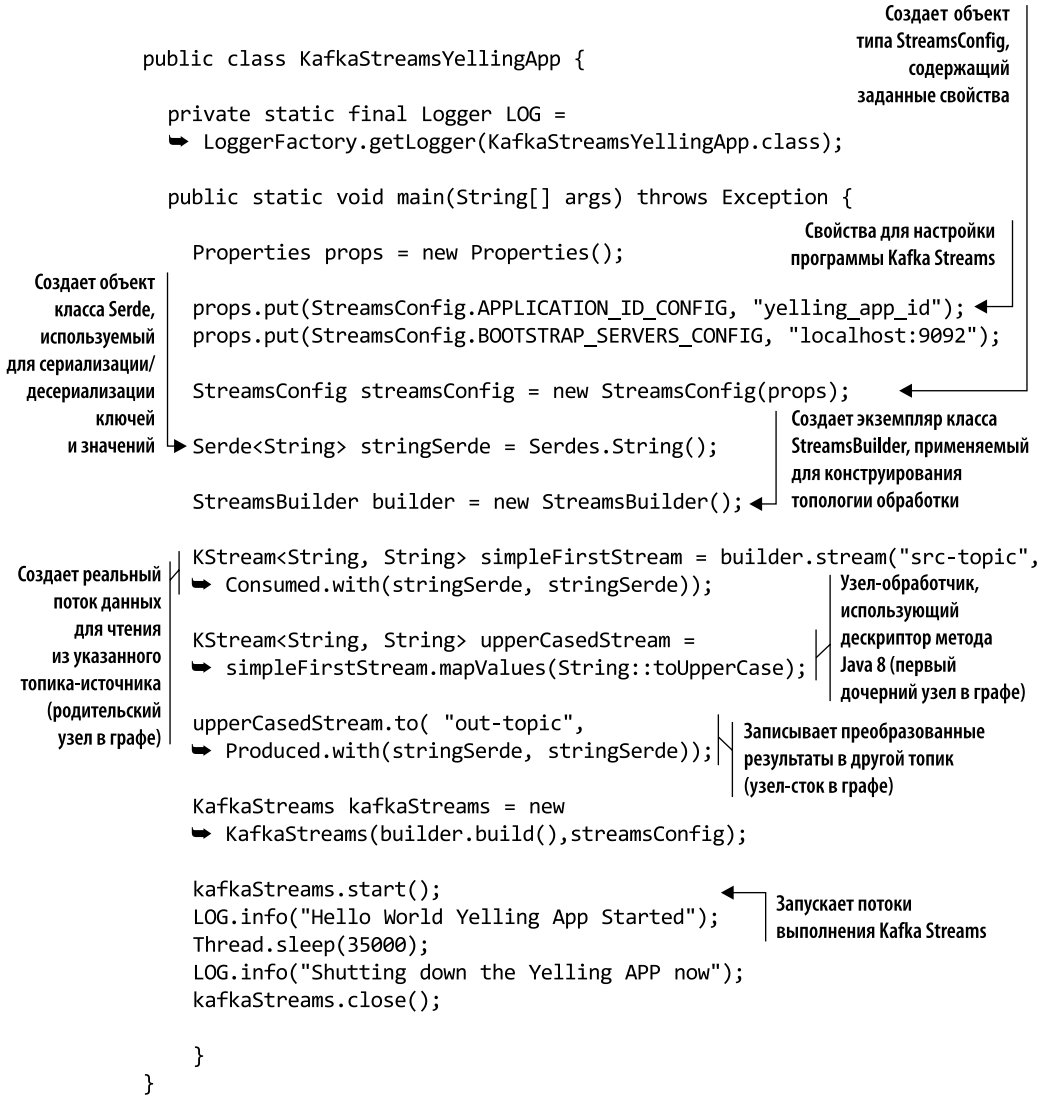
Вы создали свое первое приложение Kafka Streams. Вкратце перечислю этапы его создания, поскольку такой же паттерн встретится вам в большинстве приложений Kafka Streams.

1. Создание экземпляра `StreamsConfig`.
2. Создание объекта `Serde`.
3. Построение топологии обработки.
4. Запуск программы Kafka Streams.

---

[1] От англ. Serializer/Deserializer — «сериализатор/десериализатор». — *Примеч. пер.*

Листинг 3.4. Программа Hello World: приложение Yelling




```
public class KafkaStreamsYellingApp {

  private static final Logger LOG =
  ➥ LoggerFactory.getLogger(KafkaStreamsYellingApp.class);

  public static void main(String[] args) throws Exception {

    Properties props = new Properties();

    props.put(StreamsConfig.APPLICATION_ID_CONFIG, "yelling_app_id");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "localhost:9092");

    StreamsConfig streamsConfig = new StreamsConfig(props);

    Serde<String> stringSerde = Serdes.String();

    StreamsBuilder builder = new StreamsBuilder();

    KStream<String, String> simpleFirstStream = builder.stream("src-topic",
    ➥ Consumed.with(stringSerde, stringSerde));

    KStream<String, String> upperCasedStream =
    ➥ simpleFirstStream.mapValues(String::toUpperCase);

    upperCasedStream.to( "out-topic",
    ➥ Produced.with(stringSerde, stringSerde));

    KafkaStreams kafkaStreams = new
    ➥ KafkaStreams(builder.build(),streamsConfig);

    kafkaStreams.start();
    LOG.info("Hello World Yelling App Started");
    Thread.sleep(35000);
    LOG.info("Shutting down the Yelling APP now");
    kafkaStreams.close();

    }
}
```

Помимо общей информации о создании приложений Kafka Streams, надеюсь, вы сделали из вышеизложенного вывод о необходимости применять лямбда-выражения везде, где это только возможно, ради повышения лаконичности ваших программ.

Приступим теперь к более сложному примеру, который позволит вам изучить дополнительные API потоковых узлов-обработчиков. Хотя пример новый, сценарий использования вам уже хорошо знаком — обработка данных для ZMart.

3.3. Работа с данными покупателей

В главе 1 мы обсуждали новые требования ZMart по обработке данных покупателей, целью которых было повышение эффективности ведения бизнеса. Вы узнали, как создать топологию узлов для обработки записей о покупках по мере их поступления из потока данных о транзакциях в магазинах ZMart. На рис. 3.5 снова показан полный граф топологии.

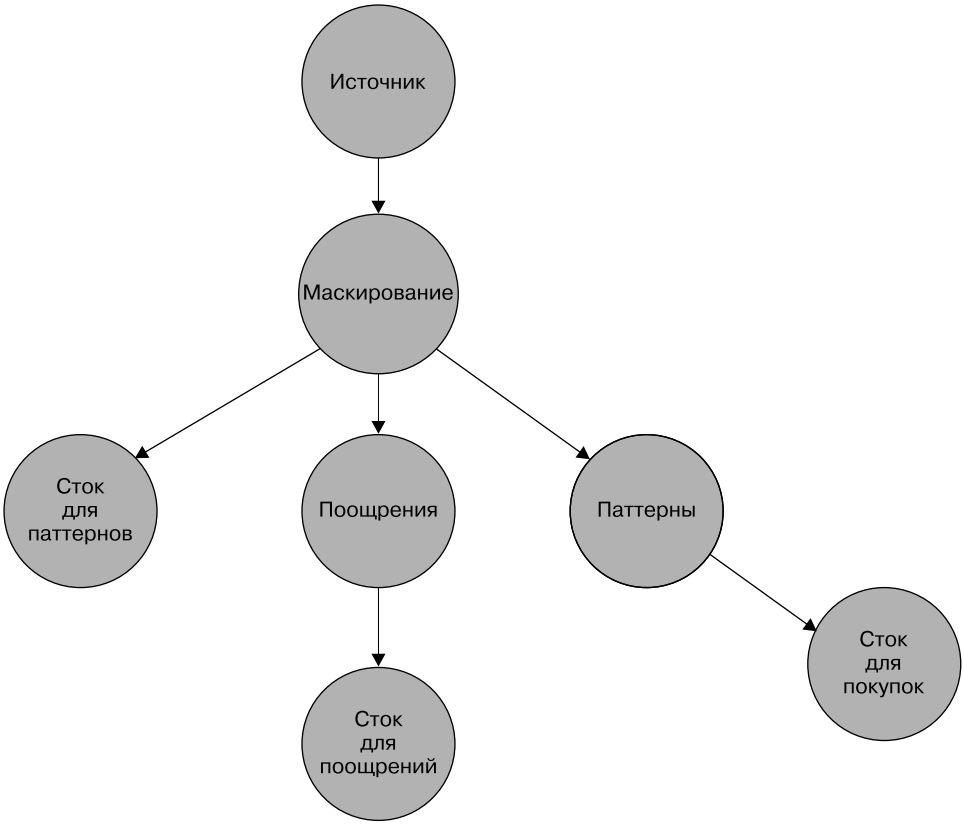


Рис. 3.5. Топология программы Kafka Streams для ZMart

Вкратце рассмотрим требования к программе потоковой обработки, которые заодно отлично описывают, что эта программа будет делать:

- номера платежных карт во всех записях должны быть защищены, в данном случае путем маскирования первых 12 цифр;
- для определения паттернов покупок необходимо извлечь информацию о купленных товарах и почтовых индексах, которая далее будет записана в топик;

- необходимо собрать номера карт постоянных покупателей ZMart, а также данные о потраченных суммах и записать эту информацию в топик. Потребители топика на основе указанных данных смогут определить число бонусов покупателей;
- необходимо записать в топик транзакцию целиком для последующего чтения движком хранения и ситуативного анализа.

Как и в приложении Yelling, при создании приложения мы будем сочетать текучие интерфейсы с лямбда-выражениями Java 8. Хотя иногда очевидно, что при вызове метода возвращается тип `KStream`, но не всегда. Напомню, что большинство методов API-класса `KStream` возвращают *новые* экземпляры `KStream`. А теперь создадим потоковое приложение, которое бы удовлетворяло бизнес-требованиям компании ZMart.

3.3.1. Конструирование топологии

Займемся построением топологии обработки. Чтобы вам было понятнее, к какой части топологии относится определенный фрагмент создаваемого кода, я буду выделять на рисунках часть графа, над которой мы в этот момент будем работать.

Создание узла-источника

Мы начнем с создания узла-источника и первого узла-обработчика топологии, соединив цепочкой два вызова API `KStream` (выделено на рис. 3.6). Вам должно быть уже ясно, какова роль этого начального узла. Первый узел-обработчик топологии будет отвечать за маскирование номеров платежных карт с целью защиты персональной информации покупателей.

Узел-источник создается с помощью вызова метода `StreamsBuilder.stream` с передаваемыми ему в качестве параметров строковым объектом `Serde` по умолчанию, пользовательским `Serde` для объектов `Purchase` и именем топика — источника сообщений для потока. В данном случае мы указываем только один топик, но можно указывать и разделенный запятыми список имен или регулярное выражение для имен топиков.

В листинге 3.5 мы передали объекты `Serde` с помощью экземпляра класса `Consumed`, но могли и опустить его и передать только имя топика, оставив объекты `Serde` по умолчанию, из настроек конфигурации.

Листинг 3.5. Создание узла-источника и первого узла-обработчика

```
KStream<String,Purchase> purchaseKStream =
➥ streamsBuilder.stream("transactions",
➥ Consumed.with(stringSerde, purchaseSerde))
➥ .mapValues(p -> Purchase.builder(p).maskCreditCard().build());
```

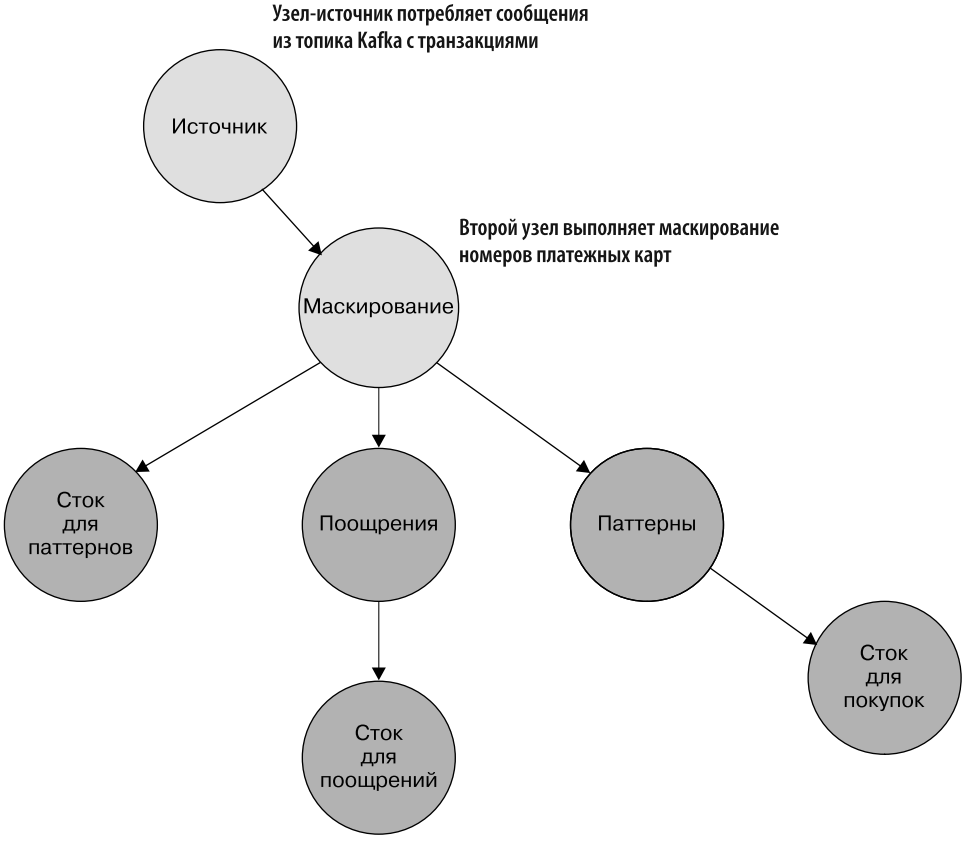


Рис. 3.6. Узел-источник потребляет из топика Kafka, поставляя при этом данные только маскирующему узлу-обработчику, вследствие чего тот становится источником для всей остальной топологии

Непосредственно за ним следует вызов метода `KStream.mapValues`, который принимает на входе экземпляр `ValueMapper<V, V1>` в качестве параметра. `ValueMapper` получают один параметр одного типа (в данном случае объект `Purchase`) и отображают его в новое значение, возможно, другого типа. В этом примере метод `KStream.mapValues` возвращает объект того же типа (`Purchase`), но уже с маскированным номером платежной карты.

Обратите внимание, что при применении метода `KStream.mapValues` исходный ключ не меняется и не учитывается при отображении в новое значение. При необходимости сгенерировать новую пару «ключ/значение» или задействовать ключ при генерации нового значения можно воспользоваться методом `KStream.map`, который принимает в качестве параметра экземпляр `KeyValueMapper<K, V, KeyValue<K1, V1>>`.

Советы по функциональному программированию

У функций `map` и `mapValues` важно учитывать то, что при их работе предполагается отсутствие побочных эффектов, то есть передаваемые как параметры объект или значения не должны изменяться. Происходит это из-за свойств функционального программирования в API `KStream`. Функциональное программирование — обширная тема, всестороннее обсуждение которой выходит за рамки данной книги, но мы вкратце рассмотрим тут два основных принципа функционального программирования.

Первый из них: избегать изменения состояния. Если необходимо изменить или обновить объект, он передается функции и создается его копия или совершенно новый экземпляр со всеми требуемыми изменениями или обновлениями. В листинге 3.5 для внесения в объект `Purchase` обновлений, связанных с маскированием номеров платежных карт, методу `KStream.mapValues` передается лямбда-выражение. Значение поля с номером платежной карты в исходном объекте `Purchase` не меняется.

Второй принцип: построение сложных операций путем композиции нескольких более мелких специализированных функций. Композиция функций — часто встречающийся при работе с API `KStream` паттерн.

ОПРЕДЕЛЕНИЕ

> В рамках этой книги я буду называть *функциональным программированием* (functional programming) такой подход к программированию, при котором функции являются объектами первого класса. Более того, предполагается, что функции стараются не создавать побочных эффектов, например не модифицировать состояния или изменяемые объекты.

Создание второго узла-обработчика

Теперь мы приступим к созданию второго узла-обработчика, отвечающего за извлечение из топика информации о паттернах, на основе которых ZMart сможет определить паттерны покупок в различных областях страны. Мы также добавим в топологию узел-сток, отвечающий за запись информации о паттернах в топик Kafka. Их устройство показано на рис. 3.7.

В листинге 3.6 видно, как узел-обработчик `purchaseKStream` использует уже знакомый нам вызов метода `mapValues` для создания экземпляра `KStream`. Этот новый объект `KStream` будет получать объекты `purchaseKStream`, созданные в результате вызова метода `mapValues`.

Листинг 3.6. Второй узел-обработчик и записывающий данные в Kafka узел-сток

```
KStream<String, PurchasePattern> patternKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ PurchasePattern.builder(purchase).build());
patternKStream.to("patterns",
➥ Produced.with(stringSerde,purchasePatternSerde));
```

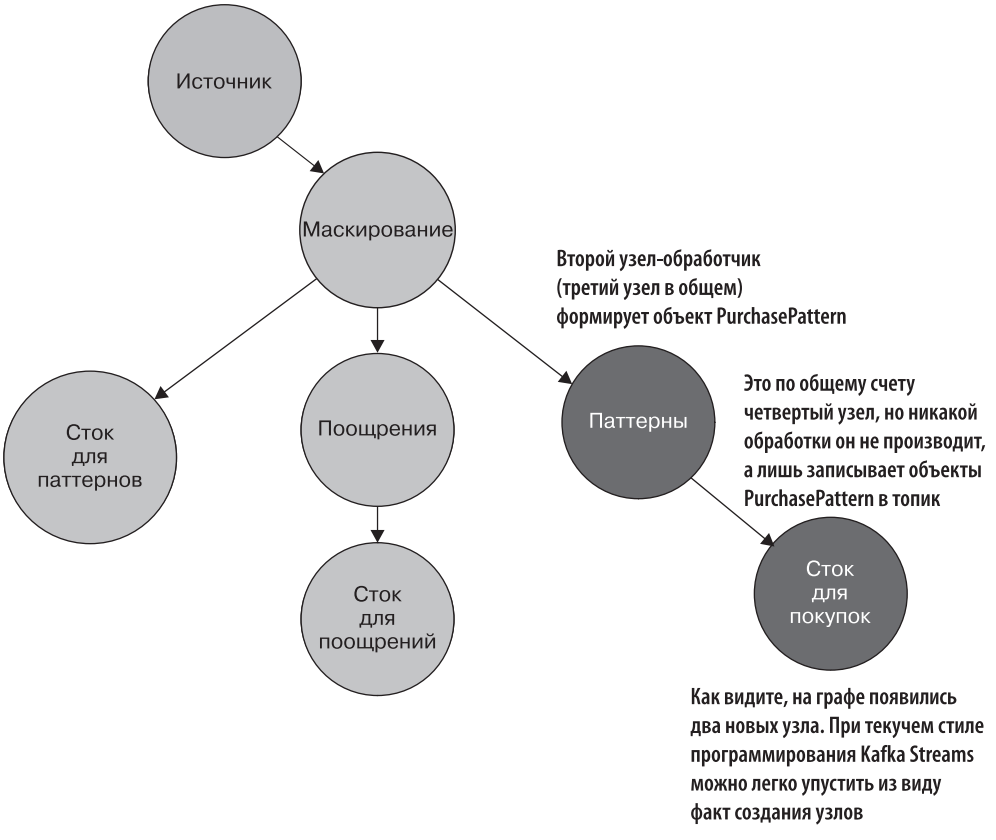


Рис. 3.7. Второй узел-обработчик создает объект с информацией о паттернах покупок. А узел-сток записывает объект PurchasePattern в топик

Тут мы объявляем переменную для хранения ссылки на новый экземпляр `KStream`, которая пригодится нам при выводе результатов потока данных в консоль с помощью вызова `print`. Это очень удобно при разработке и отладке. Узел-обработчик паттернов покупок переправляет полученные им записи своему описанному в вызове метода `KStream.to` дочернему узлу, который записывает данные в топик `patterns`. Обратите внимание, как объект `Produced` используется для передачи заранее созданного объекта `Serde`.

Метод `KStream.to` — полная противоположность методу `KStream.source`. Вместо создания источника данных для топологии метод `KStream.to` задает узел-сток, записывающий данные из экземпляра `KStream` в топик Kafka. Существуют также перегруженные версии метода `KStream.to`, в которых можно опустить параметр `Produced` и применять заданные в настройках `Serde` по умолчанию. С помощью класса `Produced` можно также задать несколько необязательных параметров, один из которых, `StreamPartitioner`, мы обсудим далее.

Создание третьего узла-обработчика

Третьим узлом-обработчиком в топологии является накопитель бонусов покупателей, показанный на рис. 3.8, благодаря которому ZMart имеет возможность отслеживать покупки, совершаемые членами клуба постоянных покупателей (листинг 3.7). Накопитель бонусов отправляет данные в топик, откуда их потребляют приложения в штаб-квартире компании ZMart, вычисляющие бонусы по завершении покупки.

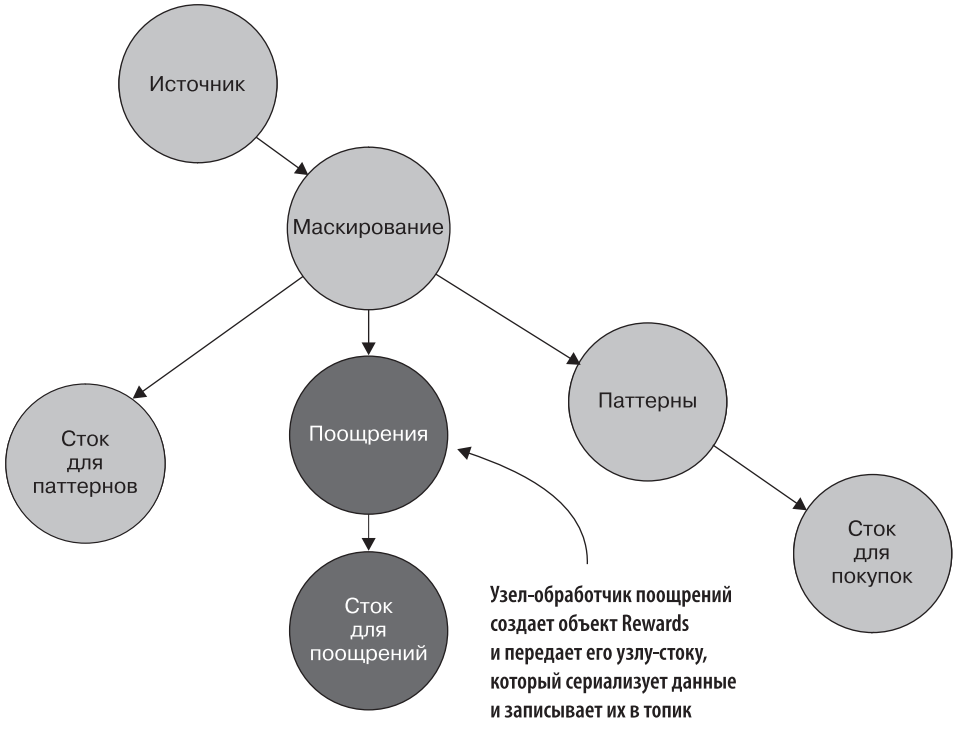


Рис. 3.8. Третий узел-обработчик создает объект RewardAccumulator. А завершающий узел записывает результаты в топик Kafka

Листинг 3.7. Третий узел-обработчик и записывающий в Kafka завершающий узел

```
KStream<String, RewardAccumulator> rewardsKStream =
➥ purchaseKStream.mapValues(purchase ->
➥ RewardAccumulator.builder(purchase).build());
rewardsKStream.to("rewards",
➥ Produced.with(stringSerde,rewardAccumulatorSerde));
```

Для создания узла-обработчика поощрений мы воспользовались уже знакомым вам паттерном: создали новый экземпляр `KStream`, отображающий содержащиеся в записях необработанные данные о покупках в объекты нового типа. Мы также присоединили к накопителю бонусов узел-сток для записи в топик объектов

KStream с информацией о поощрениях и последующего определения количества бонусов покупателей.

Создание последнего узла-обработчика

И наконец, мы присоединим к первому из созданных нами объектов KStream, purchaseKStream узел-сток для записи необработанных данных о покупках (с маскированными номерами платежных карт, конечно) в топик под названием purchases (листинг 3.8). Из него информация будет попадать в NoSQL-хранилище, например, Cassandra (http://cassandra.apache.org/), Presto (https://prestodb.io/) или Elastic Search (www.elastic.co/webinars/getting-started-elasticsearch) для последующего ситуативного анализа. Этот последний узел-обработчик показан на рис. 3.9.

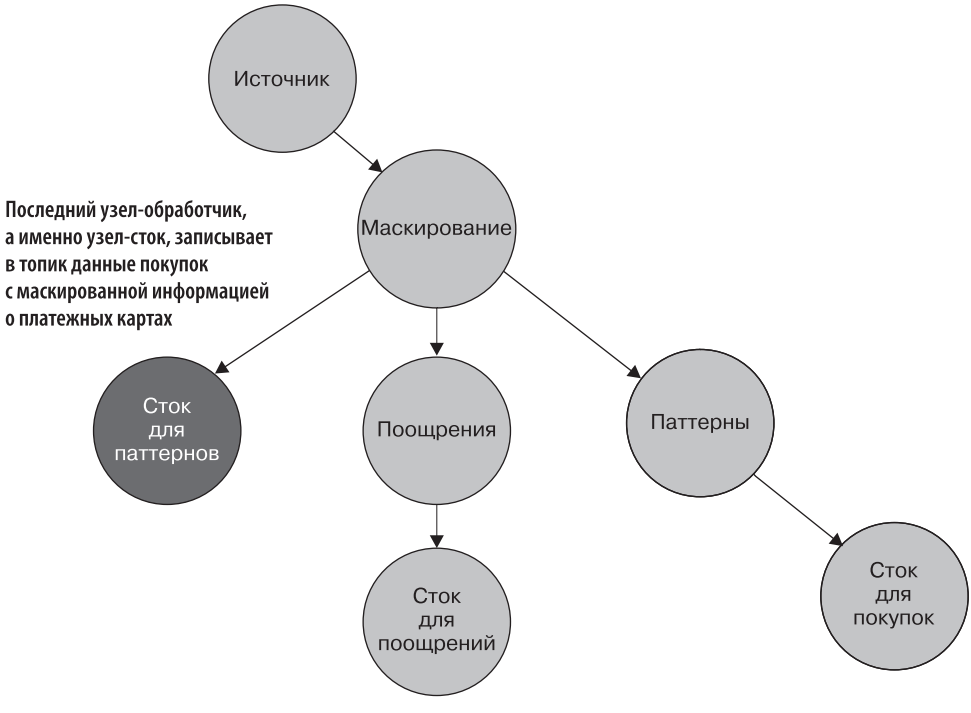


Рис. 3.9. Последний узел-обработчик записывает транзакцию покупки целиком в топик, потребителем которого является NoSQL-хранилище данных

Листинг 3.8. Последний узел-обработчик

```
purchaseKStream.to("purchases", Produced.with(stringSerde, purchaseSerde));
```

Теперь, по завершении создания приложения по кусочкам, посмотрим на него целиком (src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsApp.java) (листинг 3.9). Как легко заметить, оно намного сложнее, чем предыдущий пример Hello World (приложение Yelling).

Листинг 3.9. Программа KStream для покупок компании ZMart

```
public class ZMartKafkaStreamsApp {

  public static void main(String[] args) throws Exception {
    // Некоторые подробности для ясности опущены

    StreamsConfig streamsConfig = new StreamsConfig(getProperties());

    JsonSerializer<Purchase> purchaseJsonSerializer = new
    ➥  JsonSerializer<>();
    JsonDeserializer<Purchase> purchaseJsonDeserializer =
    ➥  new JsonDeserializer<>(Purchase.class);
    Serde<Purchase> purchaseSerde =
    ➥  Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);
    // Остальные объекты Serde опущены для большей ясности

    Serde<String> stringSerde = Serdes.String();

    StreamsBuilder streamsBuilder = new StreamsBuilder();

    KStream<String,Purchase> purchaseKStream =
    ➥  streamsBuilder.stream("transactions",
    ➥  Consumed.with(stringSerde, purchaseSerde))
    ➥  .mapValues(p -> Purchase.builder(p).maskCreditCard().build());

    KStream<String, PurchasePattern> patternKStream =
    ➥  purchaseKStream.mapValues(purchase ->
    ➥  PurchasePattern.builder(purchase).build());

    patternKStream.to("patterns",
    ➥  Produced.with(stringSerde,purchasePatternSerde));

    KStream<String, RewardAccumulator> rewardsKStream =
    ➥  purchaseKStream.mapValues(purchase ->
    ➥  RewardAccumulator.builder(purchase).build());

    rewardsKStream.to("rewards",
    ➥  Produced.with(stringSerde,rewardAccumulatorSerde));

    purchaseKStream.to("purchases",
    ➥  Produced.with(stringSerde,purchaseSerde));

    KafkaStreams kafkaStreams =
    ➥  new KafkaStreams(streamsBuilder.build(),streamsConfig);
    kafkaStreams.start();

  }
}
```

Создает объект Serde; формат данных — JSON

Создает источник и первый узел-обработчик

Создает узел-обработчик PurchasePattern

Создает узел-обработчик RewardAccumulator

Создает узел-сток для сохранения, топик используется потребителем хранилища

ПРИМЕЧАНИЕ

Я опустил некоторые подробности в листинге 3.9 для ясности. Учтите, что примеры кода из книги далеко не всегда самодостаточны. Примеры целиком можно найти в прилагаемом к книге исходном коде.

Как вы можете видеть, этот пример несколько сложнее, чем приложение Yelling, но его информационный поток аналогичен. А именно этапы его создания попрежнему были следующими.

1. Создание экземпляра `StreamsConfig`.
2. Создание одного или нескольких экземпляров класса `Serde`.
3. Формирование топологии обработки.
4. Сбор всех компонентов воедино и запуск программы Kafka Streams.

В этом приложении упоминалось создание объектов `Serde`, но я не объяснял, почему или как их создавать. Давайте потратим еще немного времени и обсудим роль класса `Serde` в приложении Kafka Streams.

3.3.2. Создание пользовательского объекта Serde

Kafka производит передачу данных в виде байтовых массивов. Поскольку формат данных — JSON, необходимо сообщить Kafka, как преобразовать объект сначала в JSON, а затем, при отправке в топик, в байтовый массив. И наоборот, необходимо указать способ преобразования прочитанных байтовых массивов в JSON, а затем — в используемый узлами-обработчиками тип объектов. Именно для такого преобразования данных в различные форматы и из них необходимы объекты `Serde`. Kafka предоставляет несколько готовых объектов `Serde` через клиентские зависимости Kafka (`String`, `Long`, `Integer` и т. п.), но для других объектов нужно создавать свои.

В нашем первом примере, приложении Yelling, требовался лишь сериализатор/десериализатор для строковых значений, реализации которых предоставляет фабричный метод `Serdes.String()`. В примере ZMart, однако, нам придется создавать пользовательские экземпляры `Serde` в силу произвольности типов объектов. Мы увидим далее, что требуется для создания объекта `Serde` для класса `Purchase`. Я не стану описывать создание других экземпляров `Serde`, поскольку оно следует тому же паттерну, только с другими типами.

Для создания объекта `Serde` необходимо реализовать интерфейсы `Deserializer<T>` и `Serializer<T>`. В дальнейших примерах мы будем использовать реализации, приведенные в листингах 3.10 и 3.11, а также библиотеку Gson от компании Google для преобразования объектов в/из формата JSON. В следующем листинге приведен сериализатор, который также можно найти в файле src/main/java/bbejeck/util/serializer/JsonSerializer.java.

Для сериализации мы сначала преобразовываем объект в формат JSON, а затем получаем для этого строкового значения представление в виде байтового массива. Для преобразования в/из JSON в данном примере используется библиотека Gson (https://github.com/google/gson).

Десериализация требует другой последовательности действий: создания нового строкового значения из байтового массива и дальнейшего преобразования строки JSON в Java-объект с помощью библиотеки Gson. Этот обобщенный десериализатор можно найти в файле src/main/java/bbejeck/util/serializer/JsonDeserializer.java.

Листинг 3.10. Обобщенный сериализатор

```
public class JsonSerializer<T> implements Serializer<T> {

    private Gson gson = new Gson();

    @Override
    public void configure(Map<String, ?> map, boolean b) {

    }

    @Override
    public byte[] serialize(String topic, T t) {
        return gson.toJson(t).getBytes(Charset.forName("UTF-8"));
    }

    @Override
    public void close() {

    }
}
```

Создает объект класса Gson

Сериализует объект в байтовое представление

Листинг 3.11. Обобщенный десериализатор

```
public class JsonDeserializer<T> implements Deserializer<T> {

    private Gson gson = new Gson();
    private Class<T> deserializedClass;

    public JsonDeserializer(Class<T> deserializedClass) {
        this.deserializedClass = deserializedClass;
        init();
    }

    public JsonDeserializer() {
    }

    @Override
    @SuppressWarnings("unchecked")
    public void configure(Map<String, ?> map, boolean b) {
        if(deserializedClass == null) {
            deserializedClass = (Class<T>) map.get("serializedClass");
        }
    }

    @Override
    public T deserialize(String s, byte[] bytes) {
         if(bytes == null){
             return null;
         }

         return gson.fromJson(new String(bytes),deserializeFrom);

    }
```

Создает объект класса Gson

Десериализуемая переменная экземпляра класса Class

Десериализует байтовый массив в экземпляр ожидаемого класса Class

```
    @Override
    public void close() {

    }
}
```

Далее вернемся к следующим строкам из листинга 3.9:

```
JsonDeserializer<Purchase> purchaseJsonDeserializer =
➥ new JsonDeserializer<>(Purchase.class);
JsonSerializer<Purchase> purchaseJsonSerializer =
➥ new JsonSerializer<>();
Serde<Purchase> purchaseSerde =
➥ Serdes.serdeFrom(purchaseJsonSerializer, purchaseJsonDeserializer);
```

Создает десериализатор для класса Purchase

Создает сериализатор для класса Purchase

Создает объект Serde для объектов класса Purchase

Как вы можете видеть, объект `Serde` удобен в качестве контейнера для сериализатора и десериализатора для заданного объекта.

Я уже рассказал немало про разработку приложения Kafka Streams. И хотя нужно рассказать еще намного больше, сделаем небольшую паузу и поговорим о самом процессе разработки, а также способах упрощения разработки приложений Kafka Streams.

3.4. Интерактивная разработка

Мы создали граф для потоковой обработки записей ZMart с тремя узлами-обработчиками, записывающими данные в отдельные топики. Конечно, вы сможете во время разработки использовать для просмотра результатов консольный потребитель, но не помешало бы более удобное решение, например возможность просмотра в консоли движения данных по топологии, как показано на рис. 3.10.

Для этой цели может пригодиться один из методов интерфейса `KStream`, а именно метод `KStream.print`, принимающий в качестве параметра экземпляр класса `Printed<K, V>`.

Рис. 3.10. При разработке очень удобна возможность вывода в консоль выходных данных каждого узла. Для включения вывода в консоль достаточно заменить вызов метода to на вызов метода print

Класс Printed содержит два статических метода: для вывода в stdout (`Printed.toSysOut()`) и для записи результатов в файл (`Printed.toFile(filePath)`).

Кроме того, можно маркировать выводимые результаты, присоединив вызов метода `withLabel()`, благодаря которому вы можете выводить перед записями заголовок. Чтобы выводимые в консоль или файл результаты были удобны для использования, важно обеспечить разумную реализацию метода `toString`.

Наконец, если вы не хотите использовать метод `toString` или хотели бы настроить способ вывода Kafka Streams записей в соответствии со своими задачами, существует метод `Printed.withKeyValueMapper`, принимающий на входе экземпляр `KeyValueMapper`, позволяющий форматировать записи произвольным образом. Повторю вышеупомянутое предостережение, которое относится и к этому методу: не следует модифицировать исходные записи.

В этой книге мы будем выводить во всех примерах записи в консоль. Вот несколько примеров использования метода `KStream.print` в листинге 3.9:

Настройки для вывода в консоль преобразованного объекта RewardAccumulator

Настройки для вывода в консоль преобразованного объекта PurchasePattern

```
patternKStream.print(Printed.<String, PurchasePattern>toSysOut()
➥ .withLabel("patterns"));

rewardsKStream.print(Printed.<String, RewardAccumulator>toSysOut()
➥ .withLabel("rewards"));

purchaseKStream.print(Printed.<String, Purchase>toSysOut()
➥ .withLabel("purchases"));
```

Выводит в консоль данные о покупках

Взглянем, какие результаты вы увидите на экране (рис. 3.11), и разберемся, чем они могут помочь при разработке. При включенном выводе можно работать с приложением Kafka Streams непосредственно из IDE, внося изменения, останавливая и снова запуская его, и визуально верифицировать, что выводимые результаты такие, как вы ожидали. Это не заменяет модульное и комплексное тестирование, но непосредственное наблюдение результатов потоковой обработки во время разработки очень помогает.

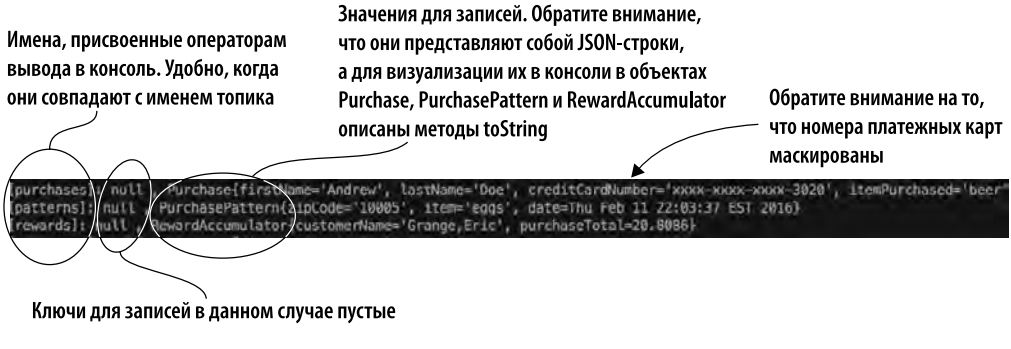


Рис. 3.11. Вид данных на экране. Включив вывод результатов в консоль, вы сразу увидите, правильно ли работают узлы-обработчики

Один из недостатков использования метода `print()` — он приводит к созданию концевого узла, который нельзя включить в цепочку узлов-обработчиков. Поэтому он должен быть отдельным оператором. Однако существует также метод `KStream.peek`, принимающий в качестве параметра экземпляр интерфейса `ForeachAction` и возвращающий новый экземпляр `KStream`. В интерфейсе `ForeachAction` содержится один метод, `apply()`, с типом возвращаемого значения `void`, так что от вызова метода `KStream.peek` ничего не передается далее по конвейеру, благодаря чему он идеально подходит для таких операций, как вывод результатов в консоль. Для его включения в цепочку узлов-обработчиков не требуется отдельный оператор вывода. Как вы увидите далее, именно так метод `KStream.peek` и используется в других примерах в этой книге.

3.5. Дальнейшие шаги

Пока что наша программа Kafka Streams для анализа покупок работает неплохо. Были разработаны и другие приложения для потребления записанных в топики `patterns-`, `rewards-` и `purchases`-сообщений, и результаты для компании ZMart оказались вполне удовлетворительными. Но, увы, ни одно хорошее дело не остается безнаказанным. Как только руководители ZMart увидели возможности вашего потокового приложения, так сразу же завалили вас новыми требованиями.

3.5.1. Новые требования

У вас появились новые требования для каждой из трех категорий генерируемых результатов. Хорошая новость: исходные данные не поменялись. От вас требуется лишь немного усовершенствовать и в некоторых случаях дополнительно разбить выдаваемые вами результаты. Новые требования могут как относиться к уже существующим топикам, так и означать необходимость создания совершенно новых:

- нужно отфильтровать покупки стоимостью меньше определенного количества долларов. Высшее руководство компании не интересуют мелкие покупки для повседневных нужд;
- компания ZMart расширилась и купила сеть магазинов электроники, а также сеть кофеен. Все данные о покупках в этих новых магазинах также будут проходить через ваше потоковое приложение. От вас требуется отправлять информацию о совершаемых в данных новых подразделениях покупках в соответствующие топики;
- выбранное вами NoSQL-решение сохраняет элементы в формате «ключ/значение». Хотя в Kafka также используются пары «ключ/значение», ключи в поступающих в ваш кластер Kafka записях не заданы. Вам нужно будет сгенерировать ключ для каждой из записей до передачи его по топологии в топик `purchases`.

В дальнейшем вас неизбежно ждут еще новые требования, но пока можно начинать работать с текущим набором новых требований. Если вы заглянете в API

KStream, то увидите там несколько уже существующих методов, сильно облегчающих выполнение этих требований.

ПРИМЕЧАНИЕ

Начиная с данного момента, мы будем ради большей ясности сокращать все примеры кода, оставляя лишь самое важное. Можно без опасений всегда предполагать, что код конфигурации и задания настроек не поменялся, разве что происходит знакомство с чем-то новым. Эти сокращенные примеры сами по себе могут не работать — полные листинги кода для данного примера можно найти в файле src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsAdvancedReqsApp.java.

Фильтрация покупок

Начнем с того, что отфильтруем покупки, стоимость которых меньше заданного порогового значения. Чтобы удалить дешевые покупки, необходимо вставить фильтрующий узел-обработчик между экземпляром KStream и узлом-стоком. При этом граф топологии обработки приобретет вид, показанный на рис. 3.12.

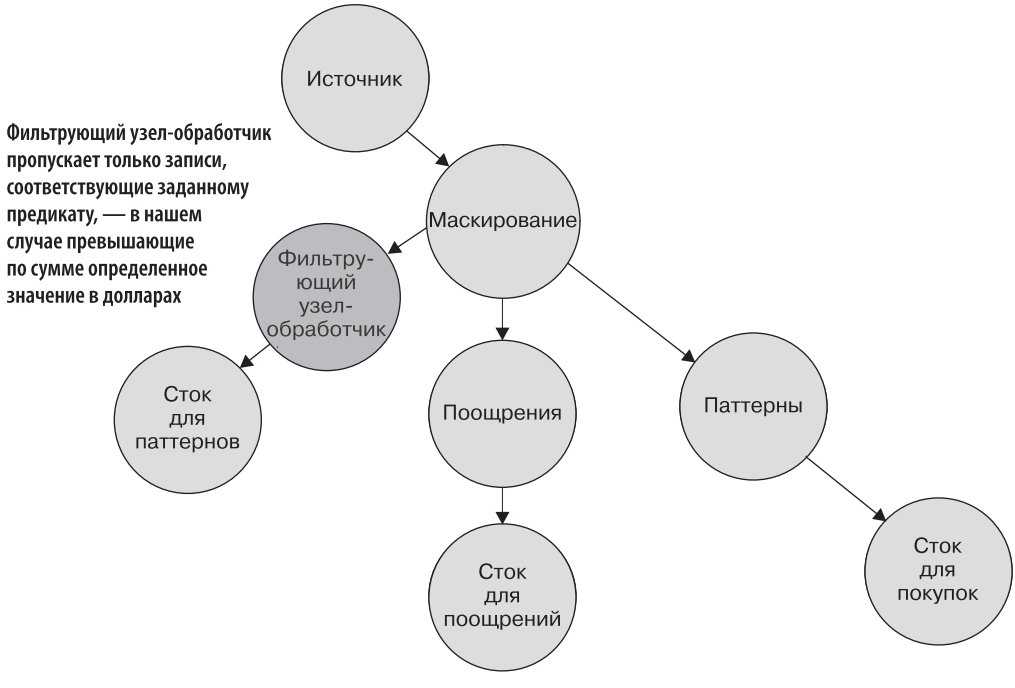


Рис. 3.12. Мы вставили узел-обработчик между маскирующим узлом и записывающим данные в Kafka концевым узлом. Этот фильтрующий узел-обработчик отбрасывает покупки, чья стоимость меньше заданного значения

Можно воспользоваться методом KStream, принимающим в качестве параметра экземпляр Predicate<K,V>. Хотя мы и связываем тут вызовы методов цепочкой, но создаем в топологии новый обрабатывающий узел (листинг 3.12).

Листинг 3.12. Фильтрация KStream

```
KStream<Long, Purchase> filteredKStream =
➥ purchaseKStream.filter((key, purchase) ->
➥ purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);
```

Этот фрагмент кода отфильтровывает покупки стоимостью менее $5 с выбором даты покупки в качестве значения типа Long для ключа.

В интерфейсе Predicate описан только один метод, test(), принимающий два параметра — ключ и значение, — хотя пока нам требуется только значение. Опять же, вы можете воспользоваться лямбда-выражением Java 8 вместо описанного в API KStream конкретного типа.

ОПРЕДЕЛЕНИЕ

Если вы знакомы с функциональным программированием, то должны чувствовать себя как дома при работе с интерфейсом Predicate. Если же термин *«предикат»* (predicate) вам в новинку — ничего страшного, это просто заданное утверждение, например $x < 100$. Объект может или соответствовать утверждению предиката, или не соответствовать.

Кроме того, мы хотели бы задействовать в качестве ключа метку даты/времени покупки, поэтому воспользуемся узлом-обработчиком выбора ключей selectKey, который с помощью упомянутого в разделе 3.4 KeyValueMapper извлекает дату покупки в виде значения типа Long. Мы обсудим выбор ключа подробнее в пункте «Генерация ключа» далее.

Обратная функция, KStreamNot, делает то же самое с точностью до наоборот: только *не* соответствующие заданному предикату записи попадают в дальнейшую обработку в топологии.

Разбиение/ветвление потока данных

Теперь нам нужно разбить поток покупок на отдельные потоки, записывающие в различные топики. К счастью, существует метод KStream.branch, который идеально подходит для этой задачи. Метод KStream.branch принимает на входе произвольное число экземпляров Predicate и возвращает массив экземпляров KStream. Размер возвращаемого массива соответствует числу переданных при вызове метода предикатов.

На предыдущем шаге мы изменили существующий лист топологии обработки. А поскольку нам нужно разбить поток данных, мы создадим совершенно новые листья на графе обрабатывающих узлов, как показано на рис. 3.13.

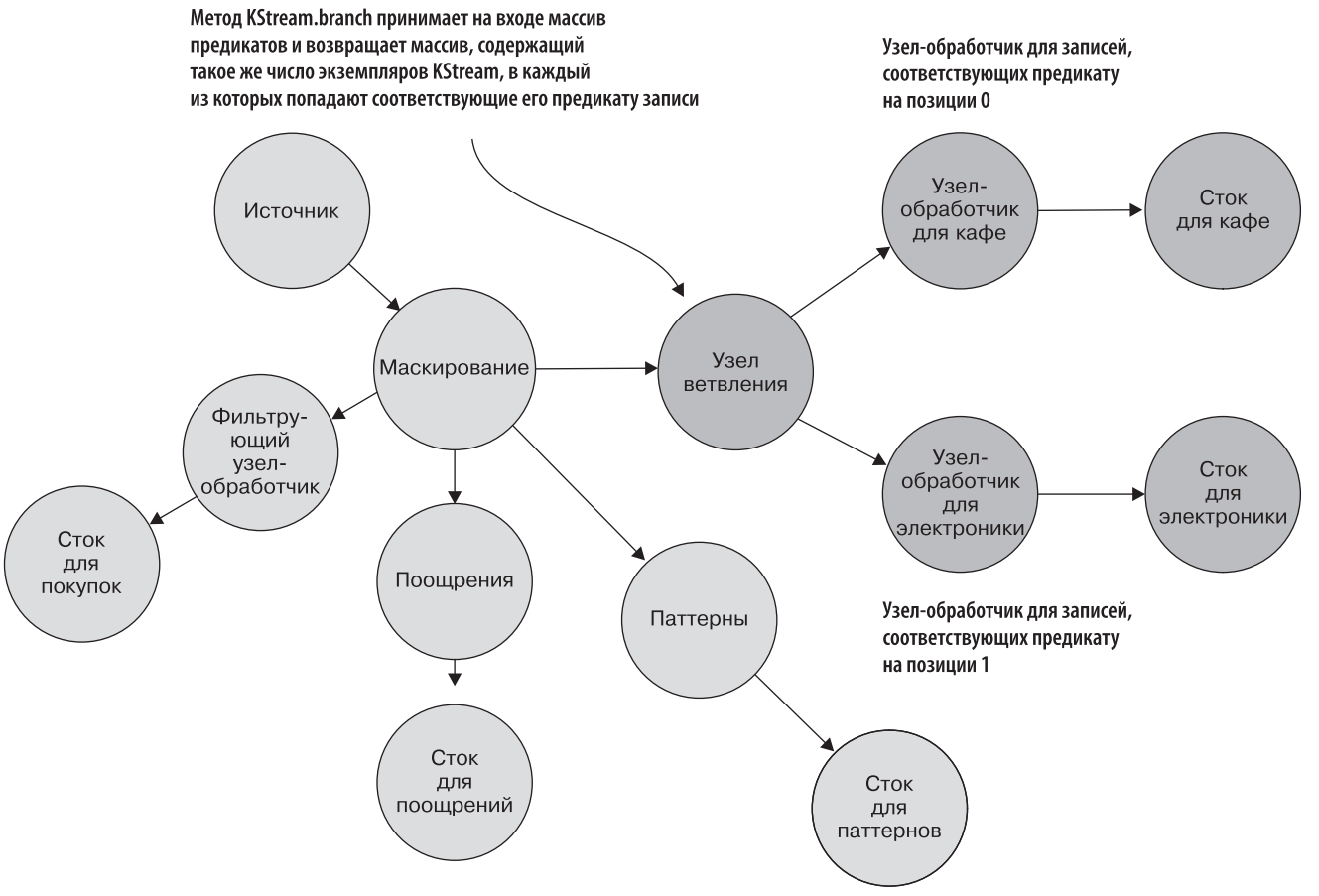


Рис. 3.13. Узел ветвления разбивает поток на два: один состоит из информации о покупках в кафе, а второй содержит сведения о покупках в магазине электроники

Записи из первоначального потока данных проходят через узел ветвления, при этом каждая запись сопоставляется с переданными предикатами в соответствующем порядке. Порядок потоков в возвращаемом массиве соответствует порядку переданных методу `branch()` предикатов. Мы будем пока что придерживаться варианта с отдельным топиком для каждого подразделения, хотя это далеко не единственный возможный подход. В конце концов, требованиям данный подход удовлетворяет, а в дальнейшем его можно при необходимости пересмотреть.

Листинг 3.13. Разбиение потока данных

```
Predicate<String, Purchase> isCoffee =
➥ (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("coffee");

Predicate<String, Purchase> isElectronics =
➥ (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("electronics");

int coffee = 0;
int electronics = 1;

KStream<String, Purchase>[] kstreamByDept =
➥ purchaseKStream.branch(isCoffee, isElectronics);

kstreamByDept[coffee].to( "coffee",
➥ Produced.with(stringSerde, purchaseSerde));
kstreamByDept[electronics].to("electronics",
➥ Produced.with(stringSerde, purchaseSerde));
```

Создает предикаты в виде лямбда-выражений Java 8

Маркировка предполагаемых позиций возвращаемого массива

Вызов метода branch для разбиения первоначального потока данных на два потока

Запись результатов каждого потока в свой топик

ПРЕДОСТЕРЕЖЕНИЕ

В примере из листинга 3.13 записи попадают в несколько различных топиков. Хотя существует возможность настроить Kafka для автоматического создания топиков при первой попытке сгенерировать данные для несуществующего топика или прочитать данные из него, не стоит полагаться на этот механизм. Если положиться на автоматическое создание топиков, их настройки будут основаны на значениях по умолчанию из файла server.config, которые могут отличаться от того, что вам нужно. Всегда следует заранее обдумывать, какие топики нужны, с каким числом секций и каким коэффициентом репликации, и создавать их до запуска приложения Kafka Streams.

В листинге 3.13 мы заранее описали предикаты, поскольку четыре параметра в виде лямбда-выражений были бы несколько громоздкими для передачи. Индексы возвращаемого массива также маркированы для большего удобства чтения.

Этот пример демонстрирует возможности и гибкость Kafka Streams. Нам потребовалось лишь несколько строк кода для разбиения исходного потока транзакций покупок на четыре потока. Также мы начали наращивать более сложную топологию обработки на основе все того же узла-источника.

Разбиение и секционирование потоков данных

Хотя принципы разбиения и секционирования кажутся схожими, в Kafka и Kafka Streams это совершенно разные вещи. Разбиение потока данных с помощью метода `KStream.branch` приводит к созданию одного или нескольких потоков, которые могут в итоге отправлять записи в отдельные топики. Секционирование же представляет собой способ, с помощью которого Kafka распределяет сообщения одного топика по нескольким серверам и, если не считать тонкой модификации настроек, является основным способом достижения высокой пропускной способности в Kafka.

Пока задача решается нормально. Мы легко выполнили два из трех новых требований. Пришло время реализовать последнее из дополнительных требований — сгенерировать ключи для сохраняемых записей о покупках.

Генерация ключа

Сообщения Kafka представляют собой пары «ключ/значение», так что все проходящие через приложения Kafka Streams записи — тоже пары «ключ/значение». Но ключи вовсе не обязаны быть непустыми. На практике, если конкретный ключ не нужен, за счет пустого ключа можно снизить общий объем передаваемых по сети данных. Ключи во всех поступающих в приложение Kafka Streams компании ZMart записях — пустые.

Это нам ничуть не мешало, пока мы не осознали, что наше NoSQL-хранилище хранит данные в формате «ключ/значение». Нам нужно найти способ создать ключ на основе данных объекта `Purchase`, прежде чем он окажется записан в топик `purchases` (листинг 3.14). Конечно, можно воспользоваться для генерации ключа и возврата новой пары «ключ/значение» (в которой «новым» будет только ключ) методом `KStream.map`, но существует метод `KStream.selectKey` с более лаконичным синтаксисом, возвращающий новый экземпляр `KStream`, который генерирует записи с новым ключом (возможно, с другим типом данных) и тем же значением. Это изменение топологии обработки аналогично фильтрации в том смысле, что между фильтром и стоком добавляется новый обрабатывающий узел, как показано на рис. 3.14.

Листинг 3.14. Генерация нового ключа

```
KeyValueMapper<String, Purchase, Long> purchaseDateAsKey =
➥ (key, purchase) -> purchase.getPurchaseDate().getTime();

KStream<Long, Purchase> filteredKStream =
➥ purchaseKStream.filter((key, purchase) ->
➥ purchase.getPrice() > 5.00).selectKey(purchaseDateAsKey);

filteredKStream.print(Printed.<Long, Purchase>
➥ toSysOut().withLabel("purchases"));
filteredKStream.to("purchases",
➥ Produced.with(Serdes.Long(),purchaseSerde));
```

KeyValueMapper извлекает дату покупки и преобразует в длинное целое (Long)

Отфильтровываем покупки и выбираем ключи в одном операторе

Выводим результаты в консоль

Материализация результатов в топик Kafka

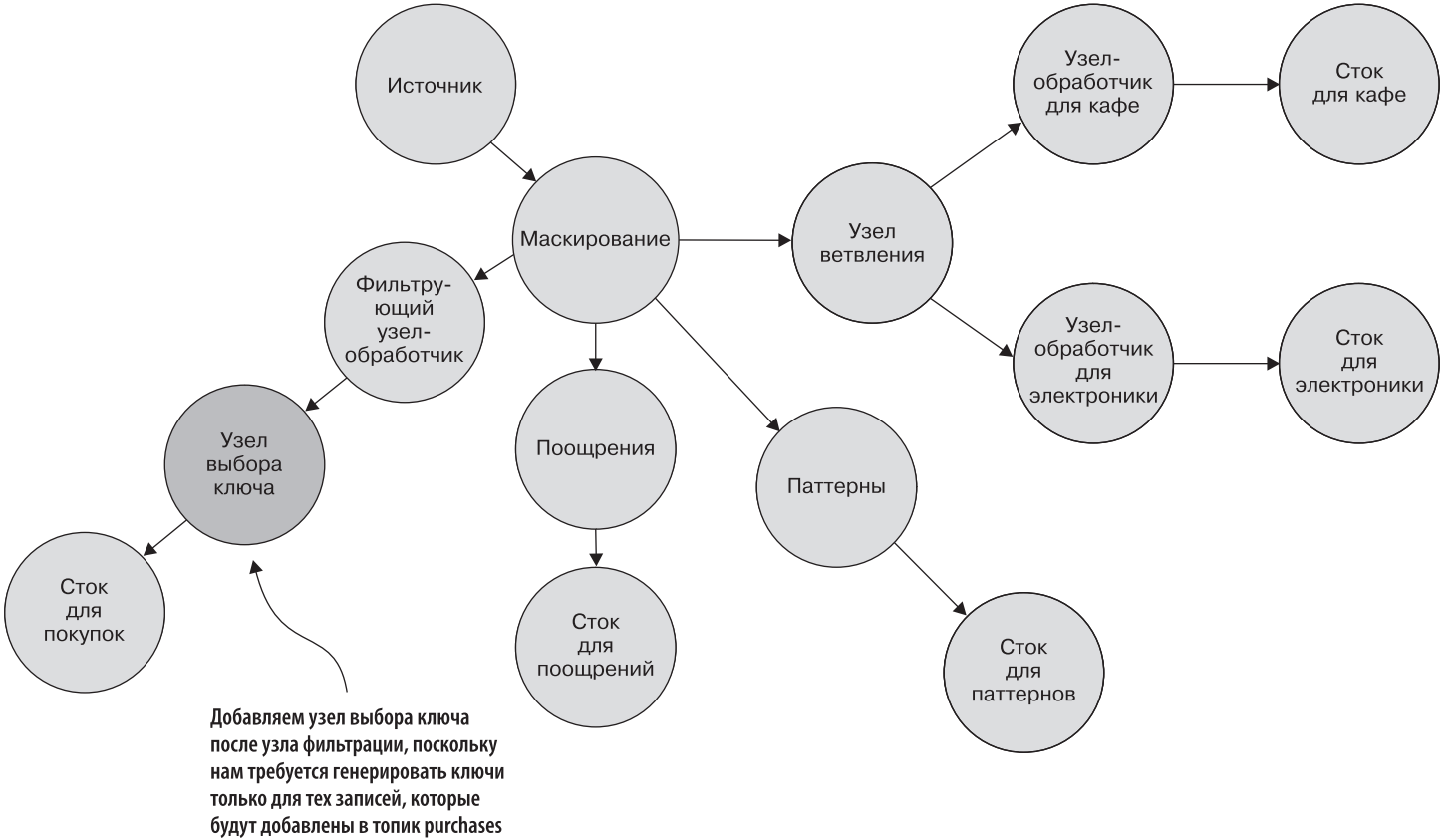


Рис. 3.14. В NoSQL-хранилище в качестве ключа хранимых данных будет использоваться дата покупки. Новый узел выбора ключа будет извлекать дату покупки (для применения в качестве ключа) непосредственно перед записью данных в Kafka

Для создания нового ключа мы преобразуем дату покупки в длинное целое. Хотя мы могли бы передать лямбда-выражение, для повышения удобочитаемости мы присвоили это значение переменной. Обратите внимание также, что необходимо изменить тип используемого в методе `KStream.to` объекта `Serde`, поскольку мы изменили тип ключа.

Это был простой пример генерации новых ключей. Далее, в другом примере, я покажу выбор ключей для соединения отдельных потоков данных. Кроме того, во всех предыдущих примерах отсутствовало сохранение состояния, но есть и несколько возможностей для преобразований с сохранением состояния, которые мы рассмотрим чуть позже.

3.5.2. Сохранение записей вне Kafka

К вам обратилась служба безопасности компании ZMart по поводу возможного мошенничества в одном из магазинов. К ним поступило несколько сообщений о том, что один из менеджеров этого магазина вводит неправильные коды скидок. Служба безопасности не знает точно, что произошло на самом деле, поэтому просит вас помочь ей.

Люди из службы безопасности не хотели бы, чтобы указанная информация попадала в топик. Вы долго рассказывали им о безопасности Kafka, о разграничении доступа к данным, о том, что можно ограничить доступ к топику, но они стоят на своем и хотят, чтобы эти записи попадали в полностью контролируемую ими реляционную базу данных. Вы чувствуете, что в этой битве вам не победить, так что сдаетесь и решаете сделать так, как они просят.

Обработка по каждой записи. Прежде всего вам нужно создать новый `KStream` для фильтрации результатов по одному идентификатору работника. Этот фильтр позволит сократить до очень небольшого количества даже самые огромные объемы проходящих через топологию данных.

Мы будем использовать тут `KStream` с предикатом, нацеленным на выбор одного идентификатора работника. Данный фильтр будет совершенно независим от предыдущего и подключен к экземпляру `KStream`-источника. Хотя фильтры можно связывать цепочкой, здесь мы этого делать не станем, поскольку для указанного фильтра нам требуется полный доступ к данным потока.

Далее мы воспользуемся методом `KStream.foreach`, как показано на рис. 3.15. Метод `KStream.foreach` принимает в качестве параметра экземпляр `ForeachAction<K, V>`. Это еще один пример концевого узла, представляющий собой простой узел-обработчик, который выполняет с помощью переданного ему экземпляра `ForeachAction` какие-либо действия над каждой получаемой записью (листинг 3.15).

Листинг 3.15. Операции над каждой записью с помощью foreach

```
ForeachAction<String, Purchase> purchaseForeachAction = (key, purchase) ->
➥ SecurityDBService.saveRecord(purchase.getPurchaseDate(),
➥ purchase.getEmployeeId(), purchase.getItemPurchased());
purchaseKStream.filter((key, purchase) ->
➥ purchase.getEmployeeId()
➥ .equals("source code has 000000"))
➥ .foreach(purchaseForeachAction);
```

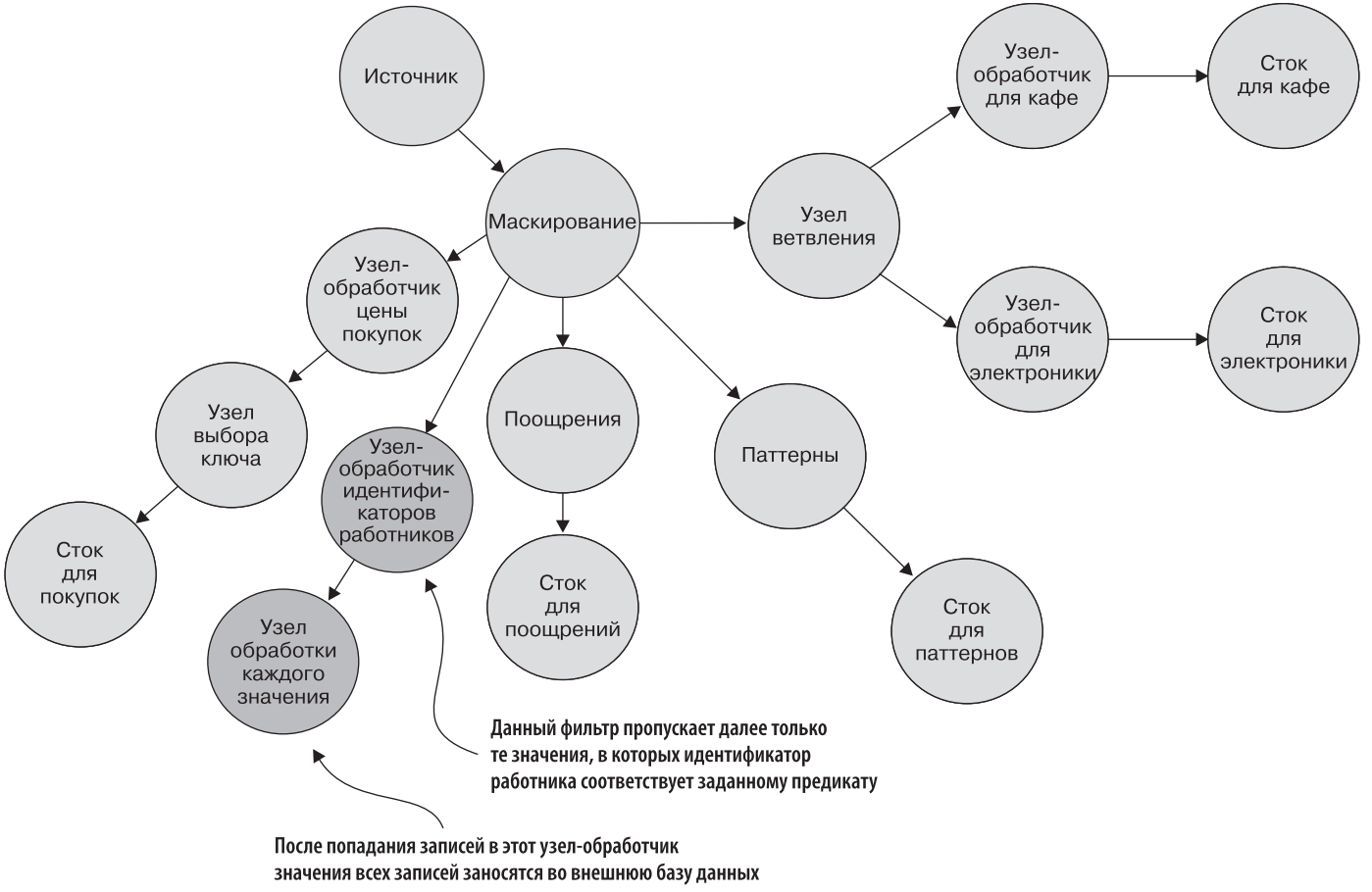


Рис. 3.15. Для записи покупок, в оформлении которых участвовал определенный работник компании вне приложения Kafka Streams, мы сначала добавим узел фильтрации (filter) для извлечения покупок с соответствующим идентификатором работника, после чего воспользуемся оператором foreach для занесения всех записей во внешнюю реляционную базу данных

`ForeachAction` опять же использует лямбда-выражение Java 8 и сохраняется в переменной `purchaseForeachAction`. Это требует лишней строки кода, что более чем компенсируется повышением понятности. В следующей строке другой экземпляр `KStream` выполняет отправку отфильтрованных результатов в описанный строкой выше `ForeachAction`.

Отмечу, что `KStream.foreach` не сохраняет состояние. Если для каких-либо действий над каждой из записей требуется состояние, можно воспользоваться методом `KStream.process`. Мы обсудим метод `KStream.process` в следующей главе, когда займемся добавлением сохранения состояния в приложение Kafka Streams.

Отступим на шаг назад и посмотрим на сделанное. Весьма впечатляюще, учитывая объем написанного кода. Впрочем, не слишком расслабляйтесь, руководство ZMart отметило вашу продуктивность и готовит для вас множество новых изменений и усовершенствований программы потокового анализа покупок.

Резюме

- Функцию `KStream.mapValues` можно использовать для отображения значений входящих записей в новые значения, возможно, с другим типом. Как вы узнали, при этом отображении исходные объекты не должны меняться. Другой метод, `KStream.map`, делает то же самое, но его можно применять для отображения как ключа, так и значения в новую сущность.
- Предикат — оператор, принимающий в качестве параметра объект и возвращающий `true` или `false` в зависимости от того, удовлетворяет ли объект заданному условию. Мы использовали в наших примерах предикаты, чтобы не удовлетворяющие заданному предикату записи не проходили дальше по топологии.
- В методе `KStream.branch` предикаты применяются для разбиения потока данных на несколько новых потоков. Узел-обработчик отправляет запись в первый поток с соответствующим записи предикатом и отбрасывает не удовлетворяющие ни одному предикату записи.
- С помощью метода `KStream.selectKey` можно модифицировать существующие записи или создавать новые.

В следующей главе мы начнем работать с состоянием, свойствами, необходимыми для использования состояния в потоковом приложении, и причинами, по которым состояние вообще может вам понадобиться. Затем мы добавим в приложение `KStream` состояние сначала с помощью сохраняющих его версий уже встречавшихся вам в этой главе методов `KStream` (`KStream.mapValues()`). А затем, в более продвинутом примере, мы выполним соединение двух различных потоков покупок для улучшения обслуживания покупателей компанией ZMart.

4 Потоки данных и состояние

В этой главе:

- операции с сохранением состояния и Kafka Streams;
- использование хранилищ состояния для поиска по справочнику и запоминания уже просмотренных данных;
- соединение потоков данных для получения дополнительной информации;
- время и метки даты/времени как движущая сила Kafka Streams.

В прошлой главе мы нырнули без оглядки в DSL Kafka Streams и создали топологию обработки для удовлетворения требований к потоковой обработке покупок в магазинах компании ZMart. Построенная нами топология, хотя и нетривиальная, была одномерной в том смысле, что все преобразования и операции были без сохранения состояния. Мы рассматривали каждую транзакцию изолированно от других, безотносительно к другим событиям, происходящим в тот же момент или в пределах определенного промежутка времени до или после транзакции. Кроме того, мы работали только с отдельными потоками данных, игнорируя возможность получения дополнительной информации за счет их соединения.

В настоящей главе вам предстоит извлечь из приложения Kafka Streams как можно больше информации. Для этого нам придется использовать состояние. *Состояние* (state) — не что иное, как возможность восстанавливать просмотренную ранее информацию и связывать ее с текущей. Состояние можно применять по-разному. Мы рассмотрим пример этого, когда будем изучать операции с сохранением состояния, такие как накопление значений с помощью DSL Kafka Streams.

Еще один пример сохранения состояния, который мы обсудим далее, — соединение потоков данных. Оно очень похоже на соединения при операциях баз данных, например соединение записей из таблиц `employee` («сотрудник») и `department` («отдел») для генерации отчета по штату различных отделов компании.

Мы также покажем, как должно выглядеть состояние и каковы требования к использованию состояния, когда будем обсуждать хранилища состояния в Kafka Streams. Наконец, мы поговорим о важности меток даты/времени и о том, чем они могут оказаться полезны при операциях с сохранением состояния, например,

для обеспечения работы лишь с событиями, происходящими в пределах определенного промежутка времени, или для упрощения работы с поступающими не в том порядке данными.

4.1. Обработка событий

Иногда при обработке событий не требуется никакой дополнительной информации или контекста. А иногда событие само по себе несет определенный смысл, но без дополнительного контекста важность происходящего остается непонятной: благодаря дополнительной информации этой событие предстает в совершенно ином свете.

Пример события, не требующего дополнительной информации, — попытка использования украденной платежной карты. Транзакцию необходимо отменить сразу же, как только выяснится, что эта платежная карта была украдена. Никакой дополнительной информации для принятия такого решения не требуется.

Но иногда одиночного события недостаточно для принятия решения. Рассмотрим, например, последовательность биржевых сделок трех отдельных инвесторов за короткий промежуток времени. На первый взгляд ничего в покупках акций компании «XYZ Фармасьютикл» (рис. 4.1) не наводит на размышления. Покупки одних и тех же акций инвесторами происходят на Уолл-стрит каждый день.

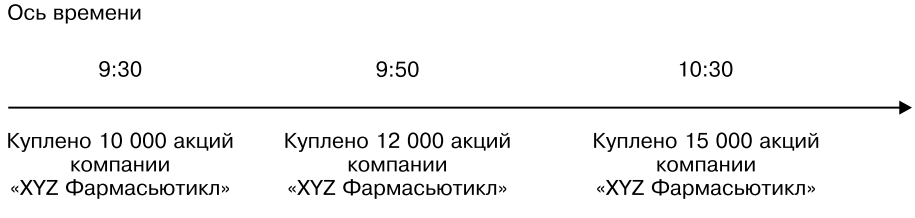


Рис. 4.1. Биржевые сделки без дополнительной информации выглядят совершенно обычными

А теперь рассмотрим эти события в контексте. В течение биржевого дня компания «XYZ Фармасьютикл» объявила об одобрении государственными органами ее нового лекарства, из-за чего цены на ее акции подскочили до исторического максимума. Кроме того, указанные три инвестора тесно связаны с «XYZ Фармасьютикл». С учетом этого приведенные на рис. 4.2 транзакции представляются в совершенно новом свете.

Хронометраж совершения этих покупок акций и публикации информации вызывает вопросы. Имела ли место заблаговременная утечка информации? Или транзакции относятся к одному инвестору, получившему внутреннюю информацию, который пытается замести следы?

Сохранение состояния необходимо для потоков данных. Предыдущий вымышленный сценарий иллюстрирует некие интуитивно понятные большинству из нас знания. Иногда понять, что происходит, несложно, но чаще всего для принятия удачных решений необходимо представлять контекст. В области потоковой обработки такой дополнительный контекст называется состоянием.

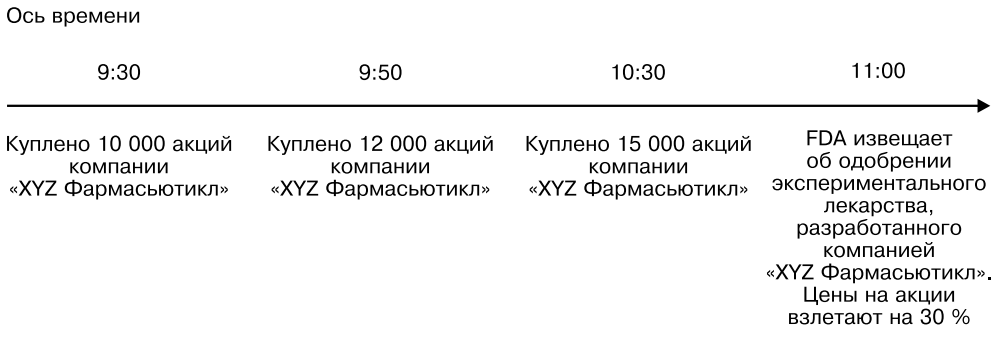


Рис. 4.2. В контексте дополнительной информации о времени совершения биржевых сделок они предстают в абсолютно новом свете

На первый взгляд, сохранение состояния и потоковая обработка между собой не согласуются. Потоковая обработка означает постоянный поток отдельных событий, особо не связанных друг с другом, которые нужно обрабатывать по мере их возникновения. Понятие состояния же скорее вызывает представление о статическом ресурсе вроде таблицы базы данных.

На деле же их можно рассматривать как одно и то же. Но изменения в потоке данных происходят намного быстрее и чаще, чем в таблице базы данных[1].

Для работы с потоковыми данными не всегда требуется состояние. В некоторых случаях отдельные события или записи несут достаточно информации, чтобы представлять значимость сами по себе. Но чаще все же входящий поток данных требует обогащения информацией из какого-либо хранилища, или на основе ранее поступивших событий, или путем соединения взаимосвязанных событий из различных потоков.

4.2. Операции с сохранением состояния в Kafka Streams

В этом разделе вы увидите, как добавить сохраняющую состояние операцию к уже существующей операции без сохранения состояния с целью обогащения собираемой приложением информации. Вам придется модифицировать первоначальную топологию из главы 3, которая для освежения вашей памяти приведена на рис. 4.3.

В этой топологии мы генерировали поток событий, относящихся к транзакциям покупок. Один из узлов-обработчиков топологии подсчитывал бонусы покупателей в зависимости от суммы покупки. Но в нем лишь подсчитывались бонусы для отдельной транзакции, а полученные результаты отправлялись дальше по топологии.

---

[1] *Kreps J.* Why Local State Is a Fundamental Primitive in Stream Processing (Джей Крепс «Почему локальное состояние — важнейший базовый элемент потоковой обработки»), http://mng.bz/sfoI.

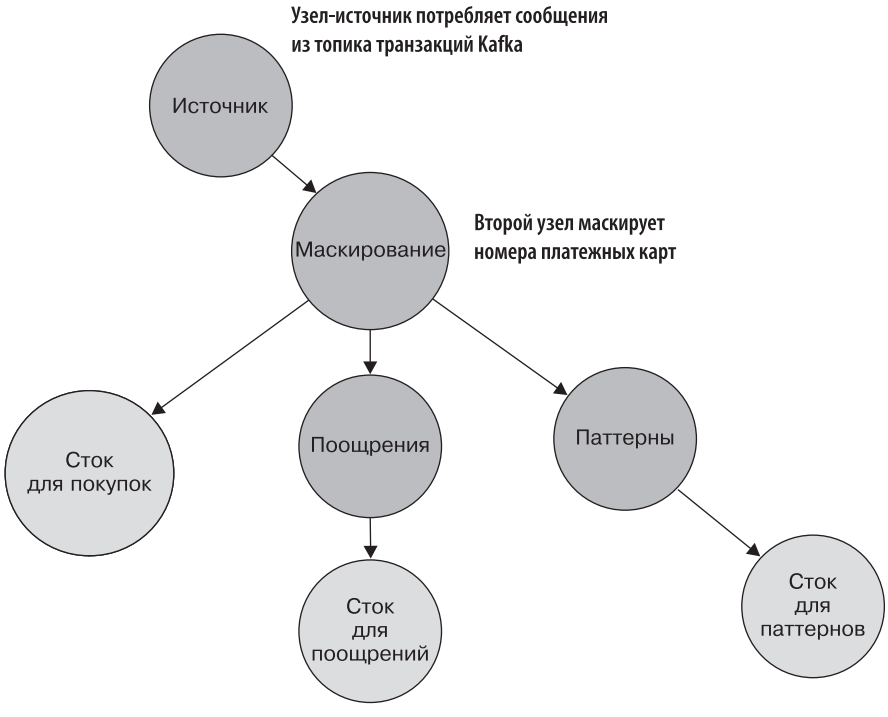


Рис. 4.3. Еще раз взглянем на топологию из главы 3

Если добавить в этот узел-обработчик сохранение состояния, можно будет отслеживать общую сумму накопленных бонусов. А приложение-потребитель ZMart сможет проверять указанную сумму и отправлять покупателю поощрение при необходимости.

Теперь, когда вы уже представляете себе, как сохранение состояния может пригодиться в приложениях Kafka Streams (или любых других потоковых приложениях), рассмотрим несколько конкретных примеров. Мы начнем с преобразования узла-обработчика для поощрений без сохранения состояния в узел-обработчик с сохранением состояния с помощью метода `transformValues` интерфейса `KStream`. Вы сможете предоставлять расширенную информацию потребителям, находящимся дальше по конвейеру, благодаря отслеживанию общего числа заработанных бонусов и промежутков времени между покупками.

4.2.1. Узел-обработчик transformValues

`KStream.transformValues` — простейшая из функций с сохранением состояния. Рисунок 4.4 иллюстрирует ее работу.

Этот метод семантически не отличается от `KStream.mapValues()` с несколькими исключениями. Одно из отличий заключается в наличии у `transformValues` доступа

к экземпляру интерфейса `StateStore`. Еще одно отличие — возможность планирования операций на выполнение через равные промежутки времени с помощью метода `punctuate()`. Мы обсудим метод `punctuate()` подробнее, когда будем рассматривать API узлов-обработчиков в главе 6.

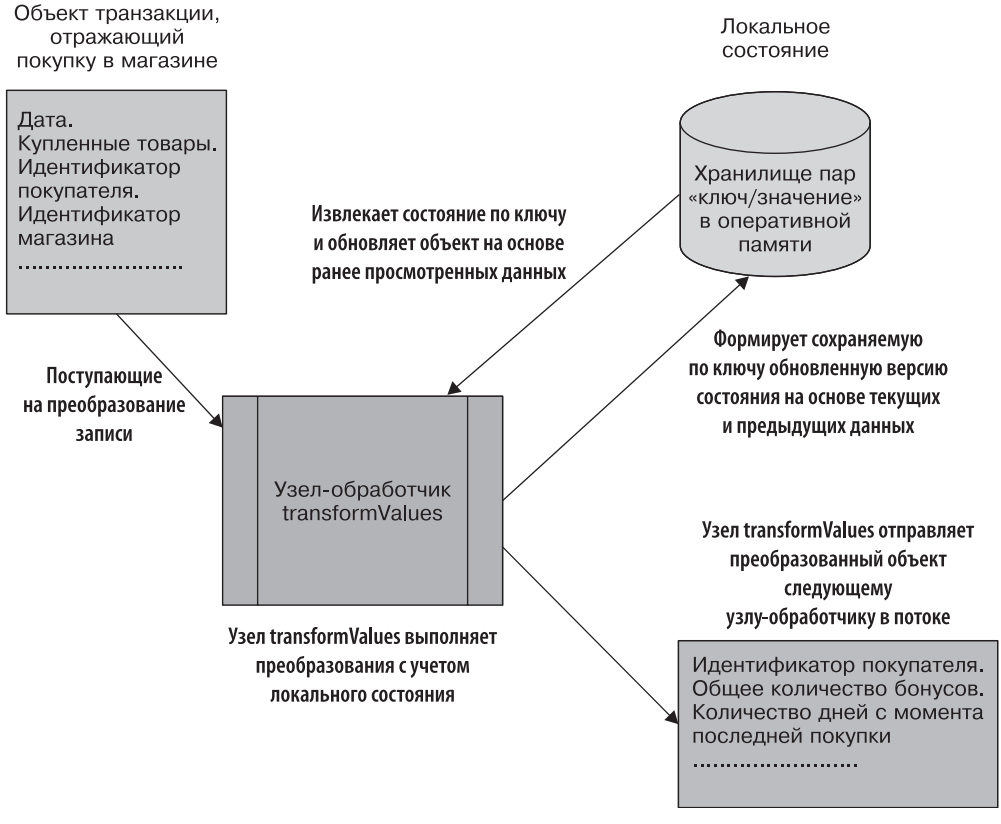


Рис. 4.4. Узел-обработчик transformValues обновляет входящие записи на основе информации, хранимой в локальном состоянии. В этом случае ключом для извлечения и сохранения состояния для заданной записи служит идентификатор покупателя

4.2.2. Поощрения покупателей с сохранением состояния

Узел-обработчик поощрений из топологии главы 3 (см. рис. 4.3) для компании ZMart извлекает информацию о членах клуба постоянных покупателей ZMart. Изначально в нем использовался метод `KStream.mapValues()` для отображения объекта `Purchase` в объект `RewardAccumulator`.

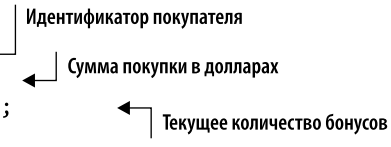
Объект `RewardAccumulator` сначала состоял всего из двух полей — идентификатора покупателя и суммы покупки по транзакции. Теперь же требования

несколько поменялись и бонусы связываются с программой поощрения покупателей ZMart:

```
public class RewardAccumulator {

    private String customerId;
    private double purchaseTotal;
    private int currentRewardPoints;

  // Для большей ясности опускаем подробности
}
```



Если раньше другое приложение читало данные из топика поощрений и подсчитывало «достижения» покупателя, то теперь руководство хочет, чтобы бонусы подсчитывало наше потоковое приложение. Кроме того, необходимо собирать данные о длительности промежутка времени между покупками этого покупателя.

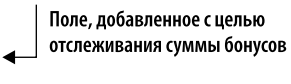
Приложению-потребителю при чтении записей из топика поощрений необходимо только проверить, превышает ли общее число бонусов установленное для выдачи вознаграждения пороговое значение. Можно добавить в объект `RewardAccumulator` поля `totalRewardPoints` и `daysFromLastPurchase` и отслеживать накопленные бонусы и последнюю дату покупки на основе локального состояния. Для этого нам понадобится переделать код класса `RewardAccumulator` следующим образом (его можно найти в файле src/main/java/bbejeck/model/RewardAccumulator.java; весь исходный код находится по адресу https://manning.com/books/kafka-streams-in-action) (листинг 4.1).

Листинг 4.1. Переделанный объект RewardAccumulator

```
public class RewardAccumulator {

    private String customerId;
    private double purchaseTotal;
    private int currentRewardPoints;
    private int daysFromLastPurchase;
    private int totalRewardPoints;

  // Для большей ясности опускаем подробности
}
```



Обновленные правила программы постоянных покупателей просты. Покупатель зарабатывает один бонус за каждый потраченный доллар, причем сумма транзакции округляется до ближайшего целого числа долларов. Общая структура топологии не меняется, но узел обработки поощрений будет использовать метод `KStream.transformValues()` вместо `KStream.mapValues()`. Семантически эти методы работают схожим образом, в том смысле что в обоих объект `Purchase` отображается в объект `RewardAccumulator`. Различие состоит в возможности использования локального состояния для преобразования.

Точнее говоря, эта обработка будет состоять из двух основных шагов.

1. Инициализация преобразователя значений.
2. Отображение на основе состояния объекта `Purchase` в объект `RewardAccumulator`.

Метод `KStream.transformValues()` принимает на входе объект `ValueTransformerSupplier<V, R>`, служащий для создания экземпляра интерфейса `ValueTransformer<V, R>`. Наша реализация `ValueTransformer` будет иметь вид `PurchaseRewardTransformer<Purchase, RewardAccumulator>`. Ради ясности я не буду воспроизводить весь код класса в тексте книги. Вместо этого мы рассмотрим наиболее важные методы для нашего примера приложения. Отмечу также, что эти фрагменты кода несамодостаточны и некоторые детали будут опущены для ясности. Полный код можно найти в исходном коде этой главы (по адресу https://manning.com/books/kafka-streams-in-action). Отправимся дальше и займемся инициализацией данного узла-обработчика.

4.2.3. Инициализация преобразователя значений

Первый этап — задание значений или создание переменных экземпляра в методе `init()` преобразователя (листинг 4.2). В методе `init()` мы извлечем хранилище состояний, созданное при построении топологии обработки (вопрос добавления хранилища состояний мы рассмотрим в подразделе 4.3.3).

Листинг 4.2. Метод init()

```
private KeyValueStore<String, Integer> stateStore;

private final String storeName;
private ProcessorContext context;

public void init(ProcessorContext context) {
  this.context = context;
  stateStore = (KeyValueStore)
  ➥ this.context.getStateStore(storeName);
}
```

Переменные экземпляра

Локальная ссылка на ProcessorContext

Извлекаем экземпляр StateStore, соответствующий переменной storeName, которая задается в конструкторе

Внутри класса преобразователя происходит преобразование к типу `KeyValueStore`. Пока нас не интересуют детали реализации внутри преобразователя, лишь возможность извлекать значения по ключу (в следующем разделе я расскажу подробнее о разновидностях реализации хранилищ состояния).

4.2.4. Отображение объекта Purchase в объект RewardAccumulator на основе состояния

После инициализации узла-обработчика можно перейти к преобразованию объекта `Purchase` с использованием данных о состоянии (листинг 4.3). Это преобразование состоит из нескольких простых этапов.

1. Проверка количества уже накопленных покупателем бонусов.
2. Подсчет числа бонусов по текущей транзакции и общего их числа.

3. Установка значения соответствующего поля объекта `RewardAccumulator` равным новому общему количеству бонусов.
4. Сохранение суммарного количества бонусов для данного идентификатора покупателя в локальном хранилище состояния.

Листинг 4.3. Преобразование объекта Purchase с использованием состояния

```
public RewardAccumulator transform(Purchase value) {
  RewardAccumulator rewardAccumulator =
  ➥ RewardAccumulator.builder(value).build();
  Integer accumulatedSoFar =
  ➥ stateStore.get(rewardAccumulator.getCustomerId());
  if (accumulatedSoFar != null) {
      rewardAccumulator.addRewardPoints(accumulatedSoFar);
  }

  stateStore.put(rewardAccumulator.getCustomerId(),
                 rewardAccumulator.getTotalRewardPoints());

  return rewardAccumulator;
}
```

Создаем объект RewardAccumulator из объекта Purchase

Извлекаем по идентификатору покупателя последнее значение количества бонусов

Если у покупателя уже накоплены какие-то бонусы, прибавляем их количество к текущей сумме

Сохраняем новое общее количество бонусов в объекте StateStore

Возвращаем новое количество накопленных бонусов

В методе `transform()` мы сначала отображаем объект `Purchase` в `RewardAccumulator` — это та же операция, что и в методе `mapValues()`. В нескольких следующих строках в процессе преобразования задействуется состояние. Производится поиск по ключу (идентификатору покупателя), и к бонусам за текущую покупку прибавляются ранее накопленные. После чего новое количество бонусов помещается в хранилище состояния, откуда может быть затребовано при необходимости.

Осталось только обновить узел-обработчик поощрений. Но сначала нужно учесть, что мы обращаемся ко всей информации о продажах по идентификатору покупателя. Группировка информации конкретного покупателя по продажам подразумевает, что все его транзакции находятся в одной секции. Но, поскольку у поступающих в приложение транзакций отсутствуют ключи, генератор распределяет их по секциям в циклическом порядке. Мы рассматривали циклическое распределение по секциям в главе 2, но имеет смысл освежить его в памяти (рис. 4.5).

Это значит, что, если не использовать топики только с одной секцией, мы столкнемся с проблемой. Поскольку ключи не заполнены, назначение секций в циклическом порядке означает, что транзакции для одного покупателя могут оказаться в разных секциях.

Важно размещать транзакции с одним идентификатором покупателя в одной и той же секции для целей поиска записей по идентификатору в хранилище состояния. В противном случае в разных секциях окажутся записи с одним идентификатором и вам придется искать одного покупателя в нескольких хранилищах состояния. (Из предыдущей фразы можно сделать вывод, что у каждой секции — свое хранилище состояния, но это не так. Секции распределяются по объектам `StreamTask`, у каждого из которых есть свое хранилище состояния.)

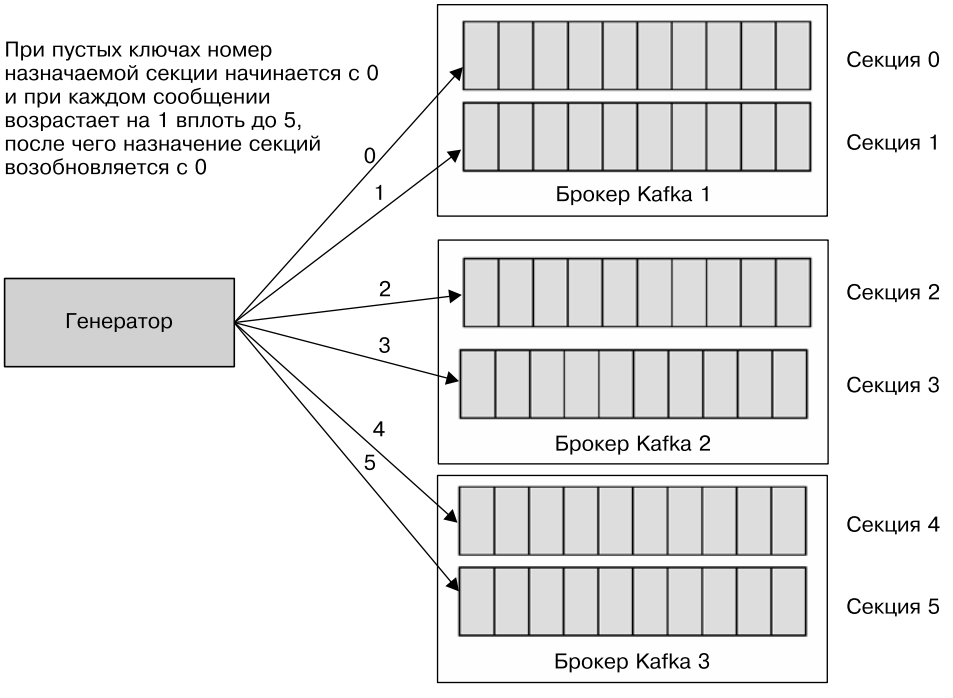


Рис. 4.5. При пустых ключах генератор Kafka равномерно распределяет записи (в циклическом порядке)

Для решения такой проблемы необходимо заново секционировать данные по идентификатору покупателя. Мы посмотрим, как это сделать, в следующем пункте.

Повторное секционирование данных

Сначала обсудим, как происходит в целом повторное секционирование данных (рис. 4.6). Для повторного секционирования записей можно сначала модифицировать или заменить ключ исходной записи, после чего отправить запись в новый топик. Далее можно снова потреблять эти записи, но в результате повторного секционирования они могут поступать из других секций, не тех, в которых находились изначально.

Хотя в этом простом примере мы заменяли пустой ключ конкретным значением, повторное секционирование не всегда требует замены ключа. С помощью интерфейса `StreamPartitioner` (http://mng.bz/9Z8A) можно применить практически любую стратегию секционирования, какая только придет вам в голову, например секционировать по значению или по части значения вместо ключа. В следующем разделе я продемонстрирую использование интерфейса `StreamPartitioner` в Kafka Streams.

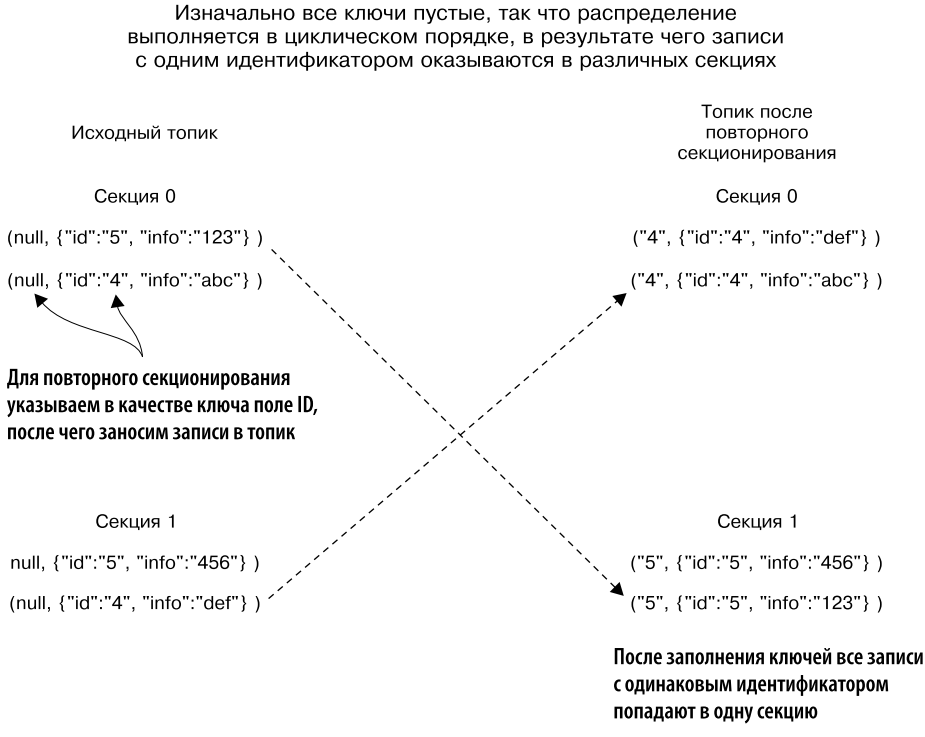


Рис. 4.6. Повторное секционирование: изменение исходного ключа с целью перемещения записей в другую секцию

Повторное секционирование в Kafka Streams

Повторное секционирование в Kafka Streams легко выполняется с помощью метода `KStream.through()`, как показано на рис. 4.7. Метод `KStream.through()` создает промежуточный топик, а *текущий* экземпляр `KStream` начинает заносить в него записи. Вызванный метод `through()` возвращает *новый* экземпляр `KStream` с тем же промежуточным топиком в качестве источника. Таким образом, повторное секционирование данных происходит без перебоев в работе.

Между тем внутри Kafka Streams происходит создание узла-стока и узла-источника. Узел-сток является дочерним узлом-обработчиком вызывающего интерфейса `KStream`, а новый узел-источник служит источником записей для нового экземпляра `KStream`. Можно создать аналогичную субтопологию с помощью DSL, но метод `KStream.through()` удобнее.

Если вы модифицировали или заменили ключи, а пользовательская стратегия секционирования вам не нужна, можете применить для секционирования метод секционирования по умолчанию встроенного генератора Kafka Streams — `KafkaProducer`. Если же вы захотите применить свой подход к секционированию, то можете воспользоваться `StreamPartitioner`. Именно этим мы и займемся в следующем примере.

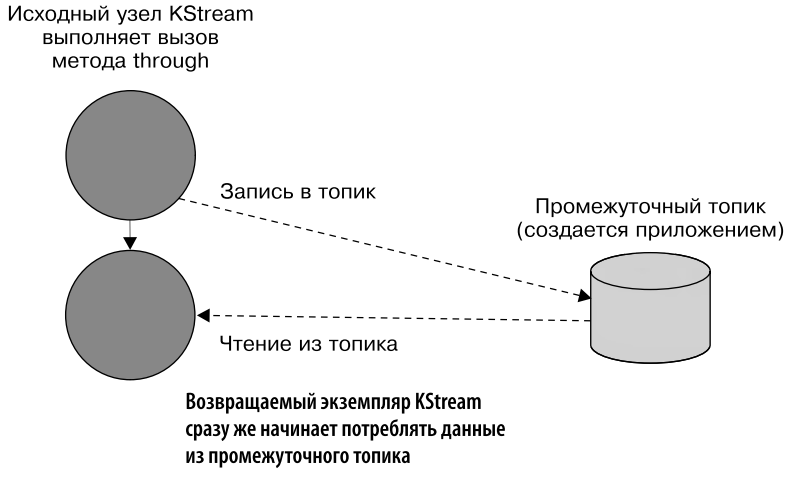


Рис. 4.7. Запись в промежуточный топик и чтение из него в новом экземпляре KStream

Код применения метода `KStream.through()` приведен в листинге 4.4. В этом примере метод `KStream.through()` принимает два параметра: название топика и экземпляр класса `Produced` с объектами `Serde` для ключа, `Serde` для значения и `StreamPartitioner`. Обратите внимание, что существует версия метода `KStream.through()`, в которую можно передать только название топика на случай, если вас устраивают экземпляры `Serde` по умолчанию для ключа и значения и вам не требуется пользовательская стратегия секционирования.

Листинг 4.4. Использование метода KStream.through()

```
RewardsStreamPartitioner streamPartitioner =
➥ new RewardsStreamPartitioner();
```

Создание конкретного экземпляра StreamPartitioner

```
KStream<String, Purchase> transByCustomerStream =
➥ purchaseKStream.through("customer_transactions",
                           Produced.with(stringSerde,
                                         purchaseSerde,
                                         streamPartitioner));
```

Создает новый KStream с помощью метода KStream.through()

Мы создали в этом фрагменте кода экземпляр `RewardsStreamPartitioner`. Посмотрим теперь на то, как он работает, а также продемонстрируем создание объекта `StreamPartitioner`.

Использование StreamPartitioner

Обычно секции назначаются на основе вычисления хеша объекта, по модулю числа секций. В этом случае мы собираемся использовать идентификатор покупателя из объекта `Purchase`, чтобы все относящиеся к одному покупателю данные оказывались в одном хранилище состояния. В листинге 4.5 показана реализация интерфейса

`StreamPartitioner` (ее можно найти в файле src/main/java/bbejeck/chapter_4/partitioner/RewardsStreamPartitioner.java).

Листинг 4.5. RewardsStreamPartitioner

```
public class RewardsStreamPartitioner implements
➥ StreamPartitioner<String, Purchase> {

    @Override
    public Integer partition(String key,
                             Purchase value,
                             int numPartitions) {
      return value.getCustomerId().hashCode() % numPartitions;
    }
}
```

Определяем секцию по идентификатору покупателя

Обратите внимание, что новый ключ мы еще не сгенерировали. Для определения нужной секции мы используем одно из свойств значения. Главный вывод из этого краткого отступления: записи, для обновления и модификации которых применяется состояние, должны находиться в одной секции.

ПРЕДУПРЕЖДЕНИЕ

Не относитесь легкомысленно к повторному секционированию, исходя из этой простой демонстрации. Хотя повторное секционирование иногда необходимо, оно сопряжено с определенными издержками: дублированием данных и дополнительными вычислительными затратами на обработку. Я бы советовал вам использовать по возможности операции mapValues(), transformValues() или flatMapValues(), поскольку map(), transform() и flatMap() могут приводить к автоматическому повторному секционированию. Лучше применять повторное секционирование умеренно.

Вернемся теперь к изменениям в узле-обработчике поощрений, необходимым для поддержки преобразований с сохранением состояния.

4.2.5. Обновление узла-обработчика поощрений

Пока мы создали новый узел-обработчик, записывающий объекты с информацией о покупках в топик, секционированный по идентификатору покупателя. Новый топик станет также источником узла-обработчика поощрений, который мы скоро модифицируем. Это нам было нужно, чтобы гарантировать попадание всех покупок одного покупателя в одну секцию, так что мы будем далее использовать одно хранилище состояния для всех покупок конкретного покупателя. На рис. 4.8 показана обновленная топология обработки с новым узлом повторного секционирования между узлом маскирования платежных карт (который служит источником для всех транзакций покупок) и узлом-обработчиком поощрений.

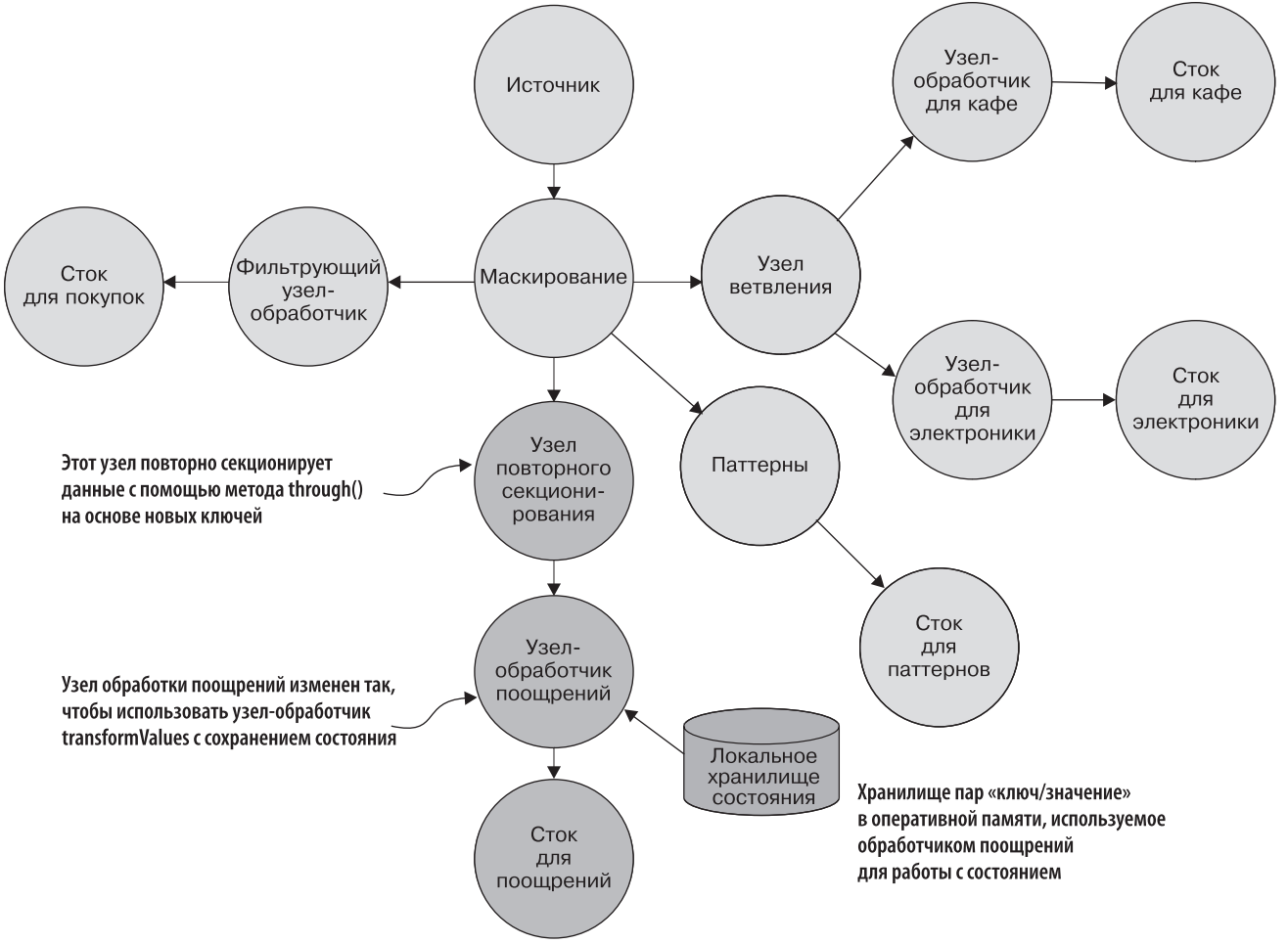


Рис. 4.8. Новый узел повторного секционирования обеспечивает распределение покупок по секциям в соответствии с идентификаторами покупателей, что позволяет узлу-обработчику поощрений правильно обновлять данные на основе локального состояния

Теперь мы воспользуемся в следующем коде новым экземпляром `Stream` (созданным методом `KStream.through()`) для обновления узла-обработчика поощрений и применения подхода к преобразованию с сохранением состояния (листинг 4.6).

Листинг 4.6. Меняем узел-обработчик поощрений так, чтобы использовать преобразования с сохранением состояния

```
KStream<String, RewardAccumulator> statefulRewardAccumulator =
➥ transByCustomerStream.transformValues(() ->
➥ new PurchaseRewardTransformer(rewardsStateStoreName),
                                 rewardsStateStoreName);
statefulRewardAccumulator.to("rewards",
                             Produced.with(stringSerde,
                                           rewardAccumulatorSerde));
```

Использует преобразование с сохранением состояния

Записывает результаты в топик

Метод `KStream.transformValues` принимает посредством лямбда-выражения Java 8 экземпляр интерфейса `ValueTransformerSupplier<V, R>`.

В данном разделе мы добавили сохранение состояния к узлу, в котором оно ранее не сохранялось. Благодаря этому ZMart может быстрее отреагировать на покупку, подходящую под критерии поощрения. Вы увидели, как использовать хранилище состояния и выгоды от него, но мы умолчали о важных нюансах, касающихся возможного влияния состояния на ваши приложения. С учетом этого мы посвятим следующий раздел обсуждению разновидностей хранилищ состояния, требований, которые нужно удовлетворить для эффективного применения состояния, и способов добавления хранилища состояния в программу Kafka Streams.

4.3. Использование хранилищ состояния для поиска и ранее просмотренные данные

В этом разделе мы займемся основами использования хранилищ состояния в Kafka Streams и ключевыми аспектами применения состояния в потоковых приложениях в целом. Благодаря таким знаниям вы сможете на практике принимать верные решения при использовании состояния в своих приложениях Kafka Streams.

Пока мы обсудили, почему может понадобиться сохранять состояние при работе с потоками данных, и рассмотрели пример одной из простейших операций с сохранением состояния из числа доступных в Kafka Streams. Прежде чем углубиться в использование хранилищ состояния в Kafka Streams, коротко рассмотрим два важных атрибута состояния: локальность данных и восстановление после сбоя.

4.3.1. Локальность данных

Локальность данных критически важна для высокой производительности. Хотя поиск по ключу обычно выполняется очень быстро при работе с большими объемами данных, задержка вследствие использования удаленных хранилищ оказывается узким местом.

Рисунок 4.9 иллюстрирует принципы локальности данных. Штриховая линия символизирует обращение по сети с целью извлечения данных из удаленной базы данных. Сплошная линия изображает обращение к хранилищу данных в оперативной памяти, расположенному на том же сервере. Как вы можете видеть, обращаться за данными локально более выгодно с точки зрения производительности по сравнению с сетевым обращением к удаленной базе данных.

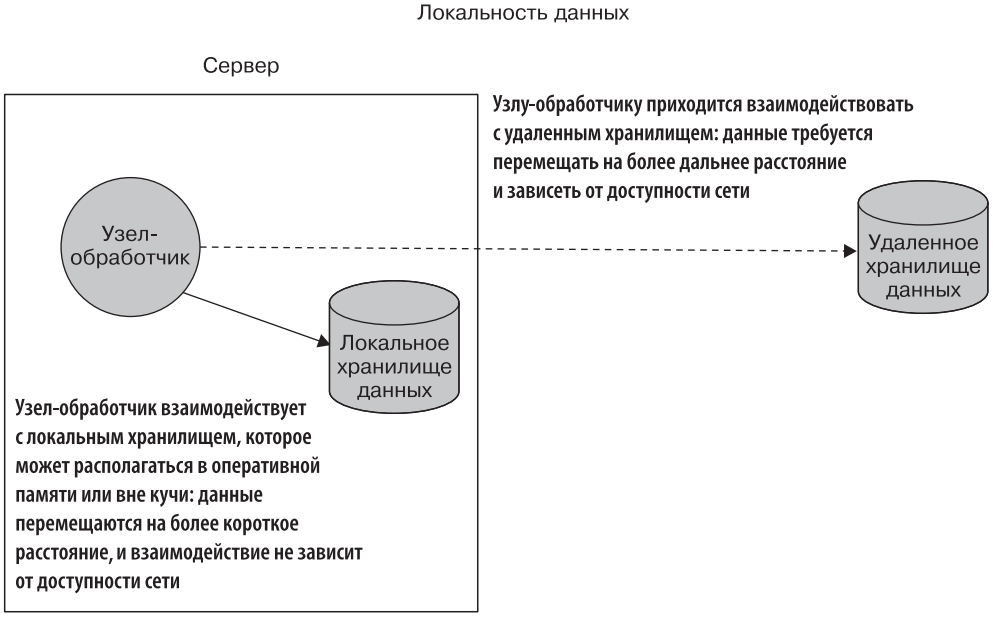


Рис. 4.9. Локальность данных жизненно необходима для потоковой обработки

Главное здесь не длительность задержки в пересчете на одну извлеченную запись, которая может быть минимальной. Суть в том, что потоковое приложение потенциально может обрабатывать миллионы или даже миллиарды записей, так что даже минимальная сетевая задержка, умноженная на такое число записей, может сыграть роковую роль.

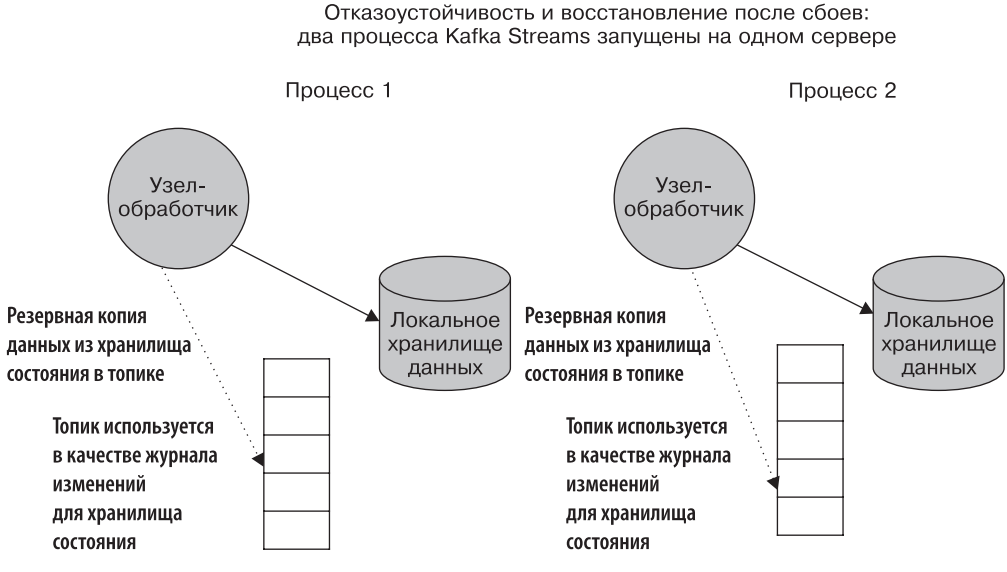
Локальность данных также означает локальность хранилища по отношению ко всем обрабатывающим узлам и отсутствие совместного его использования различными процессами или потоками выполнения. Таким образом, сбой одного процесса не влияет на другие процессы или потоки выполнения при потоковой обработке.

Ключевой момент состоит в том, что, хотя для потоковых приложений иногда требуется сохранение состояния, оно должно быть локальным по отношению к месту обработки. У каждого сервера или узла приложения должно быть свое хранилище состояния.

4.3.2. Восстановление после сбоя и отказоустойчивость

Сбои в приложениях неизбежны, особенно когда речь идет о распределенных приложениях. Необходимо переместить акцент с предотвращения сбоев на быстрое восстановление после них или даже восстановление после перезапуска.

Рисунок 4.10 демонстрирует основные принципы локальности данных и отказоустойчивости. У каждого узла-обработчика — свое локальное хранилище данных, а топик для журнала изменений играет роль резервной копии хранилища состояния.



В силу наличия у каждого из процессов своего собственного хранилища состояния и архитектуры, не предусматривающей разделения ресурсов, никакого влияния на второй процесс не оказывается. Кроме того, ключи/значения каждого из топиков реплицируются в топик, используемый для восстановления значений, утраченных при сбое или перезапуске процесса

Рис. 4.10. Способность восстанавливаться после сбоев очень важна для потоковых приложений. Kafka Streams сохраняет во внутреннем топике данные из локальных хранилищ в оперативной памяти, так что при возобновлении работы после сбоя или перезапуска производится повторное заполнение данными

Высокая стоимость резервного копирования хранилища состояния в топик смягчается за счет того, что `KafkaProducer` отправляет записи в пакетном режиме, и по умолчанию они кэшируются. Kafka Streams заносит записи в хранилище только при сбросе кэша на диск, так что сохраняется только последняя запись с заданным ключом. Мы обсудим механизм кэширования хранилищ состояния подробнее в главе 5.

Хранилища состояния Kafka Streams удовлетворяют требованию как локальности, так и отказоустойчивости. Они локальны по отношению к заданным узлам-обработчикам и не используются совместно различными процессами и потоками выполнения. Кроме того, для резервного копирования и быстрого восстановления хранилища состояния применяются топики.

Мы рассмотрели требования к использованию состояния в потоковых приложениях. Далее мы обсудим, как обеспечить применение состояния в приложении Kafka Streams.

4.3.3. Использование хранилищ состояния в Kafka Streams

Для добавления хранилища состояния достаточно создать экземпляр интерфейса `StoreSupplier` с помощью одного из статических фабричных методов класса `Stores`. Для настройки хранилища состояния под свои задачи существует два дополнительных класса: `Materialized` и `StoreBuilder`. Какой использовать — зависит от способа добавления хранилища в топологию. В случае высокоуровневого DSL обычно применяется класс `Materialized`, а при работе с низкоуровневым API узлов-обработчиков — `StoreBuilder`.

Хотя в текущем примере используется высокоуровневый DSL, хранилище состояния мы будем добавлять в `Transformer`, предоставляющий семантику API узлов-обработчиков. Поэтому для настройки хранилища состояния под наши нужды мы будем применять `StoreBuilder` (листинг 4.7).

Листинг 4.7. Добавление хранилища состояния

```
String rewardsStateStoreName = "rewardsPointsStore";
KeyValueBytesStoreSupplier storeSupplier =
➥ Stores.inMemoryKeyValueStore(rewardsStateStoreName);

StoreBuilder<KeyValueStore<String, Integer>> storeBuilder =
➥ Stores.keyValueStoreBuilder(storeSupplier,
                               Serdes.String(),
                               Serdes.Integer());

builder.addStateStore(storeBuilder);
```

Создаем объект — поставщик StateStore

Создаем объект StoreBuilder и задаем типы ключа и значения

Добавляем хранилище состояния в топологию

Сначала мы создаем объект `StoreSupplier`, который, в свою очередь, создает хранилище (в оперативной памяти) пар «ключ/значение». Далее мы передаем этот `StoreSupplier` в качестве параметра при создании `StoreBuilder`, указывая также ключи в виде `String` и значения в виде `Integer`. Наконец, мы добавляем `StateStore` в топологию, передавая объект `StoreBuilder` в метод `addStateStore`. В результате мы получаем возможность обращаться к объекту для состояния в узлах-обработчиках по созданному выше имени `rewardsStateStoreName`.

Мы показали вам пример создания хранилища состояния в оперативной памяти, но существует немало опций для создания различных видов экземпляров `StateStore`. Рассмотрим их вкратце.

4.3.4. Другие поставщики хранилищ пар «ключ/значение»

Помимо метода `Stores.inMemoryKeyValueStore`, существуют и другие статические фабричные методы для генерации поставщиков хранилищ:

- `Stores.persistentKeyValueStore`;
- `Stores.lruMap`;
- `Stores.persistentWindowStore`;
- `Stores.persistentSessionStore`.

Стоит отметить, что все экземпляры `StateStore`, созданные с помощью методов, которые содержат слово `persistent` в названии, осуществляют локальное хранение данных с помощью RocksDB (http://rocksdb.org).

Прежде чем завершить обсуждение хранилищ состояния, я хотел бы рассказать еще о двух важных аспектах хранилищ состояния Kafka Streams: об обеспечении отказоустойчивости с помощью топиков журналов изменений и настройке этих топиков.

4.3.5. Отказоустойчивость StateStore

Во всех типах `StateStoreSupplier` журналирование включено по умолчанию. *Журналирование* в данном контексте означает использование журнала изменений в виде топика Kafka для резервного копирования значения из хранилища и обеспечения отказоустойчивости.

Например, предположим, что произошел отказ одной из машин, на которых работает Kafka Streams. После восстановления сервера и перезапуска приложения Kafka Streams восстанавливается исходное содержимое хранилищ состояния этого экземпляра (последнее зафиксированное перед сбоем смещение в журнале изменений).

Журналирование можно отключить при использовании фабрики `Stores` с помощью метода `disableLogging()`. Но при этом хранилища состояния теряют отказоустойчивость и способность восстанавливаться после аварийных сбоев, так что делать это без веских оснований не стоит.

4.3.6. Настройки топиков журналов изменений

Журналы изменений для хранилищ состояния можно настраивать с помощью метода `withLoggingEnabled(Map <String, String> config)`. При этом можно использовать любые доступные в интерфейсе `java.util.Map` для топиков параметры конфигурации. Настройки журналов изменений для хранилищ состояния играют важную роль

при создании приложений Kafka Streams. Но не забывайте, что создавать вручную топики журналов изменений не нужно — Kafka Streams делает это за вас.

ПРИМЕЧАНИЕ

Журналы изменений хранилищ состояния представляют собой сжатые топики, обсуждавшиеся в главе 2. Как вы помните, семантика удаления требует, чтобы ключ был пустым, так что, если нужно удалить запись из хранилища состояния навсегда, необходимо выполнить операцию put(key, null).

Длительность сохранения данных для сегмента журнала в топиках Kafka по умолчанию равна одной неделе. В зависимости от объема ваших данных это может оказаться вполне удовлетворительным значением, но очень может быть, что вы захотите изменить такую настройку. Кроме того, стратегия очистки по умолчанию — `delete`.

Рассмотрим сначала, как настроить топик журнала изменений так, чтобы установить объем сохраняемой информации равным 10 Гбайт и период сохранения — двум дням (листинг 4.8).

Листинг 4.8. Задание свойств журналов изменений

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms","172800000" );
changeLogConfigs.put("retention.bytes", "10000000000");

// Для использования с классом StoreBuilder
storeBuilder.withLoggingEnabled(changeLogConfigs);

// Для использования с классом Materialized
Materialized.as(Stores.inMemoryKeyValueStore("foo")
                      .withLoggingEnabled(changeLogConfigs));
```

В главе 2 мы обсуждали предоставляемую Kafka возможность сжатия топиков. Освежим вашу память: в сжатых топиках применяется другой подход к очистке. Вместо удаления сегментов журнала время от времени сегменты журналов *сжимаются* (compacted): для каждого ключа оставляется только последняя запись, а более старые записи с тем же ключом удаляются. По умолчанию Kafka Streams создает топики журналов изменений со стратегией очистки `compact`.

Но если в вашем топике журнала изменений множество уникальных ключей, то сжатия может оказаться недостаточно, так как размер сегмента журнала будет продолжать расти. Существует простое решение этой проблемы — достаточно задать стратегию очистки `delete` и `compact` (листинг 4.9).

Листинг 4.9. Задание стратегии очистки

```
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms","172800000" );
changeLogConfigs.put("retention.bytes", "10000000000");
changeLogConfigs.put("cleanup.policy", "compact,delete");
```

Теперь размер вашего топика журнала изменений останется в разумных пределах даже с уникальными ключами. Это было лишь краткое описание настроек топиков, в приложении А вы найдете дополнительную информацию о топиках журналов изменений и их настройках.

Мы рассмотрели основы операций с сохранением состояния и хранилищами состояния. Вы узнали о предоставляемых Kafka Streams хранилищах состояния — в оперативной памяти и постоянных — и разобрались, как включить их в свое приложение. Вы также узнали, насколько важны локальность данных и отказоустойчивость при использовании состояния в потоковом приложении. Займемся теперь соединением потоков данных.

4.4. Получение дополнительной информации путем соединения потоков данных

Как мы уже обсуждали ранее в этой главе, сохранение состояния необходимо для потоков данных тогда, когда события в них неавтономны. Иногда нужные вам состояние или контекст содержатся в другом потоке данных. В этом разделе мы займемся формированием нового события путем сочетания различных событий из двух потоков с одинаковым ключом.

Лучше всего посмотреть на соединение потоков данных на конкретном примере, так что мы вернемся к примеру компании ZMart. Как вы помните, компания ZMart открыла новую линейку магазинов, торгующих электроникой и сопутствующими товарами (CD и DVD, смартфонами и т. п.). В попытке найти новый подход компания ZMart начала сотрудничество с общенациональной сетью кофеен и открыла на территории всех своих магазинов кафе. В главе 3 мы занимались разделением транзакций покупок в этих магазинах на два отдельных потока данных. На рис. 4.11 показана топология для удовлетворения этого требования.

Открытие на территории магазинов кафе было очень удачным решением для компании ZMart, и этот успех она хотела бы закрепить. Поэтому было решено запустить новую программу. ZMart хотела бы поддерживать посещаемость покупателями магазинов электроники за счет предложения купонов на посещение кафе (в надежде, что повышение посещаемости приведет к дополнительным продажам).

Компания ZMart хотела бы идентифицировать покупателей, которые заказали кофе и совершили затем покупку в магазине электроники, и отправлять им купон почти сразу после этой второй транзакции (рис. 4.12). ZMart хочет выяснить, удастся ли им выработать у своих покупателей своеобразный условный рефлекс.

Чтобы определить, когда выдавать купон, необходимо соединить информацию о заказах в кафе с информацией о продажах в магазине электроники. Код соединения потоков данных довольно прост. Начнем с подготовки данных, которые необходимо обработать для выполнения соединения.

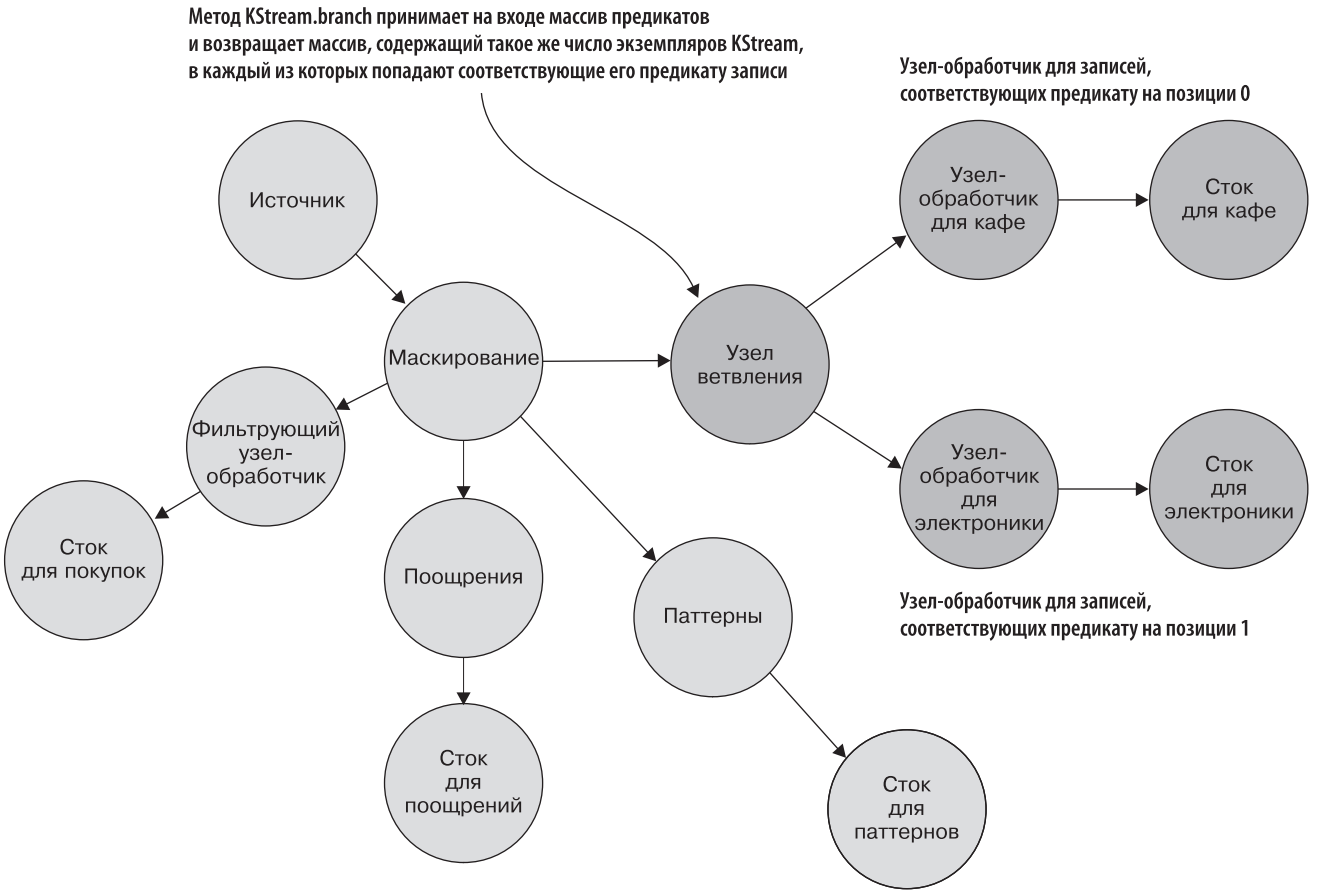


Рис. 4.11. Узел ветвления и его место в общей топологии

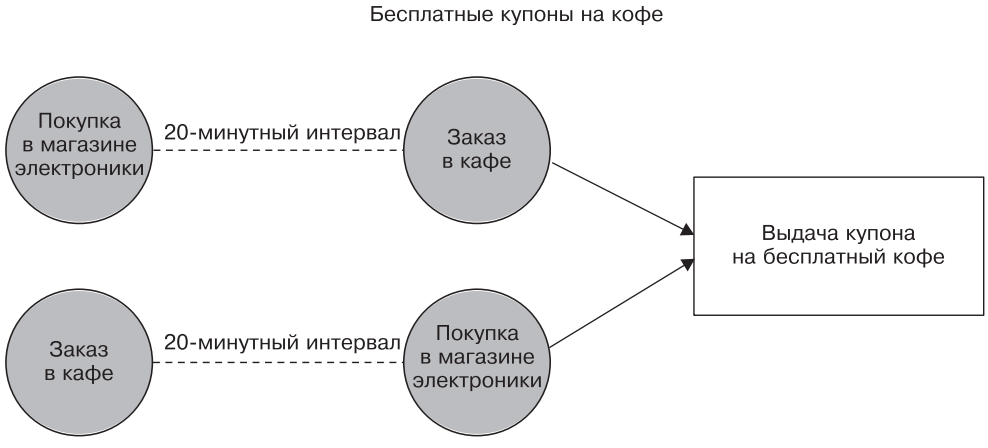


Рис. 4.12. Записи о покупках с метками даты/времени, отдаленными друг от друга не более чем на 20 минут, соединяются по идентификатору покупателя, после чего на их основе покупателю выдается поощрительный бесплатный купон на кофе

4.4.1. Подготовка данных

Во-первых, взглянем еще раз на ту часть топологии, которая отвечает за ветвление потоков данных (рис. 4.13). И проанализируем код, реализующий ветвление (его можно найти в файле src/main/java/bbejeck/chapter_3/ZMartKafkaStreamsAdvancedReqsApp.java) (листинг 4.10).

Листинг 4.10. Ветвление на два потока данных

```
Predicate<String, Purchase> coffeePurchase = (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("coffee");

Predicate<String, Purchase> electronicPurchase = (key, purchase) ->
➥ purchase.getDepartment().equalsIgnoreCase("electronics");

final int COFFEE_PURCHASE = 0;
final int ELECTRONICS_PURCHASE = 1;

KStream<String, Purchase>[] branchedTransactions =
➥ transactionStream.branch(coffeePurchase, electronicPurchase);
```

Описание предикатов для выбора записей

Для большей ясности при обращении к соответствующему массиву задаем имена для целочисленных констант

Ветвление потока данных

Этот код демонстрирует реализацию ветвления: входящие записи отбираются в соответствии с предикатами и возвращается массив экземпляров `KStream`. Индексы объектов `KStream` в массиве точно такие же, как и порядок предикатов. Любые записи, не соответствующие ни одному из предикатов, в процессе ветвления отбрасываются.

Хотя у нас уже есть два потока данных для соединения, остался еще один шаг. Как вы помните, записи о покупках поступают в приложение Kafka Streams без

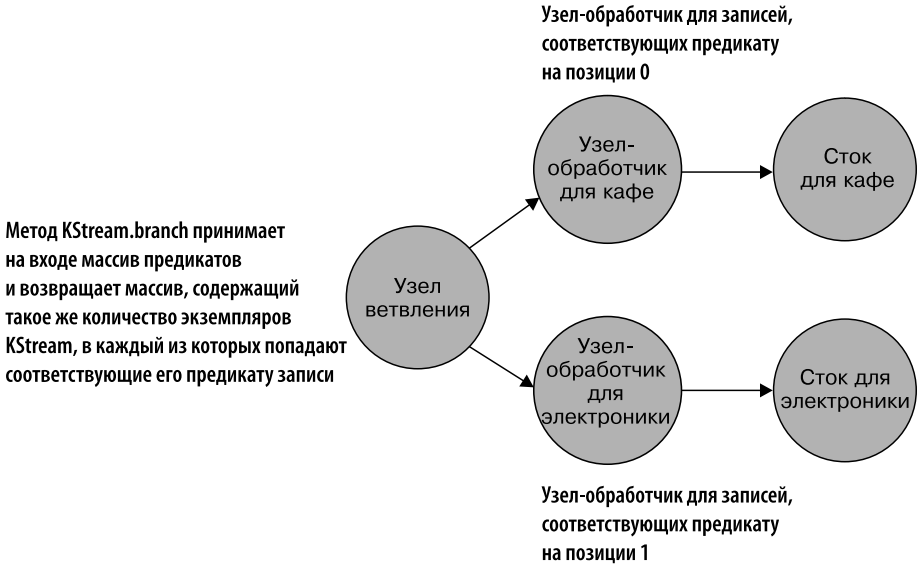


Рис. 4.13. Для соединения необходимо более одного потока данных. Узел-обработчик ветвления обеспечивает это за счет создания двух потоков: одного с заказами в кафе, а второго — с покупками электроники

ключей. В результате нам нужно добавить еще один узел-обработчик для генерации ключа с идентификатором покупателя. Ключи должны быть заполнены, поскольку именно их мы и будем использовать для соединения записей.

4.4.2. Генерация ключей с идентификаторами покупателей для соединения

Для генерации ключа выбирается идентификатор покупателя из данных о покупке, содержащихся в потоке. Для этого необходимо обновить исходный экземпляр `KStream` (`transactionStream`) и создать еще один обрабатывающий узел между ним и узлом ветвления. Это выполняется с помощью следующего кода (располагается в файле src/main/java/bbejeck/chapter_4/KafkaStreamsJoinsApp.java) (листинг 4.11).

Листинг 4.11. Генерация новых ключей

```
KStream<String, Purchase>[] branchesStream =
➥ transactionStream.selectKey((k,v)->
➥ v.getCustomerId()).branch(coffeePurchase, electronicPurchase);
```

Вставка узла выбора ключей

На рис. 4.14 показано, как выглядит обновленная топология обработки с учетом листинга 4.11. Как вы уже видели, при изменении ключа может потребоваться повторное секционирование данных. Это справедливо и для настоящего примера. Почему же на рисунке нет шага повторного секционирования?

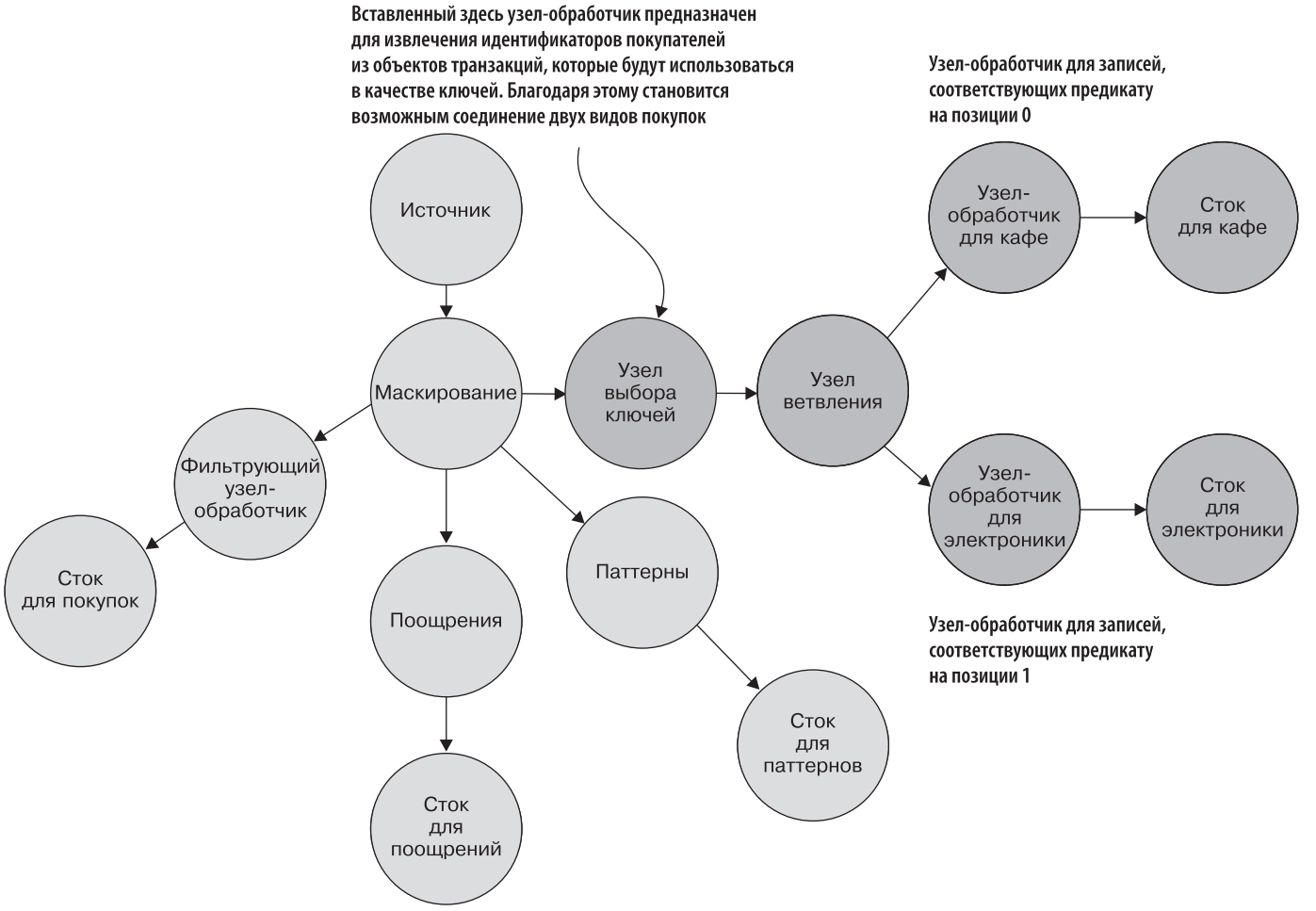


Рис. 4.14. Необходимо повторно отобразить записи о покупках типа «ключ/значение» в такие записи, где ключ содержит идентификатор покупателя. К счастью, идентификатор покупателя можно извлечь из объекта Purchase

При каждом вызове в Kafka Streams метода, в результате выполнения которого может быть сгенерирован новый ключ (`selectKey`, `map` или `transform`), значение специального внутреннего булева флага устанавливается в `true`, указывая, что этот новый экземпляр `KStream` требует повторного секционирования. А при таком значении указанного флага Kafka Streams автоматически осуществляет вместо вас повторное секционирование при выполнении вами любой операции соединения, свертки или агрегирования.

В данном примере мы выполнили над потоком `transactionStream` операцию `selectKey()`, так что полученный `KStream` помечен как подлежащий повторному секционированию. Кроме того, мы сразу после этого выполняем операцию ветвления, так что все возвращаемые из вызова метода `branch()` объекты `KStream` также помечены для повторного секционирования.

ПРИМЕЧАНИЕ

В этом примере повторное секционирование производится только по ключам. Но бывают случаи, когда использовать ключи нежелательно или требуется применить какое-либо сочетание ключа и значения. В подобных случаях можно воспользоваться интерфейсом StreamPartitioner<K, V>, как вы видели в листинге 4.5 в пункте «Использование StreamPartitioner».

Теперь у нас есть два отдельных потока данных с заполненными ключами, и мы готовы к следующему шагу: соединению потоков данных по ключу.

4.4.3. Конструирование соединения

Следующий шаг — выполнение собственно соединения. Мы возьмем два разветвленных ранее потока и соединим их с помощью метода `KStream.join()`. Получившаяся топология показана на рис. 4.15.

Соединение записей о покупках

Для создания соединенной записи необходимо создать сначала экземпляр интерфейса `ValueJoiner<V1, V2, R>`. `ValueJoiner` принимает в качестве параметров два объекта одного или разных типов и возвращает единый объект, возможно, третьего типа. В данном случае `ValueJoiner` принимает два объекта `Purchase` и возвращает объект `CorrelatedPurchase`. Взглянем на код (его можно найти в файле src/main/java/bbejeck/chapter_4/joiner/PurchaseJoiner.java) (листинг 4.12).

Для создания объекта `CorrelatedPurchase` мы извлекаем определенную информацию из каждого объекта `Purchase`. Поскольку требуемое для конструирования нового объекта число элементов велико, мы воспользуемся паттерном проектирования «Строитель», чтобы сделать код понятнее и избежать возможных ошибок вследствие перепутанных местами параметров. Кроме того, `PurchaseJoiner` проверяет, не пусты ли значения обоих передаваемых объектов `Purchase`, так что его можно применять для внутреннего, внешнего и левого внешнего соединений. Мы обсудим различные варианты соединений в подразделе 4.4.4. А пока перейдем к реализации соединения потоков данных.

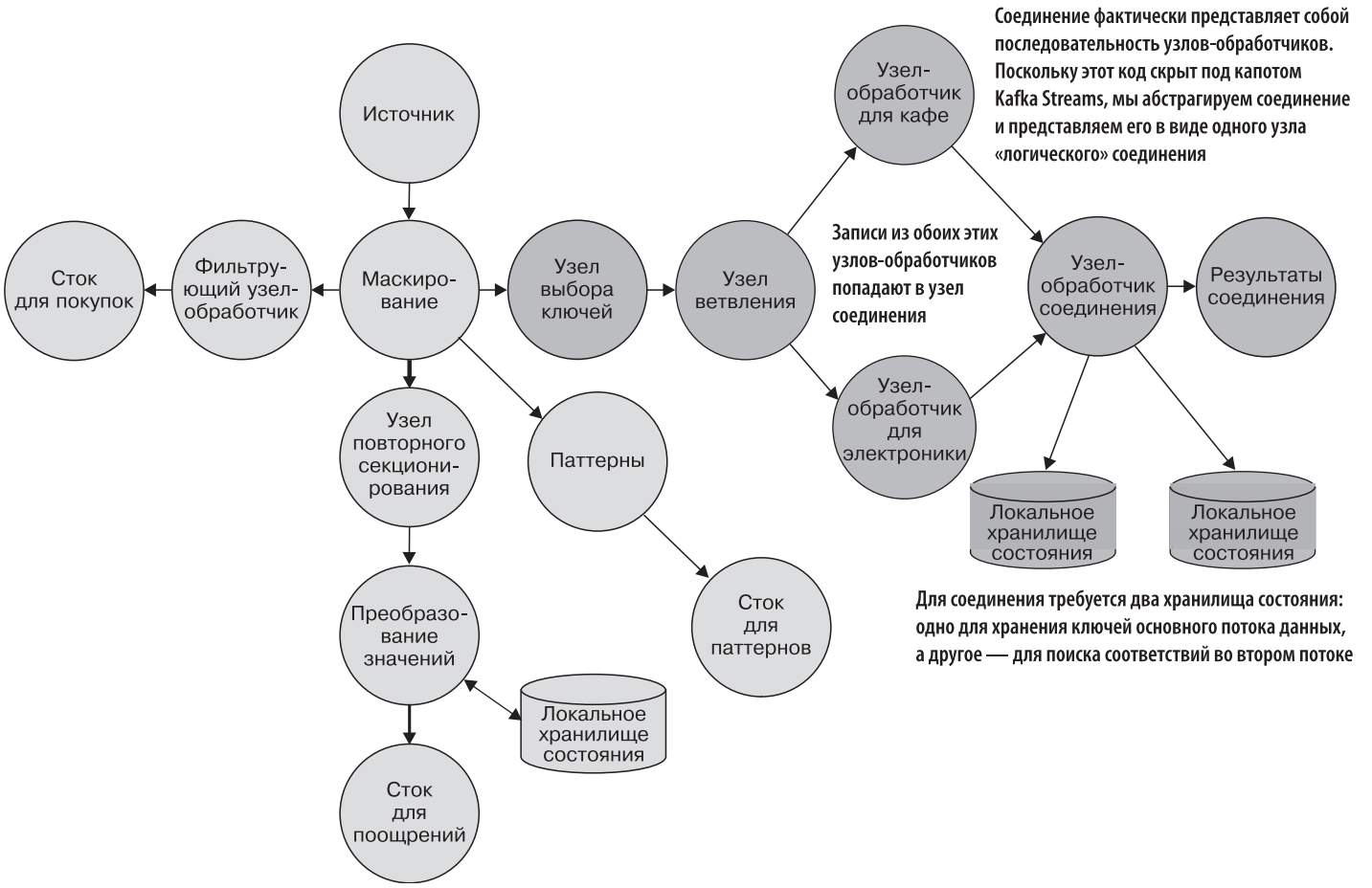


Рис. 4.15. В этой обновленной топологии оба узла-обработчика (для кафе и для магазина электроники) пересылают свои записи в узел соединения. В узле соединения для поиска соответствий во втором потоке используется два хранилища состояния

Листинг 4.12. Реализация ValueJoiner

```
public class PurchaseJoiner
➥ implements ValueJoiner<Purchase, Purchase, CorrelatedPurchase> {

   @Override
   public CorrelatedPurchase apply(Purchase purchase, Purchase purchase2) {
     CorrelatedPurchase.Builder builder =
     ➥ CorrelatedPurchase.newBuilder();

     Date purchaseDate =
     ➥ purchase != null ? purchase.getPurchaseDate() : null;

     Double price = purchase != null ? purchase.getPrice() : 0.0;

     String itemPurchased =
     ➥ purchase != null ? purchase.getItemPurchased() : null;

     Date otherPurchaseDate =
     ➥ otherPurchase != null ? otherPurchase.getPurchaseDate() : null;

     Double otherPrice =
     ➥ otherPurchase != null ? otherPurchase.getPrice() : 0.0;

     String otherItemPurchased =
     ➥ otherPurchase != null ? otherPurchase.getItemPurchased() : null;

     List<String> purchasedItems = new ArrayList<>();

     if (itemPurchased != null) {
         purchasedItems.add(itemPurchased);
     }

     if (otherItemPurchased != null) {
         purchasedItems.add(otherItemPurchased);
     }

     String customerId =
     ➥ purchase != null ? purchase.getCustomerId() : null;

     String otherCustomerId =
     ➥ otherPurchase != null ? otherPurchase.getCustomerId() : null;

     builder.withCustomerId(customerId != null ? customerId :
     ➥ otherCustomerId)
            .withFirstPurchaseDate(purchaseDate)
            .withSecondPurchaseDate(otherPurchaseDate)
            .withItemsPurchased(purchasedItems)
            .withTotalAmount(price + otherPrice);

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

Создает экземпляр Builder

Обработчик для пустого объекта Purchase в случае внешнего соединения

Обработчик для пустого объекта Purchase в случае левого внешнего соединения

Возвращает новый объект CorrelatedPurchase

Реализация соединения

Вы уже видели, как выполняется слияние записей, получившихся в результате соединения потоков данных, так что перейдем сразу к вызову метода `KStream.join` (его можно найти в файле src/main/java/bbejeck/chapter_4/KafkaStreamsJoinsApp.java) (листинг 4.13).

Листинг 4.13. Использование метода join()

```
KStream<String, Purchase> coffeeStream =
➥ branchesStream[COFFEE_PURCHASE];
KStream<String, Purchase> electronicsStream =
➥ branchesStream[ELECTRONICS_PURCHASE];

ValueJoiner<Purchase, Purchase, CorrelatedPurchase> purchaseJoiner =
➥ new PurchaseJoiner();

JoinWindows twentyMinuteWindow =  JoinWindows.of(60 * 1000 * 20);

KStream<String, CorrelatedPurchase> joinedKStream =
➥ coffeeStream.join(electronicsStream,
                     purchaseJoiner,
                     twentyMinuteWindow,
                     Joined.with(stringSerde,
                                 purchaseSerde,
                                 purchaseSerde));

joinedKStream.print("joined KStream");
```

Извлекаем разветвленные потоки

Экземпляр ValueJoiner, используемый для соединения

Вызов метода join, вызывающий автоматическое повторное секционирование объектов coffeeStream и electronicsStream

Конструирование соединения

Вывод результатов соединения в консоль

Мы передаем в метод `KStream.join` четыре параметра:

- `electronicsStream` — поток данных о покупках электроники для соединения;
- `purchaseJoiner` — реализация интерфейса `ValueJoiner<V1, V2, R>`. В `ValueJoiner` передается два значения (возможно, различных типов). Метод `ValueJoiner.apply` выполняет логику реализации и возвращает (возможно, новый) объект типа `R` (возможно, совершенно нового типа). В этом примере `purchaseJoiner` добавляет соответствующую информацию из обоих объектов `Purchase` и возвращает объект `CorrelatedPurchase`;
- `twentyMinuteWindow` — экземпляр класса `JoinWindows`. Метод `JoinWindows.of` позволяет задать максимальную разницу во времени между двумя включаемыми в соединение значениями. В данном случае разница между метками даты/времени не должна превышать 20 минут;
- экземпляр класса `Joined` — осуществляет передачу параметров для соединения. В этом случае параметры представляют собой объекты `Serde` ключа и значения для основного потока данных и объект `Serde` значения для дополнительного.

Экземпляр `Serde` для ключа только один, поскольку при соединении записей ключи должны быть одного типа.

ПРИМЕЧАНИЕ

Объекты Serde необходимы при выполнении соединений потому, что участники соединения материализуются в оконных хранилищах состояния. В этом примере передается только один объект Serde для ключа, поскольку ключи для обеих сторон соединения должны быть одного типа.

Мы указали, что события покупки должны отстоять друг от друга не более чем на 20 минут, но это не подразумевает какой-либо упорядоченности. Соединение будет выполняться для любых меток даты/времени, расположенных в пределах 20 минут друг от друга, независимо от порядка.

Существует два дополнительных метода `JoinWindows()`, с помощью которых можно задать относительный порядок событий:

- `JoinWindows.after` — вызов `streamA.join(streamB,...,twentyMinuteWindow.after(5000)....)` означает, что метка даты/времени записи из потока `streamB` должна находиться в пределах 5 секунд *после* метки даты/времени записи из потока `streamA`. Начальная граница окна не меняется;
- `JoinWindows.before` — вызов `streamA.join(streamB,...,twentyMinuteWindow.before(5000),...)` означает, что метка даты/времени записи из потока `streamB` должна находиться в пределах 5 секунд *до* метки даты/времени записи из потока `streamA`. Конечная граница окна не меняется.

В обоих методах, `before()` и `after()`, длительность промежутка времени выражается в миллисекундах. Используемые в соединениях временные диапазоны представляют собой примеры так называемых *скользящих окон* (sliding windows). Мы обсудим оконные операции подробнее в следующей главе.

ПРИМЕЧАНИЕ

В листинге 4.13 задействуются реальные метки даты/времени транзакций, а не метки даты/времени Kafka. Для применения включенных в транзакции меток даты/времени необходимо задать пользовательское средство извлечения меток даты/времени путем установки значения свойства StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG равным TransactionTimestampExtractor.class.

Мы сформировали соединенный поток данных: покупка электроники, произведенная в пределах 20 минут от заказа кофе, приведет к выдаче покупателю купона на получение при следующем посещении им ZMart бесплатного кофе.

Прежде чем двигаться дальше, я хотел бы дать некоторые пояснения по поводу важного требования при соединении данных — *совместного секционирования* (co-partitioning).

Совместное секционирование

Для выполнения соединения в Kafka Streams необходимо, чтобы все его участники были *совместно секционированы*, то есть у них было одинаковое число секций и ключи одного типа. В результате при вызове метода `join()` в листинге 4.13 оба экземпляра `KStream` проверяются на предмет необходимости повторного секционирования.

ПРИМЕЧАНИЕ

Участвующие в соединении экземпляры интерфейса GlobalKTable повторного секционирования не требуют.

В подразделе 4.4.2 мы применяли к `transactionStream` метод `selectKey()` и сразу после этого осуществляли ветвление по возвращаемым объектам `KStream`. А поскольку метод `selectKey()` изменяет ключи, то как объект `coffeeStream`, так и `electronicsStream` требуют повторного секционирования. Имеет смысл еще раз сказать, что повторное секционирование требуется для того, чтобы гарантировать попадание записей с идентичными ключами в одну секцию. Повторное секционирование выполняется автоматически. Кроме того, при запуске приложения Kafka Streams проверяется, одинаковое ли число секций в участвующих в соединении топиках, и при обнаружении расхождений генерирует исключение `TopologyBuilderException`. За то, чтобы участвующие в соединении ключи были одного типа, отвечает разработчик.

Совместное секционирование также требует, чтобы все генераторы Kafka использовали один и тот же класс секционирования при записи в топики-источники Kafka Streams. Аналогично вы должны применять один и тот же объект `StreamPartitioner` при любых операциях записи в топики-стоки Kafka Streams посредством метода `KStream.to()`. Если же вы придерживаетесь стратегий секционирования по умолчанию, то можете о них не волноваться.

Продолжим разговор о соединениях и посмотрим на остальные их варианты.

4.4.4. Другие варианты соединений

Соединение из листинга 4.13 представляет собой *внутреннее соединение* (inner join). При нем, если запись с одной из сторон отсутствует, соединение не производится и объект `CorrelatedPurchase` не возвращается. Существуют и другие варианты соединений, не требующие наличия записей с обеих сторон. Они удобны в тех случаях, когда информация нужна, но необходимая запись для соединения отсутствует.

Внешние соединения

При внешнем соединении всегда возвращается запись, хотя отправляемая дальше соединенная запись может не включать оба задаваемых соединением события. Если одна из сторон соединения отсутствует в пределах временного окна, то внешнее

соединение отправляет далее по конвейеру имеющуюся запись. Конечно, если в пределах временного окна есть оба события, то возвращаемая запись содержит их оба.

Например, в листинге 4.13 можно было бы воспользоваться внешним соединением следующим образом:

```
coffeeStream.outerJoin(electronicsStream,..)
```

Рисунок 4.16 демонстрирует три возможных результата внешнего соединения.

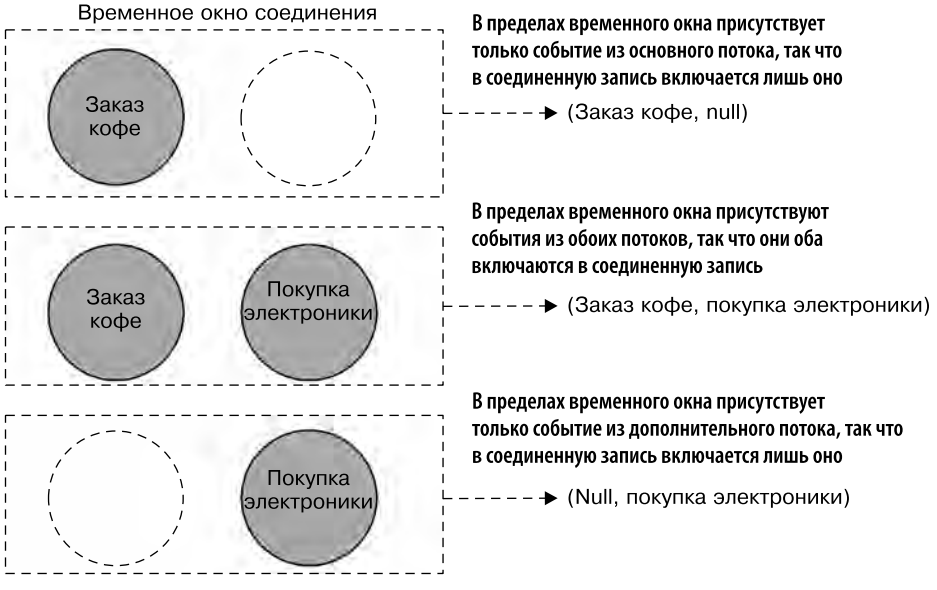


Рис. 4.16. При внешнем соединении возможны три результата: только событие из основного потока данных, оба события и только событие из второго потока

Левое внешнее соединение

Отправляемые далее по конвейеру из левого внешнего соединения записи аналогичны полному внешнему соединению за одним исключением. Если в окне соединения присутствует только событие из второго потока, то никакого результата вообще не возвращается. В листинге 4.13 можно было бы воспользоваться внешним соединением следующим образом:

```
coffeeStream.leftJoin(electronicsStream..)
```

Рисунок 4.17 демонстрирует возможные результаты левого внешнего соединения.

Мы охватили в этом разделе соединение потоков данных, но одна концепция заслуживает более подробного обсуждения: метки даты/времени и их влияние на приложение Kafka Streams. В примере с соединением мы задавали максимальную разницу между моментами событий равной 20 минутам. В нашем случае речь шла

о времени между покупками, но мы не конкретизировали, как задавать или извлекать эти метки даты/времени. Рассмотрим этот вопрос подробнее.

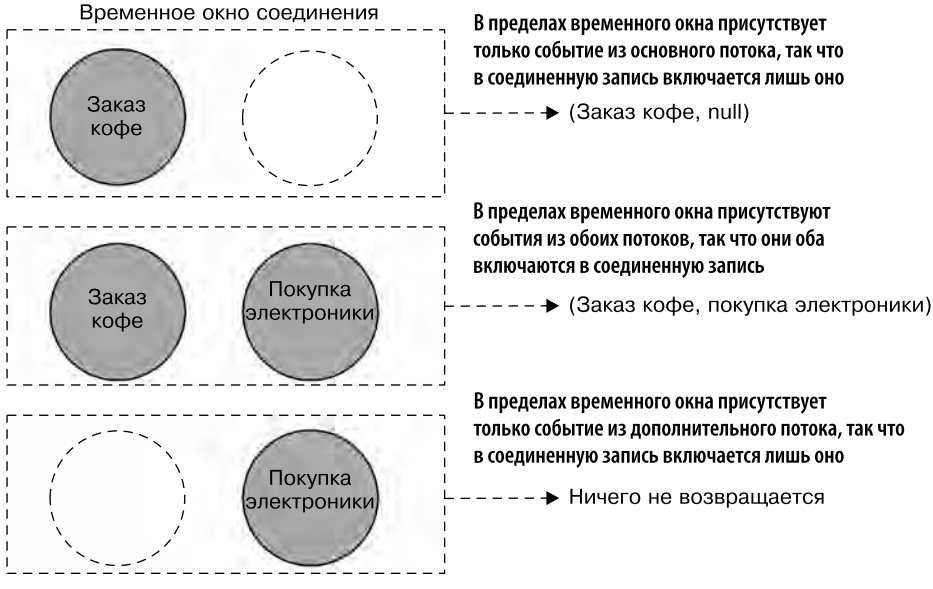


Рис. 4.17. При левом внешнем соединении тоже возможны три результата, но если присутствует только событие из второго потока, то ничего не возвращается

4.5. Метки даты/времени в Kafka Streams

В подразделе 2.4.4 мы обсуждали метки даты/времени в записях Kafka. В этом разделе мы поговорим про использование меток даты/времени в Kafka Streams. Метки даты/времени играют важную роль в следующих ключевых аспектах функциональности Kafka Streams:

- в соединении потоков данных;
- обновлении журналов изменений (API `KTable`);
- определении момента запуска на выполнение метода `Processor.punctuate()` (API узлов-обработчиков).

Мы пока не рассматривали API `KTable` и узлов-обработчиков, но ничего страшного. Для понимания изложенного в данном разделе это вам не понадобится.

При обсуждении потоковой обработки можно сгруппировать метки даты/времени по трем категориям, как показано на рис. 4.18.

- *Время события (event time)* — метка даты/времени, задаваемая в момент генерации события, обычно включается в объект, представляющий событие. Здесь мы будем считать временем события также и метку даты/времени, задаваемую при создании объекта `ProducerRecord`.

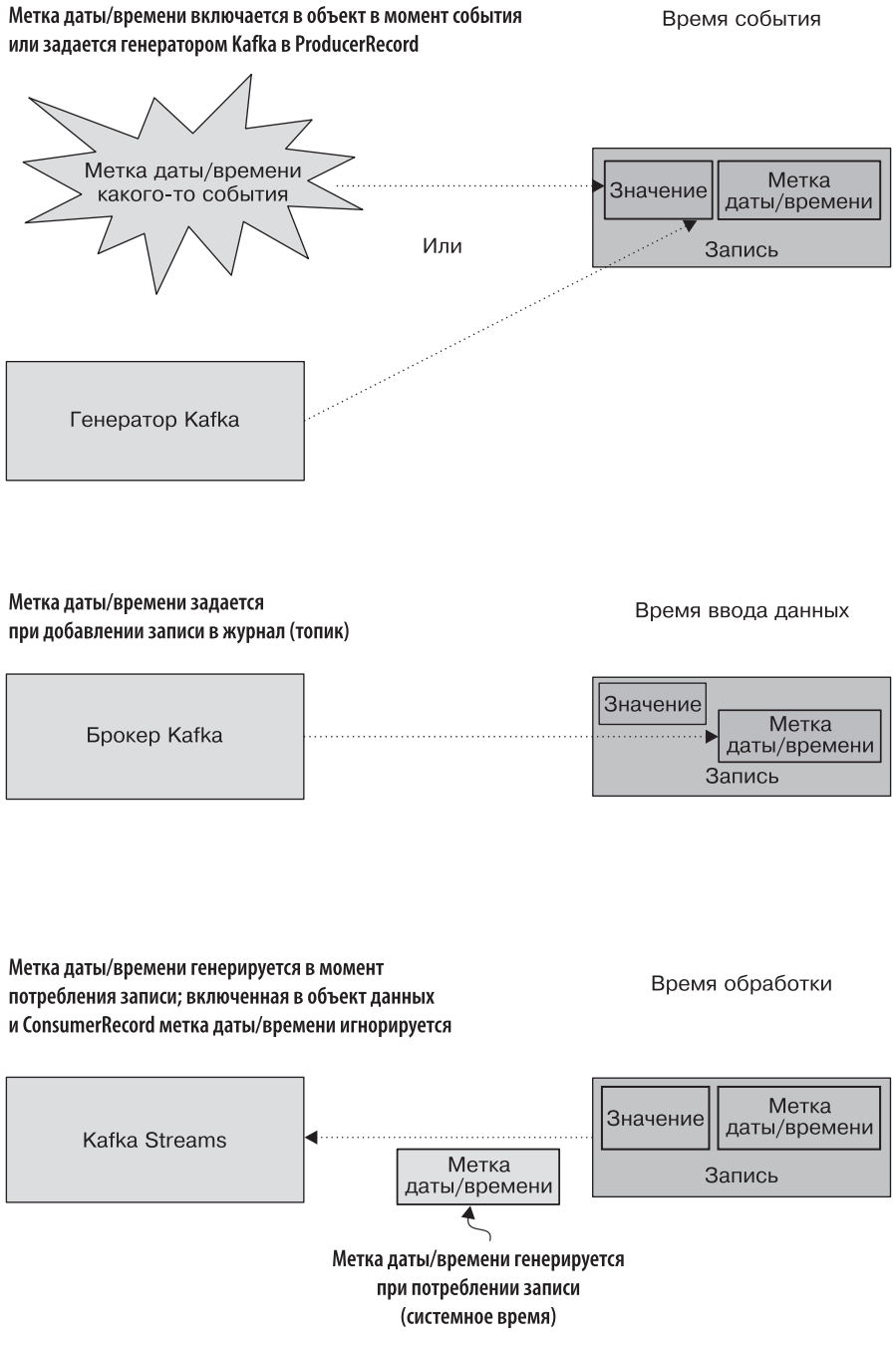


Рис. 4.18. В Kafka Streams существует три категории меток даты/времени: время события, время ввода и время обработки

- *Время ввода данных (ingestion time)* — метка даты/времени, задаваемая в момент первого попадания данных в конвейер их обработки. Можно считать временем ввода данных метку даты/времени, устанавливаемую брокером Kafka (при параметре конфигурации `log.message.timestamp.type`, равном `LogAppendTime`).
- *Время обработки (processing time)* — метка даты/времени, задаваемая в момент начала прохождения данных или записи о событии через конвейер обработки.

В этом разделе я покажу, как API Kafka Streams поддерживает все три типа меток даты/времени.

ПРИМЕЧАНИЕ

До сих пор мы неявно предполагали, что клиенты и брокеры располагаются в одном часовом поясе, но это может быть не так. При использовании меток даты/времени безопаснее будет применять всемирное координированное время (UTC), чтобы исключить возможные недоразумения при использовании различными брокерами и клиентами различных часовых поясов.

Мы рассмотрим три случая семантики обработки меток даты/времени:

- метку даты/времени, включаемую в объект самого события или сообщения (семантика времени события);
- использование метки даты/времени, задаваемой в метаданных записи при создании объекта `ProducerRecord` (семантика времени события);
- использование текущей метки даты/времени (текущее локальное время) при вводе записи[1] в приложение Kafka Streams (семантика времени обработки).

Для семантики времени события вполне достаточно использовать метку даты/времени, которую помещает в метаданные `ProducerRecord`. Но встречаются случаи, когда этого недостаточно. Рассмотрим следующие примеры:

- вы отправляете в Kafka сообщения с событиями, с зафиксированными в объектах сообщений метками даты/времени. Эти объекты событий становятся доступными генератору Kafka с некоторым запозданием, так что необходимо учитывать только включенную в объект метку даты/времени;
- требуется различать время потребления записей приложением Kafka Streams и время, указанное в метках даты/времени записей.

Чтобы сделать возможными различные семантики обработки, Kafka Streams предоставляет интерфейс `TimestampExtractor` с одной абстрактной и четырьмя конкретными его реализациями. Для работы с включенными в значения записей метками даты/времени необходимо создать пользовательскую реализацию интерфейса `TimestampExtractor`. Рассмотрим вкратце уже готовые реализации и реализуем свой `TimestampExtractor`.

---

[1] Возможно, автор неточно выражается или ошибается в данном месте. По логике, здесь должно говориться не о «вводе записи» (ingest), а о «начале ее движения по конвейеру» (flow). — *Примеч. пер.*

4.5.1. Готовые реализации интерфейса TimestampExtractor

Практически все предоставляемые Kafka Streams реализации интерфейса `TimestampExtractor` работают с метками даты/времени, указываемыми генератором или брокером в метаданных сообщения, что означает или семантику времени события (метку даты/времени устанавливает генератор), или семантику времени добавления в журнал (метку даты/времени устанавливает брокер). Рисунок 4.19 демонстрирует извлечение метки даты/времени из объекта `ConsumerRecord`.

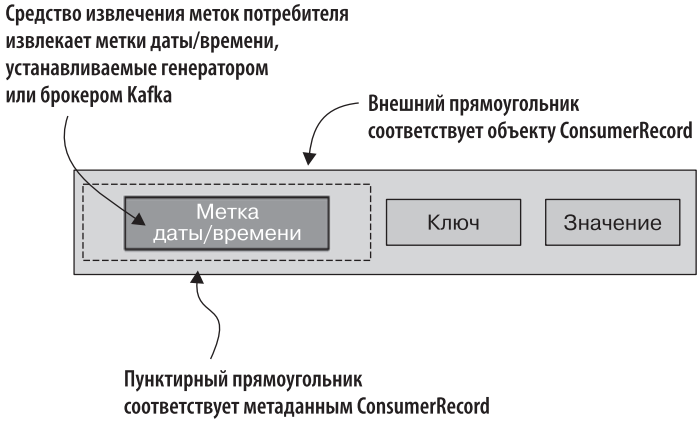


Рис. 4.19. Метки даты/времени в объекте ConsumerRecord: метку устанавливает генератор или брокер в зависимости от настроек

Хотя мы предполагаем, что используется настройка по умолчанию `CreateTime` для меток даты/времени, не забывайте, что при настройке `LogAppendTime` возвращается значение метки даты/времени, соответствующее времени добавления записи в журнал. Базовую функциональность для извлечения метки даты/времени из метаданных объекта `ConsumerRecord` предоставляет абстрактный класс `ExtractRecordMetadataTimestamp`. Большинство конкретных реализаций расширяют именно его. Для обработки некорректных меток даты/времени (случаев, когда метка даты/времени меньше 0) разработчики переопределяют абстрактный метод `ExtractRecordMetadataTimestamp.onInvalidTimestamp`.

Вот список классов, расширяющих класс `ExtractRecordMetadataTimestamp`:

- `FailOnInvalidTimestamp` — генерирует исключение в случае некорректной метки даты/времени;
- `LogAndSkipOnInvalidTimestamp` — возвращает некорректную метку даты/времени и заносит в журнал предупреждение о том, что запись будет отброшена из-за некорректной метки даты/времени;
- `UsePreviousTimeOnInvalidTimestamp` — возвращает последнюю извлеченную корректную метку даты/времени в случае некорректной метки.

Мы рассмотрели в этом разделе средства извлечения меток времени события, но существует одно более продвинутое средство извлечения меток даты/времени, которое я хотел бы обсудить.

4.5.2. Класс WallclockTimestampExtractor

Класс `WallclockTimestampExtractor` предоставляет семантику времени обработки и не извлекает никаких меток даты/времени. Вместо этого он возвращает время в миллисекундах, получаемое с помощью вызова метода `System.currentTimeMillis()`.

Вот и все, что я хотел рассказать о готовых средствах извлечения меток даты/времени. В следующем разделе мы рассмотрим вопрос создания их пользовательского варианта.

4.5.3. Пользовательская реализация интерфейса TimestampExtractor

Для работы с метками даты/времени (или их вычисления) в объекте-значении из `ConsumerRecord` вам понадобится пользовательское средство извлечения меток, реализующее интерфейс `TimestampExtractor`. Рисунок 4.20 демонстрирует применение включенной в объект-значение метки даты/времени по сравнению с меткой даты/времени, устанавливаемой (генератором или брокером) Kafka.

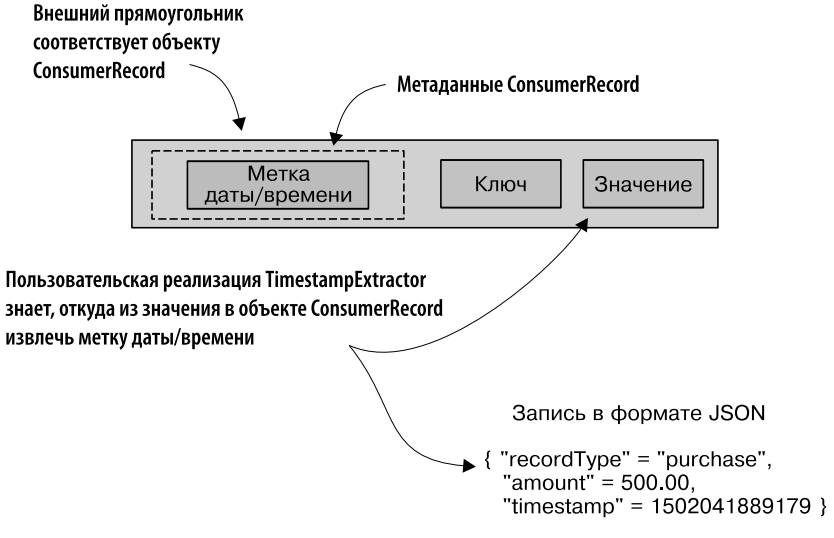


Рис. 4.20. Пользовательский TimestampExtractor возвращает метку даты/времени на основе значения, содержащегося в объекте ConsumerRecord. Эта метка даты/времени может быть уже существующим значением или значением, вычисленным на основе свойств объекта-значения

Ниже приведен пример реализации `TimestampExtractor` (ее можно найти в файле src/main/java/bbejeck/chapter_4/timestamp_extractor/TransactionTimestampExtractor.java) (ли-

стинг 4.14), используемый также в примере соединения (листинг 4.13 из пункта «Реализация соединения»), хотя как параметр конфигурации и не показанный в тексте.

Листинг 4.14. Пользовательский TimestampExtractor

```
public class TransactionTimestampExtractor implements TimestampExtractor {

    @Override
    public long extract(ConsumerRecord<Object, Object> record,
    ➥ long previousTimestamp) {
        Purchase purchasePurchaseTransaction = (Purchase) record.value();
        return purchasePurchaseTransaction.getPurchaseDate().getTime();
    }
}
```

Извлекает объект Purchase из отправленной в Kafka пары «ключ/значение»

Возвращает метку даты/времени, зафиксированную в момент продажи

В примере с соединением мы применили пользовательский `TimestampExtractor` потому, что хотели задействовать метки даты/времени фактического момента покупки. Подобный подход дает возможность соединять записи даже в случае задержек доставки или поступления данных в неправильном порядке.

ПРЕДУПРЕЖДЕНИЕ

Старайтесь не слишком «умничать» при создании пользовательской реализации TimestampExtractor. Сохранение и архивирование журналов основываются на метках даты/времени, так что возвращаемая средством извлечения метка может стать меткой сообщения, которая затем будет применяться в журналах изменений и топиках результатов далее по конвейеру.

4.5.4. Указываем, какой TimestampExtractor использовать

Теперь, когда мы обсудили принципы функционирования средств для извлечения меток даты/времени, осталось указать приложению, какое из них использовать. Существует два способа задания средства извлечения меток.

Первая возможность: задать глобальное средство извлечения меток, указав его в свойствах при настройке приложения Kafka Streams. Если же значение свойства не задано, будет применяться значение по умолчанию — `FailOnInvalidTimestamp.class`. Например, следующий фрагмент кода указывает приложению с помощью задания свойства при его настройке использовать класс `TransactionTimestampExtractor`:

```
props.put(StreamsConfig.DEFAULT_TIMESTAMP_EXTRACTOR_CLASS_CONFIG,
➥ TransactionTimestampExtractor.class);
```

Вторая возможность — передать экземпляр `TimestampExtractor` через объект `Consumed`:

```
Consumed.with(Serdes.String(), purchaseSerde)
        .withTimestampExtractor(new TransactionTimestampExtractor()))
```

Преимущество второго способа состоит в возможности использования своего `TimestampExtractor` для каждого источника входных данных, в то время как при первом варианте приходится обрабатывать записи из различных топиков в одном экземпляре `TimestampExtractor`.

Мы подошли к концу обсуждения вопросов применения меток даты/времени. В следующих главах вам предстоит столкнуться с ситуациями, в которых разница между метками даты/времени будет инициировать какие-либо действия, например сброс на диск кэша `KTable`. Я не жду, что вы запомните все три вида средств извлечения меток даты/времени, но важно понимать, что эти метки даты/времени играют важную роль в функционировании Kafka и Kafka Streams.

Резюме

- Сохранение состояния необходимо для потоковой обработки. Иногда события бывают автономны, но обычно для принятия верных решений требуется дополнительная информация.
- Kafka Streams предоставляет удобные абстракции для преобразований с сохранением состояния, включая соединения.
- Хранилища состояния в Kafka Streams обеспечивают необходимые для потоковой обработки атрибуты состояния: локальность данных и отказоустойчивость.
- Метки даты/времени управляют движением данных в Kafka Streams. Следует тщательно и обдуманно выбирать источники меток даты/времени.

В следующей главе мы продолжим изучать вопросы сохранения состояния в потоковой обработке, перейдя к более важным операциям, например таким, как агрегирование и группировка. Мы изучим также API `KTable`. Если основные задачи API `KStream` связаны с обработкой отдельных записей, то `KTable` представляет собой реализацию журнала изменений, где записи с одинаковыми ключами считаются обновлениями одной записи. Мы также обсудим соединения экземпляров `KStream` и `KTable`. Наконец, мы поговорим об одном из наиболее интересных усовершенствований Kafka Streams — *доступном для запроса состоянии* (queryable state). Доступное для запроса состояние позволяет непосредственно наблюдать состояние потока без необходимости материализации информации с помощью чтения данных из топика во внешнем приложении.

5 API KTable

В этой главе:

- описание связи между потоками данных и таблицами;
- обновление записей и абстракция `KTable`;
- агрегирование, оконные операции и соединение объектов `KStream` и `KTable`;
- глобальные `KTable`;
- доступные для запросов хранилища состояния.

Пока мы рассмотрели API `KStream` и вопросы добавления состояния в приложения Kafka Streams. В этой главе мы обсудим вопрос сохранения состояния более подробно. А попутно я познакомлю вас с новой абстракцией — `KTable`.

При обсуждении API `KStream` речь шла об отдельных событиях или потоках событий. В первоначальном примере ZMart, при покупке Джейн Доу товара, мы рассматривали покупку как отдельное событие. Мы не отслеживали количество совершенных Джейн покупок или их частоту. В контексте базы данных поток событий покупок можно рассматривать как последовательность операций вставки. А поскольку все записи новые и не связаны друг с другом, можно спокойно вставлять их в таблицу.

Теперь пусть у каждого события покупки есть первичный ключ (идентификатор покупателя). Покупки Джейн Доу сейчас представляют собой последовательность *связанных* событий покупки (обновлений). Благодаря использованию первичного ключа происходит обновление каждой из покупок в соответствии с новыми покупками Джейн. Связь между потоками данных и таблицами можно описать следующим образом: поток событий рассматривается как операции вставки, а события с ключами — как обновления.

В этой главе мы обсудим взаимосвязь между потоками данных и таблицами подробнее. Подобная взаимосвязь важна для понимания того, как функционирует `KTable`.

Во-вторых, мы рассмотрим интерфейс `KTable`. API `KTable` необходимо, потому что специально предназначено для работы с обновлениями записей. Эта возможность

нужна нам для таких операций, как агрегирование и подсчеты количеств. Мы вкратце затронули вопрос обновлений в главе 4, при знакомстве с преобразованиями с сохранением состояния; в подразделе 4.2.5 мы обновляли узел-обработчик поощрений, чтобы отслеживать покупки.

В-третьих, мы займемся оконными операциями. Оконная операция — операция, при которой группируются данные за определенный промежуток времени. Например, чтобы узнать, сколько покупок было совершено за последний час при обновлении каждые 10 минут. Оконные операции дают возможность собирать данные порциями, в отличие от сбора неограниченных данных.

ПРИМЕЧАНИЕ

Оконная обработка данных (windowing) и *обработка по корзинам* (bucketing) — в чем-то синонимичные понятия. И то и другое относится к разбиению информации на меньшие порции или категории. Оконная обработка данных подразумевает категоризацию по времени, но результат в обоих случаях одинаков.

Последней темой обсуждения в данной главе будут доступные для запросов хранилища состояния. Доступные для запросов хранилища состояния — замечательная возможность Kafka Streams: они позволяют выполнять запросы непосредственно к хранилищам состояния. Другими словами, дают возможность просматривать данные (с сохранением состояния) без потребления их из топика Kafka или чтения их из базы данных. Итак, начнем с первой из анонсированных тем.

5.1. Взаимосвязь между потоками данных и таблицами

В главе 1 было дано определение потока данных как бесконечной последовательности событий. Это довольно расплывчатая формулировка, так что конкретизируем ее на конкретном примере.

5.1.1. Поток записей

Допустим, мы хотели бы увидеть последовательность обновлений курса акций. Можно переделать «жемчужную» диаграмму из главы 1, чтобы она выглядела так, как показано на рис. 5.1. Как видите, каждая котировка акций представляет собой дискретное событие, они не связаны друг с другом. Даже если за несколько котировок отвечает одна компания, они все равно рассматриваются по отдельности. Такое представление событий соответствует потоку данных событий, описываемому `KStream`.

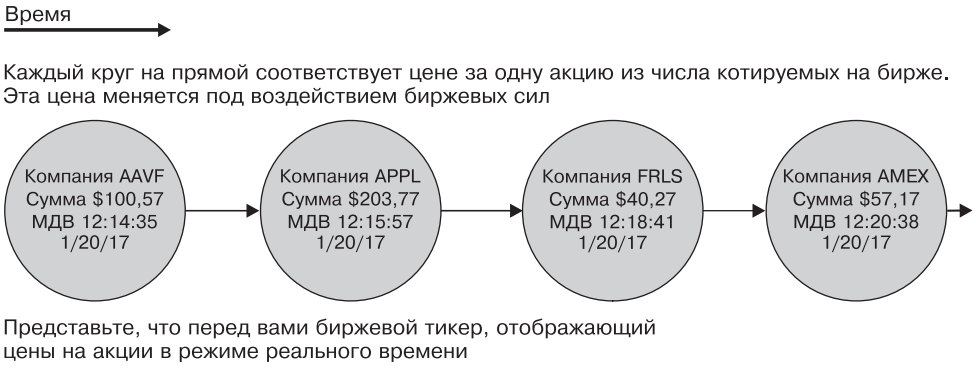


Рис. 5.1. «Жемчужная» диаграмма для неограниченного потока котировок акций

Взглянем теперь на то, как эта концепция соотносится с таблицами базы данных. Рассмотрим простую таблицу биржевых котировок, показанную на рис. 5.2.

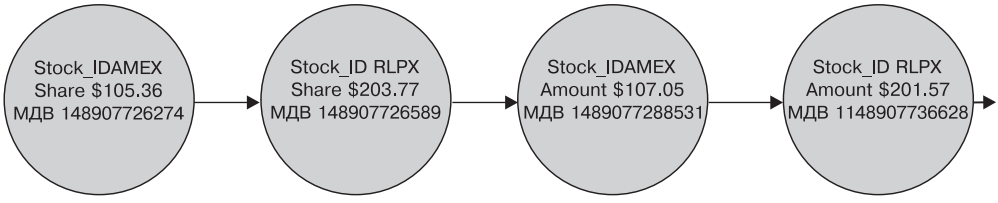
Рис. 5.2. Простая таблица базы данных с курсом акций различных компаний. В ней есть столбец с ключом, а также другие столбцы со значениями. Можно рассматривать ее строки как пары «ключ/значение», если «свалить» все остальные столбцы в контейнер value

ПРИМЕЧАНИЕ

Для упрощения будем считать, что ключ — одиночное значение.

Далее обратимся снова к потоку записей. Поскольку записи автономны, данный поток соответствует операциям вставки в таблицу. На рис. 5.3 эти две концепции объединены в целях иллюстрации.

Самое главное здесь — возможность рассматривать поток событий аналогично вставкам в таблицу, благодаря чему можно лучше разобраться в использовании потоков данных для работы с событиями. Следующий этап — изучение случая, когда события в потоке взаимосвязаны.



Ключ (key)	Stock_ID	Метка даты/времени	Share_Price
1	AMEX	148907726274	105.36
2	RLPX	148907726589	203.77
3	AMEX	1489077288531	107.05
4	RLPX	148907736628	201.57

Этот рисунок демонстрирует взаимосвязь между событиями и операциями вставки в таблицу. Хотя здесь приведены курсы акций только для двух компаний, событий — четыре, поскольку мы рассматриваем каждый элемент в потоке как отдельное событие.

В результате каждое событие представляет собой операцию вставки, при которой значение ключа увеличивается на единицу.

С учетом этого каждое событие представляет собой новую, независимую запись (вставку в таблицу базы данных)

Рис. 5.3. Поток индивидуальных событий по сравнению со вставками в таблицу базы данных. Аналогично можно представить себе построчную потоковую обработку таблицы

5.1.2. Обновления записей (журнал изменений)

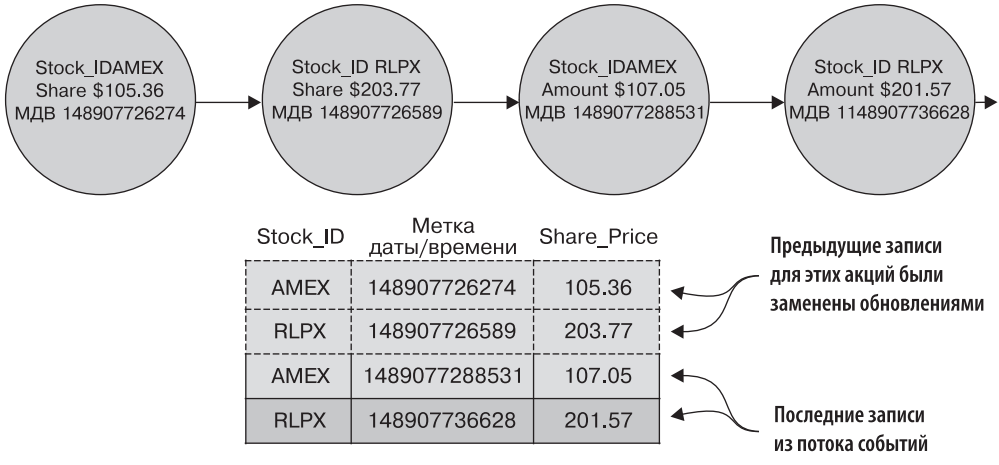
Рассмотрим тот же поток транзакций покупателя, но теперь с учетом их действий в различные моменты времени. Если добавить ключ — идентификатор покупателя, то можно связать события покупки друг с другом и получить поток обновлений вместо потока событий.

Если поток событий мы сравнивали с журналом, то поток обновлений можно сравнить с журналом изменений. Рисунок 5.4 иллюстрирует эту концепцию.

На этом рисунке видна взаимосвязь между потоком обновлений и таблицей базы данных. Как журнал (log), так и журнал изменений (changelog) отражают добавляемые в конец файла входящие записи. В журнале видны все записи, а в журнале изменений — только последняя запись для каждого ключа.

ПРИМЕЧАНИЕ

Как в журнале, так и в журнале изменений записи при поступлении добавляются в конец файла. Различие в том, что журнал используется, когда нужно видеть *все* записи, а журнал изменений — только когда нужно видеть *последнюю* запись для каждого ключа.



При использовании идентификатора акции в качестве первичного ключа в журнале изменений последующие события с тем же ключом рассматриваются как обновления. В данном случае записей будет только две, по одной на компанию. Хотя в будущем могут поступить новые записи, относящиеся к тем же компаниям, они не будут накапливаться

Рис. 5.4. В журнале изменений каждая входящая запись заменяет предыдущую запись с тем же ключом (если таковая есть). В случае потока записей общее число событий было равно четырем, а в случае обновлений (журнала изменений) их только два

Для сокращения журнала с сохранением последних записей для всех ключей можно воспользоваться сжатием журналов, обсуждавшимся в главе 2. Результат сжатия журнала показан на рис. 5.5. Раз нас интересуют только последние значения, то можно удалить более старые пары «ключ/значение».

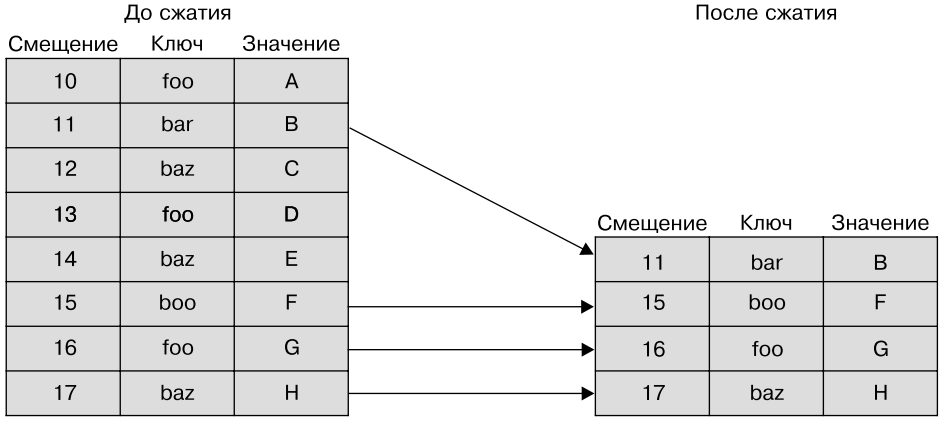


Рис. 5.5. *Слева* показан журнал до сжатия, в котором можно заметить дублирующиеся ключи с разными значениями — обновления. *Справа* показан журнал после сжатия — для каждого ключа остается только последнее значение, и размер журнала за счет этого уменьшился

Вы уже знакомы с потоками событий по работе с `KStream`. Для журналов изменений (потоков обновлений) мы будем использовать абстракцию `KTable`. Следующий этап, после того как мы разобрались во взаимосвязи между потоками и таблицами, — сравнение потока событий с потоком обновлений.

5.1.3. Поток событий по сравнению с потоком обновлений

Для сравнения потока событий с потоком обновлений мы воспользуемся `KStream` и `KTable`. С этой целью мы запустим простое приложение — биржевой тикер, записывающий текущие курсы акций для трех (вымышленных!) компаний. От него мы получим три порции котировок акций, всего девять записей. А `KStream` и `KTable` будут читать эти записи и выводить их в консоль с помощью метода `print()`.

На рис. 5.6 показаны результаты работы данного приложения. Как можно видеть, `KStream` вывел в консоль все девять записей. Именно так и должно было быть, поскольку `KStream` рассматривает каждую запись как отдельную сущность. `KTable` же вывел только три записи, поскольку рассматривает их как обновления предыдущих.

Простой биржевой тикер для трех вымышленных компаний с генератором данных, генерирующим три обновления курсов акций. KStream выводит все полученные записи. KTable выводит лишь последнюю порцию записей — последние обновления для конкретных символов акций

Рис. 5.6. Вывод записей с одинаковыми ключами: KTable по сравнению с KStream

ПРИМЕЧАНИЕ

На рис. 5.6 показано, как KTable работает с обновлениями. При этом мной было сделано неявное допущение, о котором хотелось бы упомянуть. При работе с KTable в записях в парах «ключ/значение» должны быть заполнены ключи. Ключи необходимы для работы KTable, поскольку без ключа невозможно обновить запись в таблице базы данных.

С точки зрения объекта `KTable`, он получил не девять отдельных записей, а три исходных записи и две порции обновлений и вывел в консоль только последнюю их порцию. Обратите внимание, что записи `KTable` совпадают с последними тремя опубликованными `KStream` записями. Мы обсудим механизмы, с помощью которых `KTable` выводит только обновления, в следующем разделе.

В листинге 5.1 приведена программа для вывода в консоль результатов биржевых тикеров (находится в файле src/main/java/bbejeck/chapter_5/KStreamVsKTableExample.java; исходный код можно найти по адресу https://manning.com/books/kafka-streams-in-action).

Листинг 5.1. Вывод в консоль KTable и KStream

```
KTable<String, StockTickerData> stockTickerTable =
➥ builder.table(STOCK_TICKER_TABLE_TOPIC);
KStream<String, StockTickerData> stockTickerStream =
➥ builder.stream(STOCK_TICKER_STREAM_TOPIC);

stockTickerTable.toStream()
➥ .print(Printed.<String, StockTickerData>toSysOut()
➥ .withLabel("Stocks-KTable"));

stockTickerStream
➥ .print(Printed.<String, StockTickerData>toSysOut()
➥ .withLabel("Stocks-KStream"));
```

Создание экземпляра KTable

Создание экземпляра KStream

KTable выводит результаты в консоль

KStream выводит результаты в консоль

Использование объектов Serde по умолчанию

При создании `KTable` и `KStream` мы не задавали никаких объектов `Serde`. А равно и при обоих вызовах метода `print()`. А все благодаря тому, что мы внесли в конфигурацию объекты `Serde` для использования по умолчанию, примерно вот так:

```
props.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
➥ Serdes.String().getClass().getName());
props.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
➥ StreamsSerdes.StockTickerSerde().getClass().getName());
```

Если же вы хотели бы использовать другие типы данных, то для чтения или внесения записей необходимо передать объекты `Serde` через перегруженные методы.

Главный вывод: записи с одинаковыми ключами в потоке данных по своей сути обновления, а не самостоятельные новые записи. Именно понятие потока обновлений лежит в основе интерфейса `KTable`.

Теперь, когда мы увидели интерфейс `KTable` в действии, можно обсудить механизмы, которые лежат в основе его функциональности.

5.2. Обновления записей и настройки KTable

Чтобы разобраться, как работает интерфейс `KTable`, нужно задать себе два вопроса:

- где хранятся записи;
- как `KTable` определяет, когда отправлять записи дальше?

Ответы на эти вопросы жизненно важны для операций агрегирования и свертки. Например, при агрегировании хотелось бы получить обновленные количества, но получать обновление при каждом увеличении количества на единицу — вряд ли то, что вам нужно.

Для ответа на первый вопрос взглянем на строку кода для создания объекта `KTable`:

```
builder.table(STOCK_TICKER_TABLE_TOPIC);
```

С помощью этого простого оператора `StreamsBuilder` создает экземпляр `KTable` и одновременно незаметно создает объект `StateStore` для отслеживания состояния потока, таким образом создавая поток обновлений. У созданного при этом подходе хранилища состояния `StateStore`, недоступного для интерактивных запросов, есть только внутреннее имя.

Существует перегруженная версия метода `StreamsBuilder.table`, принимающая в качестве параметра экземпляр `Materialized`, благодаря чему у вас появляется возможность настроить тип хранилища и задать для него имя, чтобы сделать его доступным для запросов. Мы обсудим интерактивные запросы позднее в этой главе.

Вот и ответ на наш первый вопрос: `KTable` использует для хранения локальное состояние в сочетании с Kafka Streams (мы обсуждали хранилища состояния в разделе 4.3).

Перейдем теперь к следующему вопросу: что определяет, когда `KTable` отправляет обновления узлам-обработчикам далее по конвейеру? Для ответа на этот вопрос нам придется учесть несколько факторов:

- число поступающих в приложение записей. При повышении темпов поступления данных обычно повышается и темп отправки обновленных записей;
- количество уникальных ключей в данных. Опять же, чем больше уникальных ключей, тем больше обновлений отправляется дальше по конвейеру;
- параметры конфигурации `cache.max.bytes.buffering` и `commit.interval.ms`.

Из этого списка мы рассмотрим только то, чем можем управлять: параметры конфигурации. Сначала поговорим о настройке `cache.max.bytes.buffering`.

5.2.1. Задание размера буфера кэша

Кэш объекта `KTable` служит для дедупликации обновлений записей с одним ключом. Благодаря этой дедупликации дочерние узлы получают только самые свежие обновления вместо всех обновлений, что существенно снижает объемы обрабатываемых данных. Кроме того, в хранилище состояния помещаются только самые свежие обновления, что означает весьма существенный рост производительности при использовании постоянных хранилищ состояния.

Рисунок 5.7 иллюстрирует кэширование. Как вы можете видеть, при включенном кэшировании не все обновления записей перенаправляются дальше по конвейеру. В кэше содержится только последняя запись для конкретного ключа.

ПРИМЕЧАНИЕ

Приложение Kafka Streams представляет собой топологию (граф) соединенных между собой узлов (узлов-обработчиков). У каждого узла, если только он не концевой, есть один или несколько дочерних узлов. По завершении работы с записью узел-обработчик отправляет ее «дальше по конвейеру», своим дочерним узлам.

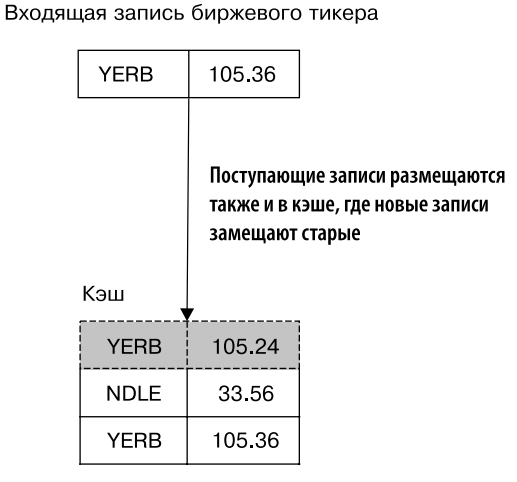


Рис. 5.7. Кэширование KTable удаляет избыточные обновления записей с одинаковым ключом, предотвращая таким образом переполнение дочерних узлов KTable в топологии непрерывными обновлениями

Поскольку `KTable` соответствует журналу изменений в потоке данных, можно ожидать, что в каждый конкретный момент придется иметь дело только с последним изменением. А применение кэша обеспечивает реализацию этого поведения. При необходимости обработать все записи в потоке лучше воспользоваться описанным ранее интерфейсом `KStream`.

Чем больше кэш, тем меньше изменений будет отправляться далее. Кроме того, кэширование снижает объем данных, записываемых на диск постоянными хранилищами (RocksDB), а в случае включенного журналирования — и число записей для любого хранилища, отправляемых в топик журнала изменений.

Размер кэша определяется настройкой `cache.max.bytes.buffering`, которая задает объем памяти, выделяемой под кэш записей. Заданное количество памяти поровну распределяется между потоками выполнения для этого потока данных (число потоков выполнения для потока данных задается параметром `StreamsConfig.NUM_STREAM_THREADS_CONFIG` конфигурации, равным по умолчанию 1).

ПРИМЕЧАНИЕ

Для отключения кэширования необходимо установить параметр cache.max.bytes.buffering в 0. Но в результате этого далее по конвейеру будут отправляться все обновления KTable, то есть поток журнала изменений фактически превратится в поток событий. Кроме того, без кэширования будет больше операций ввода/вывода, ведь постоянным хранилищам придется записывать на диск все обновления, а не только последние.

5.2.2. Задание интервала фиксации

Второй параметр конфигурации — `commit.interval.ms`. Он определяет частоту (в миллисекундах) сохранения состояния узла-обработчика. При сохранении (фиксации) состояния узла-обработчика происходит сброс кэша на диск с отправкой дальше по конвейеру последних обновленных и дедуплицированных записей.

В полном технологическом процессе кэширования (рис. 5.8) можно видеть две основные движущие силы, участвующие в отправке записей дальше по конвейеру. Отправка записей дальше по конвейеру происходит в результате или фиксации, или достижения максимального размера кэша. И наоборот, отключение кэширования приведет к отправке дальше по конвейеру всех записей, включая записи с дублирующимися ключами. Вообще говоря, при использовании `KTable` лучше, чтобы кэширование было включено.

Как видите, необходим компромисс между размером кэша и интервалом фиксации. Большой размер кэша при коротком интервале фиксации приведет к частым обновлениям. А более длинный интервал фиксации может привести к меньшему числу обновлений (в зависимости от настроек памяти), поскольку для освобождения памяти предусмотрен механизм вытеснения кэша. Жестких правил тут не существует — узнать, что лучше подходит для вас, можно только методом проб и ошибок. Рекомендую начать со значений по умолчанию: 30 секунд (интервал фиксации) и 10 Мбайт (размер кэша). Главное — помнить, что частотой отправления из `KTable` обновленных записей можно управлять.

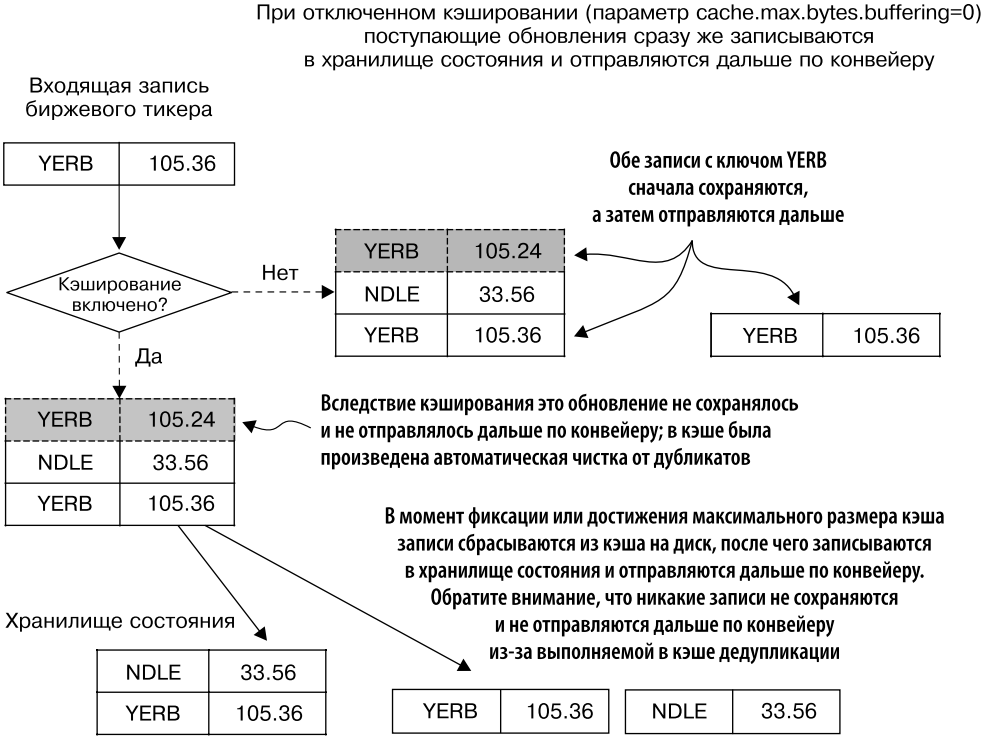


Рис. 5.8. Полный технологический процесс кэширования: если кэширование включено, записи с дублированием удаляются и отправляются дальше по конвейеру при сбросе кэша на диск или фиксации

Теперь посмотрим, как использовать `KTable` в своем приложении.

5.3. Агрегирование и оконные операции

В этом разделе мы перейдем к изучению наиболее многообещающих частей Kafka Streams. Пока мы рассмотрели следующие аспекты Kafka Streams:

- создание топологии обработки;
- использование состояния в потоковых приложениях;
- выполнение соединений потоков данных;
- различия между потоками событий (`KStream`) и потоками обновлений (`KTable`).

В следующих же примерах мы соберем все эти элементы воедино. Кроме того, вы познакомитесь с оконными операциями — еще одной замечательной возможностью потоковых приложений. Первым нашим примером будет простое агрегирование.

5.3.1. Агрегирование объема продаж акций по отраслям промышленности

Агрегирование и группировка — жизненно необходимые инструменты при работе с потоковыми данными. Исследования отдельных записей по мере поступления часто оказывается недостаточно. Для извлечения из данных дополнительной информации необходимы их группировка и комбинирование.

В этом примере вам предстоит примерить костюм внутридневного трейдера, которому нужно отслеживать объемы продаж акций компаний в нескольких отраслях промышленности. В частности, вас интересуют пять компаний с наибольшими объемами продаж акций в каждой из отраслей промышленности.

Для подобного агрегирования потребуется несколько следующих шагов по переводу данных в нужный вид (если говорить в общих чертах).

1. *Создать источник на основе топика, публикующий необработанную информацию по торговле акциями*. Нам придется отобразить объект типа `StockTransaction` в объект типа `ShareVolume`. Дело в том, что объект `StockTransaction` содержит метаданные продаж, а нам нужны только данные о количестве продаваемых акций.
2. *Сгруппировать данные* `ShareVolume` *по символам акций*. После группировки по символам можно свернуть эти данные до промежуточных сумм объемов продаж акций. Стоит отметить, что метод `KStream.groupBy` возвращает экземпляр типа `KGroupedStream`. А получить экземпляр `KTable` можно, вызвав далее метод `KGroupedStream.reduce`.

Что такое интерфейс KGroupedStream

Методы `KStream.groupBy` и `KStream.groupByKey` возвращают экземпляр `KGroupedStream`. `KGroupedStream` является промежуточным представлением потока событий после группировки по ключам. Он вовсе не предназначен для непосредственной работы с ним. Вместо этого `KGroupedStream` используется для операций агрегирования, результатом которых всегда является `KTable`. А поскольку результатом операций агрегирования является `KTable` и в них применяется хранилище состояния, то, возможно, не все обновления в результате отправляются дальше по конвейеру.

Метод `KTable.groupBy` возвращает аналогичный `KGroupedTable` — промежуточное представление потока обновлений, перегруппированных по ключу.

Сделаем небольшой перерыв и посмотрим на рис. 5.9, на котором показано, чего мы добились. Эта топология должна быть вам уже хорошо знакома.

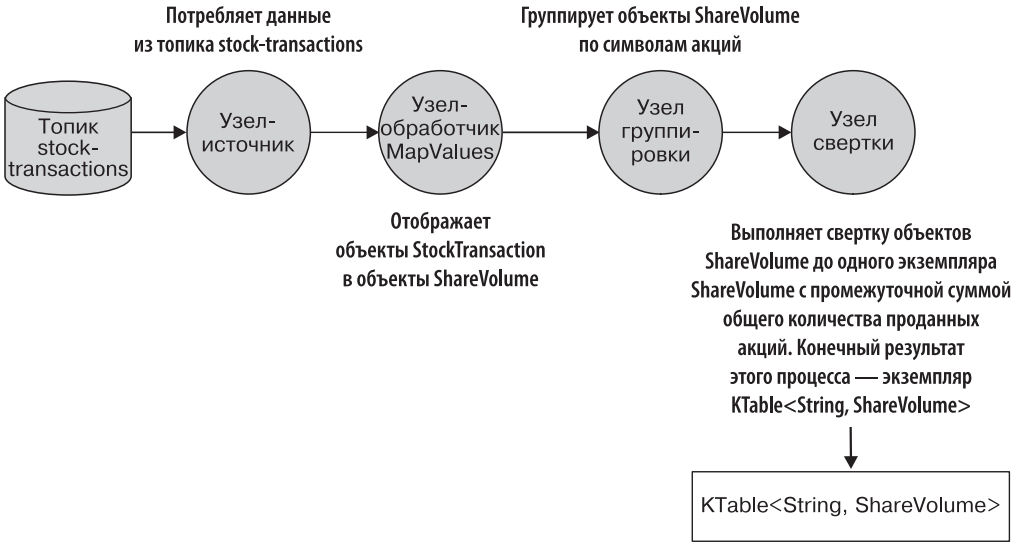


Рис. 5.9. Отображение и свертка объектов StockTransaction в объекты ShareVolume с последующей сверткой их до промежуточных сумм

Взглянем теперь на код для этой топологии (его можно найти в файле src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java) (листинг 5.2).

Листинг 5.2. Источник для преобразования-свертки биржевых транзакций

```
KTable<String, ShareVolume> shareVolume =
➥ builder.stream(STOCK_TRANSACTIONS_TOPIC,
                  Consumed.with(stringSerde, stockTransactionSerde)
➥ .withOffsetResetPolicy(EARLIEST))
➥ .mapValues(st -> ShareVolume.newBuilder(st).build())
➥ .groupBy((k, v) -> v.getSymbol(),
                    Serialized.with(stringSerde, shareVolumeSerde))
➥ .reduce(ShareVolume::reduce);
```

Узел-источник потребляет данные из топика

Отображает объекты StockTransaction в объекты ShareVolume

Выполняет свертку объектов ShareVolume до промежуточного количества проданных акций

Группирует объекты ShareVolume по символам акций

Приведенный код отличается краткостью и большим объемом производимых в нескольких строках действий. В первом параметре метода `builder.stream` вы можете заметить нечто новое для себя: значение перечисляемого типа `AutoOffsetReset.EARLIEST` (существует также и `LATEST`), задаваемое с помощью метода `Consumed.withOffsetResetPolicy`. С помощью этого перечисляемого типа можно указать стратегию сброса смещений для каждого из `KStream` или `KTable`, он обладает приоритетом над параметром сброса смещений из конфигурации.

GroupByKey и GroupBy

В интерфейсе `KStream` есть два метода для группировки записей: `GroupByKey` и `GroupBy`. Оба возвращают `KGroupedTable`, так что у вас может появиться закономерный вопрос: в чем же различие между ними и когда использовать какой из них?

Метод `GroupByKey` применяется, когда ключи в `KStream` уже непустые. А главное, флаг «требует повторного секционирования» никогда не устанавливался.

Метод `GroupBy` предполагает, что вы меняли ключи для группировки, так что флаг повторного секционирования установлен в `true`. Выполнение после метода `GroupBy` соединений, агрегирования и т. п. приведет к автоматическому повторному секционированию.

Резюме: следует при малейшей возможности использовать `GroupByKey`, а не `GroupBy`.

Что делают методы `mapValues` и `groupBy` — понятно, так что взглянем на метод `sum()` (его можно найти в файле src/main/java/bbejeck/model/ShareVolume.java) (листинг 5.3).

Листинг 5.3. Метод ShareVolume.sum

```
public static ShareVolume sum(ShareVolume csv1, ShareVolume csv2) {
      Builder builder = new Builder(csv1);
      builder.shares = csv1.shares + csv2.shares;
      return builder.build();
}
```

Используем конструктор копирования класса Builder

Устанавливаем количество акций равным сумме количеств акций из обоих объектов ShareVolume

Вызываем метод build() и возвращаем новый объект ShareVolume

ПРИМЕЧАНИЕ

Мы уже встречали паттерн проектирования «Строитель» ранее в этой книге, но здесь он используется несколько в другом контексте. В данном примере «Строитель» применяется для создания копии объекта и обновления поля без модификации исходного объекта.

Метод `ShareVolume.sum` возвращает промежуточную сумму объема продаж акций, а результат всей цепочки вычислений представляет собой объект `KTable<String, ShareVolume>`. Теперь вы понимаете, какую роль играет `KTable`. При поступлении объектов `ShareVolume` в соответствующем объекте `KTable` сохраняется последнее актуальное обновление. Важно не забывать, что все обновления отражаются в предшествующем `shareVolumeKTable`, но не все отправляются далее.

ПРИМЕЧАНИЕ

Зачем выполнять свертку вместо агрегирования? Хотя свертка — одна из разновидностей агрегирования, при ней вы получаете объект того же типа. Хотя операция агрегирования также суммирует значения, но может возвращать объект другого типа.

Далее с помощью этого `KTable` мы выполняем агрегирование (по количеству продаваемых акций), чтобы получить пять компаний с наибольшими объемами продаж акций в каждой из отраслей промышленности. Наши действия при этом будут аналогичны действиям при первом агрегировании.

1. Выполнить еще одну операцию `groupBy` для группировки отдельных объектов `ShareVolume` по отраслям промышленности.
2. Приступить к суммированию объектов `ShareVolume`. На этот раз объект агрегирования представляет собой очередь по приоритету фиксированного размера. В такой очереди фиксированного размера сохраняются только пять компаний с наибольшими количествами проданных акций.
3. Отобразить очереди из предыдущего пункта в строковое значение и вернуть пять наиболее продаваемых по количеству акций по отраслям промышленности.
4. Записать результаты в строковом виде в топик.

На рис. 5.10 показан граф топологии движения данных. Как вы видите, второй круг обработки достаточно прост.

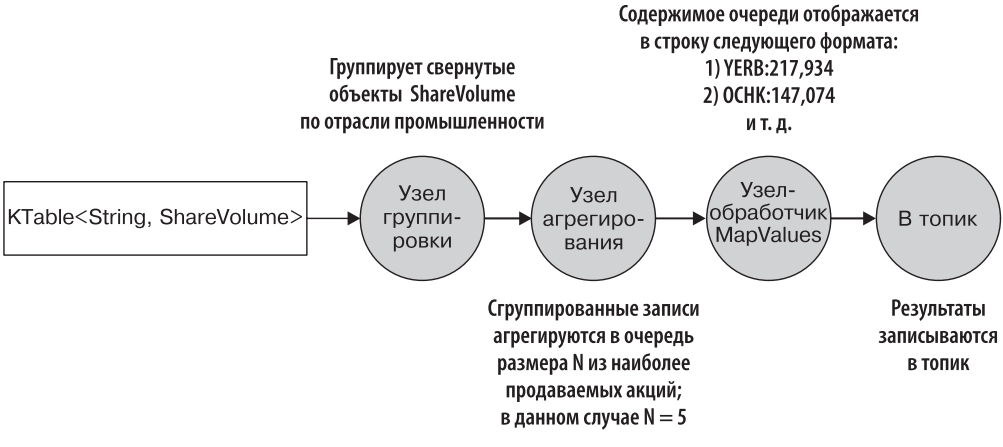
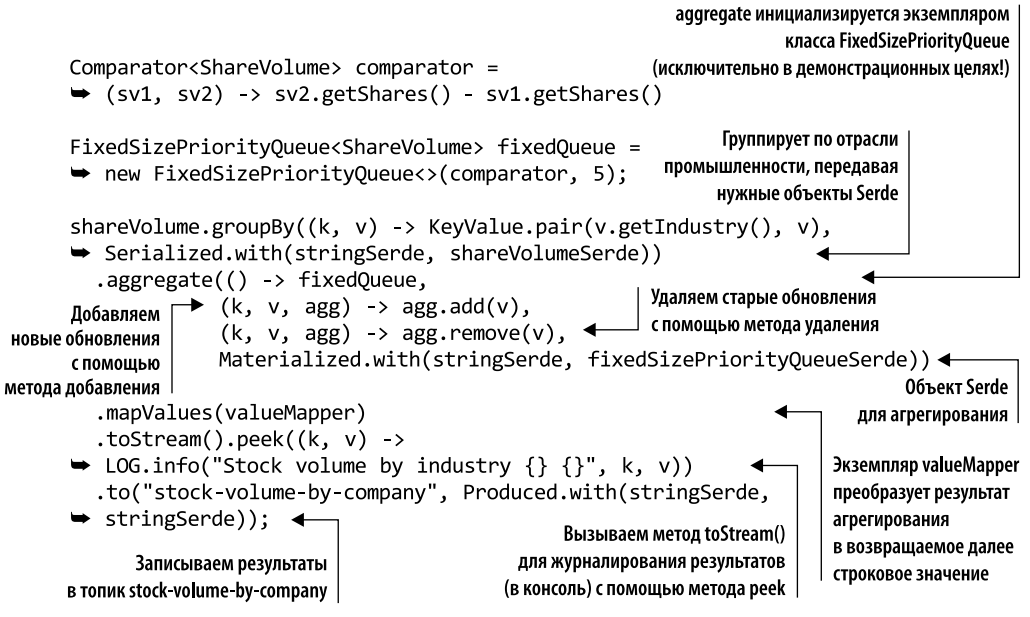


Рис. 5.10. Топология для группировки по отрасли промышленности, агрегирования для получения пяти самых продаваемых, отображения очереди из этих пяти в строковое значение и записи данной строки в топик

Теперь, четко уяснив себе структуру этого второго круга обработки, можно обратиться к его исходному коду (вы найдете его в файле src/main/java/bbejeck/chapter_5/AggregationsAndReducingExample.java) (листинг 5.4).

В данном инициализаторе есть переменная `fixedQueue`. Это пользовательский объект — адаптер для `java.util.TreeSet`, который применяется для отслеживания *N* наибольших результатов в порядке убывания количества проданных акций.

Листинг 5.4. KTable: группировка и агрегирование




```
Comparator<ShareVolume> comparator =
➥ (sv1, sv2) -> sv2.getShares() - sv1.getShares()

FixedSizePriorityQueue<ShareVolume> fixedQueue =
➥ new FixedSizePriorityQueue<>(comparator, 5);

shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v),
➥ Serialized.with(stringSerde, shareVolumeSerde))
  .aggregate(() -> fixedQueue,
             (k, v, agg) -> agg.add(v),
             (k, v, agg) -> agg.remove(v),
             Materialized.with(stringSerde, fixedSizePriorityQueueSerde))
  .mapValues(valueMapper)
  .toStream().peek((k, v) ->
➥ LOG.info("Stock volume by industry {} {}", k, v))
  .to("stock-volume-by-company", Produced.with(stringSerde,
➥ stringSerde));
```

Вы уже встречались с вызовами `groupBy` и `mapValues`, так что не будем на них останавливаться (мы вызываем метод `KTable.toStream`, поскольку метод `KTable.print` считается устаревшим). Но вы пока еще не видели `KTable`-версию метода `aggregate()`, так что мы потратим немного времени на его обсуждение.

Как вы помните, `KTable` отличает то, что записи с одинаковыми ключами считаются обновлениями. `KTable` заменяет старую запись новой. Агрегирование происходит подобным же образом: агрегируются последние записи с одним ключом. При поступлении записи она добавляется в экземпляр класса `FixedSizePriorityQueue` с помощью сумматора (второй параметр в вызове метода `aggregate`), но если уже существует другая запись с тем же ключом, то старая запись удаляется с помощью вычитателя (третий параметр в вызове метода `aggregate`).

Это все значит, что наш агрегатор, `FixedSizePriorityQueue`, вовсе не агрегирует *все* значения с одним ключом, а хранит скользящую сумму количеств N наиболее продаваемых видов акций. В каждой поступающей записи содержится общее количество проданных до сих пор акций. `KTable` даст вам информацию о том, акций каких компаний продается больше всего в настоящий момент, скользящее агрегирование каждого из обновлений не требуется.

Мы научились делать две важные вещи:

- группировать значения в `KTable` по общему для них ключу;
- выполнять над этими сгруппированными значениями такие полезные операции, как свертка и агрегирование.

Умение выполнять эти операции важно для понимания смысла данных, движущихся через приложение Kafka Streams, и выяснения того, какую информацию они несут.

Мы также соединили воедино некоторые из ключевых понятий, обсуждавшихся ранее в этой книге. В главе 4 мы рассказывали, насколько важно для потокового приложения отказоустойчивое, локальное состояние. Первый пример из этой главы продемонстрировал, почему настолько важно локальное состояние — оно дает возможность отслеживать, какую информацию вы уже видели. *Локальный* доступ позволяет избежать сетевых задержек, благодаря чему приложение становится более производительным и устойчивым к ошибкам.

При выполнении любой операции свертки или агрегирования необходимо указать название хранилища состояния. Операции свертки и агрегирования возвращают экземпляр `KTable`, а `KTable` использует хранилище состояния для замены старых результатов новыми. Как вы видели, далеко не все обновления отправляются далее по конвейеру, и это важно, поскольку операции агрегирования предназначены для получения итоговой информации. Если не применять локальное состояние, `KTable` будет отправлять дальше *все* результаты агрегирования и свертки.

Далее мы посмотрим на выполнение таких операций, как агрегирование, в пределах конкретного промежутка времени — так называемых *оконных операций* (windowing operations).

5.3.2. Оконные операции

В предыдущем разделе мы познакомились со «скользящими» сверткой и агрегированием. Приложение производило непрерывную свертку объема продаж акций с последующим агрегированием пяти наиболее продаваемых на бирже акций.

Иногда подобные непрерывные агрегирование и свертка результатов необходимы. А иногда нужно выполнить операции только над заданным промежутком времени. Например, вычислить, сколько было произведено биржевых операций с акциями конкретной компании за последние 10 минут. Или сколько пользователей нажало на новый рекламный баннер за последние 15 минут. Приложение может производить такие операции многократно, но с результатами, относящимися только к заданным промежуткам времени (временным окнам).

Подсчет биржевых транзакций по покупателю

В следующем примере мы займемся отслеживанием биржевых транзакций по нескольким трейдерам — либо крупным организациям, либо смышленым финансистам-одиночкам.

Существует две возможные причины для подобного отслеживания. Одна из них — необходимость знать, что покупают/продают лидеры рынка. Если эти крупные игроки и искушенные инвесторы видят для себя открывающиеся возможности,

имеет смысл следовать их стратегии. Вторая причина заключается в желании заметить любые возможные признаки незаконных сделок с использованием внутренней информации. Для этого вам понадобится проанализировать корреляцию крупных всплесков продаж с важными пресс-релизами.

Такое отслеживание состоит из таких этапов, как:

- создание потока для чтения из топика `stock-transactions`;
- группировка входящих записей по идентификатору покупателя и биржевому символу акции. Вызов метода `groupBy` возвращает экземпляр класса `KGroupedStream`;
- возвращение методом `KGroupedStream.windowedBy` потока данных, ограниченного временным окном, что позволяет выполнять оконное агрегирование. В зависимости от типа окна возвращается либо `TimeWindowedKStream`, либо `SessionWindowedKStream`;
- подсчет транзакций для операции агрегирования. Оконный поток данных определяет, учитывается ли при этом подсчете конкретная запись;
- запись результатов в топик или вывод их в консоль во время разработки.

Топология данного приложения проста, но наглядная ее картинка не помешает. Взглянем на рис. 5.11.

Далее мы рассмотрим функциональность оконных операций и соответствующий код.

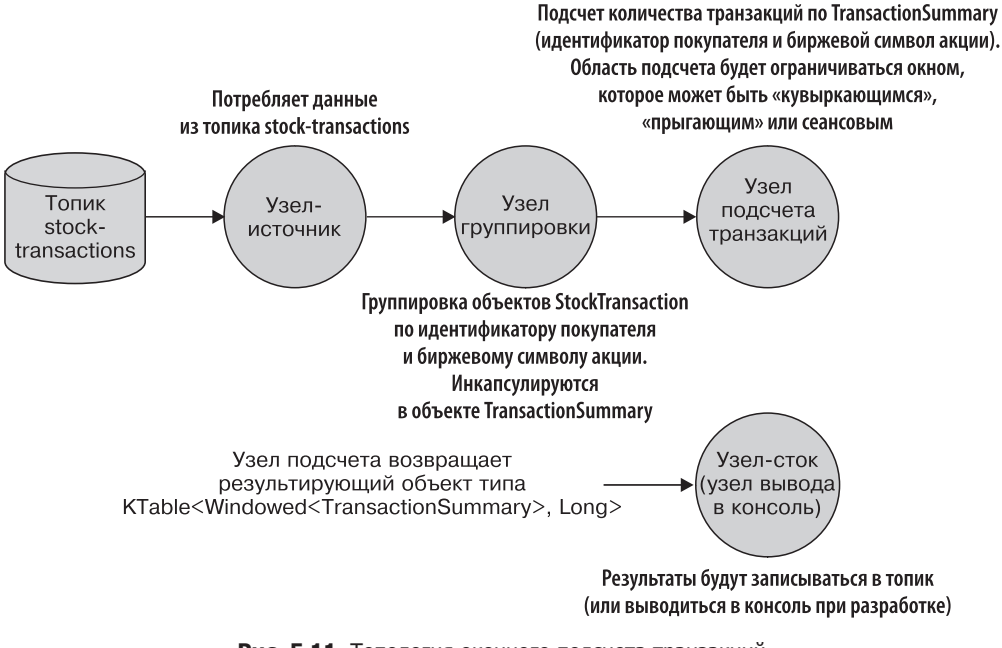


Рис. 5.11. Топология оконного подсчета транзакций

Типы окон

В Kafka Streams существует три типа окон:

- сеансовые;
- «кувыркающиеся» (tumbling);
- скользящие/«прыгающие» (sliding/hopping).

Какое выбрать — зависит от бизнес-требований. «Кувыркающиеся» и «прыгающие» окна ограничиваются по времени, в то время как ограничения сеансовых связаны с действиями пользователей — длительность сеанса (-ов) определяется исключительно тем, насколько активно ведет себя пользователь. Главное — не забывать, что все типы окон основываются на метках даты/времени записей, а не на системном времени.

Далее мы реализуем нашу топологию с каждым из типов окон. Полный код будет приведен только в первом примере, для других типов окон ничего не изменится, кроме типа оконной операции.

Сеансовые окна

Сеансовые окна сильно отличаются от всех остальных типов окон. Они ограничиваются не столько по времени, сколько активностью пользователя (или активностью той сущности, которую вы хотели бы отслеживать). Сеансовые окна разграничиваются периодами бездействия.

Рисунок 5.12 иллюстрирует понятие сеансовых окон. Меньший сеанс будет сливаться с сеансом слева от него. А сеанс справа будет отдельным, поскольку следует за длительным периодом бездействия. Сеансовые окна основываются на действиях пользователей, но применяют метки даты/времени из записей для определения того, к какому сеансу относится запись.

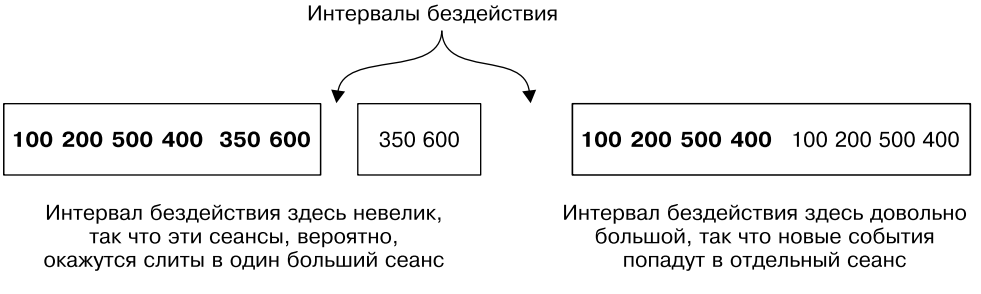


Рис. 5.12. Сеансовые окна, разделяемые коротким периодом бездействия, объединяются в новый, более крупный сеанс

Использование сеансовых окон для отслеживания биржевых транзакций

Воспользуемся сеансовыми окнами для захвата информации о биржевых транзакциях. Реализация сеансовых окон показана в листинге 5.5 (который можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKTableJoinExample.java).

Листинг 5.5. Отслеживание биржевых транзакций с помощью сеансовых окон

```
Serde<String> stringSerde = Serdes.String();
Serde<StockTransaction> transactionSerde =
➥ StreamsSerdes.StockTransactionSerde();

Serde<TransactionSummary> transactionKeySerde =
➥ StreamsSerdes.TransactionSummarySerde();

long twentySeconds = 1000 * 20;
long fifteenMinutes = 1000 * 60 * 15;
KTable<Windowed<TransactionSummary>, Long>
➥ customerTransactionCounts =
➥ builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
➥ transactionSerde)
.withOffsetResetPolicy(LATEST))
.groupBy((noKey, transaction) ->
➥ TransactionSummary.from(transaction),
➥ Serialized.with(transactionKeySerde, transactionSerde))
.windowedBy(SessionWindows.with(twentySeconds).
➥ until(fifteenMinutes)).count();


customerTransactionCounts.toStream()
➥ .print(Printed.<Windowed<TransactionSummary>, Long>toSysOut()
➥ .withLabel("Customer Transactions Counts"));
```

Создает поток данных из строковой константы STOCK_TRANSACTIONS_TOPIC. Для этого потока используется значение LATEST перечисляемого типа для стратегии сброса смещений

Объект KTable, получающийся в результате вызовов методов groupBy и count

Группирует записи по идентификатору покупателя и символу акции, хранящимся в объекте TransactionSummary

Разбиваем группы по сеансовым окнам с интервалом бездействия 20 секунд и интервалом сохранения 15 минут, после чего выполняем агрегирование с помощью метода count

Преобразуем результаты в виде KTable в KStream и выводим полученное в консоль

Большинство операций этой топологии вы уже встречали, так что нет нужды рассматривать их тут снова. Но есть здесь и несколько новых элементов, которые мы сейчас обсудим.

При всякой операции `groupBy` обычно выполняется какая-либо операция агрегирования (агрегирование, свертка или подсчет количества). Можно выполнить или накопительное агрегирование с нарастающим итогом, или оконное агрегирование, при котором учитываются записи в пределах заданного временного окна.

Код из листинга 5.5 выполняет подсчет количества транзакций в пределах сеансовых окон. На рис. 5.13 эти действия анализируются пошагово.

С помощью вызова `windowedBy(SessionWindows.with(twentySeconds).until(fifteenMinutes))` мы создаем сеансовое окно с интервалом бездействия 20 секунд и интервалом сохранения 15 минут. Интервал бездействия 20 секунд означает, что приложение будет включать любую запись, которая поступит в пределах 20 секунд от окончания или начала текущего сеанса в текущий (активный) сеанс.

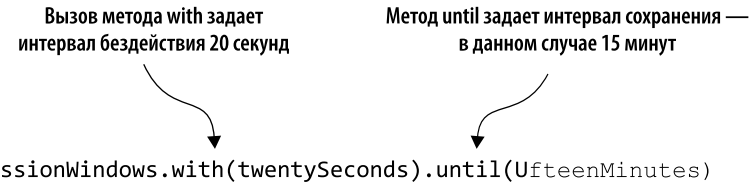


Рис. 5.13. Создание сеансовых окон с заданием интервалов бездействия и сохранения

Далее мы указываем, какую операцию агрегирования нужно выполнить в сеансовом окне — в данном случае `count`. Если входящая запись выходит за пределы интервала бездействия (с любой из сторон от метки даты/времени), то приложение создает новый сеанс. Интервал сохранения означает поддержание сеанса в течение определенного времени и допускает запоздавшие данные, которые выходят за период бездействия сеанса, но все еще могут быть присоединены. Кроме того, начало и конец нового сеанса, получившегося в результате объединения, соответствуют самой ранней и самой поздней метке даты/времени.

Рассмотрим несколько записей из метода `count`, чтобы увидеть, как работают сеансы (табл. 5.1).

Таблица 5.1. Таблица сеансов с интервалом бездействия 20 секунд

Порядок поступления	Ключ	Метка даты/времени
1	{123-345-654,FFBE}	00:00:00
2	{123-345-654,FFBE}	00:00:15
3	{123-345-654,FFBE}	00:00:50
4	{123-345-654,FFBE}	00:00:05

При поступлении записей мы ищем уже существующие сеансы с тем же ключом, временем окончания меньше чем `текущая метка даты/времени – интервал бездействия` и временем начала больше чем `текущая метка даты/времени + интервал бездействия`. С учетом этого четыре записи из табл. 5.1 сливаются в единый сеанс следующим образом.

1. Первой поступает запись 1, так что время начала равно времени окончания и равно 00:00:00.
2. Далее поступает запись 2, и мы ищем сеансы, заканчивающиеся не раньше 23:59:55 и начинающиеся не позднее 00:00:35. Находим запись 1 и объединяем сеансы 1 и 2. Берем время начала сеанса 1 (более раннее) и время окончания сеанса 2 (более позднее), так что наш новый сеанс начинается в 00:00:00 и заканчивается в 00:00:15.
3. Поступает запись 3, мы ищем сеансы между 00:00:30 и 00:01:10 и не находим ни одного. Добавляем второй сеанс для ключа `123-345-654,FFBE`, начинающийся и заканчивающийся в 00:00:50.

4. Поступает запись 4, и мы ищем сеансы между 23:59:45 и 00:00:25. На этот раз находятся оба сеанса — 1 и 2. Все три сеанса объединяются в один, с временем начала 00:00:00 и временем окончания 00:00:15.

Из рассказанного в этом разделе стоит запомнить следующие важные нюансы:

- сеансы — не окна фиксированного размера. Длительность сеанса определяется активностью в рамках заданного промежутка времени;
- метки даты/времени в данных определяют, попадает событие в существующий сеанс или в промежуток бездействия.

Далее мы обсудим следующую разновидность окон — «кувыркающиеся» окна.

«Кувыркающиеся» окна

«Кувыркающиеся» (tumbling) окна захватывают события, попадающие в определенный промежуток времени. Представьте себе, что вам нужно захватывать все биржевые транзакции какой-то компании каждые 20 секунд, так что вы собираете все события за этот промежуток времени. По окончании 20-секундного интервала окно «кувыркается» и переходит на новый 20-секундный интервал наблюдения. Рисунок 5.14 иллюстрирует эту ситуацию.

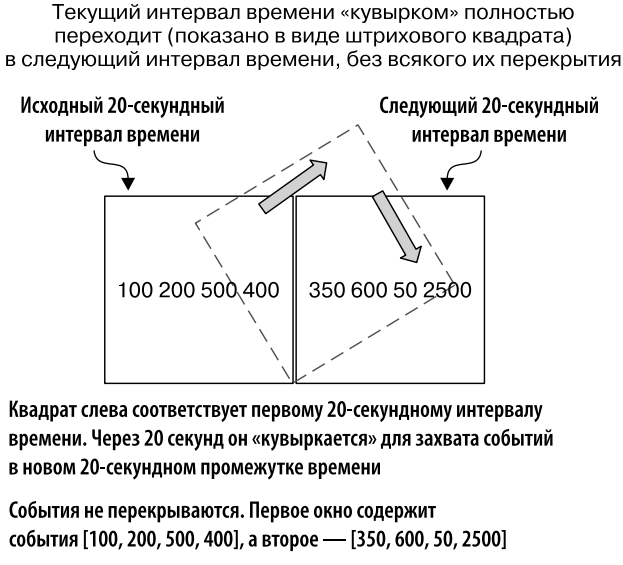


Рис. 5.14. Смещение «кувыркающегося» окна через фиксированный промежуток времени

Как вы можете видеть, все поступившие за последние 20 секунд события включены в окно. По окончании этого промежутка времени создается новое окно.

В листинге 5.6 приведен код, демонстрирующий использование «кувыркающихся» окон для захвата каждые 20 секунд биржевых транзакций (его можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Листинг 5.6. Использование «кувыркающихся» окон для подсчета транзакций

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
➥ builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
                                                          transactionSerde)
➥ .withOffsetResetPolicy(LATEST))
    .groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
➥ Serialized.with(transactionKeySerde, transactionSerde))
    .windowedBy(TimeWindows.of(twentySeconds)).count();
```

Задает длительность отображения «кувыркающегося» окна равной 20 секундам

Благодаря этому небольшому изменению вызова метода `TimeWindows.of` можно использовать «кувыркающееся» окно. В данном примере нет вызова метода `until()`, вследствие чего будет использоваться интервал сохранения по умолчанию, равный 24 часам.

Наконец, пора перейти к последнему из вариантов окон — «прыгающим» (hopping) окнам.

Скользящие («прыгающие») окна

Скользящие/*«прыгающие»* (sliding/hopping) окна похожи на «кувыркающиеся», но с небольшим отличием. Скользящие окна не ждут окончания интервала времени перед созданием нового окна для обработки недавних событий. Они запускают новые вычисления после интервала ожидания, меньшего чем длительность окна.

Для иллюстрации различий «кувыркающихся» и «прыгающих» окон вернемся к примеру с подсчетом биржевых транзакций. Наша цель по-прежнему состоит в подсчете числа транзакций, но нам не хотелось бы ждать весь промежуток времени перед обновлением счетчика. Вместо этого мы будем обновлять счетчик через *более короткие* промежутки времени. Например, подсчитывать число транзакций мы будем по-прежнему каждые 20 секунд, но обновлять счетчик — каждые 5 секунд, как показано на рис. 5.15. При этом у нас оказывается три окна результатов с перекрывающимися данными.

Квадрат слева — первое 20-секундное окно, которое каждые 5 секунд скользит вправо (обновляется), создавая новое окно. На рисунке видно, что события пересекаются. Окно 1 содержит [100, 200, 500, 400], окно 2 содержит [500, 400, 350, 600], а окно 3 — [350, 600, 50, 2500]

Рис. 5.15. Скользящие окна обновляются чаще и могут содержать пересекающиеся данные

В листинге 5.7 приведен код для задания скользящих окон (его можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java).

Листинг 5.7. Использование «прыгающих» окон для подсчета транзакций

```
KTable<Windowed<TransactionSummary>, Long> customerTransactionCounts =
➥ builder.stream(STOCK_TRANSACTIONS_TOPIC, Consumed.with(stringSerde,
➥ transactionSerde)
➥ .withOffsetResetPolicy(LATEST))
.groupBy((noKey, transaction) -> TransactionSummary.from(transaction),
➥ Serialized.with(transactionKeySerde, transactionSerde))
.windowedBy(TimeWindows.of(twentySeconds)
➥ .advanceBy(fiveSeconds).until(fifteenMinutes)).count(); ← Применяем 20-секундное «прыгающее» окно, сдвигающееся вперед каждые 5 секунд
```

«Кувыркающееся» окно можно преобразовать в «прыгающее» с помощью добавления вызова метода `advanceBy()`. В приведенном примере интервал сохранения равен 15 минутам.

ПРИМЕЧАНИЕ

Как вы, наверное, заметили, единственное отличие в коде во всех приведенных примерах оконных операций заключается в вызове метода windowedBy. Вместо включения в код примеров четырех практически идентичных классов я вставил четыре различные строки для оконных операций в файл src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java. Чтобы посмотреть на ту или иную оконную операцию в действии, закомментируйте текущую оконную операцию и раскомментируйте ту, которую хотите выполнить.

Вы увидели в этом разделе, как ограничивать результаты агрегирования временными окнами. В частности, хотелось бы, чтобы вы запомнили из этого раздела следующие три вещи:

- размер сеансовых окон ограничивается не промежутком времени, а активностью пользователей;
- «кувыркающиеся» окна дают представление о событиях в рамках заданного периода времени;
- длительность работы «прыгающих» окон фиксирована, но они часто обновляются и могут содержать во всех окнах пересекающиеся записи.

Далее мы узнаем, как преобразовать `KTable` обратно в `KStream` для соединения.

5.3.3. Соединение объектов KStream и KTable

В главе 4 мы обсуждали соединение двух объектов `KStream`. Теперь нам предстоит научиться соединять `KTable` и `KStream`. Понадобиться это может по следующей простой причине. `KStream` — поток записей, а `KTable` — поток обновлений записей, но

иногда может быть нужно добавить дополнительный контекст к потоку записей с помощью обновлений из `KTable`.

Возьмем данные о количестве биржевых транзакций и соединим их с биржевыми новостями по соответствующим отраслям промышленности. Вот что нужно сделать, что добиться этого с учетом уже имеющегося кода.

1. Преобразовать объект `KTable` с данными о количестве биржевых транзакций в `KStream` с последующей заменой ключа на ключ, обозначающий отрасль промышленности, соответствующую данному символу акций.
2. Создать объект `KTable`, читающий данные из топика с биржевыми новостями. Этот новый `KTable` будет категоризован по отраслям промышленности.
3. Соединить обновления новостей с информацией о количестве биржевых транзакций по отраслям промышленности.

Теперь посмотрим, как реализовать этот план действий.

Преобразование KTable в KStream

Для преобразования `KTable` в `KStream` необходимо сделать следующее.

1. Вызвать метод `KTable.toStream()`.
2. С помощью вызова метода `KStream.map` заменить ключ названием отрасли промышленности, после чего извлечь из экземпляра `Windowed` объект `TransactionSummary`.

Мы свяжем эти операции цепочкой следующим образом (код можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java) (листинг 5.8).

Листинг 5.8. Преобразование KTable в KStream

```
KStream<String, TransactionSummary> countStream =
➥ customerTransactionCounts.toStream().map((window, count) -> {
    TransactionSummary transactionSummary = window.key();
    String newKey = transactionSummary.getIndustry();
    transactionSummary.setSummaryCount(count);
    return KeyValue.pair(newKey, transactionSummary);
  });
```

Извлекаем объект TransactionSummary из экземпляра Windowed

Вызываем метод toStream, а сразу же за ним — map

Задаем новый ключ по отрасли промышленности, где продаются акции

Возвращаем новую пару «ключ/значение» для KStream

Получаем из агрегатора количество транзакций и помещаем это значение в объект TransactionSummary

Поскольку мы выполняем операцию `KStream.map`, то повторное секционирование для возвращаемого экземпляра `KStream` производится автоматически при его использовании в соединении.

Мы завершили процесс преобразования, далее нам нужно создать объект `KTable` для чтения биржевых новостей.

Создание KTable для биржевых новостей

К счастью, для создания объекта `KTable` достаточно одной строки кода (этот код можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java) (листинг 5.9).

Листинг 5.9. Объект KTable для биржевых новостей

```
KTable<String, String> financialNews =
➥  builder.table( "financial-news", Consumed.with(EARLIEST));
```

Создает объект KTable с перечислением EARLIEST и топиком financial-news

Стоит отметить, что никаких объектов `Serde` указывать не требуется, поскольку в настройках используются строковые `Serde`. Также благодаря применению перечисления `EARLIEST` таблица заполняется записями в самом начале.

Теперь мы можем перейти к заключительному шагу — соединению.

Соединение обновлений новостей с данными о числе транзакций

Создание соединения не представляет сложностей. Мы воспользуемся левым соединением на случай, если по соответствующей отрасли промышленности нет биржевых новостей (нужный код можно найти в файле src/main/java/bbejeck/chapter_5/CountingWindowingAndKtableJoinExample.java) (листинг 5.10).

Листинг 5.10. Выполнение соединения объектов KStream и KTable

```
ValueJoiner<TransactionSummary, String, String> valueJoiner =
➥  (txnct, news) ->
➥  String.format("%d shares purchased %s related news [%s]",
➥  txnct.getSummaryCount(), txnct.getStockTicker(), news);

KStream<String,String> joined =
➥  countStream.leftJoin(financialNews, valueJoiner,
➥  Joined.with(stringSerde, transactionKeySerde, stringSerde));

joined.print(Printed.<String, String>toSysOut()
➥  .withLabel("Transactions and News"));
```

ValueJoiner комбинирует значения из результатов соединения

Оператор левого соединения для объекта countStream типа KStream и объекта KTable с биржевыми новостями; нужные объекты Serde передаются с помощью экземпляра Joined

Выводим результаты в консоль (при промышленной эксплуатации они бы записывались в топик с помощью вызова to("название_топика"))

Этот оператор `leftJoin` достаточно прост. В отличие от соединений из главы 4, метод `JoinWindow` не используется, поскольку при выполнении соединения `KStream-KTable` для каждого ключа в `KTable` присутствует только одна запись. Такое соединение не ограничивается по времени: запись или есть в `KTable`, или отсутствует. Основной вывод: с помощью объектов `KTable` можно обогащать `KStream` реже обновляемыми справочными данными.

А теперь мы рассмотрим более эффективный способ обогащения событий из `KStream`.

5.3.4. Объекты GlobalKTable

Как вы поняли, существует необходимость обогащения потоков событий или добавления к ним контекста. В главе 4 вы видели соединения двух объектов `KStream`, а в предыдущем разделе — соединение `KStream` и `KTable`. Во всех этих случаях необходимо повторное секционирование потока данных при отображении ключей на новый тип или значение. Иногда повторное секционирование выполняется явным образом, а иногда Kafka Streams делает это автоматически. Повторное секционирование необходимо, поскольку ключи изменились и записи должны оказаться в новых секциях, иначе соединение окажется невозможным (это обсуждалось в главе 4, в пункте «Повторное секционирование данных» подраздела 4.2.4).

Повторное секционирование имеет свою цену

Повторное секционирование требует затрат — дополнительных затрат ресурсов на создание промежуточных топиков, сохранение дублирующихся данных в еще одном топике; оно также означает повышение задержки вследствие записи и чтения из этого топика. Кроме того, при необходимости выполнить соединение более чем по одному аспекту или измерению нужно организовать соединения цепочкой, отобразить записи с новыми ключами и снова провести процесс повторного секционирования.

Соединение с наборами данных меньшего размера

В некоторых случаях объем справочных данных, с которыми планируется соединение, относительно невелик, так что полные их копии вполне могут поместиться локально на каждом из узлов. Для подобных ситуаций в Kafka Streams предусмотрен класс `GlobalKTable`.

Экземпляры `GlobalKTable` уникальны, поскольку приложение реплицирует все данные на каждый из узлов. А поскольку на каждом из узлов присутствуют все данные, нет необходимости секционировать поток событий по ключу справочных данных, чтобы он был доступен всем секциям. С помощью объектов `GlobalKTable` можно также выполнять бесключевые соединения. Вернемся к одному из предыдущих примеров для демонстрации этой возможности.

Соединение объектов KStream с объектами GlobalKTable

В подразделе 5.3.2 мы выполнили оконное агрегирование биржевых транзакций по покупателям. Результаты этого агрегирования выглядели примерно следующим образом:

```
{customerId='074-09-3705', stockTicker='GUTM'}, 17
{customerId='037-34-5184', stockTicker='CORK'}, 16
```

Хотя эти результаты соответствовали поставленной цели, было бы удобнее, если бы выводилось также имя клиента и полное название компании. Чтобы добавить имя покупателя и название компании, можно выполнять обычные соединения, но при этом понадобится произвести два отображения ключей и повторное секционирование. С помощью `GlobalKTable` можно избежать затрат на подобные операции.

Для этого мы воспользуемся объектом `countStream` из листинга 5.11 (соответствующий код можно найти в файле src/main/java/bbejeck/chapter_5/GlobalKTableExample.java), соединив его с двумя объектами `GlobalKTable`.

Листинг 5.11. Агрегирование биржевых транзакций с помощью сеансовых окон

```
KStream<String, TransactionSummary> countStream =
builder.stream( STOCK_TRANSACTIONS_TOPIC,
➥ Consumed.with(stringSerde, transactionSerde)
➥ .withOffsetResetPolicy(LATEST)).groupBy((noKey, transaction) ->
➥ TransactionSummary.from(transaction),
➥ Serialized.with(transactionSummarySerde, transactionSerde))
     .windowedBy(SessionWindows.with(twentySeconds)).count()
     .toStream().map(transactionMapper);
```

Мы уже обсуждали это ранее, так что не стану повторяться. Но отмечу, что код в функции `toStream().map` ради удобочитаемости абстрагирован в объект-функцию вместо встраиваемого лямбда-выражения.

Следующий этап — объявление двух экземпляров `GlobalKTable` (приведенный код можно найти в файле src/main/java/bbejeck/chapter_5/GlobalKTableExample.java) (листинг 5.12).

Листинг 5.12. Объявление экземпляров GlobalKTable для справочных данных

```
GlobalKTable<String, String> publicCompanies =
➥ builder.globalTable(COMPANIES.topicName());

GlobalKTable<String, String> clients =
➥ builder.globalTable(CLIENTS.topicName());
```

publicCompanies предназначен для поиска компаний по биржевому символу их акций

clients предназначен для получения имен клиентов по их идентификаторам

Обратите внимание, что названия топиков описываются с помощью перечисляемых типов.

Теперь, когда мы подготовили все компоненты, осталось написать код для соединения (который можно найти в файле src/main/java/bbejeck/chapter_5/GlobalKTableExample.java) (листинг 5.13).

Листинг 5.13. Соединение объекта KStream с двумя GlobalKTable

```
countStream.leftJoin(publicCompanies, (key, txn) ->
➥ txn.getStockTicker(),TransactionSummary::withCompanyName)
     .leftJoin(clients, (key, txn) ->
➥ txn.getCustomerId(), TransactionSummary::withCustomerName)
     .print(Printed.<String, TransactionSummary>toSysOut()
➥ .withLabel("Resolved Transaction Summaries"));
```

Выполняет левое соединение с таблицей publicCompanies с биржевыми символами акций в качестве ключей и возвращает объект TransactionSummary с добавленными названиями компаний

Выводит результаты в консоль

Выполняет левое соединение с таблицей clients с идентификаторами клиентов в качестве ключей и возвращает объект TransactionSummary с добавленными именами клиентов

Хотя в этом коде присутствуют два соединения, они организованы в виде цепочки, поскольку отдельно ни один из их результатов не используется. Результаты выводятся в конце всей операции.

При запуске вышеприведенной операции соединения вы получите результаты следующего вида:

```
{customer='Barney, Smith' company="Exxon", transactions= 17}
```

Суть не изменилась, но эти результаты выглядят более понятно.

Если считать главу 4, вы уже видели несколько типов соединений в действии. Они перечислены в табл. 5.2. Эта таблица отражает возможности соединения, актуальные для версии 1.0.0 Kafka Streams; в будущих выпусках, возможно, что-то поменяется.

Таблица 5.2. Соединения в Kafka Streams

Левое соединение	Внутреннее соединение	Внешнее соединение
KStream-KStream	KStream-KStream	KStream-KStream
KStream-KTable	KStream-KTable	–
KTable-KTable	KTable-KTable	KTable-KTable
KStream-GlobalKTable	KStream-GlobalKTable	–

В заключение напомню основное: вы можете соединять потоки событий (`KStream`) и потоки обновлений (`KTable`) с помощью локального состояния. Кроме того, если размер справочных данных не слишком велик, можно воспользоваться объектом `GlobalKTable`. `GlobalKTable` реплицируют все секции на каждый из узлов приложения Kafka Streams, обеспечивая тем самым доступность всех данных независимо от того, какой секции соответствует ключ.

Далее мы увидим возможность Kafka Streams, благодаря которой можно наблюдать изменения состояния без потребления данных из топика Kafka.

5.3.5. Доступное для запросов состояние

Мы уже выполняли несколько операций с участием состояния и всегда выводили результаты в консоль (для целей разработки) или записывали их в топик (для целей промышленной эксплуатации). При записи результатов в топик приходится использовать потребитель Kafka для их просмотра.

Чтение данных из этих топиков можно считать разновидностью *материализованных представлений* (materialized views). Для наших задач можно использовать определение материализованного представления из «Википедии»: «...физический объект базы данных, содержащий результаты выполнения запроса. Например, оно может быть локальной копией удаленных данных, или подмножеством строк и/или столбцов таблицы или результатов соединения, или сводной таблицей, полученной с помощью агрегирования» (https://en.wikipedia.org/wiki/Materialized_view).

Kafka Streams также позволяет выполнять *интерактивные запросы* (interactive queries) к хранилищам состояния, что дает возможность непосредственного чтения этих материализованных представлений. Важно отметить, что запрос к хранилищу состояния носит характер операции «только для чтения». Благодаря этому вы можете не бояться случайно сделать состояние несогласованным во время обработки данных приложением.

Возможность непосредственных запросов к хранилищам состояния имеет большое значение. Она значит, что можно создавать приложения — информационные панели без необходимости сначала получать данные от потребителя Kafka. Повышает она и эффективность приложения, благодаря тому что не требуется снова записывать данные:

- благодаря локальности данных к ним можно быстро обратиться;
- исключается дублирование данных, поскольку они не записываются во внешнее хранилище[1].

Главное, что я хотел бы, чтобы вы запомнили: можно напрямую выполнять запросы к состоянию из приложения. Нельзя переоценить возможности, которые это вам дает. Вместо того чтобы потреблять данные из Kafka и сохранять записи в базе данных для приложения, можно выполнять запросы к хранилищам состояния с тем же результатом. Непосредственные запросы к хранилищам состояния означают меньший объем кода (отсутствие потребителя) и меньше программного обеспечения (отсутствие потребности в таблице базы данных для хранения результатов).

Мы охватили немалый объем информации в настоящей главе, поэтому на время прекратим наше обсуждение интерактивных запросов к хранилищам состояния. Но не волнуйтесь: в главе 9 мы будем создавать простое приложение — информационную панель с интерактивными запросами. Для демонстрации интерактивных запросов и возможностей их добавления в приложения Kafka Streams в нем будут использоваться некоторые из примеров этой и предыдущих глав.

Резюме

- Объекты `KStream` олицетворяют потоки событий, сравнимые со вставками в базу данных. Объекты `KTable` олицетворяют потоки обновлений, они больше схожи с обновлениями в базе данных. Размер объекта `KTable` не растет, старые записи заменяются новыми.
- Объекты `KTable` необходимы для операций агрегирования.

---

[1] Этот раздел основан на информации из статей Джея Крепса (Jay Kreps) Introducing Kafka Streams: Stream Processing Made Simple («Знакомимся с Kafka Streams: упрощение потоковой обработки», http://mng.bz/49HO) и The Log: What Every Software Engineer Should Know About Real-time Data's Unifying Abstraction («Журнал: что каждый разработчик ПО должен знать об абстракции, объединяющей всю обработку поступающих в реальном времени данных», http://mng.bz/eE3w).

- С помощью оконных операций можно разбить агрегированные данные по временны́м корзинам.
- Благодаря объектам `GlobalKTable` можно получить доступ к справочным данным в любой точке приложения, независимо от разбиения по секциям.
- Возможны соединения между собой объектов `KStream`, `KTable` и `GlobalKTable`.

До сих пор мы концентрировали внимание на создании приложений Kafka Streams с помощью высокоуровневого DSL `KStream`. Хотя высокоуровневый подход позволяет создавать аккуратные и лаконичные программы, его использование представляет собой определенный компромисс. Работа с DSL `KStream` означает повышение лаконичности кода за счет снижения степени контроля. В следующей главе мы рассмотрим низкоуровневый API узлов-обработчиков и попробуем другие компромиссы. Программы станут длиннее, чем были до сих пор, зато у нас появится возможность создания практически любого узла-обработчика, который только может нам понадобиться.

6 API узлов-обработчиков

В этой главе:

- что лучше: абстракции более высокого уровня или больше возможностей контроля;
- создание топологии с использованием источников, узлов-обработчиков и стоков;
- углубляемся в API узлов-обработчиков на примере узла финансовой аналитики;
- создаем узел совместной группировки;
- интеграция API узлов-обработчиков и API Kafka Streams.

До сих пор мы работали в этой книге с высокоуровневым API Kafka Streams. Но именно DSL дает возможность разработчикам создавать ошибкоустойчивые приложения с помощью минимального количества кода. Способность быстрой компоновки топологий обработки — важная возможность DSL Kafka Streams. Благодаря этой возможности вы можете быстро воплощать в жизнь свои идеи по обработке данных, не увязая в запутанных настройках вроде тех, что бывают у других фреймворков.

В какой-то момент даже при использовании самых лучших инструментов вы натолкнетесь на какой-нибудь нестандартный случай: задачу, которая требует отхода от привычного пути. Как бы то ни было, вам придется перейти на уровень ниже и создать код, который просто невозможно было написать при использовании абстракции более высокого уровня.

6.1. Компромисс между повышением уровня абстракции и расширением возможностей контроля

Классический пример компромисса между повышением уровня абстракции и расширением возможностей контроля — использование фреймворков объектно-реляционного отображения (object-relational mapping, ORM). Хороший фреймворк ORM отображает объекты предметной области в таблицы базы данных и создает нужные SQL-запросы во время выполнения программы. При SQL-операциях не выше среднего уровня сложности (простые операторы `SELECT` или `JOIN`) фрейм-

ворки ORM экономят массу времени. Но как бы хорош ORM-фреймворк ни был, всегда найдется хотя бы несколько запросов (очень сложные соединения, операторы `SELECT` со вложенными подзапросами), которые просто не работают так, как вам нужно. Вам придется самим написать код SQL для получения информации из базы данных в нужном формате. Здесь и возникает необходимость компромисса между повышением уровня абстракции и расширением возможностей контроля. Зачастую можно сочетать свой код SQL с высокоуровневыми отображениями, реализуемыми фреймворком.

Эта глава посвящена тем случаям, когда требуется такая потоковая обработка, которую использование DSL Kafka Streams только усложняет. Например, как вы видели при работе с API `KTable`, фреймворк управляет временем отправки записей далее по конвейеру. Но вам может понадобиться управлять отправкой записей явным образом. Например, при отслеживании сделок на Уолл-стрит вам нужно отправлять записи далее только в случае превышения ценой акции определенного порогового значения. Для подобного контроля можно задействовать API узлов-обработчиков. Хотя API узлов-обработчиков повышает сложность разработки, но предоставляет более широкие возможности. С его помощью можно создавать пользовательские узлы-обработчики, которые выполняют практически все, что только может вам понадобиться.

Из этой главы вы узнаете, как с помощью API узлов-обработчиков решать следующие задачи:

- планировать выполнение каких-либо действий через равные промежутки времени (на основе меток даты/времени записей или системного времени);
- получить полный контроль над тем, когда записи отправляются далее по конвейеру;
- отправлять записи далее конкретным дочерним узлам;
- создавать отсутствующую в API Kafka Streams функциональность (я покажу пример этого при создании узла совместной группировки).

Сначала посмотрим на использование API узлов-обработчиков на примере создания топологии шаг за шагом.

6.2. Создание топологии с использованием источников, узлов-обработчиков и стоков

Допустим, вы владелец успешной пивоварни (Pops Hops) с несколькими точками продаж. Недавно вы начали принимать онлайн-заказы от оптовых торговцев, включая сбыт за рубежом, в Европе. Вам нужно организовать маршрутизацию заказов внутри компании, в зависимости от того, местный заказ или международный, с преобразованием валюты любых продаж в рамках ЕС из фунтов стерлингов в евро или доллары США.

Упрощенная схема такого технологического процесса будет выглядеть примерно так, как показано на рис. 6.1. При реализации этого примера вы увидите, насколько

гибко позволяет API узлов-обработчиков выбирать конкретные дочерние узлы для пересылки записей. Начнем с создания узла-источника.

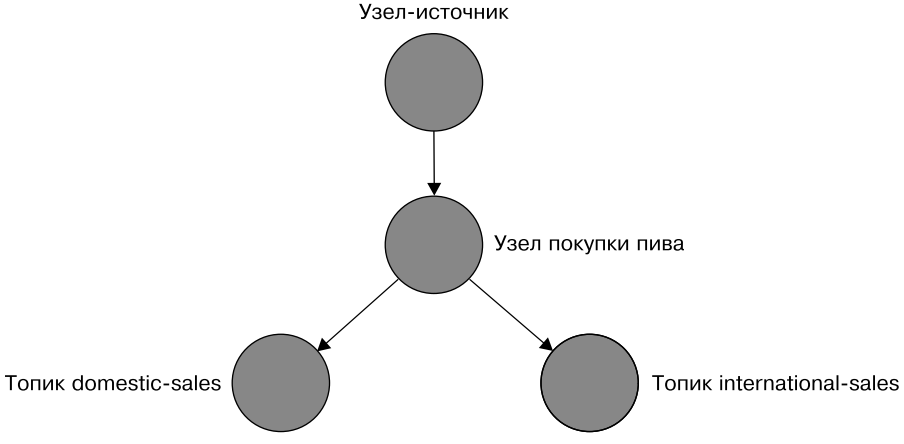


Рис. 6.1. Конвейер сбыта пива

6.2.1. Добавление узла-источника

Первый шаг формирования топологии — создание узлов-источников. В листинге 6.1 (код которого можно найти в файле src/main/java/bbejeck/chapter_6/PopsHopsApplication.java) задается узел-источник для нашей новой топологии.

Листинг 6.1. Создание узла-источника нашего «пивного» приложения

```
topology.addSource(LATEST,
                   purchaseSourceNodeName,
                   new UsePreviousTimeOnInvalidTimestamp(),
                   stringDeserializer,
                   beerPurchaseDeserializer,
                   Topics.POPS_HOPS_PURCHASES.topicName())
```

Задает режим сброса смещений

Задает название узла

Задает используемый для этого источника объект TimestampExtractor

Задает десериализатор ключей

Задает десериализатор значений

Задает имя топика, из которого будут потребляться данные

В методе `Topology.addSource()` есть несколько параметров, которые не использовались нами в DSL. Во-первых, мы указываем название узла-источника. При применении DSL Kafka Streams передавать название не требовалось, поскольку название для узла генерировал экземпляр `KStream`. Но при использовании API узлов-обработчиков необходимо указывать названия узлов топологии. Имена узлов применяются для привязки дочерних узлов к родительским.

Далее мы указываем используемое для этого источника средство извлечения метки даты/времени. В подразделе 4.5.1 мы обсуждали различные средства извлечения меток даты/времени, которые можно применять для источников потоков данных.

В этом случае мы задействовали класс `UsePreviousTimeOnInvalidTimestamp`; все остальные источники в данном приложении будут использовать класс по умолчанию `FailOnInvalidTimestamp`.

Затем мы указали десериализаторы ключей и значений — еще одно отличие от DSL Kafka Streams. В DSL при создании узлов-источников и узлов-стоков мы передавали экземпляры `Serde`. Объект `Serde` сам содержит сериализатор и десериализатор, а DSL Kafka Streams использует нужный в зависимости от того, требуется ли преобразование объекта в байтовый массив или байтового массива в объект. А поскольку API узлов-обработчиков — абстракция более низкого уровня, необходимо напрямую указывать десериализатор при создании узла-источника и сериализатор при создании узла-стока. Наконец, мы указали название топика-источника.

Взглянем теперь на то, как мы будем обрабатывать записи покупок, поступающие в приложение.

6.2.2. Добавление узла-обработчика

Сейчас мы добавим узел для обработки поступающих из узла-источника записей (соответствующий код можно найти в файле src/main/java/bbejeck/chapter_6/PopsHopsApplication.java) (листинг 6.2).

Листинг 6.2. Добавление узла-обработчика

```
BeerPurchaseProcessor beerProcessor =
➥ new BeerPurchaseProcessor(domesticSalesSink, internationalSalesSink);

topology.addSource(LATEST,
                   purchaseSourceNodeName,
                   new UsePreviousTimeOnInvalidTimestamp(),
                   stringDeserializer,
                   beerPurchaseDeserializer,
                   Topics.POPS_HOPS_PURCHASES.topicName())
        .addProcessor(purchaseProcessor,
                      () -> beerProcessor,
                      purchaseSourceNodeName);
```

Наименование узла-обработчика

Добавление описанного выше узла-обработчика

Указываем название (-ия) родительского (-их) узла (-ов)

В этом коде для формирования топологии используется паттерн «текучего» интерфейса. Отличие от API Kafka Streams заключается в возвращаемом типе. В случае API Kafka Streams любой вызов оператора `KStream` возвращает новый экземпляр `KStream` или `KTable`. В API узлов-обработчиков же любое обращение к `Topology` возвращает тот же экземпляр `Topology`.

Второй комментарий описывает передачу узла-обработчика, экземпляр которого был создан в первой строке кода примера. Метод `Topology.addProcessor` принимает в качестве второго параметра экземпляр интерфейса `ProcessorSupplier`, но, поскольку `ProcessorSupplier` — интерфейс с одним методом, его можно заменить лямбда-выражением.

Важнее всего в этом разделе то, что третий параметр, `purchaseSourceNodeName`, метода `addProcessor()` совпадает со вторым параметром метода `addSource()`, как

показано на рис. 6.2. Благодаря этому между узлами устанавливается связь типа «предок — потомок», которая, в свою очередь, определяет путь перемещения записей от одного узла-обработчика к другому в приложении Kafka Streams. На рис. 6.3 показано, что мы уже создали.

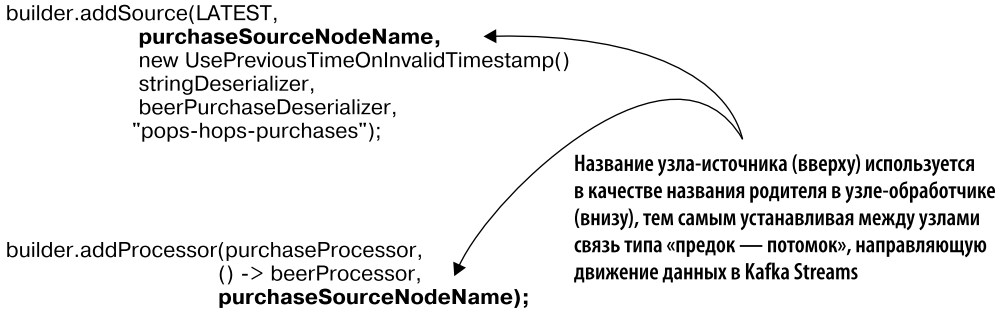


Рис. 6.2. Связываем родительский и дочерний узлы в API узлов-обработчиков

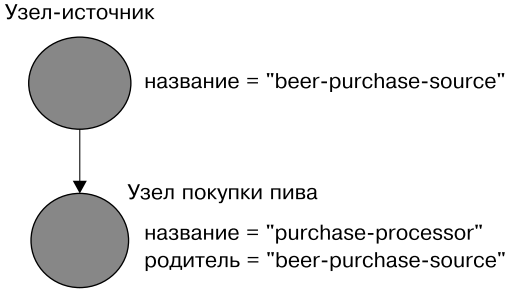


Рис. 6.3. Топология API узлов-обработчиков на текущий момент, включая названия узлов и названия родителей

Уделим немного времени обсуждению функций узла-обработчика `BeerPurchaseProcessor`, созданного в листинге 6.1. У него есть две задачи:

- преобразование сумм продаж на внешнем рынке (в евро) в доллары США;
- маршрутизация записи к соответствующему узлу-стоку в зависимости от типа продажи (за рубежом или на внутреннем рынке).

Все эти действия происходят в методе `process()`. Подытожим вкратце, какие действия он производит:

- проверяет тип валюты. Если сумма указана не в долларах — переводит в доллары;
- если речь идет о продаже за рубежом — направляет обновленную запись в топик `international-sales`;
- в противном случае отправляет запись непосредственно в топик `domestic-sales`.

В листинге 6.3 приведен код для этого узла-обработчика (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/BearPurchaseProcessor.java).

Листинг 6.3. Узел покупки пива (BeerPurchaseProcessor)

```
public class BeerPurchaseProcessor extends
➥ AbstractProcessor<String, BeerPurchase> {

    private String domesticSalesNode;
    private String internationalSalesNode;

    public BeerPurchaseProcessor(String domesticSalesNode,
                                 String internationalSalesNode) {
        this.domesticSalesNode = domesticSalesNode;
        this.internationalSalesNode = internationalSalesNode;
    }

    @Override
    public void process(String key, BeerPurchase beerPurchase) {

        Currency transactionCurrency = beerPurchase.getCurrency();

        if (transactionCurrency != DOLLARS) {
            BeerPurchase dollarBeerPurchase;
            BeerPurchase.Builder builder =
            ➥ BeerPurchase.newBuilder(beerPurchase);
            double internationalSaleAmount = beerPurchase.getTotalSale();
            String pattern = "###.##";
            DecimalFormat decimalFormat = new DecimalFormat(pattern);
            builder.currency(DOLLARS);
            builder.totalSale(Double.parseDouble(decimalFormat.
            ➥ format(transactionCurrency
            ➥ .convertToDollars(internationalSaleAmount))));
            dollarBeerPurchase = builder.build();
            context().forward(key,
            ➥ dollarBeerPurchase, internationalSalesNode);
        } else {
            context().forward(key, beerPurchase, domesticSalesNode);
        }

    }
}
```

Задает имена различных узлов, в которые будут направляться записи

Метод process(), в котором происходят основные действия

Преобразует суммы продаж за рубежом в доллары США

Отправляет записи, относящиеся к местным продажам, в дочерний узел-обработчик местных продаж

С помощью объекта ProcessorContext, возвращаемого из метода context(), направляет записи в дочерний узел-обработчик продаж на внешнем рынке

Класс из этого примера расширяет `AbstractProcessor` — класс, переопределяющий методы интерфейса `Processor`, за исключением метода `process()`. Именно в методе `Processor.process()` мы производим действия над проходящими через топологию записями.

ПРИМЕЧАНИЕ

В интерфейсе Processor имеются методы init(), process(), punctuate() и close(). Processor — основная движущая сила логики любого работающего с записями потокового приложения. В примерах мы в основном будем использовать класс AbstractProcessor, так что переопределять станем только необходимые нам методы. Класс AbstractProcessor сам инициализирует объект ProcessorContext, так что переопределять метод init() не нужно, разве что вам понадобится выполнить какие-либо дополнительные настройки.

Последние несколько строк листинга 6.3 демонстрируют главное в этом примере — способность направлять записи конкретным дочерним узлам. Метод `context()` в данных строках извлекает ссылку на объект `ProcessorContext` для нужного узла-обработчика. Все узлы-обработчики топологии получают ссылки на `ProcessorContext` с помощью метода `init()`, выполняемого потоковой задачей (объектом `StreamTask`) при инициализации топологии.

Мы узнали, как обрабатывать записи, теперь нам нужно подключиться к узлу-стоку (топику), чтобы занести записи обратно в Kafka.

6.2.3. Добавление узла-стока

Сейчас вы уже, наверное, хорошо представляете себе схему применения API узлов-обработчиков. Для добавления источника мы воспользовались методом `addSource`, а для добавления узла-обработчика — методом `addProcessor`. Как можно ожидать, для подключения узла-стока (топика) к узлу-обработчику мы воспользуемся методом `addSink()`. На рис. 6.4 показана обновленная топология.

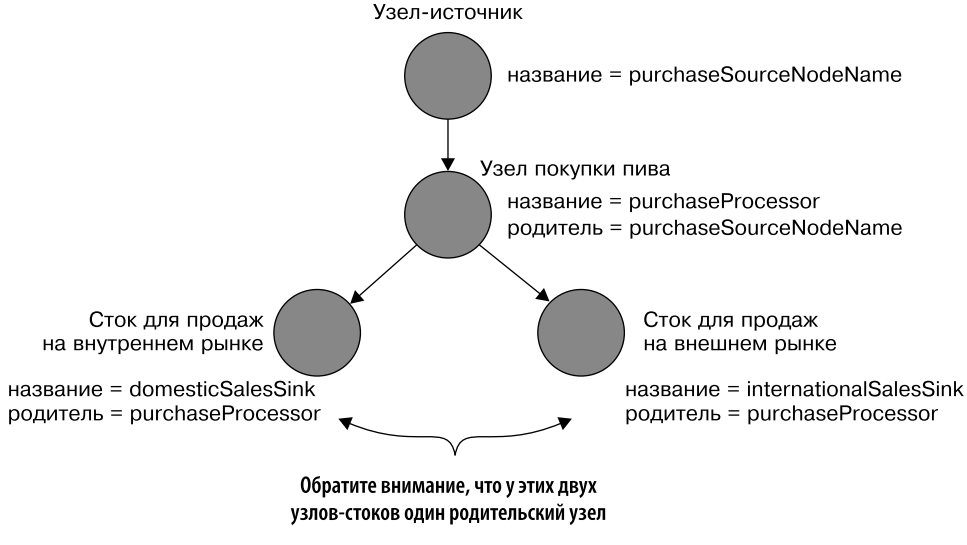


Рис. 6.4. Завершаем топологию, добавляя узлы-стоки

Теперь мы можем обновить нашу топологию, добавив код для узлов-стоков (его можно найти в файле src/main/java/bbejeck/chapter_6/PopsHopsApplication.java) (листинг 6.4).

Листинг 6.4. Добавление узла-стока

```
toplogy.addSource(LATEST,
                  purchaseSourceNodeName,
                  new UsePreviousTimeOnInvalidTimestamp(),
                  stringDeserializer,
                  beerPurchaseDeserializer,
                  Topics.POPS_HOPS_PURCHASES.topicName())
        .addProcessor(purchaseProcessor,
                      () -> beerProcessor,
                      purchaseSourceNodeName)
        .addSink(internationalSalesSink,
                 "international-sales",
                 stringSerializer,
                 beerPurchaseSerializer,
                 purchaseProcessor)
        .addSink(domesticSalesSink,
                 "domestic-sales",
                 stringSerializer,
                 beerPurchaseSerializer,
                 purchaseProcessor);
```

Название стока

Топик, которому этот сток соответствует

Сериализатор для значения

Родительский узел для данного стока

Сериализатор для ключа

Название стока

Топик, которому этот сток соответствует

Сериализатор для значения

Родительский узел для данного стока

В этом листинге мы добавили два узла-стока, один — для долларов и второй — для евро. Записи будут заноситься в топик, соответствующий валюте транзакции.

Важно отметить, что название родительского узла у обоих добавляемых стоков совпадает. Благодаря этому мы связали оба узла-стока с одним нашим узлом-обработчиком (как показано на рис. 6.4).

В этом нашем первом примере были продемонстрированы компоновка топологий и направление записей в конкретные дочерние узлы. Хотя API узлов-обработчиков требует несколько большего объема кода, чем API Kafka Streams, создание топологий с его помощью все равно остается несложной задачей. В следующем примере я покажу, насколько гибок API узлов-обработчиков.

6.3. Углубляемся в API узлов-обработчиков на примере узла биржевой аналитики

Вернемся к сфере финансовых операций и примерим шляпу внутридневного трейдера. Внутридневному трейдеру необходимо анализировать изменения курсов акций, чтобы выбрать оптимальные моменты для покупки и продажи. Его задача — извлечь выгоду из колебаний рынка и сорвать быструю прибыль. Мы будем учитывать

несколько ключевых показателей в надежде, что они укажут, когда вам имеет смысл начинать действовать.

Вот список требований:

- отображать текущий курс акций;
- показывать, повышается или понижается цена за одну акцию;
- отображать общее количество проданных на текущий момент акций, а также его тренд (понижение/повышение);
- отправлять записи далее по конвейеру только для акций с 2%-ным трендом (понижение/повышение);
- собирать не менее 20 выборок заданных акций, прежде чем производить с ними какие-либо вычисления.

Рассмотрим, как осуществить этот анализ вручную. Рисунок 6.5 демонстрирует пример дерева решений, которое понадобится нам для принятия подобных решений.

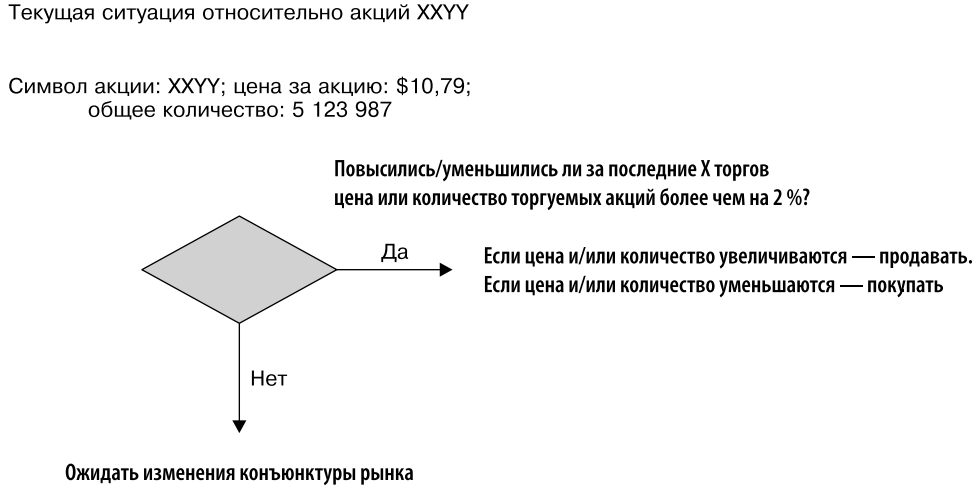


Рис. 6.5. Обновления тренда курса акций

Для нашего анализа необходимо провести несколько расчетов. Кроме того, мы воспользуемся результатами этих расчетов для определения того, нужно ли (и если да, то когда) отправлять записи далее по конвейеру.

Данное ограничение на отправку записей означает, что мы не можем положиться на стандартный механизм фиксации времени или сброса кэша на диск для управления их движением, что исключает использование API Kafka Streams. Само собой разумеется, что нам придется сохранять состояние, чтобы отслеживать изменения в динамике. Нам необходима возможность создания пользовательских узлов-обработчиков. Посмотрим, как решить эту задачу.

Только для демонстрационных целей

Я совершенно уверен, что это и так понятно, но все равно констатирую очевидное: настоящие оценки курсов акций приводятся только для демонстрационных целей. Не делайте никаких рыночных прогнозов на основе этого примера. Данная модель абсолютно не похожа на реальные и представлена только для демонстрации более сложного сценария обработки. Я отнюдь не внутридневной трейдер!

6.3.1. Узел-обработчик показателей акций

В листинге 6.5 приведена реализация приложения для обработки показателей акций (его можно найти в файле src/main/java/bbejeck/chapter_6/StockPerformanceApplication.java).

Листинг 6.5. Приложение для обработки показателей акций с пользовательским узлом-обработчиком

```
Topology topology = new Topology();
String stocksStateStore = "stock-performance-store";
double differentialThreshold = 0.02;

KeyValueBytesStoreSupplier storeSupplier =
➥ Stores.inMemoryKeyValueStore(stocksStateStore);
StoreBuilder<KeyValueStore<String, StockPerformance>> storeBuilder
➥ = Stores.keyValueStoreBuilder(
➥ storeSupplier, Serdes.String(), stockPerformanceSerde);

  topology.addSource("stocks-source",
                     stringDeserializer,
                     stockTransactionDeserializer,
                     "stock-transactions")
          .addProcessor("stocks-processor",
➥ () -> new StockPerformanceProcessor(
➥ stocksStateStore, differentialThreshold), "stocks-source")
          .addStateStore(storeBuilder,"stocks-processor")
          .addSink("stocks-sink",
                   "stock-performance",
                   stringSerializer,
                   stockPerformanceSerializer,
                   "stocks-processor");
```

Задаем разницу в процентах, при превышении которой информация об акциях будет отправляться далее

Создаем хранилище состояния в виде пар «ключ/значение» (в оперативной памяти)

Создаем объект storeBuilder для вставки в топологию

Добавляем узел-обработчик в топологию

Добавляем хранилище состояния к узлу-обработчику показателей акций

Добавляем сток для вывода результатов, хотя мы можем выводить результаты и в консоль

Технологический процесс в этой топологии такой же, как и в предыдущем примере, так что сосредоточим внимание на новых элементах узла-обработчика. В предыдущем примере не нужны никакие настройки, так что можно было положиться на инициализацию объекта `ProcessorContext` методом `AbstractProcessor.init`. В этом же примере, однако, мы собираемся использовать хранилище состояния, а также запланировать время отправки записей вместо пересылки их сразу при получении.

Рассмотрим сначала метод `init()` узла-обработчика (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/StockPerformanceProcessor.java) (листинг 6.6).

Листинг 6.6. Задачи метода init()

```
@Override
public void init(ProcessorContext processorContext) {
  super.init(processorContext);
  keyValueStore =
➥ (KeyValueStore) context().getStateStore(stateStoreName);
  StockPerformancePunctuator punctuator =
➥ new StockPerformancePunctuator(differentialThreshold,
                                  context(),
                                  keyValueStore);
 context().schedule(10000, PunctuationType.WALL_CLOCK_TIME,
➥ punctuator);
 }
}
```

Инициализируем ProcessorContext с помощью метода суперкласса AbstractProcessor

Извлекаем хранилище состояния, созданное при формировании топологии

Инициализируем объект Punctuator, отвечающий за выполнение обработки по расписанию

Планируем вызов метода Punctuator.punctuate() каждые 10 секунд

Во-первых, нам нужно инициализировать `AbstractProcessor` с помощью `ProcessorContext`, так что мы вызываем метод `init()` суперкласса. Далее мы получаем ссылку на созданное в топологии хранилище состояния. Все, что от нас требуется, — сохранить ее в переменной для дальнейшего использования в узле-обработчике. В листинге 6.5 также появляется `Punctuator` — интерфейс, представляющий собой обратный вызов для выполнения логики узла-обработчика в соответствии с расписанием, инкапсулированный в методе `Punctuator.punctuate()`.

СОВЕТ

Метод ProcessorContext.schedule(long, PunctuationType, Punctuator) возвращает тип Cancellable, благодаря чему вы можете отменить выполнение пунктуации и работать с более продвинутыми сценариями, например перечисленными в обсуждении Punctuate Use Cases («Сценарии использования метода пунктуации», http://mng.bz/YSKF). Я не стану приводить примеры в тексте или обсуждать здесь этот вопрос, но некоторые примеры вы можете найти в каталоге src/main/java/bbejeck/chapter_6/cancellation.

В последней строке листинга 6.5 мы используем `ProcessorContext`, чтобы запланировать выполнение `Punctuator` каждые 10 секунд. Второй параметр, `PunctuationType.WALL_CLOCK_TIME`, определяет, что мы хотели бы вызывать метод `Punctuator.punctuate()` каждые 10 секунд по системному времени (`WALL_CLOCK_TIME`). Можно также задать другую опцию, `PunctuationType.STREAM_TIME`, означающую, что выполнение `Punctuator.punctuate()` тоже будет производиться каждые 10 секунд, но в соответствии с временем, прошедшим судя по меткам даты/времени данных. Выясним, в чем различие между этими двумя настройками `PunctuationType`.

Семантика пунктуации. Начнем наш разговор о семантике пунктуации со `STREAM_TIME`, поскольку эта опция требует некоторых пояснений. Рисунок 6.6 иллюстрирует понятие времени потока (stream time). Давайте проясним некоторые

нюансы, чтобы лучше разобраться с тем, как формируется расписание выполнения (отмечу, что некоторые из внутренних механизмов Kafka Streams тут не показаны).

- `StreamTask` извлекает из `PartitionGroup` *минимальную* метку даты/времени. `PartitionGroup` представляет собой набор секций заданного `StreamThread`, который содержит всю информацию о метках даты/времени для всех секций группы.
- Во время обработки записей `StreamThread` обрабатывает в цикле свой объект `StreamTask`, при этом каждая задача в конце концов вызывает метод `punctuate` для каждого из своих подходящих для пунктуации узлов-обработчиков. Напомню, что вам нужно собрать не менее 20 сделок с конкретным видом акций, прежде чем оценивать его показатели.
- Если метка даты/времени с прошлого вызова `punctuate` (плюс запланированный промежуток времени) меньше извлеченной из `PartitionGroup` или равна ей, то Kafka Streams вызывает метод `punctuate` данного узла-обработчика.

Рис. 6.6. Расписание пунктуации при использовании настройки STREAM_TIME

Главное в этом процессе то, что приложение наращивает метки даты/времени с помощью `TimestampExtractor`, так что вызовы метода `punctuate()` будут единообразными только в том случае, если данные поступают с постоянной скоростью. Если же данные поступают от случая к случаю, то метод `punctuate()` не будет выполняться через запланированные регулярные интервалы времени.

При настройке `PunctuationType.WALL_CLOCK_TIME`, с другой стороны, выполнение `Punctuator.punctuate` будет более предсказуемым благодаря использованию системного времени. Отмечу, что семантика системного времени гарантий не дает — в пределах интервала опроса системное время меняется и шаг изменения зависит от того, сколько времени требуется для выполнения цикла опроса. Так что в примере из листинга 6.6 можно ожидать, что пунктуация будет производиться через интервалы, близкие к 10-секундным, вне зависимости от темпов поступления данных.

Какой подход использовать — зависит исключительно от ваших потребностей. Если вам нужно регулярно выполнять какие-либо действия независимо от движения данных, то оптимальным вариантом будет применение системного времени. С другой стороны, если требуется только производить вычисления над входящими данными и небольшой разрыв во времени между выполнениями допустим, попробуйте семантику потокового времени.

ПРИМЕЧАНИЕ

> В версиях Kafka, предшествующих 0.11.0, для пунктуации использовался метод ProcessorContext.schedule(long time), вызывавший, в свою очередь, метод Processor.punctuate через заданные промежутки времени. Такой подход работал только при семантике потокового времени, и оба метода в настоящий момент считаются устаревшими. Я иногда упоминаю в этой книге устаревшие методы, но в примерах используются только новейшие методы пунктуации.

Мы рассмотрели вопросы планирования, выполнения и пунктуации. Время перейти к обработке входящих записей.

6.3.2. Метод process()

Именно в методе `process()` производятся все расчеты для оценки показателей акций. При получении записи необходимо выполнить несколько шагов:

- проверить хранилище состояния на наличие объекта `StockPerformance`, соответствующего биржевому символу акции из этой записи;
- создать соответствующий объект `StockPerformance`, если в хранилище состояния его нет. Далее экземпляр `StockPerformance` суммирует значения для текущей цены акции и количества проданных акций и обновляет результаты;
- после достижения количества 20 транзакций для какого-либо вида акций начать расчеты.

Хотя вопрос финансового анализа и выходит за пределы данной книги, мы вкратце рассмотрим эти расчеты. Мы будем вычислять как для цены за акцию, так и для количества продаваемых акций простое скользящее среднее (simple moving average, SMA). В сфере биржевой торговли SMA используются для вычисления среднего значения для наборов данных размером *N*.

В этом примере $N = 20$. Установка подобного ограничения означает, что при поступлении новой информации о сделках будут собираться данные о цене за акцию и количестве проданных акций для первых 20 транзакций. По достижении этого порогового значения самое старое значение будет удаляться и добавляться самое последнее. С помощью SMA мы получаем скользящее среднее цены акции и числа проданных акций за последние 20 сделок. Важно отметить, что пересчитывать все данные при поступлении новых значений не требуется.

На рис. 6.7 приведена общая картина работы метода `process()`, где показано, какие действия производились бы при выполнении этих шагов вручную. Именно в методе `process()` проводятся все вычисления.

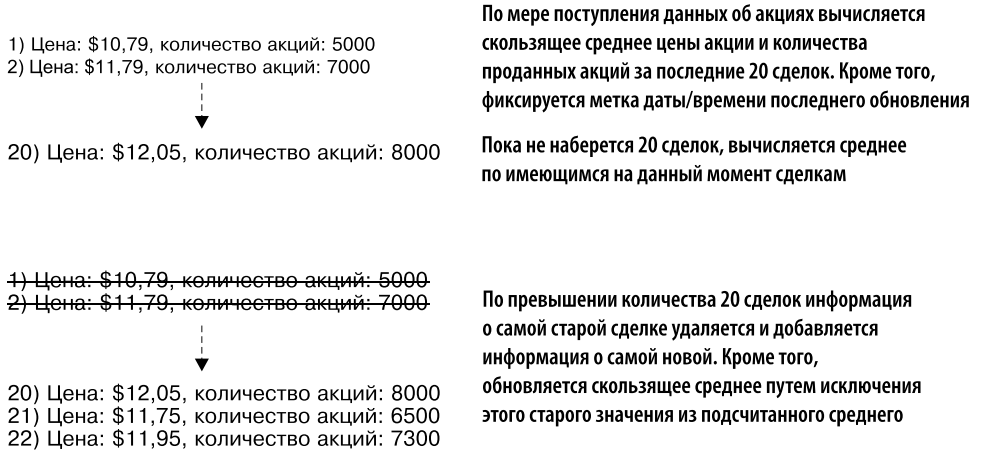


Рис. 6.7. Пошаговое выполнение метода process() для анализа динамики акций

Взглянем теперь на код метода `process()` (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/StockPerformanceProcessor.java) (листинг 6.7).

В методе `process()` мы прибавляем последние цену акции и число участвовавших в транзакции акций к значениям из объекта `StockPerformance`. Обратите внимание, что все детали выполнения обновления абстрагированы в объекте `StockPerformance`. В целом считается хорошим тоном не включать бизнес-логику в узел-обработчик — мы вернемся к этому вопросу в главе 8, когда будем обсуждать тестирование.

В этом коде выполняется два основных вычисления: определение скользящего среднего и вычисление разницы со скользящими средними цены акции/количества акций. Мы не хотим вычислять среднее до тех пор, пока не соберем данные о 20 транзакциях, так что откладываем выполнение каких-либо действий до момента

получения узлом-обработчиком данных о 20 сделках. Накопив информацию о 20 сделках для конкретного вида акций, можно вычислить наше первое среднее значение. А затем мы делим текущее значение цены акции или числа проданных акций на скользящее среднее, преобразуя результат в процентное соотношение.

Листинг 6.7. Реализация метода process()

```
@Override
public void process(String symbol, StockTransaction transaction) {
 StockPerformance stockPerformance = keyValueStore.get(symbol);

 if (stockPerformance == null) {
  stockPerformance = new StockPerformance();
 }

 stockPerformance.updatePriceStats(transaction.getSharePrice());
 stockPerformance.updateVolumeStats(transaction.getShares());
 stockPerformance.setLastUpdateSent(Instant.now());

 keyValueStore.put(symbol, stockPerformance);
}
```

Извлекаем предыдущую статистику по динамике акций, возможно, пустую

Создаем новый объект StockPerformance, если в хранилище состояния соответствующего объекта нет

Обновляем статистику цены для этого вида акций

Обновляем статистику по количеству проданных акций для этого вида акций

Задаем метку даты/времени последнего обновления

Помещаем обновленный объект StockPerformance в хранилище состояния

ПРИМЕЧАНИЕ

Если вы хотите взглянуть на вычисления, то код StockPerformance можно найти в файле src/main/java/bejeck/model/StockPerformance.java.

В примере с интерфейсом `Processor` в листинге 6.3 мы по завершении выполнения метода `process()` отправляли записи далее по конвейеру. В данном же случае мы сохраняем итоговые результаты в хранилище состояния и оставляем отправку записей методу `Punctuator.punctuate`.

6.3.3. Выполнение пунктуатора

Мы уже обсудили семантику пунктуации и выполнение по расписанию, так что перейдем прямо к изучению кода метода `Punctuator.punctuate` (который можно найти в файле src/main/java/bejeck/chapter_6/processor/punctuator/StockPerformancePunctuator.java) (листинг 6.8).

Последовательность действий в методе `Punctuator.punctuate` проста. Мы проходим в цикле по парам «ключ/значение» из хранилища состояния и, если значение превысило заданный порог, отправляем запись далее по конвейеру.

Здесь важно запомнить, что если ранее отправка записей дальше осуществлялась посредством механизмов фиксации и сброса кэша на диск, то теперь мы сами определяем, когда это будет происходить. Кроме того, предполагаемое выполнение этого кода каждые 10 секунд не гарантирует фактическую отправку записей — они

должны достичь разностного порога. Отмечу также, что методы `Processor.process` и `Punctuator.punctuate` не выполняются в конкурентном режиме.

Листинг 6.8. Код пунктуации

```
@Override
public void punctuate(long timestamp) {
  KeyValueIterator<String, StockPerformance> performanceIterator =
  ➥ keyValueStore.all();

  while (performanceIterator.hasNext()) {
        KeyValue<String, StockPerformance> keyValue =
        ➥ performanceIterator.next();
        String key = keyValue.key;
        StockPerformance stockPerformance = keyValue.value;

        if (stockPerformance != null) {
            if (stockPerformance.priceDifferential()
            ➥ >= differentialThreshold ||
                stockPerformance.volumeDifferential()
            ➥ >= differentialThreshold) {
                context.forward(key, stockPerformance);
            }
        }
    }
}
```

Извлекаем итератор для прохода по всем парам «ключ/значение» в хранилище состояния

Проверяем пороговое значение для текущего вида акций

При достижении порогового значения отправляем запись

ПРИМЕЧАНИЕ

Хотя мы иллюстрируем возможности доступа к хранилищу состояния, не помешает вспомнить архитектуру Kafka Streams и обсудить несколько важных нюансов. У каждой задачи StreamTask есть *своя* копия *локального* хранилища состояния, и объекты StreamThread не используют совместно задачи или состояние. Записи движутся по топологии посредством обхода графа «вглубь», а значит, ни один узел-обработчик не производит конкурентного обращения к хранилищам состояния.

Этот пример — превосходное введение в написание пользовательских узлов-обработчиков, но можно пойти дальше и добавить новые структуры данных, а также совершенно новые способы агрегирования данных, отсутствующие в API. С учетом этого перейдем к добавлению узла совместной группировки.

6.4. Узел совместной группировки

В главе 4 мы обсуждали соединения двух потоков, а точнее, соединения покупок из различных подразделений компании в пределах заданного промежутка времени. Соединения можно использовать для сочетания записей с одинаковым ключом, поступивших в пределах одного временно*г*о окна. В случае соединений предполагается

взаимно-однозначное соответствие записей в потоке A и потоке B. Рисунок 6.8 иллюстрирует эту связь.

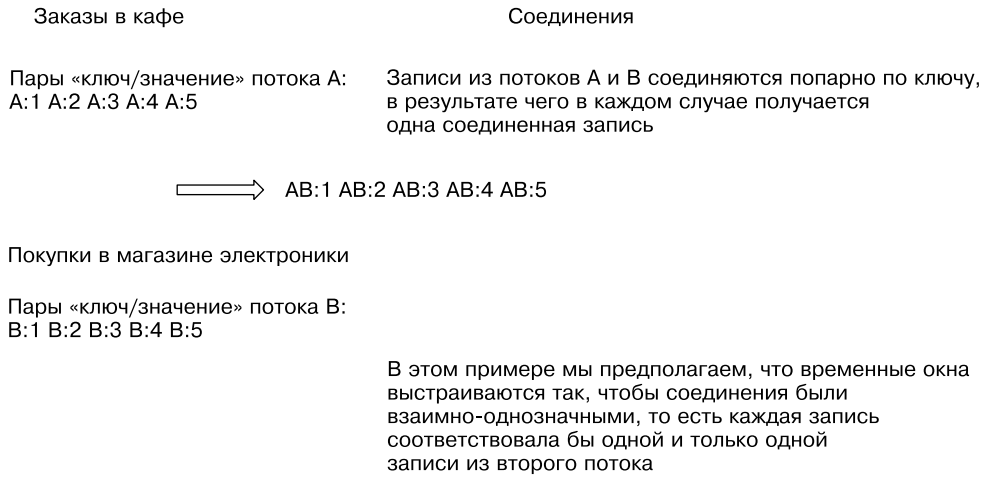


Рис. 6.8. Соединение записей из потоков A и B по общему ключу

Представьте себе теперь, что требуется выполнить подобный анализ, только вместо взаимно-однозначного соединения по ключу нам нужно соединить две коллекции данных по общему ключу, то есть произвести так называемую *совместную группировку* (co-grouping) данных. Представьте себе, что вы управляете популярным приложением для внутридневного трейдинга. Внутридневные трейдеры применяют ваше приложение несколько часов в день — иногда все время, когда открыта биржа. Одна из отслеживаемых вашим приложением метрик — понятие события. Мы считаем *событием* (event) щелчок пользователя на символе акции для чтения более подробной информации о компании и ее финансовых показателях. Нам хотелось бы проанализировать подробнее взаимосвязь между этими щелчками пользователей в приложении и покупкой ими акций. Нам нужны результаты сравнения щелчков и покупок крупным планом для понимания картины в целом. Нам требуется кортеж с двумя коллекциями — для каждого из типов событий, сгруппированных по биржевым символам компаний, как показано на рис. 6.9.

Наша цель состоит в комбинации текущего состояния событий щелчков и биржевых транзакций для заданной компании каждые N секунд, не ожидая поступления записей ни из одного потока. Когда проходит заданный промежуток времени, мы хотим получить совместную группировку событий щелчков и биржевых транзакций по тикерному символу компании. А если события одного из видов отсутствуют, то одна из коллекций в кортеже окажется пустой. Если вы знакомы с фреймворком Apache Spark или Apache Flink, то эта функциональность напоминает метод `PairRDDFunctions.cogroup` (http://mng.bz/LaD4) и класс `CoGroupDataSet`

(http://mng.bz/FH9m) соответственно. Рассмотрим, из каких шагов состоит создание этого узла-обработчика.

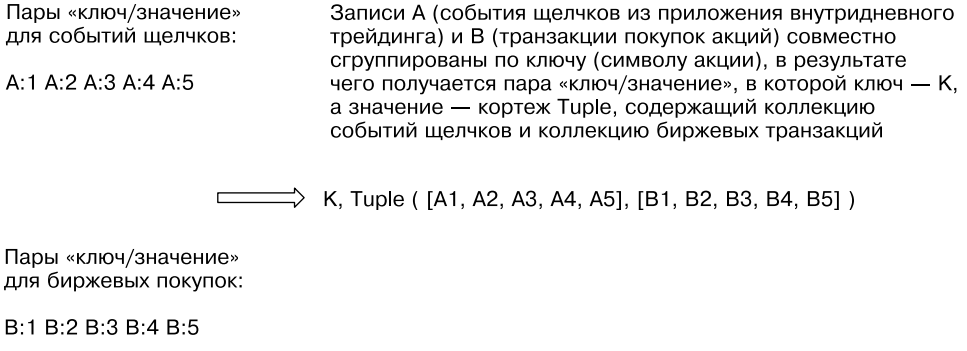


Рис. 6.9. Результаты совместной группировки по ключу с помощью кортежа, содержащего две коллекции данных

6.4.1. Создание узла совместной группировки

Для создания узла совместной группировки необходимо связать воедино несколько элементов:

- описать два топика (`stock-transactions`, `events`);
- добавить два узла-обработчика для потребления записей из вышеуказанных топиков;
- добавить третий узел-обработчик в качестве агрегатора/узла совместной группировки для двух предыдущих узлов;
- добавить хранилище состояния для узла агрегирования, в котором будет храниться состояние для обоих событий;
- добавить узел-сток, в который будут записываться результаты (и/или узел вывода результатов в консоль).

Рассмотрим теперь каждый из этих шагов подробнее.

Описание узлов-источников

Первый шаг — создание узлов-источников — уже вам хорошо знаком. На этот раз мы создадим два узла-источника — для чтения как потока событий щелчков, так и потока биржевых транзакций. Чтобы видеть, где в топологии мы находимся в данный

момент, мы будем отталкиваться от рис. 6.10, дорисовывая его постепенно. Код создания узлов-источников показан в листинге 6.9 (его можно найти в файле src/main/java/bbejeck/chapter_6/CoGroupingApplication.java).

Листинг 6.9. Узлы-источники для узла совместной группировки

```
topology.addSource("Txn-Source",
                   stringDeserializer,
                   stockTransactionDeserializer,
                   "stock-transactions")
       .addSource("Events-Source",
                   stringDeserializer,
                   clickEventDeserializer,
                   "events")
```

Узел-источник для топика stock-transactions

Узел-источник для топика events

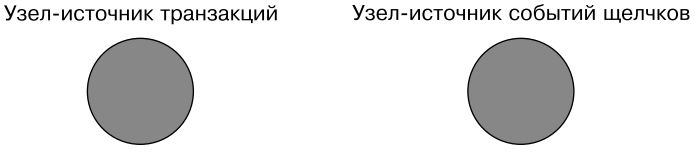


Рис. 6.10. Узлы-источники для совместной группировки

После создания источников для топологии перейдем к следующему шагу.

Добавление узлов-обработчиков

Добавим теперь «рабочих лошадок» топологии — узлы-обработчики. На рис. 6.11 показан обновленный граф топологии. Вот код для добавления этих двух узлов-обработчиков (находится в файле src/main/java/bbejeck/chapter_6/CoGroupingApplication.java) (листинг 6.10).

Листинг 6.10. Узлы-обработчики

```
.addProcessor("Txn-Processor",
              StockTransactionProcessor::new,
              "Txn-Source")

.addProcessor("Events-Processor",
              ClickEventProcessor::new,
              "Events-Source")

.addProcessor("CoGrouping-Processor",
              CogroupingProcessor::new,
              "Txn-Processor",
              "Events-Processor")
```

Добавляет объект StockTransactionProcessor

Добавляет объект ClickEventProcessor

Добавляет объект CogroupingProcessor, который является дочерним узлом для обоих процессоров

В первых двух строках названия родительских узлов представляют собой названия узлов-источников, читающих данные из топиков `stock-transactions` и `events` соответственно. У третьего узла-обработчика в качестве названий родительских

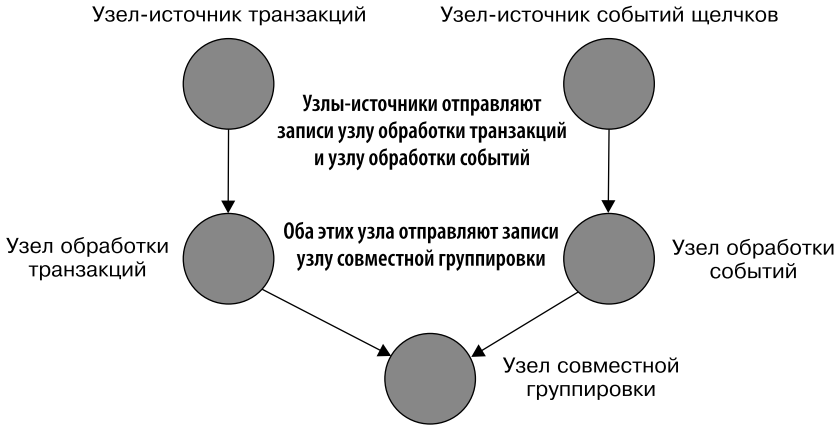


Рис. 6.11. Добавление узлов-обработчиков

узлов указаны названия обоих предыдущих узлов-обработчиков. Это значит, что оба первых узла-обработчика будут поставлять данные для узла агрегирования.

Для экземпляров `ProcessorSupplier` мы опять же воспользуемся сокращенным написанием Java 8. На этот раз мы сократили форму написания еще сильнее путем применения дескриптора метода — в данном случае для вызова конструктора, создающего соответствующий узел-обработчик.

СОВЕТ

В случае интерфейсов с одним методом без аргументов в Java 8 можно воспользоваться лямбда-выражением в виде ()->doSomething. Но, поскольку от ProcessorSupplier требуется только вернуть (возможно, новый) объект Processor, можно сократить эту форму еще больше и воспользоваться дескриптором метода для конструктора типа Processor. Обратите внимание, что это возможно только для конструкторов без аргументов.

Теперь поясню, почему наша топология имеет именно такой вид. Наш пример представляет собой операцию агрегирования, и задача `StockTransactionProcessor` и `ClickEventProcessor` состоит в обертывании соответствующих объектов в меньшие агрегирующие объекты с последующей отправкой их другому узлу-обработчику для полного агрегирования. Как `StockTransactionProcessor`, так и `ClickEventProcessor` выполняют это частичное агрегирование и отправляют записи в `CogroupingProcessor`. После чего `CogroupingProcessor` выполняет совместную группировку и отправляет результаты через равные интервалы времени (определяемые метками даты/времени) в топик вывода результатов.

Код узла-обработчика `StockTransactionProcessor` приведен в листинге 6.11 (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/StockTransactionProcessor.java).

Листинг 6.11. StockTransactionProcessor

```
public class StockTransactionProcessor extends
➥ AbstractProcessor<String, StockTransaction> {

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);
    }

    @Override
    public void process(String key, StockTransaction value) {
        if (key != null) {
            Tuple<ClickEvent,StockTransaction> tuple =
            ➥ Tuple.of(null, value);
            context().forward(key, tuple);
        }
    }
}
```

Создает агрегирующий объект на основе объекта StockTransaction

Отправляет кортеж в узел совместной группировки (CogroupingProcessor)

Как вы можете видеть, `StockTransactionProcessor` добавляет объект `StockTransaction` в агрегатор (`Tuple`) и отправляет запись дальше.

ПРИМЕЧАНИЕ

Показанный в листинге 6.11 Tuple<L, R> представляет собой пользовательский объект, предназначенный для примеров из этой книги. Вы можете найти его описание в файле src/main/java/bbejeck/util/collection/Tuple.java.

Теперь посмотрим на код `ClickEventProcessor` (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/ClickEventProcessor.java) (листинг 6.12).

Листинг 6.12. ClickEventProcessor

```
public class ClickEventProcessor extends
➥ AbstractProcessor<String, ClickEvent> {

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);

    }

    @Override
    public void process(String key, ClickEvent clickEvent) {
        if (key != null) {
            Tuple<ClickEvent, StockTransaction> tuple =
            ➥ Tuple.of(clickEvent, null);
            context().forward(key, tuple);
        }
    }
}
```

Добавляет объект clickEvent в исходный агрегирующий объект

Отправляет кортеж в узел совместной группировки (CogroupingProcessor)

Как видите, `ClickEventProcessor` добавляет `clickEvent` в агрегатор `Tuple` аналогично предыдущему листингу.

Для полноты картины агрегирования нам осталось посмотреть на код `CogroupingProcessor`. Он сложнее, так что мы рассмотрим каждый из его методов по очереди, начиная с `CogroupingProcessor.init()` (который можно найти в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingProcessor.java) (листинг 6.13).

Листинг 6.13. Метод CogroupingProcessor.init()

```
public class CogroupingProcessor extends
➥ AbstractProcessor<String, Tuple<ClickEvent,StockTransaction>> {

    private KeyValueStore<String,
    ➥ Tuple<List<ClickEvent>,List<StockTransaction>>> tupleStore;
    public static final  String TUPLE_STORE_NAME = "tupleCoGroupStore";

    @Override
    @SuppressWarnings("unchecked")
    public void init(ProcessorContext context) {
        super.init(context);
        tupleStore = (KeyValueStore)
        ➥ context().getStateStore(TUPLE_STORE_NAME);
        CogroupingPunctuator punctuator =
        ➥ new CogroupingPunctuator(tupleStore, context());
        context().schedule(15000L, STREAM_TIME, punctuator);
    }
}
```

Извлекает сконфигурированное хранилище состояния

Создает экземпляр пунктуатора — CogroupingPunctuator, отвечающий за все запланированные вызовы

Планирует вызов метода Punctuator.punctuate() каждые 15 секунд

Как вы могли ожидать, метод `init()` отвечает за нюансы настройки класса. Мы сохраняем сконфигурированное в главном приложении хранилище состояния в переменной для дальнейшего использования. А также создаем `CogroupingPunctuator`, отвечающий за запланированные вызовы пунктуации.

Дескрипторы методов для класса Punctuator

Для экземпляра класса `Punctuator` можно задать дескриптор метода. Для этого нужно объявить метод в узле-обработчике, принимающий один параметр типа `long`, с возвращаемым типом `void`. После этого запланируйте пунктуацию следующим образом:

```
context().schedule(15000L, STREAM_TIME, this::myPunctuationMethod);
```

Пример этого вы можете найти в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingMethodHandleProcessor.java.

В листинге 6.13 вызов `punctuate()` запланирован через каждые 15 секунд. Благодаря использованию семантики `PunctuationType.STREAM_TIME` вызовы `punctuate()` определяются метками даты/времени в *поступающих* данных. Помните, что если

движение данных относительно неравномерно, то между вызовами `Punctuator.punctuate()` может пройти более 15 секунд.

ПРИМЕЧАНИЕ

Как вы помните из обсуждения семантики punctuate, существует два варианта: PunctuationType.STREAM_TIME и PunctuationType.WALL_CLOCK_TIME. В листинге 6.13 используется семантика PunctuationType.STREAM_TIME. В прилагаемом к книге коде есть дополнительный пример узла-обработчика, демонстрирующий семантику PunctuationType.WALL_CLOCK_TIME, в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingSystemTimeProcessor.java, что дает вам возможность посмотреть на различия в производительности и поведении.

Посмотрим теперь, как `CogroupingProcessor` выполняет одну из своих основных задач в методе `process()` (его можно найти в файле src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingProcessor.java) (листинг 6.14).

Листинг 6.14. Метод CogroupingProcessor.process()

```
@Override
    public void process(String key,
    ➥ Tuple<ClickEvent, StockTransaction> value) {

      Tuple<List<ClickEvent>, List<StockTransaction>> cogroupedTuple
      ➥ = tupleStore.get(key);
      if (cogroupedTuple == null) {
        cogroupedTuple =
        ➥ Tuple.of(new ArrayList<>(), new ArrayList<>());
      }

      if(value._1 != null) {
        cogroupedTuple._1.add(value._1);
      }

      if(value._2 != null) {
        cogroupedTuple._2.add(value._2);
      }

      tupleStore.put(key, cogroupedTuple);
  }
}
```

Инициализирует общий агрегирующий объект, если тот еще не существует

Если объект ClickEvent не пуст, добавляет его в список событий щелчков

Если объект StockTransaction не пуст, добавляет его в список биржевых транзакций

Помещает обновленный агрегирующий объект в хранилище состояния

Первый шаг в обработке входящих меньших агрегирующих объектов для общей совместной группировки — проверка, нет ли уже соответствующего объекта в хранилище состояния. Если нет, то мы создаем `Tuple` с пустыми коллекциями объектов `ClickEvent` и `StockTransaction`.

Далее мы проверяем входящие меньшие агрегирующие объекты и при наличии непустого объекта `ClickEvent` или `StockTransaction` добавляем его в общий агре-

гирующий объект. Последний шаг в методе `process()` — поместить `Tuple` обратно в хранилище состояния, обновив тем самым итоговый агрегирующий объект.

ПРИМЕЧАНИЕ

Хотя в нашем примере два узла-обработчика отправляют записи одному и обращаются к одному хранилищу состояния, беспокоиться о вопросах конкурентности не стоит. Помните, что родительские узлы-обработчики отправляют записи дочерним посредством обхода графа «вглубь», так что все узлы последовательно вызывают дочерние. Кроме того, в Kafka Streams на одну задачу приходится только один поток выполнения, так что ни о каких вопросах конкурентности речь никогда не идет.

Следующий шаг — выполнение пунктуации (см. файл src/main/java/bbejeck/chapter_6/processor/cogrouping/CogroupingPunctuator.java). Мы используем новый API, так что не будем рассматривать устаревший метод `Processor.punctuate` (листинг 6.15).

Листинг 6.15. Метод CogroupingPunctuator.punctuate()

```
// Опускаем объявление класса и конструктор для удобочитаемости
@Override
public void punctuate(long timestamp) {
  KeyValueIterator<String, Tuple<List<ClickEvent>,
  ➥ List<StockTransaction>>> iterator = tupleStore.all();

  while (iterator.hasNext()) {
    KeyValue<String, Tuple<List<ClickEvent>, List<StockTransaction>>>
    ➥ cogrouped = iterator.next();

    // Если хотя бы один из списков не пуст — отправляем результаты
    if (cogrouped.value != null &&
    ➥ (!cogrouped.value._1.isEmpty() ||
    ➥ !cogrouped.value._2.isEmpty())) {
      List<ClickEvent> clickEvents =
      ➥ new ArrayList<>(cogrouped.value._1);
      List<StockTransaction> stockTransactions =
      ➥ new ArrayList<>(cogrouped.value._2);

      context.forward(cogrouped.key,
      ➥ Tuple.of(clickEvents, stockTransactions));
      cogrouped.value._1.clear();
      cogrouped.value._2.clear();
      tupleStore.put(cogrouped.key, cogrouped.value);
    }
  }
  iterator.close();
}
```

Получаем итератор для всех совместно сгруппированных данных в хранилище состояния

Извлекаем следующий элемент совместно сгруппированных данных

Убеждаемся, что значение не пусто и хотя бы одна из коллекций содержит данные

Делаем защитные копии совместно сгруппированных коллекций

Отправляем далее ключ и агрегированные совместно сгруппированные данные

Помещаем очищенный кортеж обратно в хранилище состояния

При каждом вызове метода `punctuate` мы извлекаем все сохраненные записи в объект `KeyValueIterator` и начинаем извлекать по очереди все содержащиеся

в нем совместно сгруппированные результаты. Далее мы создаем защитные копии коллекций, создаем новый совместно сгруппированный кортеж `Tuple` и отправляем его далее по конвейеру. В данном случае мы отправляем совместно сгруппированные результаты в узел-сток. Наконец, мы удаляем текущие совместно сгруппированные результаты и снова сохраняем кортеж в хранилище, готовые к поступлению следующей порции записей.

Мы закончили описание функциональности совместной группировки и можем теперь завершить создание топологии.

Добавление хранилища состояния

Как вы уже видели, для агрегирования данных в потоковых приложениях Kafka требуется сохранение состояния. Для должного функционирования `CogroupingProcessor` необходимо добавить к нему хранилище состояния. На рис. 6.12 показана обновленная топология.

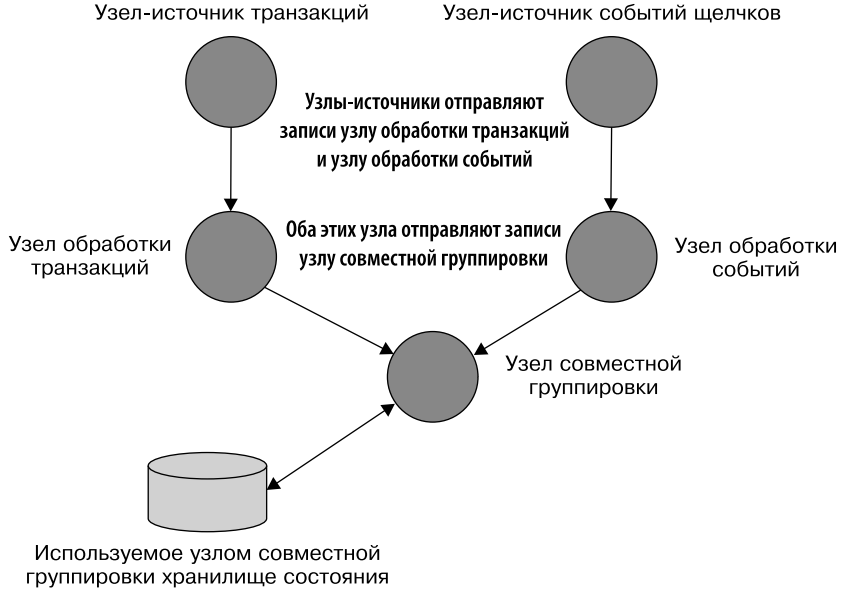


Рис. 6.12. Добавляем хранилище состояния к узлу совместной группировки в топологии

Посмотрим теперь на код добавления хранилища состояния (его можно найти в файле src/main/java/bbejeck/chapter_6/CoGroupingApplication.java) (листинг 6.16).

В этом коде мы добавили в топологию постоянное хранилище состояния. Нам нужно именно постоянное хранилище, поскольку изредка возможны обновления некоторых ключей. В случае хранилищ в оперативной памяти и хранилищ типа

LRU (с вытеснением давно не используемых данных) редко используемые ключи и значения могут со временем удаляться, а тут нам требуется возможность извлечь информацию для любого ключа, с которым мы ранее работали.

Листинг 6.16. Добавление узла хранилища состояния

```
// Этот фрагмент располагается выше в исходном коде,
// сюда он включен для того, чтобы был виден контекст
Map<String, String> changeLogConfigs = new HashMap<>();
changeLogConfigs.put("retention.ms", "120000");
changeLogConfigs.put("cleanup.policy", "compact,delete");

KeyValueBytesStoreSupplier storeSupplier =
➥ Stores.persistentKeyValueStore(TUPLE_STORE_NAME);
StoreBuilder<KeyValueStore<String,
➥ Tuple<List<ClickEvent>, List<StockTransaction>>>> storeBuilder =
        Stores.keyValueStoreBuilder(storeSupplier,
        Serdes.String(),
        eventPerformanceTuple)
➥ .withLoggingEnabled(changeLogConfigs);

topology.addStateStore(storeBuilder, "CoGrouping-Processor");
```

Задаем длительность хранения записей, а также использование сжатия и удаления для очистки

Создаем объект-поставщик для постоянного хранилища (RocksDB)

Создаем объект-строитель для хранилища

Добавляем настройки журнала изменений в объект-строитель

Добавляем хранилище в топологию с указанием названия узла-обработчика, который будет к нему обращаться

СОВЕТ

В первых трех строках листинга 6.16 создаются настройки для хранилища состояния с целью не допустить чрезмерного разрастания журнала изменений. Помните: для топиков журналов изменений можно задавать любые подходящие для обычных топиков настройки.

Этот код достаточно прост. Стоит отметить, однако, что единственный узел, который может обращаться к хранилищу состояния, — `CoGroupingProcessor`.

Нам остался один шаг до завершения топологии — обеспечить возможность чтения результатов совместной группировки.

Добавление узла-стока

Чтобы от топологии совместной группировки была польза, необходимо записывать данные в топик (или консоль). Модифицируем топологию еще один, последний, раз, как показано на рис. 6.13.

ПРИМЕЧАНИЕ

В нескольких примерах я говорил о добавлении узла-стока, но в исходном коде сток выводит данные в консоль; код для стока, записывающего данные в топик, закомментирован. При разработке я попеременно использую узел-сток, записывающий в топик, и узел-сток, выводящий результаты в stdout.

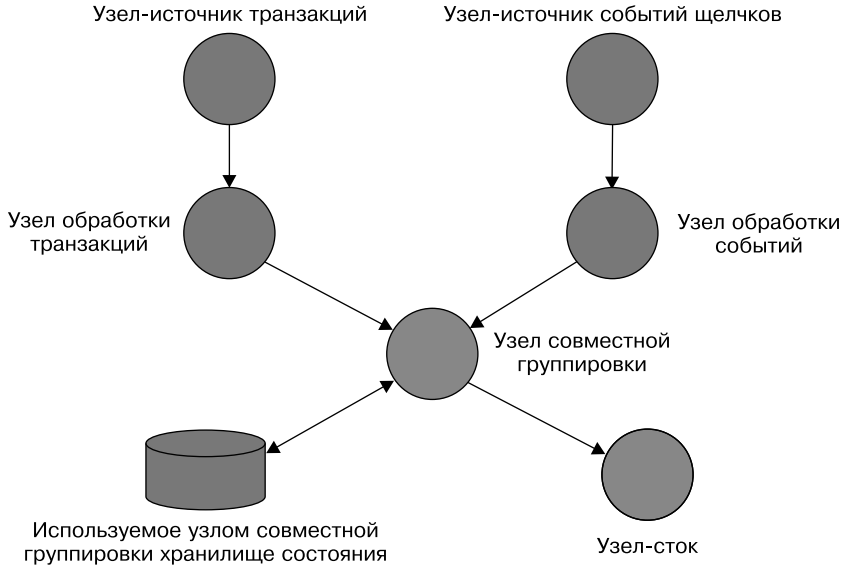


Рис. 6.13. Добавление узла-стока завершает создание топологии совместной группировки

Теперь результаты совместной группировки записываются в топик для применения при дальнейшем анализе. Вот соответствующий код (который можно найти в файле src/main/java/bbejeck/chapter_6/CoGroupingApplication.java) (листинг 6.17).

Листинг 6.17. Узел-сток и узел вывода в консоль

```
.addSink("Tuple-Sink",
         "cogrouped-results",
          stringSerializer,
          tupleSerializer,
         "CoGrouping-Processor");
```

Узел-сток записывает совместно сгруппированные кортежи в топик

```
topology.addProcessor("Print",
                      new KStreamPrinter("Co-Grouping"),
                      "CoGrouping-Processor");
```

Этот узел-обработчик выводит результаты в stdout для использования при разработке

В этом последнем элементе топологии мы добавили узел-сток — в виде дочернего узла `CoGrouping-Processor`. Напомню, что при использовании API узлов-обработчиков порядок описания узлов не устанавливает связи «предок — потомок». Для задания такой связи необходимо указывать в качестве родительских узлов названия ранее созданных узлов-обработчиков.

Мы завершили создание узла совместной группировки. Главное, что вам стоит запомнить из этого раздела: хотя использование API узлов-обработчиков требует написания большего объема кода, оно и обеспечивает гибкость, достаточную для создания практически любой мыслимой потоковой топологии.

В завершение этой главы посмотрим, как можно интегрировать функциональность API узлов-обработчиков в приложение Kafka Streams.

6.5. Интеграция API узлов-обработчиков и API Kafka Streams

До сих пор мы изучали Kafka Streams и API узлов-обработчиков по отдельности, но кто сказал, что эти подходы нельзя сочетать? Давайте разберемся, зачем нам может понадобиться подобное сочетание.

Допустим, вы уже давно используете как Kafka Streams, так и API узлов-обработчиков. Вам больше нравится подход Kafka Streams, но вам хотелось бы включить некоторые свои уже готовые узлы-обработчики в приложение Kafka Streams, поскольку они дают нужные вам возможности управления на более низком уровне.

API Kafka Streams предоставляет три метода, с помощью которых можно интегрировать созданную с помощью API узлов-обработчиков функциональность: `KStream.process`, `KStream.transform` и `KStream.transformValues`. Вам должен быть знаком этот подход, поскольку вы уже работали с интерфейсом `ValueTransformer` в подразделе 4.2.2.

Метод `KStream.process` создает концевой узел, в то время как методы `KStream.transform` и `KStream.transformValues` возвращают новый экземпляр `KStream`, благодаря чему вы можете продолжать добавлять новые узлы-обработчики к данному узлу. Отмечу также, что методы преобразования сохраняют состояние, поэтому при их использовании необходимо передавать название хранилища состояния. Поскольку `KStream.process` создает концевой узел, то обычно используется или `KStream.transform`, или `KStream.transformValues`.

Там вы можете заменить свой экземпляр `Processor` на `Transformer`. Основное различие между этими двумя интерфейсами состоит в том, что метод, в котором происходят основные действия у `Processor`, — `process()` с возвращаемым типом `void`, а у `Transformer` это `transform()` с возвращаемым типом `R`. Оба интерфейса предоставляют одинаковую семантику пунктуации.

В большинстве случаев для замены `Processor` достаточно переместить код логики из метода `Processor.process` в метод `Transformer.transform`. Нужно учесть только необходимость возврата значения, но можно вернуть пустое значение и отправлять результаты далее с помощью метода `ProcessorContext.forward`.

СОВЕТ

Преобразователь возвращает значение: в данном случае значение пусто, так что оно отфильтровывается, а для отправки нескольких значений далее по конвейеру используется метод ProcessorContext.forward. Можно также вернуть List<KeyValue<K,V>>, присоединив к нему вызов flatMap или flatMapValues для отправки далее по конвейеру отдельных записей. Пример этого вы можете найти в файле src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorMultipleValuesApplication.java. Для завершения замены экземпляра Processor необходимо подключить экземпляр Transformer (или ValueTransformer) с помощью метода KStream.transform (или KStream.transformValues).

Отличный пример сочетания API `KStream` и API узлов-обработчиков можно найти в src/main/java/bbejeck/chapter_6/StockPerformanceStreamsAndProcessorApplication.java. Я не включал этот пример в текст книги, поскольку его логика, по большому счету, не отличается от логики примера `StockPerformanceApplication` из подраздела 6.3.1. Вы можете взглянуть на него, если интересно. Кроме того, вы найдете версию исходного приложения ZMart на основе API узлов-обработчиков в файле src/main/java/bbejeck/chapter_6/ZMartProcessorApp.java.

Резюме

- API узлов-обработчиков обеспечивает большую гибкость за счет большего объема кода.
- Хотя код при использовании API узлов-обработчиков менее лаконичен, чем в случае API Kafka Streams, применять его достаточно просто и «под капотом» API Kafka Streams задействует именно API узлов-обработчиков.
- При выборе используемого API учтите и возможность применения API Kafka Streams с интеграцией низкоуровневых методов (`process()`, `transform()`, `transformValues()`) по мере необходимости.

Пока мы охватили вопрос *создания* приложений с помощью Kafka Streams. Следующий этап — их оптимальная настройка, мониторинг для обеспечения максимальной производительности и обнаружение потенциальных проблем.

Часть III
Администрирование Kafka Streams

В этих главах мы переключим внимание на оценку производительности приложений Kafka Streams. Вы также узнаете, как организовать мониторинг и тестирование кода Kafka Streams, чтобы убедиться, что он работает ожидаемым образом и достойно обрабатывает ошибки.

7 Мониторинг и производительность

В этой главе:

- основы мониторинга Kafka;
- перехват сообщений;
- оценка производительности;
- отслеживание состояния приложения.

Вы уже научились создавать приложения Kafka Streams с нуля, поработали с высокоуровневым DSL Kafka Streams и ощутили мощь декларативного API. Вы также узнали об API узлов-обработчиков и поняли, что иногда можно отказаться от некоторых удобств ради более широких возможностей контроля при написании потокового приложения.

Время переключиться на немного другую тематику. Вам предстоит надеть шляпу частного детектива и взглянуть на свое приложение с иной точки зрения. Мы переключимся с того, *как* заставить приложение работать, на то, что *происходит* в приложении. В определенном смысле проще всего — создать приложение. Гораздо более сложная задача — добиться его успешного запуска, должного масштабирования и правильного функционирования. Как бы вы ни старались, всегда возникают какие-либо непредвиденные обстоятельства.

В этой главе вы научитесь проверять состояние своего приложения Kafka Streams. Вы узнаете, как оценить производительность приложения и найти узкие места производительности. Я расскажу вам также о методах, применяемых для оповещения о различных состояниях приложения и для просмотра структуры топологии. Вы узнаете, какие существуют метрики, как их собирать и как просматривать собранные метрики во время работы приложения. Начнем с мониторинга приложения Kafka Streams.

7.1. Основы мониторинга Kafka

Поскольку API Kafka Streams — составная часть Kafka, то, само собой, мониторинг вашего приложения потребует и мониторинга Kafka. Полномасштабное наблюдение за кластером Kafka — объемная тема, так что мы ограничим наше обсуждение

производительности Kafka вопросами мониторинга потребителей и генераторов Kafka. Более подробную информацию о мониторинге кластера Kafka можно найти в документации (https://kafka.apache.org/documentation/#monitoring).

ПРИМЕЧАНИЕ

Стоит отметить, что для оценки производительности Kafka Streams необходимо измерить и производительность Kafka. Иногда наше обсуждение производительности будет немного затрагивать и Kafka, но, поскольку данная книга посвящена Kafka Streams, именно на Kafka Streams мы и сосредоточим наше внимание.

7.1.1. Оценка производительности потребителей и генераторов

Мы начнем наше обсуждение производительности потребителей и генераторов с рис. 7.1, иллюстрирующего один из базовых аспектов, заботящих нас в плане производительности потребителей и генераторов. Как вы можете видеть, производительность как генератора, так и потребителя зависит от пропускной способности. Различия ровно таковы, что можно считать их двумя сторонами одной монеты.

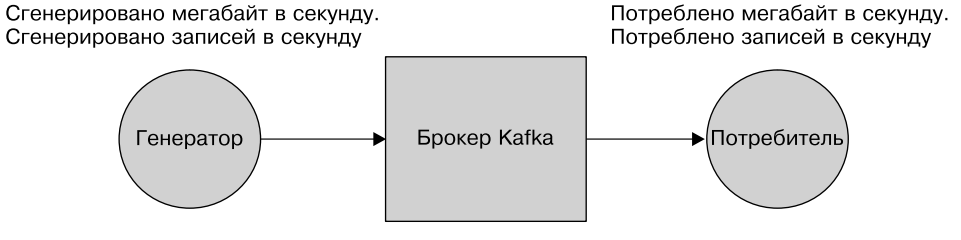


Рис. 7.1. Вопросы производительности потребителей и генераторов, запись на брокер и чтение с него

В плане генераторов нас интересует в основном скорость отправки сообщений брокеру. Разумеется, чем выше пропускная способность — тем лучше.

В плане потребителей нас также интересует производительность, а именно: насколько быстро мы можем читать сообщения от брокера. Но существует и другой показатель производительности потребителя: отставание потребителя. Взгляните на рис. 7.2. На нем производительность генератора и потребителя оценивается несколько с другой точки зрения, чем на рис. 7.1.

Как видите, нас интересует, какой объем сообщений и насколько быстро могут наши генераторы отправлять брокеру, а одновременно и то, насколько быстро наши потребители могут прочитать сообщения с него. Разница между скоростью записи генераторами сообщений на брокер и скоростью чтения потребителями сообщений с него называется *отставанием потребителя* (consumer lag).

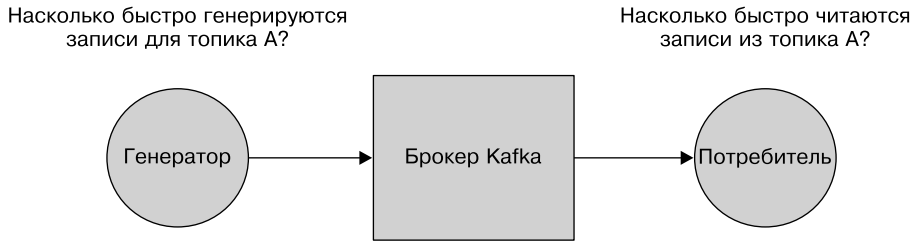


Рис. 7.2. Пересмотр вопроса оценки производительности генератора и потребителя Kafka

Рисунок 7.3 демонстрирует, что отставание потребителя представляет собой разницу между последним зафиксированным потребителем смещением и последним смещением записанного на брокер сообщения. Некоторое отставание потребителя неизбежно, но в идеале потребитель наверстывает упущенное или по крайней мере отставание не растет.

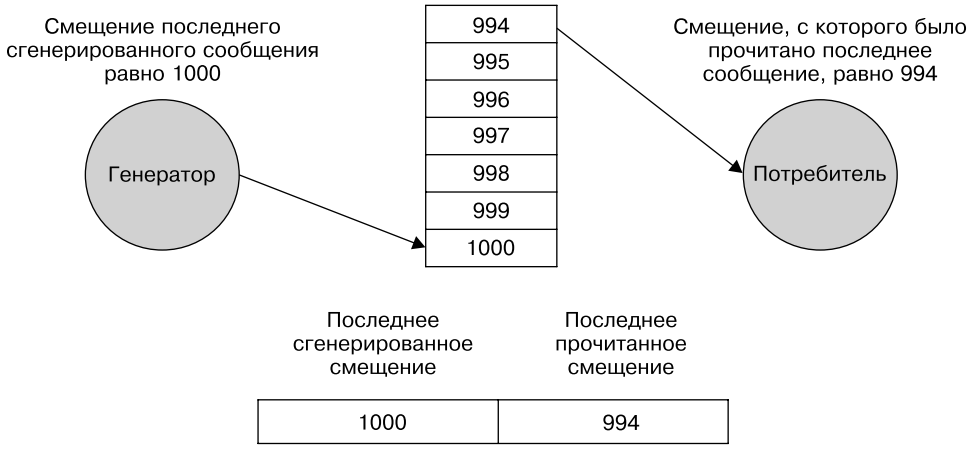


Рис. 7.3. Отставание потребителя равно разнице зафиксированных потребителем и записанных генератором смещений

Теперь, когда мы определились с параметрами производительности для потребителей и генераторов, давайте разберемся с их мониторингом в целях диагностики и устранения проблем с производительностью.

7.1.2. Проверка отставания потребителя

Для проверки отставания потребителя Kafka предоставляет удобную утилиту командной строки `kafka-consumer-groups.sh`, расположенную в каталоге <kafka-install-dir>/bin. У этого сценария есть несколько опций, из которых мы рассмотрим тут `list` и `describe`. Этих двух опций вам хватит для получения всей необходимой информации о производительности группы потребителей.

Во-первых, воспользуемся командой `list` для вывода списка всех активных групп потребителей. Результаты ее выполнения приведены на рис. 7.4.

```
<kafka-install-dir>/bin/kafka-consumer-groups.sh \
                  --bootstrap-server localhost:9092 \
                  --list
```

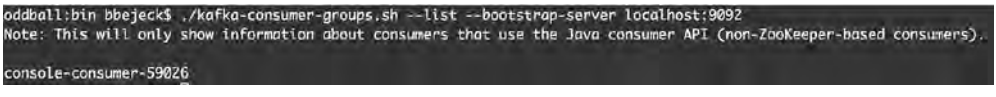


Рис. 7.4. Вывод перечня доступных групп потребителей из командной строки

Получив эту информацию, можно выбрать название группы потребителей и выполнить следующую команду:

```
<kafka-install-dir>/bin/kafka-consumer-groups.sh \
                  --bootstrap-server localhost:9092 \
                  --group <GROUP-NAME> \
                  --describe
```

Результаты (состояние работы потребителя) показаны на рис. 7.5.

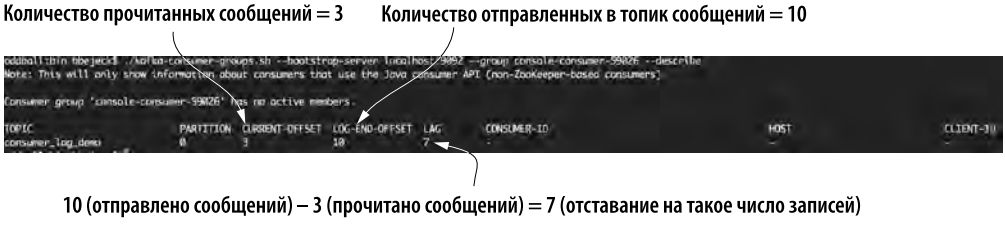


Рис. 7.5. Состояние группы потребителей

Эти результаты показывают, что отставание потребителя невелико. Наличие отставания потребителя не всегда означает наличие проблем — потребители читают сообщения пакетами и не получают следующий пакет до окончания обработки текущего. Обработка записей занимает определенное время, так что небольшое отставание неудивительно.

Небольшое либо постоянное отставание вполне нормально, но когда отставание растет со временем — это признак того, что потребителю требуется больше ресурсов. Например, может понадобиться увеличить число секций, а значит, и число потоков

выполнения, потребляющих данные из топика. Или, возможно, обработка прочитанного сообщения требует слишком больших ресурсов. После потребления сообщения его можно передать в асинхронную очередь, из которой другой поток выполнения сможет его извлечь для обработки.

Из этого раздела вы узнали, как определить, насколько быстро потребитель читает сообщения с брокера. Далее мы слегка углубимся в наблюдение за поведением в целях отладки — вы увидите, как перехватывать отправляемые генераторами и получаемые потребителями сообщения *до* отправки/потребления записей приложением Kafka Streams.

7.1.3. Перехват информации о поведении генераторов и потребителей

В начале 2016 года появилось предложение об усовершенствовании Kafka № 42 (Kafka Improvement Proposal 42, KIP-42), говорившее о возможности мониторинга (перехвата) информации о поведении клиентов (генераторов и потребителей). Целью этого KIP было предоставить «возможность быстрого развертывания инструментов для наблюдения, оценки и мониторинга поведения клиентов Kafka, вплоть до уровня сообщения»[1].

Хотя перехватчики обычно не первый эшелон средств отладки, они могут оказаться полезны при наблюдении за потоковым приложением Kafka и представляют собой ценное дополнение к вашему инструментарию. Прекрасный пример использования перехватчика (для генератора) — отслеживание смещений сообщений, отправляемых приложением Kafka Streams обратно в Kafka.

ПРИМЕЧАНИЕ

> Kafka Streams может потреблять и генерировать произвольное число типов ключей и значений, поэтому его внутренние Consumer и Producer используют байтовые массивы ключей и значений (byte[]) и поэтому всегда обрабатывают несериализованные данные. А если данные сериализованные, то для их просмотра обязательно требуется дополнительный шаг сериализации/десериализации.

Начнем с обсуждения перехватчика для потребителя.

Перехватчик для потребителя

Перехватчик для потребителя позволяет осуществлять перехват в двух точках доступа. Первая — метод `ConsumerInterceptor.onConsume()`, читающий объект `ConsumerRecords` в промежутке между извлечением записей с брокера и возвра-

[1] Apache Kafka, KIP-42: Add Producer and Consumer Interceptors («KIP-42: Добавление перехватчиков для генераторов и потребителей»), http://mng.bz/g8oX.

щением сообщений из метода `Consumer.poll()`. Представление о том, где перехватчик потребителя производит перехват, вы можете получить из следующего псевдокода:

```
ConsumerRecords<String, String> poll(long timeout) {
   ConsumerRecords<String, String> consumerRecords =
➥  ... потребляем сообщения
   return interceptors.onConsume(consumerRecords);
```

Извлекаем новые записи с брокера

Записи проходят через цепочку операций перехватчика, после чего возвращаются результаты

Хотя этот псевдокод не имеет ничего общего с настоящим кодом `KafkaConsumer`, он отлично иллюстрирует концепцию. Перехватчики принимают в качестве параметра возвращаемый с брокера внутри метода `Consumer.poll()` объект `consumerRecords` и получают возможность выполнить над ним любые операции, включая фильтрацию и модификацию, до того как `KafkaConsumer` вернет эти записи из метода `poll`.

`ConsumerInterceptor` указываются через ключ конфигурации `ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG` и принимают на входе объект `Collection`, состоящий из одного или нескольких классов реализации `ConsumerInterceptor`. При этом несколько перехватчиков соединяются цепочкой и выполняются в указанном в настройках порядке.

`ConsumerInterceptor` принимает в качестве параметра и возвращает экземпляр `ConsumerRecords`. В случае нескольких перехватчиков возвращаемый из одного перехватчика объект `ConsumerRecords` служит входным параметром для следующего перехватчика в цепочке. Таким образом, все произведенные одним перехватчиком изменения передаются далее по цепочке.

Обработка исключений играет важную роль при соединении цепочкой нескольких перехватчиков. Если в перехватчике возникает исключение, он заносит в журнал сообщение об ошибке, но *не* обрывает выполнение цепочки. А значит, объект `ConsumerRecords` отправляется далее и проходит через оставшиеся перехватчики.

Допустим, у нас есть три перехватчика: А, Б и В. Все три модифицируют записи, причем для каждого из двух последних важны изменения, выполненные его предшественниками. Но в случае возникновения в перехватчике А исключения объект `ConsumerRecords` попадет далее в перехватчики Б и В без ожидаемых изменений, после чего результаты работы цепочки перехватчиков окажутся некорректными. Поэтому лучше не полагаться в перехватчиках на изменения, вносимые в `ConsumerRecords` предыдущим перехватчиком в цепочке.

Вторая точка перехвата — метод `ConsumerInterceptor.onCommit()`. После фиксации потребителем смещений на брокере брокер возвращает объект `Map<TopicPartition, OffsetAndMetadata>`, содержащий информацию о топике, секции и зафиксированных смещениях, а также соответствующие метаданные (время фиксации и т. п.). Информация о фиксации может пригодиться для отслеживания. В листинге 7.1 приведен пример простого `ConsumerInterceptor`, используемого для

журналирования (его можно найти в файле src/main/java/bbejeck/chapter_7/interceptors/StockTransactionConsumerInterceptor.java).

Листинг 7.1. Журналирующий перехватчик для потребителя

```
public class StockTransactionConsumerInterceptor implements
➥ ConsumerInterceptor<Object, Object> {

    // Некоторые подробности были опущены для простоты
    private static final Logger LOG =
    ➥ LoggerFactory.getLogger(StockTransactionConsumerInterceptor.class);

    public StockTransactionConsumerInterceptor() {
        LOG.info("Built StockTransactionConsumerInterceptor");
    }

    @Override
    public ConsumerRecords<Object, Object> onConsume
    ➥ (ConsumerRecords<Object, Object> consumerRecords) {
        LOG.info("Intercepted ConsumerRecords {}",
                 buildMessage(consumerRecords.iterator()));
        return consumerRecords;
    }

    @Override
    public void onCommit(Map<TopicPartition, OffsetAndMetadata> map) {
         LOG.info("Commit information {}",  map);
    }
```

Заносит в журнал записи и метаданные перед обработкой записей

Заносит в журнал информацию о фиксации потребителем Kafka Streams смещений на брокере

Рассмотрим теперь перехват с точки зрения генераторов.

Перехватчики генераторов

`ProducerInterceptor` ведет себя аналогично `ConsumerInterceptor`, у него также есть две точки доступа: `ProducerInterceptor.onSend()` и `ProducerInterceptor.onAcknowledgement()`. С помощью метода `onSend()` перехватчик может выполнить любое нужное действие, включая изменение объекта `ProducerRecord`. Каждый из перехватчиков генераторов в цепочке получает объект, возвращаемый предыдущим перехватчиком.

Обработка исключений производится так же, как и на стороне потребителей, так что для нее актуальны те же предостережения. Метод `ProducerInterceptor.onAcknowledgement()` вызывается при подтверждении брокером получения записи. Метод `onAcknowledgement()` вызывается и после того, как отправка записи завершилась неудачей.

В следующем листинге показан простой пример журналирующего перехватчика генератора (его можно найти в файле src/main/java/bbejeck/chapter_7/interceptors/ZMartProducerInterceptor.java).

`ProducerInterceptor` указывается через ключ конфигурации `ProducerConfig.INTERCEPTOR_CLASSES_CONFIG` и принимает на входе объект `Collection`, состо-

ящий из одного или нескольких классов реализации `ProducerInterceptor`. При этом несколько перехватчиков соединяются цепочкой и выполняются в указанном в настройках порядке.

Листинг 7.2. Журналирующий перехватчик для генератора

```
public class ZMartProducerInterceptor implements
➥ ProducerInterceptor<Object, Object> {

  // Некоторые подробности были опущены для простоты
  private static final Logger LOG =
  ➥ LoggerFactory.getLogger(ZMartProducerInterceptor.class);

  @Override
  public ProducerRecord<Object, Object> onSend(ProducerRecord<Object,
  ➥ Object> record) {
   LOG.info("ProducerRecord being sent out {} ", record);
   return record;
  }

  @Override
  public void onAcknowledgement(RecordMetadata metadata, Exception
    ➥ exception) {
    if (exception != null) {
        LOG.warn("Exception encountered producing record {}",
    ➥ exception);
    } else {
        LOG.info("record has been acknowledged {} ", metadata);
    }
  }
```

Заносим информацию в журнал непосредственно перед отправкой сообщения брокеру

Заносим в журнал информацию о том, подтвердил брокер получение или произошла ошибка (на стороне брокера) во время фазы генерации

СОВЕТ

При настройке перехватчиков в приложении Kafka Streams необходимо указать перед названиями свойств перехватчиков потребителей и генераторов props.put(StreamsConfig.consumerPrefix(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG) и props.put(StreamsConfig.producerPrefix(ProducerConfig.INTERCEPTOR_CLASSES_CONFIG) соответственно.

Посмотреть перехватчики в действии можно в классе src/main/java/bbejeck/chapter_7/StockPerformanceStreamsAndProcessorMetricsApplication.java, где используется перехватчик потребителя, и в классе src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java, где используется перехватчик генератора. Оба класса включают необходимые для применения перехватчиков настройки.

Отмечу, что, поскольку перехватчики обрабатывают каждую запись в приложении Kafka Streams, объем выводимой журналирующими перехватчиками информации весьма значителен. Результаты работы перехватчиков выводятся в файлы consumer_interceptor.log и producer_interceptor.log, располагающиеся в подкаталоге logs корневого каталога установки исходного кода.

Мы потратили уже немало времени на изучение метрик производительности потребителей и перехват записей, поступающих в приложение Kafka Streams

и исходящих из него. Но эта информация дает лишь общую картину, причем *извне* приложения Kafka Streams. Заглянем внутрь приложения Kafka Streams и посмотрим, что происходит у него «под капотом». Следующий наш шаг — оценка производительности внутри топологии посредством сбора метрик.

7.2. Метрики приложения

Когда речь заходит об оценке производительности приложения, вы можете получить представление о том, сколько времени нужно для обработки одной записи, да и оценка длительности сквозной задержки — неплохой показатель общей производительности. Но для повышения производительности нужно знать точно, в каком месте возникает затор.

Оценка производительности очень важна для потоковых приложений. Сам факт использования потокового приложения подразумевает ваше желание обрабатывать данные сразу же после их появления. Вполне логично, что если вашему бизнесу требуется потоковое решение, то вы хотели бы получить самый *эффективный* и *безошибочный* процесс потоковой обработки, какой только доступен в рамках вашего бюджета.

Прежде чем перейти к обсуждению собственно метрик, вернемся к одному из созданных нами в главе 3 приложений, а именно к продвинутому приложению ZMart. Это хороший кандидат на измерение метрик, поскольку содержит несколько узлов обработки, поэтому мы воспользуемся его топологией для следующего примера. На рис. 7.6 показана созданная нами топология.

С учетом топологии ZMart рассмотрим следующие категории метрик:

- метрики потока выполнения:
  - среднее время выполнения операций `commit`, `poll`, `process`;
  - количество созданных задач в секунду, завершенных задач в секунду;
- метрики задачи:
  - среднее число фиксаций в секунду;
  - среднее время фиксации;
- метрики узлов-обработчиков:
  - средняя и максимальная длительность обработки;
  - среднее число операций `process` в секунду;
- метрики хранилищ состояния:
  - среднее время выполнения операций `put`, `get` и `flush`;
  - среднее число операций `put`, `get` и `flush` в секунду.

Обратите внимание, что это далеко не исчерпывающий список возможных метрик. Я выбрал именно данные метрики потому, что они прекрасно охватывают наиболее частые сценарии в смысле производительности. Полный список вы можете найти на сайте Confluent: http://mng.bz/4bcA.

Теперь, когда мы уже знаем, что будем измерять, выясним, как нам захватить эту информацию.

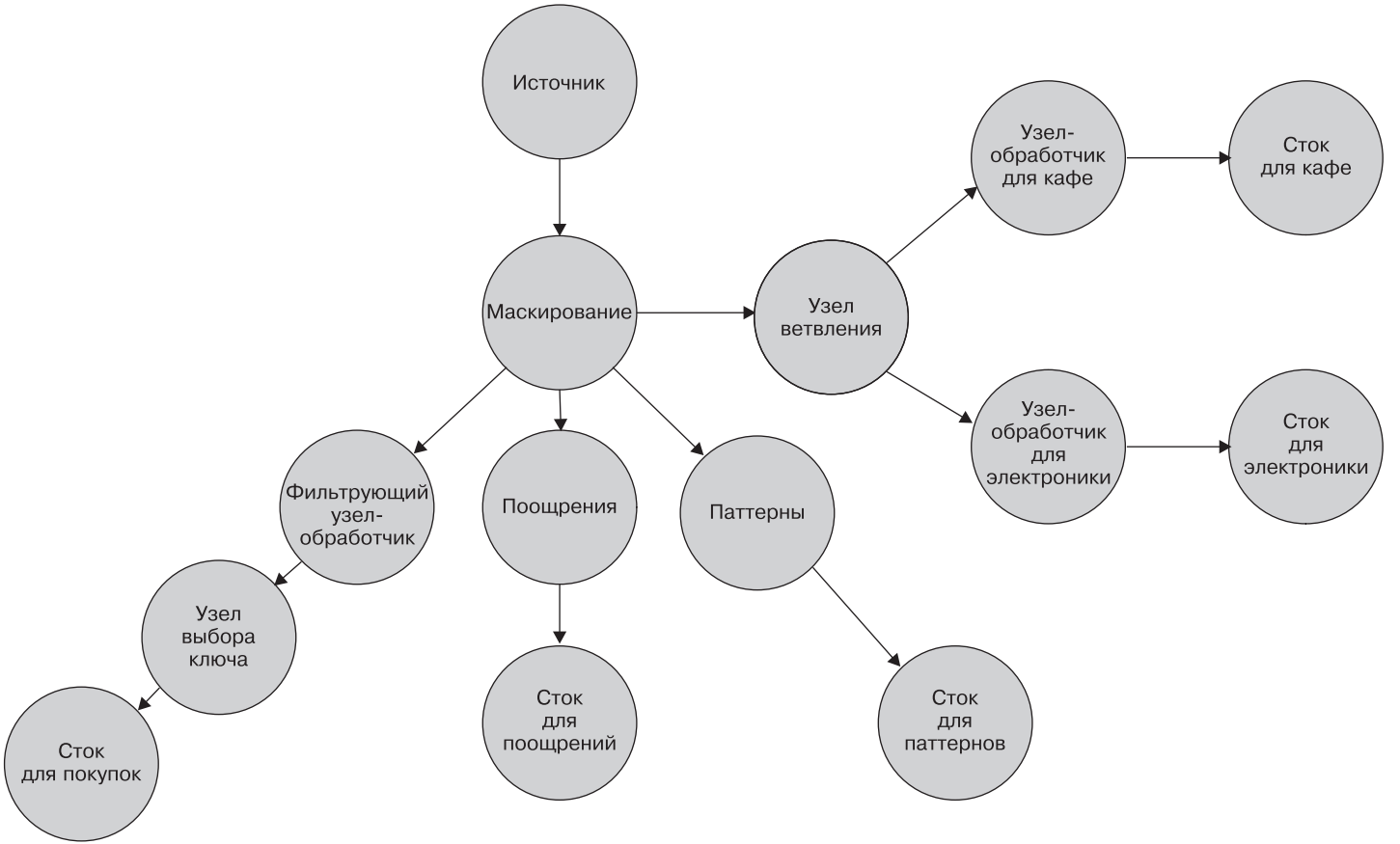


Рис. 7.6. Топология продвинутого приложения ZMart с множеством узлов

7.2.1. Настройки метрик

В Kafka Streams есть уже готовый механизм сбора метрик производительности. Чаще всего достаточно только указать несколько параметров конфигурации. Поскольку сбор метрик сам по себе снижает производительность, существует два его уровня: `INFO` и `DEBUG`. Сбор отдельной метрики может требовать не слишком большого расхода ресурсов, но, если учесть, что некоторые метрики могут относиться к операциям, связанным с обработкой каждой проходящей через приложение Kafka Streams записи, суммарное воздействие на производительность может оказаться весьма существенным.

Уровни метрик подобны уровням журналирования. При поиске причин проблемы или при наблюдении за поведением приложения необходимо больше информации, так что можно воспользоваться уровнем `DEBUG`. В остальное время вся информация не требуется и достаточно уровня `INFO`.

Обычно в промышленной эксплуатации уровень `DEBUG` не используется, поскольку падение производительности оказалось бы слишком большим. Ранее перечисленные метрики доступны на различных уровнях, как показано в табл. 7.1. Как вы можете видеть, метрики потоков выполнения доступны на обоих уровнях, в то время как остальные категории метрик собираются только при применении уровня `DEBUG`.

Таблица 7.1. Возможность использования метрик по уровням

Категория метрик	**DEBUG**	**INFO**
Поток выполнения	×	×
Задача	×	
Узел-обработчик	×	
Хранилище состояния	×	
Кэш записей	×	

Уровень задается при описании конфигурации приложения Kafka Streams. Этот параметр конфигурации всегда был там, во всех конфигурациях наших приложений, но до сих пор мы использовали значение по умолчанию. Уровень сбора метрик по умолчанию — `INFO`.

Модифицируем настройки продвинутого приложения ZMart и включим сбор всех метрик (src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java) (листинг 7.3).

Мы включили сбор и запись метрик уровня `DEBUG`. Главное, что стоит запомнить из этого раздела: существуют встроенные метрики для всесторонней оценки производительности приложения Kafka Streams, но следует внимательно обдумать возможные последствия для производительности, прежде чем включать сбор метрик на уровне `DEBUG`.

Листинг 7.3. Модификация настроек для сбора метрик уровня DEBUG

```
private static Properties getProperties() {
    Properties props = new Properties();
    props.put(StreamsConfig.CLIENT_ID_CONFIG,
➥   "metrics-client-id");
    props.put(ConsumerConfig.GROUP_ID_CONFIG,
➥   "metrics-group-id");
    props.put(StreamsConfig.APPLICATION_ID_CONFIG,
➥   "metrics-app-id");
    props.put(StreamsConfig.METRICS_RECORDING_LEVEL_CONFIG,
➥   "DEBUG");
    props.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
➥   "localhost:9092");
  return props;
}
```

Идентификатор клиента

Идентификатор группы

Идентификатор приложения

Устанавливает уровень DEBUG сбора метрик

Параметры соединения для брокеров

Мы обсудили, какие существуют метрики и как их собирать. Следующий шаг — просмотр собранных метрик.

7.2.2. Как получить доступ к собранным метрикам

Метрики в приложениях Kafka Streams собираются, после чего отправляются в подпрограммы генерации отчетов. Как вы могли догадаться, Kafka Streams предоставляет подпрограмму генерации отчетов по умолчанию через Java Management Extensions (JMX).

Если вы уже включили сбор метрик на уровне `DEBUG`, то ничего не остается, кроме как изучить собранное. Помните, однако, что для работы JMX требуется работающее в данный момент приложение, так что просматривать метрики мы будем во время работы приложения.

СОВЕТ

Можно также получить доступ к метрикам программным способом. Пример этого вы найдете в файле src/main/java/bbejeck/chapter_7/StockPerformanceStreamsAndProcessorMetricsApplication.java.

Вероятно, вы уже имели дело с технологией JMX или по крайней мере слышали о ней. Я расскажу вкратце, с чего начать ее использование, но, если вы уже хорошо с ней знакомы, можете следующий раздел смело пропускать.

7.2.3. Использование JMX

JMX — стандартный способ наблюдения за поведением программ, запущенных на виртуальной машине Java. С помощью JMX можно также узнать, насколько эффективно работает виртуальная машина Java (Java VM). По сути, JMX — это

инфраструктура, с помощью которой можно сделать видимыми различные части вашей работающей программы.

К счастью, для такого мониторинга не требуется писать никакого кода. Достаточно просто подключиться с помощью утилит Java VisualVM (http://mng.bz/euif), JConsole (http://mng.bz/Ea71) или Java Mission Control (http://mng.bz/0r5B).

СОВЕТ

Java Mission Control (JMC) — замечательная утилита для мониторинга с большими возможностями, однако для промышленной эксплуатации она требует получения лицензии. Поскольку JMC поставляется в комплекте JDK, ее можно запустить прямо из командной строки с помощью команды `jmc` (конечно, если каталог bin JDK внесен в ваш системный путь). Кроме того, необходимо добавить при запуске вашего потокового приложения Kafka следующие флаги: -XX:+UnlockCommercialFeatures -XX:+FlightRecorder.

Поскольку самой простой в использовании является утилита JConsole, то на ней мы пока и остановимся.

Что такое JMX?

В документе Lesson: Overview of the JMX Technology («Обзор технологии JMX», http://mng.bz/Ej29) Oracle сообщает:

«Технология Java Management Extensions представляет собой стандартную часть платформы Java SE (Стандартный выпуск), добавленную в платформу в выпуске 5.0.

Технология JMX позволяет легко и единообразно управлять такими ресурсами, как приложения, устройства и сервисы. А поскольку это динамическая технология, ее можно использовать для мониторинга и управления ресурсами при их создании, установке и реализации. Можно также применять технологию JMX для мониторинга и управления виртуальной машиной Java.

В спецификации JMX описываются архитектура, паттерны проектирования, API и сервисы языка Java, предназначенные для управления приложениями и сетями и их мониторинга».

Запуск JConsole

JConsole поставляется вместе с JDK, так что если у вас установлен выпуск Java, то есть и JConsole. Запуск JConsole требует всего лишь выполнения команды `jconsole` из командной строки (Java должна быть включена в вашу системную переменную `path`). После запуска появляется GUI, как показано на рис. 7.7. А дальше можно сразу использовать ее для просмотра информации метрик!

Рис. 7.7. Начальное меню JConsole

Начинаем мониторинг запущенной программы

Если вы посмотрите в центр GUI JConsole, то увидите диалоговое окно New Connection (Новое подключение). Эта отправная точка работы с JConsole показана на рис. 7.8. Пока нас интересуют только процессы Java, перечисленные в разделе Local Process (Локальный процесс).

ПРИМЕЧАНИЕ

JConsole можно применять для мониторинга удаленных приложений, равно как и защищенный доступ к JMX. На рис. 7.8 можно увидеть текстовые поля Remote Process (Удаленный процесс), Username (Имя пользователя) и Password (Пароль). В этой книге, однако, мы ограничимся локальным доступом при разработке. В Интернете можно найти множество инструкций по удаленному и защищенному доступу к JConsole, и в качестве отправной точки отлично подойдет документация Oracle (http://mng.bz/Ea71).

Если вы еще не запустили свое приложение Kafka Streams — самое время сделать это. Далее, чтобы оно отобразилось в окне Local Process (Локальный процесс), нажмите Connection ▸ New Connection (Подключение ▸ Новое подключение), как на рис. 7.8. При этом список перечисленных ниже Local Process (Локальный процесс) процессов обновится и вы увидите в нем свое приложение Kafka Streams. Щелкните на соответствующем процессе дважды.

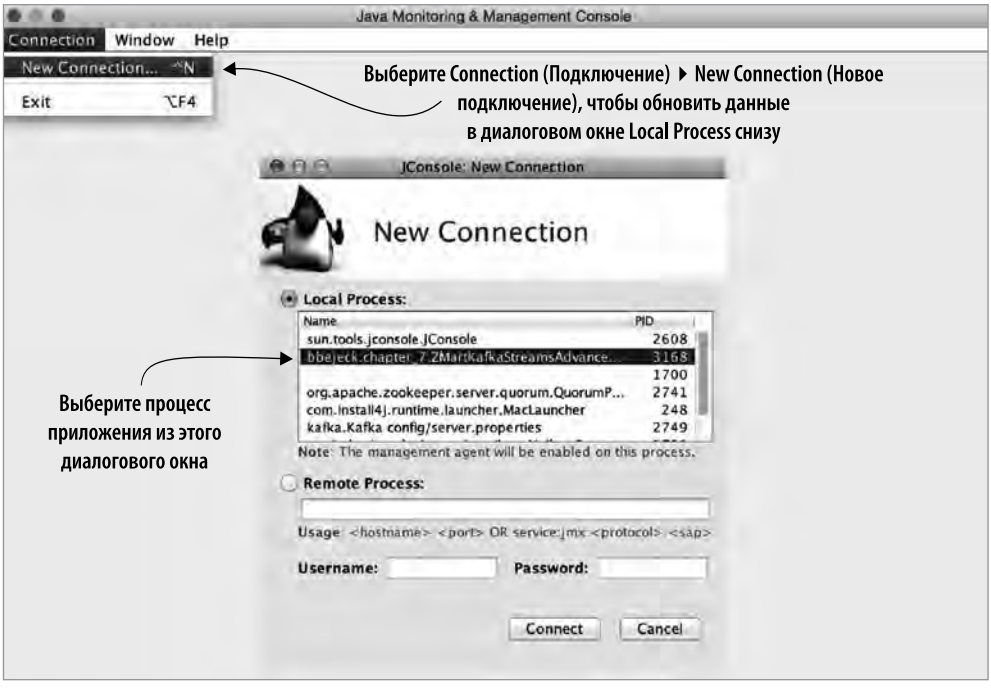


Рис. 7.8. JConsole подключается к программе

Весьма вероятно, что после двойного щелчка на программе, к которой вы хотели бы подключиться, вы увидите предупреждение, аналогичное показанному на рис. 7.9. Поскольку вы работаете в пределах своей локальной машины, можете нажать кнопку Insecure Connection (Ненадежное подключение).

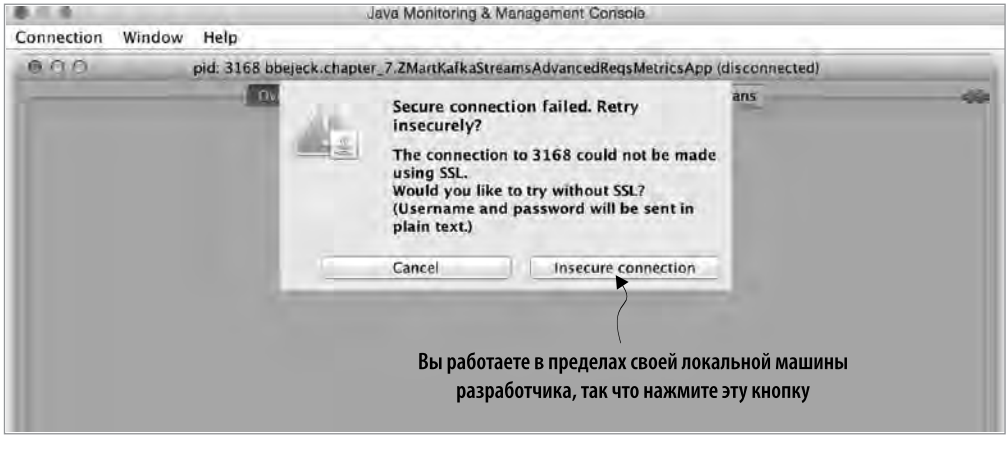


Рис. 7.9. Предупреждение подключения JConsole об отсутствии SSL

Теперь все готово для просмотра метрик, собранных вашим приложением Kafka Streams. Осталось только проверить имеющуюся информацию.

ПРЕДУПРЕЖДЕНИЕ

Вы используете ненадежное подключение для разработки на локальной машине. На практике следует всегда обращаться только по защищенным каналам к любым удаленным сервисам для просмотра внутреннего состояния ваших приложений.

Просмотр информации

После подключения вы увидите экран GUI, примерно такой, как на рис. 7.10. JConsole предоставляет несколько удобных способов заглянуть внутрь работающих приложений.

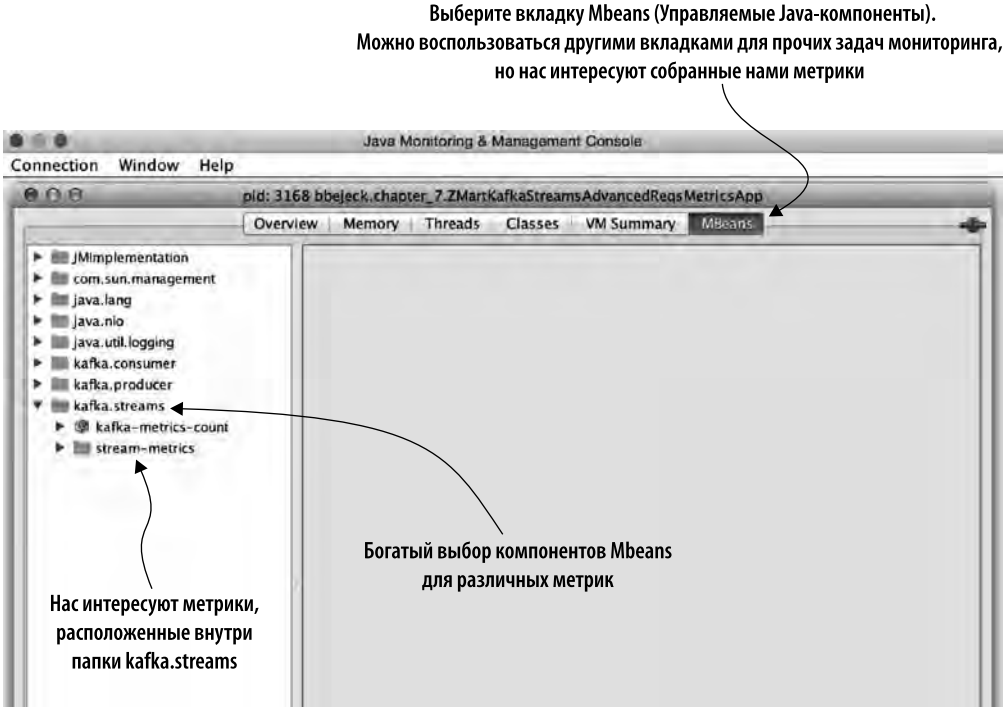


Рис. 7.10. Запущенный JConsole

Из числа вкладок Overview (Обзор), Memory (Оперативная память), Threads (Потоки выполнения), Classes (Классы), VM Summary (Сводка по VM) и MBeans (Управляемые Java-компоненты) мы воспользуемся только последней. Информация на остальных вкладках тоже интересна, но относится больше к общему состоянию

приложения и применению им ресурсов. Собранные же на MBeans (Управляемые Java-компоненты) метрики содержат информацию о внутренних рабочих характеристиках топологии.

На этом мы завершим обзор использования JConsole. На следующем шаге мы приступим к просмотру метрик, записанных для нашей топологии.

7.2.4. Просмотр метрик

Рисунок 7.11 демонстрирует возможности просмотра метрик через JConsole во время работы приложения ZMart (src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java). Как вы видите, можно просмотреть характеристики (пропускную способность или задержку) всех элементов, вплоть до отдельных узлов-обработчиков топологии.

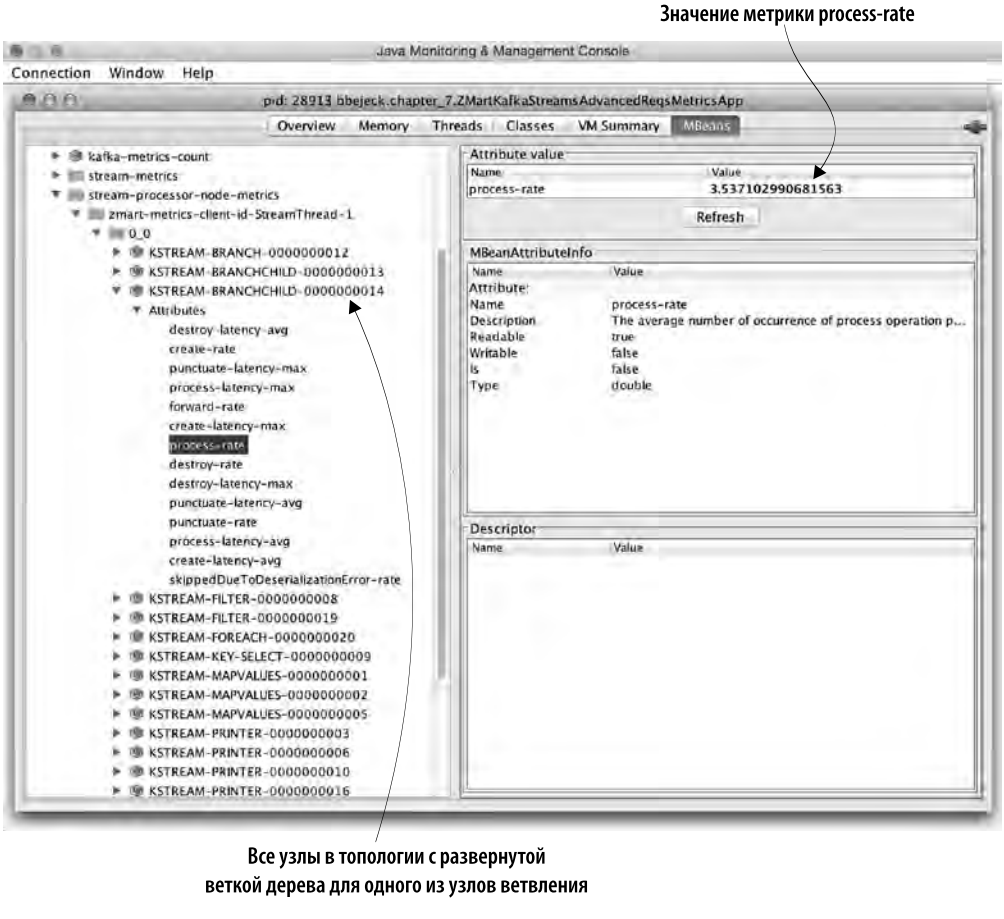


Рис. 7.11. Метрики JConsole для ZMart

СОВЕТ

Поскольку JMX взаимодействует только с работающими в данный момент приложениями, то некоторые из примеров приложений из папки src/main/java/bbejeck/chapter_7 будут работать безостановочно, чтобы вы могли поэкспериментировать с метриками. А значит, вам придется останавливать их вручную, или через IDE, или нажав Ctrl+C в командной строке.

На рис. 7.11 показана метрика `process-rate`, означающая среднее число обработанных записей в миллисекунду. Если посмотреть на правый верхний угол рисунка, можно увидеть, что скорость обработки равна 3,537 записи в миллисекунду (3537 записей в секунду). Кроме того, как уже обсуждалось ранее, из JConsole можно просматривать метрики потребителя и генератора.

СОВЕТ

Хотя имеющиеся метрики охватывают практически все, возможны случаи, когда вам понадобятся свои, пользовательские метрики. Это довольно редкий и специфический сценарий применения, так что мы не будем рассматривать подобный пример подробно. Но при желании вы можете взглянуть на метод StockPerformanceMetricsTransformer.init в качестве примера добавления пользовательской метрики и метод StockPerformanceMetricsTransformer.transform в качестве примера их использования. Класс StockPerformanceMetricsTransformer находится в файле src/main/java/bbejeck/chapter_7/transformer/StockPerformanceMetrics-Transformer.java.

Мы обсудили вопрос просмотра метрик Kafka Streams. Время перейти к другим удобным способам выяснить, что происходит внутри приложения.

7.3. Дополнительные методики отладки Kafka Streams

Мы рассмотрим еще несколько методик наблюдения за потоковыми приложениями Kafka Streams и их отладки. Предыдущий раздел был посвящен в основном производительности, в этом же разделе мы сосредоточим наше внимание на том, как узнать своевременно о различных состояниях приложения, и на просмотре структуры топологии.

7.3.1. Просмотр структуры приложения

После запуска готового приложения могут возникнуть ситуации, требующие его отладки. Допустим, для определенной задачи не помешала бы вторая пара рук, но по каким-либо причинам вы не можете никому показать код. Или, например, вам может понадобиться просмотреть, какие `TopicPartition` назначены различным задачам приложения.

Метод `Topology.describe()` дает возможность просмотреть общую структуру приложения. Он выводит информацию о структуре программы, включая все внутренние топики, созданные для целей повторного секционирования. На рис. 7.12 приведены результаты вызова метода `describe` для `CoGroupingListeningExampleApplication` из главы 7 (src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java).

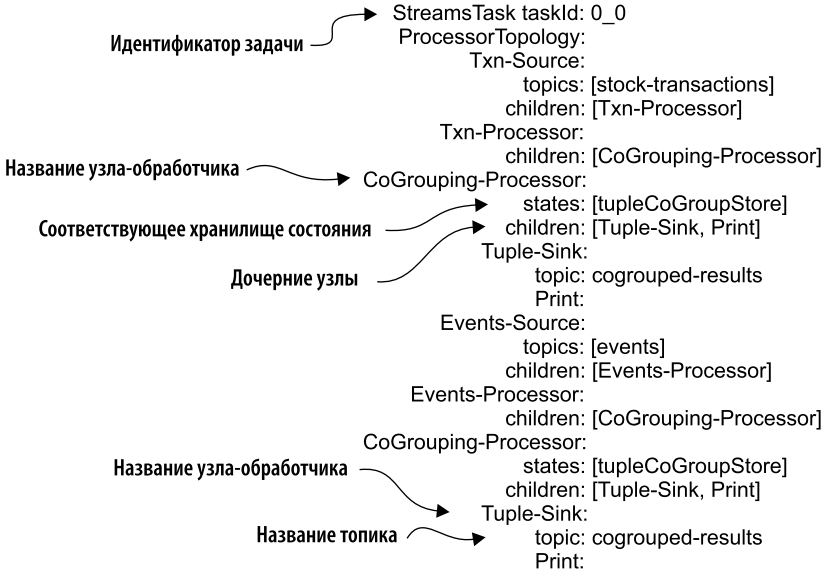


Рис. 7.12. Отображает названия узлов, соответствующие дочерние узлы и другую информацию

Как вы можете видеть, метод `Topology.describe()` выводит краткий, аккуратный обзор структуры приложения. Отмечу, что приложение `CoGroupingListeningExampleApplication` использовало API узлов-обработчиков, так что названия всех узлов топологии выбраны разработчиком. В случае API Kafka Streams названия были бы несколько более стандартизированы.

```
KSTREAM-SOURCE-0000000000:
    topics: [transactions]
    children: [KSTREAM-MAPVALUES-0000000001]
```

СОВЕТ

При работе с API DSL Kafka Streams класс `Topology` непосредственно не применяется, но к нему можно легко обратиться при необходимости. Если вам нужно вывести в консоль физическую топологию приложения, воспользуйтесь методом StreamsBuilder.build(), возвращающим объект Topology, а затем вызовите метод Topology.describe(), как мы вам только что показывали.

Может оказаться полезно также получить в приложении информацию о времени выполнения для объектов `StreamThread`. Для доступа к информации объектов `StreamThread` воспользуйтесь методом `KafkaStreams.localThreadsMetadata()`.

7.3.2. Получение уведомлений о различных состояниях приложения

Запущенное приложение Kafka Streams не начинает сразу же обрабатывать данные — сначала нужно произвести определенную координацию действий. Потребитель должен извлечь информацию о метаданных и подписках, приложение должно запустить экземпляры `StreamThread` и распределить `TopicPartition` по `StreamTask`.

Этот процесс распределения или перераспределения задач (рабочей нагрузки) называется *перебалансировкой* (rebalancing). Перебалансировка означает возможность автоматического вертикального масштабирования Kafka Streams в обе стороны. Это ключевое его преимущество — возможность добавить новые экземпляры приложения *во время работы существующего приложения* в расчете на то, что процесс перебалансировки перераспределит нагрузку.

Например, допустим, что ваше приложение Kafka Streams содержит два узла-источника, по две секции в каждом топике — итого четыре требующих распределения объекта `TopicPartition`. Сначала вы запускаете приложение с одним потоком выполнения. Kafka Streams определяет, сколько нужно создать задач, на основе максимального размера секции среди всех входных топиков. В данном случае у каждого топика две секции, так что максимум равен 2, следовательно, будет создано две задачи. Затем в процессе перебалансировки каждой из двух задач назначается по два объекта `TopicPartition`.

После того как приложение поработало какое-то время, вы решаете, что хотите обрабатывать записи быстрее. Для этого вам достаточно запустить еще один экземпляр приложения *с тем же идентификатором приложения*, после чего процесс перебалансировки распределит нагрузку с учетом нового потока выполнения приложения. В результате две задачи окажутся распределенными по двум потокам выполнения.

Мы только что удвоили возможности приложения, в то время как исходная версия продолжала работать — останавливать приложение не понадобилось.

Среди других причин перебалансировки — останов или запуск другого экземпляра Kafka Streams (с тем же идентификатором приложения), добавление секций в топик или — в случае узла-источника на основе регулярного выражения — добавление/удаление топиков, соответствующих паттерну регулярного выражения.

Во время фазы перебалансировки внешние взаимодействия временно приостанавливаются до того момента, как приложение завершит распределение секций топиков по потоковым задачам, так что вам хотелось бы получить оповещение об этом моменте жизненного цикла приложения. Например, отсутствует возможность

обращения к доступным для запроса хранилищам состояния, а значит, желательно ограничить запросы на просмотр содержимого хранилища состояния до тех пор, пока это снова не станет возможным.

Но как же проверить, происходит ли сейчас перебалансировка ваших других приложений? К счастью, такой механизм в виде интерфейса `StateListener` в Kafka Streams есть, и мы рассмотрим его в следующем разделе.

7.3.3. Использование интерфейса StateListener

Приложение Kafka Streams в каждый заданный момент времени может находиться в одном из шести состояний. На рис. 7.13 показаны допустимые для приложения Kafka Streams состояния. Как вы видите, существует несколько сценариев смены состояния, из которых мы сосредоточим внимание на одном — переходе из состояния выполнения в состояние перебалансировки. Это наиболее частый и сильнее всего влияющий на производительность тип перехода, поскольку в фазе перебалансировки не производится никакой обработки.

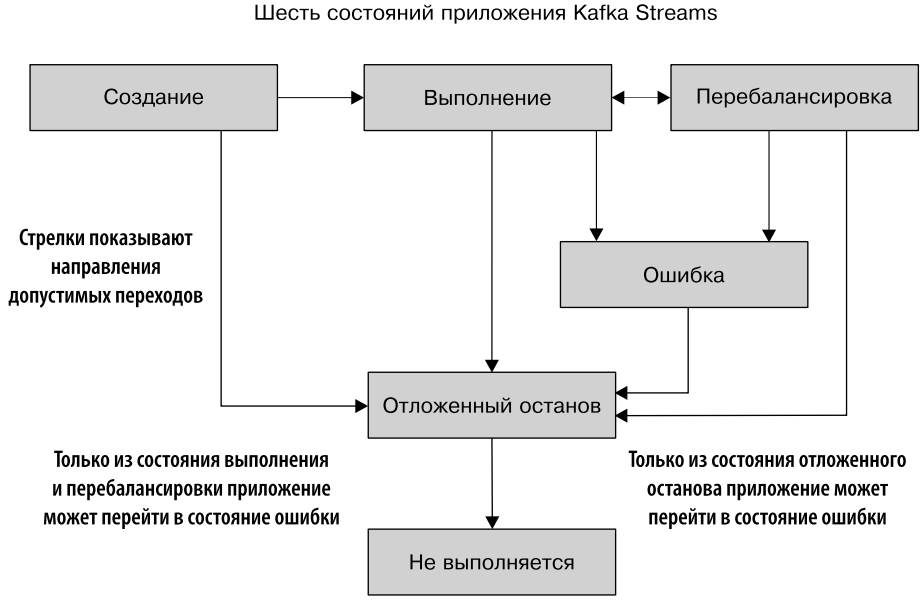


Рис. 7.13. Возможные состояния приложения Kafka Streams

Чтобы уловить эти смены состояния, мы воспользуемся методом `KafkaStreams.setStateListener`, принимающим на входе экземпляр интерфейса `StateListener`. Это интерфейс с одним методом, так что можно воспользоваться синтаксисом лямбда-выражений Java 8, как показано в листинге 7.4 (его можно найти в файле src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java).

Листинг 7.4. Добавление прослушивателя состояния

```
KafkaStreams.StateListener stateListener = (newState, oldState) -> {
    if (newState == KafkaStreams.State.RUNNING &&
    ➥ oldState == KafkaStreams.State.REBALANCING) {
      LOG.info("Application has gone from REBALANCING to RUNNING ");
      LOG.info("Topology Layout {}",
        ➥ streamsBuilder.build().describe());
    }
};
```

Проверяем, не происходит ли переход из состояния перебалансировки (REBALANCING) в состояние выполнения (RUNNING)

Выводим структуру топологии

СОВЕТ

Листинг 7.4, демонстрирующий класс ZMartKafkaStreamsAdvancedReqsMetricsApp.java, включает просмотр метрик JMX и оповещение о переходе из одного состояния в другое, так что я отключил в нем вывод результатов потоковой обработки в консоль. Мы записываем тут результаты только в топики Kafka. При запуске приложения вы увидите в консоли выводимую прослушивателем информацию.

В нашей первой реализации интерфейса `StateListener` мы выводим в журнал (консоль) информацию об изменениях состояния. В подразделе 7.3.1, когда мы обсуждали вывод структуры топологии, я упоминал, что нужно подождать завершения перебалансировки. Именно это мы и делаем в листинге 7.4: выводим структуру после завершения всех задач и переназначений.

Расширим немного этот пример и покажем оповещение при переходе приложения в состояние перебалансировки. Для учета этого дополнительного перехода необходимо изменить код следующим образом (его можно найти в файле src/main/java/bbejeck/chapter_7/ZMartKafkaStreamsAdvancedReqsMetricsApp.java) (листинг 7.5).

Листинг 7.5. Изменяем прослушиватель состояния для учета состояния перебалансировки (REBALANCING)

```
KafkaStreams.StateListener stateListener = (newState, oldState) -> {
  if (newState == KafkaStreams.State.RUNNING &&
  ➥ oldState == KafkaStreams.State.REBALANCING) {
     LOG.info("Application has gone from REBALANCING to RUNNING ");
     LOG.info("Topology Layout {}",
     ➥ streamsBuilder.build().describe());
  }

  if (newState == KafkaStreams.State.REBALANCING) {
      LOG.info("Application is entering REBALANCING phase");
  }
};
```

Дополнительное действие при входе в фазу перебалансировки

Хотя в нашем примере используются простые операторы журналирования, вам должно быть понятно, что вы можете заменить их более сложной логикой обработки изменений состояния в своем приложении.

ПРИМЕЧАНИЕ

В силу того что Kafka Streams — библиотека, а не фреймворк, на сервере можно запустить только один экземпляр. А если у вас запущено несколько приложений на разных машинах, то вы увидите результаты изменений состояния только на своей локальной машине.

Главное в этом разделе — возможность выяснить текущее состояние своего приложения Kafka Streams, благодаря чему оно перестает быть настолько «черным ящиком».

Далее мы обсудим перебалансировку несколько подробнее. Хотя возможность автоматической перебалансировки нагрузки — одна из сильных сторон Kafka Streams, число перебалансировок лучше минимизировать. Ведь во время перебалансировки данные не обрабатываются, а хотелось бы, чтобы наше приложение обрабатывало данные как можно больше времени.

7.3.4. Прослушиватель восстановления состояния

Из главы 4 вы узнали о хранилищах состояния и важности их резервного копирования на случай сбоя. В Kafka Streams мы часто используем в качестве резервной копии хранилищ состояния топики, которые называются *журналами изменений* (changelogs).

Журналы изменений записывают происходящие обновления хранилищ состояния. В случае сбоя приложения Kafka Streams или его перезапуска вручную хранилище состояния может восстановиться из локальных файлов состояния, как показано на рис. 7.14.

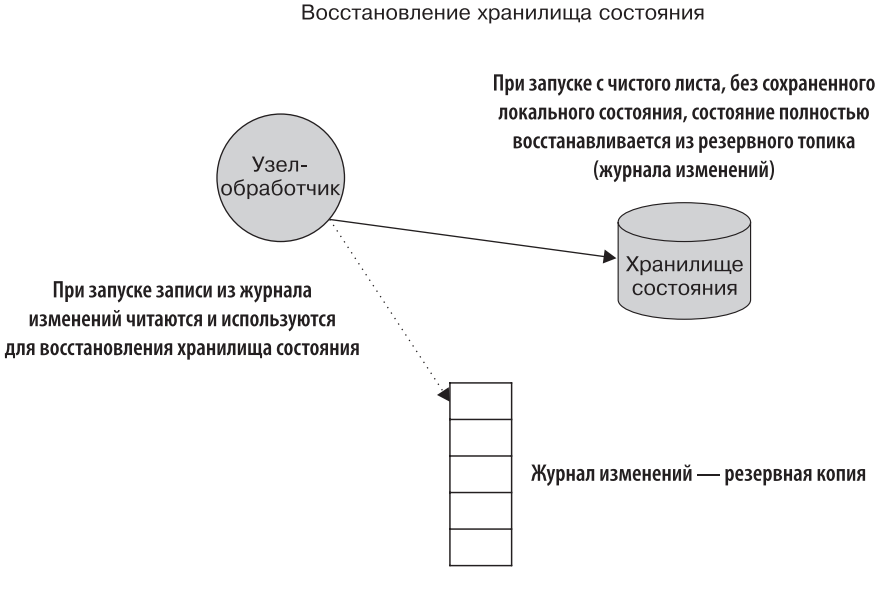


Рис. 7.14. Восстановление хранилища состояния с нуля

В некоторых случаях, однако, может понадобиться полное восстановление хранилища состояния из журнала изменений, например при работе приложения Kafka Streams в среде без сохранения состояния, вроде Mesos, или при серьезном сбое с потерей файлов на локальном диске. В зависимости от объема требующих восстановления данных такой процесс восстановления может потребовать немалого времени.

Во время этого периода восстановления все хранилища состояния, открытые вами для выполнения запросов, недоступны, так что хотелось бы понимать, сколько этот процесс восстановления займет времени и как он продвигается. Кроме того, при наличии пользовательского хранилища состояния будет не лишним получить оповещения о начале и завершении восстановления, чтобы выполнить нужные на этих этапах задачи.

Интерфейс `StateRestoreListener` аналогично `StateListener` дает возможность оповещения о происходящем внутри приложения. Интерфейс `StateRestoreListener` включает три метода: `onRestoreStart`, `onBatchRestored` и `onRestoreEnd`. Для задания глобального прослушивателя восстановления используется метод `KafkaStreams.setGlobalRestoreListener`.

ПРИМЕЧАНИЕ

Ожидается, что `StateRestoreListener` не будет сохранять состояние. Он используется для всего приложения. При необходимости отслеживания состояния в прослушивателе следует обеспечить синхронизацию.

Посмотрим на код прослушивателя, чтобы разобраться, как работает подобный процесс оповещения. Начнем с объявления переменной и метода `onRestoreStart` (его можно найти в файле src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java) (листинг 7.6).

Листинг 7.6. Журналирующий прослушиватель восстановления

```
public class LoggingStateRestoreListener implements StateRestoreListener {

    private static final Logger LOG =
    ➥ LoggerFactory.getLogger(LoggingStateRestoreListener.class);
    private final Map<TopicPartition, Long> totalToRestore =
    ➥ new ConcurrentHashMap<>();
    private final Map<TopicPartition, Long> restoredSoFar =
    ➥ new ConcurrentHashMap<>();

    @Override
    public void onRestoreStart(TopicPartition topicPartition,
    ➥ String store, long start, long end) {
        long toRestore = end - start;
        totalToRestore.put(topicPartition, toRestore);
        LOG.info("Starting restoration for {} on topic-partition {}
        ➥ total to restore {}", store, topicPartition, toRestore);
    }

    // Остальные методы, опущенные здесь для большей ясности, описываются ниже
}
```

Создает экземпляры ConcurrentHashMap для отслеживания хода восстановления

Сохраняет общий объем восстанавливаемых данных для заданного объекта TopicPartition

Первое, что мы делаем, — создаем два экземпляра `ConcurrentHashMap` для отслеживания хода восстановления. В методе `onRestoreStart` мы сохраняем общее количество записей, требующих восстановления, и заносим в журнал сообщение о начале восстановления.

Далее посмотрим на код, выполняющий собственно восстановление каждого пакета записей (его можно найти в файле src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java) (листинг 7.7).

Листинг 7.7. Метод onBatchRestored

```
@Override
 public void onBatchRestored(TopicPartition topicPartition,
 ➥ String store, long start, long batchCompleted) {
   NumberFormat formatter = new DecimalFormat("#.##");

   long currentProgress = batchCompleted +
   ➥ restoredSoFar.getOrDefault(topicPartition, 0L);
   double percentComplete =
   ➥ (double) currentProgress / totalToRestore.get(topicPartition);

   LOG.info("Completed {} for {}% of total restoration for {} on {}",
                batchCompleted,
   ➥ formatter.format(percentComplete * 100.00),
   ➥ store, topicPartition);
    restoredSoFar.put(topicPartition, currentProgress);
  }
```

Вычисляем общее количество восстановленных записей

Определяем степень завершения восстановления (в процентах)

Записываем в журнал процент восстановленных записей

Сохраняем количество восстановленных на текущий момент записей

Для чтения из топика журнала изменений в процессе восстановления используется внутренний потребитель, поэтому приложение восстанавливает записи пакетами, соответствующими вызовам метода `consumer.poll()`. Вследствие этого максимальный размер любого пакета равен значению параметра конфигурации `max.poll.records`.

После загрузки процессом восстановления последнего пакета в хранилище состояния вызывается метод `onBatchRestored`. Во-первых, мы прибавляем размер текущего пакета к накопленному количеству восстановленных записей. Далее подсчитываем степень завершения восстановления (в процентах) и выводим в журнал результат. И наконец, сохраняем ранее вычисленное новое общее количество записей.

Последний шаг, который мы обсудим, производится по завершении процесса восстановления (его можно найти в файле src/main/java/bbejeck/chapter_7/restore/LoggingStateRestoreListener.java) (листинг 7.8).

После завершения приложением процесса восстановления остается выполнить один последний вызов прослушивателя с передачей общего количества восстановленных записей. В данном примере мы записываем в журнал окончательное состояние и сбрасываем общий счетчик восстановления на 0.

Листинг 7.8. Метод, вызываемый по завершении процесса восстановления

```
@Override
  public void onRestoreEnd(TopicPartition topicPartition,
  ➥ String store, long totalRestored) {
     LOG.info("Restoration completed for {} on
     ➥ topic-partition {}", store, topicPartition);
     restoredSoFar.put(topicPartition, 0L);
  }
```

Отслеживаем ход восстановления для данного объекта TopicPartition

Наконец, мы можем воспользоваться `LoggingStateRestoreListener` в своем приложении следующим образом (см. файл src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java) (листинг 7.9).

Листинг 7.9. Задаем глобальный прослушиватель восстановления

```
kafkaStreams.setGlobalStateRestoreListener(new LoggingStateRestoreListener());
```

Это был пример использования `StateRestoreListener`. В главе 9 вы увидите пример, включающий графическое представление процесса восстановления.

СОВЕТ

Для просмотра файла журнала, сгенерированного при запуске приложения CoGroupingListeningExampleApplication, найдите файл logs/state_restore_listener.log в корневом каталоге места установки исходного кода.

7.3.5. Обработчик неперехваченных исключений

Мне кажется, что, без преувеличения, всякий разработчик время от времени сталкивается с исключениями, на появление которых не рассчитывал, и видит длинную трассу вызовов в стеке при неожиданном завершении работы программы. Хотя подобная ситуация не вполне вписывается в пример мониторинга, рекомендуемой практикой является отправка уведомления и выполнение необходимой очистки в случае возникновения неожиданного исключения. Для обработки подобных неожиданных ошибок Kafka Streams предоставляет метод `KafkaStreams.setUncaughtExceptionHandler` (листинг 7.10) (его можно найти в файле src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.java).

Листинг 7.10. Простейший обработчик неперехваченных исключений

```
kafkaStreams.setUncaughtExceptionHandler((thread, exception) -> {
       CONSOLE_LOG.info("Thread [" + thread + "]
➥ encountered [" + exception.getMessage() +"]");
});
```

Это, конечно, реализация, сокращенная до абсолютного минимума, но ее достаточно для демонстрации того, где можно обрабатывать неожиданные исключения — или

посредством журналирования ошибки, как показано выше, или путем выполнения необходимой очистки и останова потоков данных приложения.

На этом мы завершаем обсуждение мониторинга приложений Kafka Streams.

Резюме

- Для мониторинга Kafka Streams требуется также мониторинг и брокеров Kafka.
- Для отслеживания производительности приложения следует включить периодическое создание отчетов о метриках.
- Чтобы понимать, что происходит в вашем приложении, необходимо заглянуть к нему «под капот», а иногда и провести более низкоуровневое расследование, воспользовавшись утилитами командной строки, поставляемыми вместе с Java, например `jstack` (производит дампы потоков выполнения) или `jmap/jhat` (производит дампы кучи).

В этой главе мы сосредоточили наше внимание на *наблюдении* за поведением приложения. В следующей главе мы переключимся на вопрос согласованной и адекватной обработки ошибок. Мы также убедимся с помощью регулярного тестирования в том, что приложение ведет себя *ожидаемым образом*.

8 Тестирование приложения Kafka Streams

В этой главе:

- тестирование топологии;
- тестирование отдельных узлов-обработчиков и преобразователей;
- комплексное тестирование с помощью встроенного кластера Kafka.

Пока мы рассмотрели основные стандартные блоки, используемые при создании приложений Kafka Streams. Но мы не упоминали еще один из важнейших элементов разработки приложений: тестирование. Одна из главных концепций, на которой мы сосредоточим наше внимание, — размещение бизнес-логики в автономных классах, полностью независимых от приложения Kafka Streams, с целью упрощения тестирования. Я полагаю, что вы знаете, насколько важно тестирование, но все же упомяну две основные, с моей точки зрения, причины, почему тестирование не менее важно, чем сам процесс разработки.

Во-первых, разработкой своего кода вы создаете неписаное соглашение относительно того, что можно ожидать от кода. Единственный способ доказать, что код работает, — тщательное тестирование, так что при тестировании необходимо учесть широкий спектр возможных входных данных и сценариев, чтобы убедиться в должной работе кода в допустимых условиях.

Вторая причина, по которой необходим первоклассный набор тестов, — неизбежные изменения программного обеспечения. Хороший набор тестов обеспечивает немедленную обратную связь в случае, когда код начинает вести себя не так, как ожидалось. Кроме того, при масштабном рефакторинге или добавлении новой функциональности прохождение тестов позволяет быть уверенными в том, как изменения повлияют на приложение (при условии, что тесты качественные).

Тестировать приложения Kafka Streams не всегда просто, даже если вы понимаете, насколько это важно. Можно запустить на выполнение простую топологию и посмотреть на результаты, но у этого подхода есть один недостаток. Нам нужен набор *воспроизводимых* (repeatable) тестов, которые можно запустить в любой момент в качестве составной части сборки, так что желательно иметь возможность тестировать приложение без кластера Kafka и ансамбля ZooKeeper.

Именно об этом мы и поговорим в данной главе. Во-первых, вы увидите, как тестировать топологию *без* работающего кластера Kafka, когда вся топология работает в рамках модульного теста. Вы также научитесь тестировать узлы-обработчики или преобразователи отдельно друг от друга, с имитацией всех необходимых зависимостей.

ПРИМЕЧАНИЕ

Вероятно, вам приходилось выполнять тестирование с использованием имитационных объектов, но если нет, для знакомства вполне подойдет статья из «Википедии»: https://en.wikipedia.org/wiki/Mock_object.

Хотя модульное тестирование жизненно важно для воспроизводимости тестов и быстрой обратной связи, комплексное тестирование тоже играет важную роль, ведь иногда бывает необходимо проверить взаимодействие частей приложения между собой. Например, рассмотрим случай перебалансировки — важного элемента приложений Kafka Streams. Добиться выполнения перебалансировки в модульном тесте практически невозможно. В табл. 8.1 кратко описываются различия между модульным и комплексным тестированием.

Таблица 8.1. Подходы к тестированию

Тип тестов	Цель	Скорость тестирования	Объемы использования
Модульные	Изолированное тестирование отдельных частей функциональности	Быстрое	Составляют большинство тестов
Комплексные	Тестирование точек сопряжения различных подсистем	Более длительное	Составляют небольшую долю тестов

Для тестирования необходимо запустить реальную перебалансировку в реалистичных условиях. В подобных случаях нужно выполнять приложение вместе с экземпляром кластера Kafka. Но надеяться на наличие внешнего кластера нежелательно, так что мы научимся пользоваться для комплексного тестирования встроенными Kafka и ZooKeeper.

8.1. Тестирование топологии

Первая созданная нами в главе 3 топология была относительно сложной. Чтобы освежить вашу память, она приведена на рис. 8.1 снова.

Логика обработки достаточно проста, но, как можно видеть из структуры, она состоит из нескольких узлов. Важная особенность, которая упростит нам тестирование: топология принимает один параметр на входе — исходную покупку — и производит

над ним далее несколько преобразований. Благодаря этому тестирование несколько упрощается, в том смысле что достаточно передать одно значение с информацией о покупке, после чего можно по результатам проверить, были ли произведены все нужные преобразования.

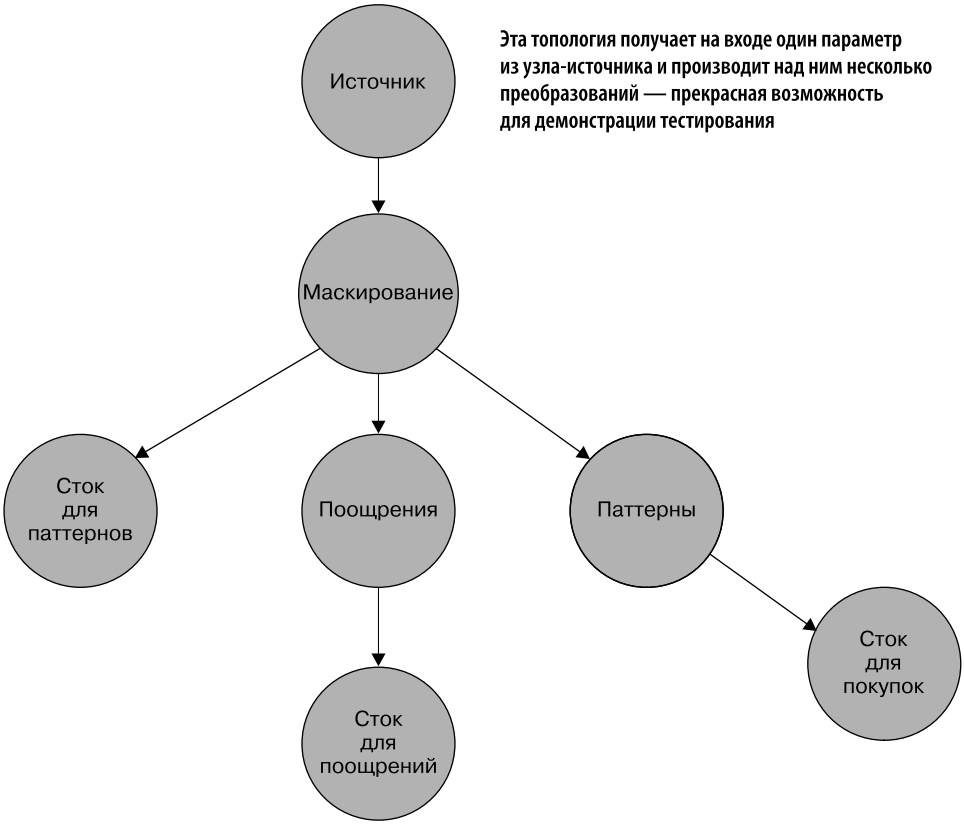


Рис. 8.1. Полная исходная топология программы Kafka Streams для компании ZMart

СОВЕТ

В большинстве случаев логика обработки должна находиться в отдельных классах, чтобы можно было модульно протестировать бизнес-логику отдельно от топологии. В случае топологии ZMart большая часть логики проста и представлена лямбда-выражениями Java 8, так что в данном случае мы будем тестировать движение данных по топологии.

Нам нужен повторяемый автономный тест, поэтому мы воспользуемся классом `ProcessorTopologyTestDriver`, благодаря которому для выполнения подобного теста не требуется Kafka. Не забывайте, что возможность тестирования

топологии без работающего экземпляра Kafka означает ускорение и упрощение тестирования, а значит, и сокращение цикла разработки. Отмечу также, что `ProcessorTopologyTestDriver` — обобщенный фреймворк, предназначенный для тестирования отдельных объектов топологии Kafka Streams.

Использование утилит тестирования библиотеки Kafka Streams

Для использования утилит тестирования библиотеки Kafka Streams необходимо модифицировать файл `build.gradle` следующим образом:

```
testCompile group:'org.apache.kafka', name:'kafka-streams',
➥ version:'1.0.0', classifier:'test'
testCompile group:'org.apache.kafka', name:'kafka-clients',
➥ version:'1.0.0', classifier:'test'
```

Если вы задействуете Maven, то воспользуйтесь следующим кодом:

```
<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-streams</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
  <classifier>test</classifier>
</dependency>

<dependency>
  <groupId>org.apache.kafka</groupId>
  <artifactId>kafka-clients</artifactId>
  <version>1.0.0</version>
  <scope>test</scope>
  <classifier>test</classifier>
</dependency>
```

СОВЕТ

При написании своих собственных проектов с применением кода тестирования Kafka и Kafka Streams желательно указывать все зависимости, приведенные в примере кода.

Начальное создание топологии полностью производилось в методе `ZMartKafkaStreamsApp.main`. Для быстрой разработки это вполне подходит, но для тестирования — не очень. Теперь нам придется переделать топологию в автономный класс, чтобы иметь возможность ее протестировать.

Логика работы не меняется, и мы будем переносить код в неизменном виде, так что я не стану демонстрировать тут данное превращение. Вместо этого я предлагаю вам при желании обратиться к файлу src/main/java/bbejeck/chapter_8/ZMartTopology.java.

По завершении переноса кода можно приступить к созданию теста.

8.1.1. Создание теста

Перейдем к созданию модульного теста для топологии ZMart. Воспользуемся стандартным тестом JUnit. Перед выполнением теста необходимо произвести кое-какие предварительные действия (см. файл src/test/java/bbejeck/chapter_8/ZMartTopologyTest.java) (листинг 8.1).

Листинг 8.1. Метод с предварительными настройками для тестирования топологии

```
@Before
public void setUp() {

 // опускаем подробности формирования свойств
 StreamsConfig streamsConfig = new StreamsConfig(props);
 Topology topology = ZMartTopology.build();

 topologyTestDriver =
  ➥ new ProcessorTopologyTestDriver(streamsConfig, topology);
}
```

Переделанная топология ZMart: теперь топологию можно получить, вызвав метод

Создает объект ProcessorTopologyTestDriver

Важнейший момент листинга 8.1 — создание объекта `ProcessorTopologyTestDriver`, которым мы воспользуемся в листинге 8.2, при запуске теста (его можно найти в файле src/test/java/bbejeck/chapter_8/ZMartTopologyTest.java).

Листинг 8.2. Тестирование топологии

```
@Test
 public void testZMartTopology() {

 Purchase purchase = DataGenerator.generatePurchase();

 topologyTestDriver.process("transactions",
                            null,
                            purchase,
                            stringSerde.serializer(),
                            purchaseSerde.serializer());

 ProducerRecord<String, Purchase> record =
 ➥ topologyTestDriver.readOutput("purchases",
                                  stringSerde.deserializer(),
                                  purchaseSerde.deserializer());

 Purchase expectedPurchase =
 ➥ Purchase.builder(purchase).maskCreditCard().build();
 assertThat(record.value(), equalTo(expectedPurchase));
```

Создаем тестовый объект; переиспользуем код генерации данных, применявшийся ранее, при выполнении топологии

Отправляем в топологию исходную запись

Читаем запись из топика purchases

Преобразуем тестовый объект в ожидаемый формат

Проверяем, соответствует ли запись из топологии ожидаемой записи

В листинге 8.2 есть два критически важных фрагмента кода. Начиная с вызова метода `topologyTestDriver.process`, мы отправляем запись в топик `transactions` — источник данных для всей топологии. По завершении прохождения данных по топологии мы сможем убедиться, что были произведены те действия, которые нужны. В следующей строке мы читаем запись из топика `purchases` с помощью метода `topologyTestDriver.readOutput` и одного из описанных в топологии стоков. В предпоследней строке мы создаем ожидаемую итоговую запись, после чего в последней строке проверяем, соответствуют ли полученные результаты ожидаемым.

В топологии есть еще два узла-стока, так что для полноты теста убедимся, что из них мы тоже получим правильные результаты (код можно найти в файле src/test/java/bbejeck/chapter_8/ZMartTopologyTest.java) (листинг 8.3).

Листинг 8.3. Тестирование оставшейся части топологии

```
@Test
  public void testZMartTopology() {

  // Продолжаем тест из предыдущего листинга

  RewardAccumulator expectedRewardAccumulator =
  ➥ RewardAccumulator.builder(expectedPurchase).build();

  ProducerRecord<String, RewardAccumulator> accumulatorProducerRecord =
  ➥ topologyTestDriver.readOutput("rewards",
                           stringSerde.deserializer(),
                           rewardAccumulatorSerde.deserializer());

  assertThat(accumulatorProducerRecord.value(),
  ➥ equalTo(expectedRewardAccumulator));

  PurchasePattern expectedPurchasePattern =
  ➥ PurchasePattern.builder(expectedPurchase).build();

  ProducerRecord<String, PurchasePattern> purchasePatternProducerRecord =
  ➥ topologyTestDriver.readOutput("patterns",
                             stringSerde.deserializer(),
                             purchasePatternSerde.deserializer());

  assertThat(purchasePatternProducerRecord.value(),
  ➥ equalTo(expectedPurchasePattern));
  }
```

Читаем запись из топика rewards

Проверяем, соответствуют ли полученные из топика rewards результаты ожидаемым

Читаем запись из топика patterns

Проверяем, соответствуют ли полученные из топика patterns результаты ожидаемым

При добавлении новых узлов-обработчиков в тест схема работы останется такой же, как в листинге 8.3. Из каждого топика читаются записи, после чего проверяется их соответствие ожидаемым с помощью оператора контроля (`assert`). Главное: мы получили воспроизводимый тест, который пропускает запись через всю топологию без всяких расходов ресурсов на выполнение Kafka.

Класс `ProcessorTopologyTestDriver` поддерживает также тестирование топологий с хранилищами состояния. Взглянем, как это осуществить.

8.1.2. Тестирование хранилища состояния в топологии

Для демонстрации тестирования хранилища состояния мы переделаем еще один класс, `StockPerformanceStreamsAndProcessorApplication`, так, чтобы вызываемый метод возвращал объект `Topology`. Этот класс можно найти в файле src/main/java/bbejeck/chapter_8/StockPerformanceStreamsProcessorTopology.java. Я не вносил никаких изменений в логику его работы, так что здесь мы не будем его рассматривать.

Предварительные настройки теста остались такими же, как и ранее, так что я дам пояснения только к новым фрагментам (src/test/java/bbejeck/chapter_8/StockPerformanceStreamsProcessorTopologyTest.java) (листинг 8.4).

Листинг 8.4. Тестирование хранилища состояния

```
StockTransaction stockTransaction =
➥ DataGenerator.generateStockTransaction();

topologyTestDriver.process("stock-transactions",
                           stockTransaction.getSymbol(),
                           stockTransaction,
                           stringSerde.serializer(),
                           stockTransactionSerde.serializer());

KeyValueStore<String, StockPerformance> store =
➥ topologyTestDriver.getKeyValueStore("s/tock-performance-store");

assertThat(store.get(stockTransaction.getSymbol()),
➥ notNullValue());
```

Генерируем тестовую запись

Отправляем запись на обработку в тестовую топологию

Извлекаем ссылку на хранилище состояния из тестовой топологии

Убеждаемся, что хранилище содержит ожидаемое значение

Как вы видите, последняя строка с оператором `assert` быстро проверяет, используется ли хранилище состояния в коде так, как и планировалось. Вы посмотрели на класс `ProcessorTopologyTestDriver` в действии, а также узнали, как выполнить сквозное тестирование топологии. Тестируемые топологии могут быть очень простыми, с одним узлом-обработчиком, или очень сложными, состоящими из нескольких субтопологий. И хотя мы производим это тестирование без брокера Kafka, не сомневайтесь: это полное тестирование топологии, с вовлечением в работу всех компонентов, включая сериализацию и десериализацию записей.

Мы увидели сквозное тестирование топологии. Но хотелось бы также протестировать внутреннюю логику объектов `Processor` и `Transformer`. Тестирование всей топологии — замечательная вещь, но проверка поведения внутри каждого из классов требует более тонкого подхода, который мы рассмотрим далее.

8.1.3. Тестирование узлов-обработчиков и преобразователей

Для проверки поведения внутри отдельного класса требуется настоящий модульный тест, предназначенный для тестирования *только* одного класса. Написание модульного теста для объектов `Processor` и `Transformer` затруднений у вас вызвать не должно, но помните, что обоим классам требуется `ProcessorContext` для получения хранилищ состояния и планирования операций пунктуации.

Нам нужно создать не реальный объект `ProcessorContext`, а его *суррогат* для целей тестирования: объект-имитацию. Для этой цели можно поступить двумя способами.

Один вариант: воспользоваться имитационным фреймворком, например Mockito, (http://site.mockito.org) для генерации имитационных объектов в тесте. Другая возможность: применить объект `MockProcessorContext` из той же библиотеки тестирования, что и `ProcessorTopologyTestDriver`. Какой вариант выбрать — зависит от ваших нужд.

Если имитационный объект нужен вам просто в качестве заместителя реальной зависимости, имеет смысл воспользоваться конкретными имитационными объектами (имитационными объектами, не создаваемыми с помощью фреймворка). Но для проверки передаваемых в имитационный объект параметров, возвращаемого значения или других элементов поведения отлично подойдет имитационный объект, сгенерированный фреймворком. Имитационные фреймворки (такие как Mockito) включают обширный API, позволяющий указывать ожидания и проверять поведение, экономя тем самым время разработки и ускоряя процесс тестирования.

В листинге 8.5 мы будем задействовать оба этих типа имитационных объектов. Для создания имитации `ProcessorContext` мы воспользуемся фреймворком Mockito, поскольку хотим проверить параметры при вызове метода `init`, а также убедиться, что из метода `punctuate()` передаются ожидаемые значения. У нас также будет пользовательский имитационный объект для хранилища пар «ключ/значение», который вы увидите в действии в нашем пошаговом разборе примера кода.

В этом листинге мы будем тестировать узел-обработчик с помощью имитационных объектов. Мы начнем с теста под названием `AggregatingMethodHandleProcessorTest` (расположенного в каталоге src/test/java/bbejeck_chapter6/processor/cogrouping/) для объекта `AggregatingMethodHandleProcessor`. Во-первых, мы хотели бы проверить используемые в методе `init` параметры (см.: src/test/java/bbejeck/chapter_6/AggregatingMethodHandleProcessorTest.java).

Этот первый шаг прост: мы вызываем метод `init` для тестируемого узла-обработчика с имитационным объектом `ProcessorContext`. Затем мы можем проверить используемые для планирования пунктуации параметры, а также извлечение хранилища состояния.

Далее протестируем метод `punctuate` и проверим, что записи отправляются далее, как и должны (его можно найти в файле src/test/java/bbejeck/chapter_6/AggregatingMethodHandleProcessorTest.java) (листинг 8.6).

Листинг 8.5. Тестирование метода init

```
// Некоторые подробности опущены для краткости
private ProcessorContext processorContext =
➥ mock(ProcessorContext.class);
private MockKeyValueStore<String, Tuple<List<ClickEvent>,
➥ List<StockTransaction>>> keyValueStore =
➥ new MockKeyValueStore<>();

private AggregatingMethodHandleProcessor processor =
➥ new AggregatingMethodHandleProcessor();

  @Test
  @DisplayName("Processor should initialize correctly")
  public void testInitializeCorrectly() {
    processor.init(processorContext);
    verify(processorContext).schedule(eq(15000L), eq(STREAM_TIME),
    ➥ isA(Punctuator.class));
    verify(processorContext).getStateStore(TUPLE_STORE_NAME);
}
```

Создаем имитацию ProcessorContext с помощью Mockito

Имитационный объект KeyValueStore

Тестируемый класс

Вызов метода init для узла-обработчика, в результате чего вызываются методы ProcessorContext

Проверка извлечения хранилища состояния

Проверка передачи параметров в метод ProcessorContext.schedule

Листинг 8.6. Тестирование метода punctuate

```
@Test
@DisplayName("Punctuate should forward records")
public void testPunctuateProcess(){
  when(processorContext.getStateStore(TUPLE_STORE_NAME))
                       .thenReturn(keyValueStore);

  processor.init(processorContext);
  processor.process("ABC", Tuple.of(clickEvent, null));
  processor.process("ABC", Tuple.of(null, transaction));

  Tuple<List<ClickEvent>,List<StockTransaction>> tuple =
  ➥ keyValueStore.innerStore().get("ABC");
  List<ClickEvent> clickEvents = new ArrayList<>(tuple._1);
  List<StockTransaction> stockTransactions = new ArrayList<>(tuple._2);

  processor.cogroup(124722348947L);

  verify(processorContext).forward("ABC",
  ➥ Tuple.of(clickEvents, stockTransactions));

  assertThat(tuple._1.size(), equalTo(0));
  assertThat(tuple._2.size(), equalTo(0));
}
```

Вызываем метод init для узла-обработчика

Указываем объекту-имитации возвращать при вызове keyValueStore

Обрабатываем объекты ClickEvent и StockTransaction

Извлекаем элементы, помещенные в хранилище состояния в методе process

Проверяем, отправляет ли ProcessorContext далее ожидаемые записи

Проверяем, очищены ли коллекции внутри кортежа

Вызываем метод совместной группировки, с помощью которого планируется пунктуация

Этот тест несколько более запутанный, в нем используется как имитационное, так и реальное поведение. Кратко рассмотрим его.

В первой строке задается поведение имитационного объекта ProcessorContext: при вызове метода `ProcessorContext.getStateStore` он должен возвращать заглушку для `KeyValueStore`. Эта строка представляет собой интересную смесь сгенерированного имитационного объекта и имитационного объекта-заглушки.

Мы вполне могли воспользоваться Mockito для генерации имитационного объекта `KeyValueStore`, но не стали этого делать по двум причинам. Во-первых, мне кажется, что сгенерированный имитационный объект, возвращающий другой сгенерированный имитационный объект, выглядит несколько неестественно. Во-вторых, мы хотим проверять и применять сохраненные значения в `KeyValueStore` во время тестирования вместо указания ожидаемых значений с помощью шаблонного ответа.

Следующие три строки, начиная с вызова `processor.init`, реализуют обычные для узла-обработчика шаги: сначала инициализацию, а потом обработку записей. На четвертом шаге особенно важно наличие работающего `KeyValueStore`. Поскольку наш KeyValueStore представляет собой простую заглушку, мы воспользуемся «под капотом» объектом `java.util.HashMap` в качестве настоящего хранилища. В трех строках, следующих за указанием ожидаемых значений для узла-обработчика, мы извлекаем из хранилища данные, помещенные туда при вызове метода `process()`. Мы создаем новые объекты `ArrayList`, включающие содержимое объекта `Tuple` (опять же это пользовательский класс, разработанный для примеров кода в настоящей книге), извлеченного из хранилища состояния по заданному ключу.

Далее мы выносим метод `punctuate` из узла-обработчика. Поскольку это модульный тест, нам не требуется проверять приращение времени — это означало бы тестирование самого API Kafka Streams, чего нам тут вовсе не нужно. Наша цель — проверить поведение метода, который мы указали в качестве своего `Punctuator` (посредством ссылки на метод).

Далее мы проверяем самое главное: что ожидаемые ключ и значение пересылаются далее по конвейеру через метод `ProcessorContext.forward`. Эта часть теста демонстрирует пользу от сгенерированного имитационного объекта. Благодаря фреймворку Mockito нам нужно только сообщить имитационному объекту, что ему следует ожидать вызова `forward` с заданными ключом и значением, и убедиться, что тест выполнил код именно так. И наконец, мы проверяем, очистил ли узел-обработчик коллекции объектов `ClickEvent` и `StockTransaction` после отправки их далее по конвейеру.

Как вы можете видеть из этого теста, с помощью комбинации имитационных объектов — сгенерированного и заглушки — можно изолировать тестируемый класс. Как я уже говорил ранее в данной главе, основную массу тестов в приложении Kafka Streams составляют модульные тесты бизнес-логики, а также отдельных объектов `Processor` и `Transformer`. Сама библиотека Kafka Streams и так тщательно протестирована, так что мы сосредоточим наши усилия на новом, непротестированном коде.

Вероятно, вам не хотелось бы дожидаться развертывания приложения, чтобы выяснить, как оно взаимодействует с кластером Kafka. Вы предпочли бы провести промежуточный контроль корректности кода, для которого потребуется комплексное тестирование. Давайте узнаем, как *локально* выполнить тест с реальным брокером Kafka.

8.2. Комплексное тестирование

Пока я рассказал вам о модульном тестировании топологии в целом или отдельных компонентов. В большинстве случаев эти тесты оптимальны, поскольку их можно быстро выполнить и проверить тем самым правильность работы конкретных фрагментов кода.

Но бывает, что требуется сквозное тестирование всех элементов приложения вместе: другими словами, комплексное тестирование. Обычно комплексное тестирование необходимо при наличии функциональности, которую невозможно охватить в модульном тесте.

В качестве примера вернемся к самому первому нашему приложению, Yelling App («Кричащее приложение»). Поскольку мы создавали топологию очень давно, напомню ее на рис. 8.2.

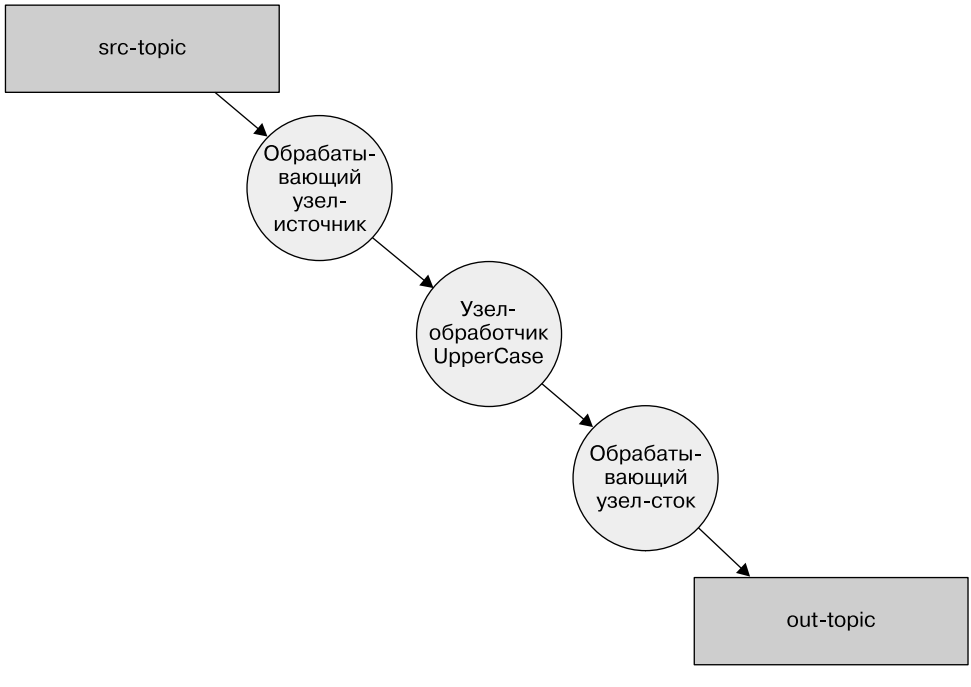


Рис. 8.2. Восстановим в памяти топологию приложения Yelling App

Допустим, что мы хотим поменять источник с конкретного поименованного топика на произвольный топик, соответствующий регулярному выражению:

```
yell-[A-Za-z0-9-]
```

В качестве примера мы хотели бы убедиться, что если при развернутом и запущенном приложении добавится топик `yell-at-everyone`, то приложение начнет читать информацию из него.

Мы не будем модифицировать исходное приложение Yelling App, поскольку оно очень маленькое. Вместо этого мы воспользуемся прямо в тесте следующей модифицированной версией (ее можно найти в файле src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java) (листинг 8.7).

Листинг 8.7. Модифицируем приложение Yelling App

```
streamsBuilder.<String,String>stream(Pattern.compile("yell.*"))
              .mapValues(String::toUpperCase)
              .to(OUT_TOPIC);
```

Преобразуем весь текст к верхнему регистру

Выводим результаты в топик («кричим»)

Подписываемся на все топики, чье название начинается с yell

Поскольку мы добавляем топики на уровне брокера Kafka, единственный способ проверить, «подхватывает» ли приложение новый топик, — добавить такой топик во время работы приложения. Подобный сценарий применения невозможен в случае модульного теста. Но значит ли это, что вам придется развертывать обновленное приложение для проверки?

К счастью, ответ на этот вопрос — нет. Вы можете воспользоваться *встраиваемым* (embedded) кластером Kafka с помощью тестовых библиотек Kafka. Под термином «встраиваемый» я понимаю тут большое приложение вроде Kafka или ZooKeeper, работающее в локальном автономном режиме, то есть «встраивание» его в существующее приложение.

Благодаря встроенному кластеру Kafka появляется возможность выполнения комплексного теста, для которого требуется кластер Kafka, на своей локальной машине в любой момент, как отдельно, так и в составе группы тестов. Это существенно сокращает цикл разработки. Приступим к созданию комплексного теста.

8.2.1. Создание комплексного теста

Первый шаг к использованию встраиваемого сервера Kafka требует добавления трех дополнительных зависимостей — `scala-library-2.12.4.jar`, `kafka_2.12-1.0.0-test.jar` и `kafka_2.12-1.0.0.jar` — в ваш файл `build.gradle` или `pom.xml`. Я уже описывал синтаксис указания тестового JAR в разделе 8.1, так что повторяться не стану.

Хотя может показаться, что число зависимостей растет, помните, что все добавляемые на данном этапе зависимости — *тестовые*. Они не включаются в дистрибутив

приложения и не развертываются вместе с его кодом; следовательно, они не повлияют на итоговый размер приложения.

Мы добавили нужные зависимости и можем теперь приступить к описанию комплексного теста с участием встраиваемого брокера Kafka. Для создания комплексного теста мы воспользуемся стандартным подходом JUnit.

Добавление EmbeddedKafkaCluster

Для добавления встраиваемого брокера Kafka в тест достаточно одной строки кода, как показано в листинге 8.8 (его можно найти в файле src/test/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java).

Листинг 8.8. Добавляем встраиваемый брокер Kafka

```
private static final int NUM_BROKERS = 1;

@ClassRule
public static final EmbeddedKafkaCluster EMBEDDED_KAFKA
➥ = new EmbeddedKafkaCluster(NUM_BROKERS);
```

Задаем количество брокеров

Аннотация ClassRule фреймворка JUnit

Создаем экземпляр EmbeddedKafkaCluster

Во второй строке листинга 8.8 мы создаем экземпляр `EmbeddedKafkaCluster`, служащий кластером для выполнения тестов в классе. Самое важное в этом примере — аннотация `ClassRule`. Полное описание фреймворков тестирования и JUnit выходит за рамки данной книги, но я уделю немного времени разъяснениям по поводу важности `@ClassRule` и ее роли в тесте.

Правила JUnit

В JUnit появилось понятие *правил* (rules), с помощью которых можно воспользоваться часто встречающейся тестовой логикой JUnit. Вот их краткое описание, взятое из https://github.com/junit-team/junit4/wiki/Rules#rules: «Правила обеспечивают возможность очень гибкого добавления или переопределения поведения любого из тестовых методов тестового класса».

В JUnit есть три типа правил, из которых класс `EmbeddedKafkaCluster` использует правила типа `ExternalResource` (https://github.com/junit-team/junit4/wiki/Rules#externalresource-rules). Правила `ExternalResource` применяются для подключения и удаления внешних ресурсов, например необходимого для теста `EmbeddedKafkaCluster`.

JUnit предоставляет класс `ExternalResource` с двумя пустыми методами: `before()` и `after()`. Всякий расширяющий `ExternalResource` класс должен переопределить эти методы, описав подключение и удаление необходимых для тестирования внешних ресурсов.

Правила — абстракция, прекрасно подходящая для использования в тестах внешних ресурсов. После создания расширяющего `ExternalResource` класса необходимо только создать в тесте переменную и применить аннотацию `@Rule` или `@ClassRule`, а все методы подключения/удаления будут выполнены автоматически.

Различие между аннотациями `@Rule` и `@ClassRule` заключается в частоте вызовов методов `before()` и `after()`. Аннотация `@Rule` выполняет методы `before()` и `after()` для *каждого отдельного* теста в классе. Аннотация же `@ClassRule` выполняет методы `before()` и `after()` *однократно*; метод `before()` выполняется до всех тестов, а метод `after()` — по завершении последнего теста в классе. Создание `EmbeddedKafkaCluster` требует довольно много ресурсов, так что имеет смысл делать это лишь один раз для каждого тестового класса.

Вернемся к созданию комплексного теста. Мы уже создали объект `EmbeddedKafkaCluster`, следующий шаг — создание необходимых в начале работы топиков.

Создание топиков

Теперь, когда у нас уже есть встраиваемый кластер Kafka, можно воспользоваться им для создания топиков следующим образом (src/test/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java) (листинг 8.9).

Листинг 8.9. Создание топиков для тестирования

```
@BeforeClass                                          ← Аннотация @BeforeClass
public static void setUpAll() throws Exception {
  EMBEDDED_KAFKA.createTopic(YELL_A_TOPIC);           ← Создаем первый топик-источник
  EMBEDDED_KAFKA.createTopic(OUT_TOPIC);              ← Создаем топик для вывода результатов
}
```

Создать топики для тестирования желательно один раз для всех тестов, так что мы воспользовались аннотацией `@BeforeClass`, создающей нужные топики до выполнения всех тестов. Для данного теста нам достаточно будет топиков из одной секции с коэффициентом репликации 1, так что можно воспользоваться удобным методом `EmbeddedKafkaCluster.createTopic(String name)`. Если же нам требуется более одной секции или коэффициент репликации больше 1, то настройки по умолчанию нам не подойдут. В подобном случае можно воспользоваться одним из следующих перегруженных методов `createTopic`:

- `EmbeddedKafkaCluster.createTopic(String topic, int partitions, int replication)`;
- `EmbeddedKafkaCluster.createTopic(String topic, int partitions, int replication, Properties topicConfig)`.

По завершении этой подготовки к запуску встраиваемого кластера Kafka мы можем перейти к тестированию топологии с помощью встраиваемого брокера.

Тестирование топологии

Все готово. Для комплексного тестирования нам нужно сделать такие шаги.

1. Запустить приложение Kafka Streams.
2. Занести какие-нибудь записи в топик-источник и проконтролировать правильность результатов.
3. Создать новый топик, соответствующий шаблону.
4. Занести еще какие-нибудь дополнительные записи в новый топик и проконтролировать правильность результатов.

Начнем с первых двух частей теста (код можно найти в файле src/test/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java) (листинг 8.10).

Листинг 8.10. Запуск приложения и контроль первого набора значений

```
// Часть кода опущена для краткости

kafkaStreams = new KafkaStreams(streamsBuilder.build(), streamsConfig);
kafkaStreams.start();                                          ← Запускаем приложение Kafka Streams

List<String> valuesToSendList =                                ← Задаем список отправляемых значений
➥ Arrays.asList("this", "should", "yell", "at", "you");
List<String> expectedValuesList =
➥ valuesToSendList.stream()
                   .map(String::toUpperCase)                   ← Создаем список ожидаемых значений
                   .collect(Collectors.toList());

→ IntegrationTestUtils.produceValuesSynchronously(YELL_A_TOPIC,
                                                  valuesToSendList,
                                                  producerConfig,
                                                  mockTime);
  int expectedNumberOfRecords = 5;
  List<String> actualValues =
  ➥ IntegrationTestUtils.waitUntilMinValuesRecordsReceived(   ← Потребляем записи из Kafka
  ➥ consumerConfig, OUT_TOPIC, expectedNumberOfRecords);

  assertThat(actualValues, equalTo(expectedValuesList));       ← Проверяем, совпадают ли прочитанные значения с ожидаемыми

Генерируем значения для отправки во встраиваемый Kafka
```

Эта часть теста — вполне заурядный код тестирования. Мы проводим «инициализацию» потокового приложения, занося записи в топик-источник. Потоковое приложение уже работает, так что оно потребляет, обрабатывает и выводит записи в рамках стандартной обработки. Для проверки того, что приложение работает так, как ожидается, тест потребляет записи из топика узла-стока и сравнивает полученные значения с ожидаемыми.

Ближе к концу листинга 8.10 вы можете видеть два статических вспомогательных метода: `IntegrationTestUtils.produceValuesSynchronously` и `IntegrationTestUtils.waitUntilMinValuesRecordsReceived`, которые сильно упрощают создание этого комплексного теста. Данные вспомогательные методы генерации и потребления — часть `kafka-streams-test.jar`. Давайте обсудим их вкратце.

Генерация и потребление записей в тесте

Метод `IntegrationTestUtils.produceValuesSynchronously` создает `ProducerRecord` для каждой записи в коллекции с пустым ключом. Этот метод — синхронный, он получает итоговый объект `Future<RecordMetadata>`, возвращаемый методом `Producer.send` и немедленно вызывает `Future.get()`, блокирующий выполнение до возврата из запроса генератора. А поскольку этот метод отправляет записи синхронно, можно быть уверенными, что после возврата из метода записи уже доступны для потребления. Другой метод, `IntegrationTestUtils.produceKeyValuesSynchronously`, принимает в качестве аргумента коллекцию `KeyValue<K,V>` на случай, если вы хотите указать значения для ключей.

Для потребления записей в листинге 8.10 мы воспользовались методом `IntegrationTestUtils.waitUntilMinValuesRecordsReceived`. Как вы, наверное, поняли из его названия, он пытается прочитать ожидаемое число записей из заданного топика. По умолчанию он ждет до 30 секунд и, если ожидаемое число записей прочитано не было, генерирует исключение `AssertionError`, сигнализируя о том, что тест провален.

На случай, если вам требуется для работы потребленное `KeyValue`, а не просто значение, существует метод `IntegrationTestUtils.waitUntilMinKeyValueRecordsReceived`, который ведет себя аналогично, но возвращает коллекцию `KeyValue`. Кроме того, существуют перегруженные версии утилиты-потребителя, в которых можно задать длительность ожидания через параметр типа `long`.

А теперь завершим наше описание теста.

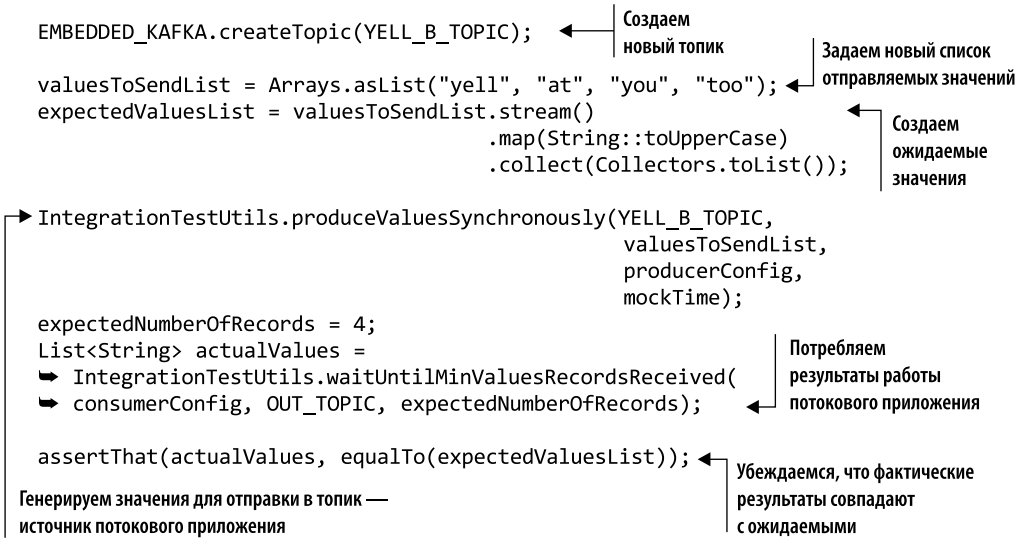
Динамическое добавление топика

Мы добрались до фазы тестирования динамического поведения, для чего нам требуется работающий брокер Kafka. Предыдущая часть теста служила для проверки начальных значений. Теперь же мы собираемся воспользоваться `EmbeddedKafkaCluster` для создания нового топика и убедиться, что приложение читает из этого топика и обрабатывает записи так, как ожидалось (см.: src/java/bbejeck/chapter_3/KafkaStreamsYellingIntegrationTest.java) (листинг 8.11).

Мы создали новый топик, соответствующий шаблону для узла-источника потокового приложения. После этого мы прошли через те же самые шаги заполнения нового топика данными и потребления записей из топика, обеспечивающего данными узел-источник потокового приложения. В конце теста мы проверяем, совпадают ли прочитанные результаты с ожидаемыми.

Вы можете запустить этот тест из своего IDE, и он должен выполняться успешно. Итак, мы завершили создание вашего первого комплексного теста!

Листинг 8.11. Запуск приложения и проверка значений



Комплексный тест не стоит использовать для всего чего угодно, поскольку модульные тесты легче в написании и сопровождении. Но комплексные тесты могут оказаться незаменимы в тех случаях, когда проверить поведение кода можно только с помощью работающего брокера Kafka.

ПРИМЕЧАНИЕ

У вас может возникнуть соблазн создавать все тесты с помощью EmbeddedKafkaCluster, но лучше так не поступать. Если запустить только что созданный нами пример комплексного теста, можно заметить, что он выполняется намного дольше модульных тестов. Несколько лишних секунд для одного теста — немного, но если умножить это время на несколько сотен или даже тысяч тестов, то расход времени на выполнение вашего набора тестов окажется весьма существенным. Кроме того, следует всегда стараться делать тесты как можно меньшими по размеру и посвященными одному конкретному элементу функциональности вместо того, чтобы задействовать весь технологический процесс приложения.

Резюме

- Старайтесь размещать бизнес-логику в автономных классах, полностью независимых от вашего приложения Kafka Streams. Это упрощает их модульное тестирование.
- Не помешает иметь хотя бы один тест, использующий `ProcessorTopologyTestDriver`, для сквозного тестирования топологии. Эти тесты выполняются быстро, поскольку не задействуют брокер Kafka.

- Старайтесь использовать имитационный фреймворк для проверки отдельных экземпляров `Processor` и `Transformer` только тогда, когда требуется проверить, как ведет себя какой-либо класс в API Kafka Streams.
- Комплексные тесты с `EmbeddedKafkaCluster` лучше применять редко, только в случае такого интерактивного поведения, которое можно проверить лишь при доступном работающем брокере Kafka.

Это была весьма интересная экскурсия в Kafka Streams, и вы узнали немало об API Kafka Streams и об его применении для решения задач по обработке данных. В завершение вашего обучения мы перейдем на более продвинутый уровень. Следующая, последняя глава этой книги представляет собой итоговый проект, который основан на всем том, что вы узнали до сих пор, а в некоторых случаях требует и написания пользовательского кода, отсутствующего в API Kafka Streams. В результате мы от начала и до конца создадим реальное приложение на основе описанной в этой книге базовой функциональности.

Часть IV
Передовые возможности Kafka Streams

В этой, заключительной, части книги мы соберем воедино все полученные вами знания и воспользуемся ими для создания продвинутого приложения. Вам предстоит интегрировать Kafka Streams с Kafka Connect, благодаря чему вы сможете осуществлять потоковую обработку данных даже в случае их записи в реляционную базу данных. Далее вы научитесь применять возможности интерактивных запросов для отображения — в режиме реального времени — информации вашего приложения непосредственно из Kafka Streams, без внешних утилит. Наконец, я расскажу вам о KSQL — новой утилите от компании Confluent (компании, основанной разработчиками Kafka из LinkedIn) — и научу вас писать операторы SQL и выполнять непрерывные запросы к поступающим в Kafka данным.

9 Создание продвинутых приложений с помощью Kafka Streams

В этой главе:

- интеграция внешних данных в Kafka Streams с помощью Kafka Connect;
- выбрасывание базы данных за борт благодаря интерактивным запросам;
- непрерывные запросы KSQL в Kafka.

Вы прошли долгий путь в своем стремлении научиться применению Kafka Streams. Мы проделали немало работы, и вы теперь уже умеете, я надеюсь, создавать потоковые приложения. До сих пор мы ограничивались базовой функциональностью Kafka Streams, но это не предел. В данной главе вам предстоит воспользоваться полученными знаниями для создания двух продвинутых приложений, подходящих для промышленного применения.

Например, во многих организациях новые технологии приходится интегрировать с унаследованными старыми технологиями и процессами. Нередко источником входящих данных служат таблицы баз данных. Как вы знаете из главы 5, таблицы — это потоки, так что таблицы баз данных можно рассматривать как потоки данных.

Первое продвинутое приложение, которым мы займемся в настоящей главе, «преобразует» физическую базу данных в потоковое приложение благодаря интеграции Kafka Connect с Kafka Streams. Kafka Connect будет прослушивать приложение на предмет новых вставок в таблицу (-ы) и помещать соответствующие записи в топик Kafka. Этот же топик будет служить источником для приложения Kafka Streams, так что таблица базы данных превратится в потоковое приложение.

При работе со старым, унаследованным ПО данные, даже при их захвате в режиме реального времени, обычно сбрасываются в базу данных, которая служит в дальнейшем источником информации для приложений-информационных панелей. Во втором продвинутом приложении этой главы вы научитесь пользоваться интерактивными запросами — средством, с помощью которого можно выполнять запросы напрямую к хранилищам состояния Kafka Streams. Приложения — информационные панели могут благодаря этому извлекать данные непосредственно из хранилищ со-

стояния и отображать их в процессе движения по потоковому приложению, а база данных становится не нужна.

А завершим мы наше обсуждение продвинутых функций замечательной новой возможностью Kafka — KSQL. KSQL позволяет писать «долгоиграющие» SQL-запросы к поступающим в Kafka данным. KSQL сочетает возможности Kafka Streams с удобством написания SQL-запроса. Внутренние механизмы KSQL используют Kafka Streams для выполнения запросов.

9.1. Интеграция Kafka с другими источниками данных

Для нашего первого продвинутого приложения предположим, что вы работаете в известной финансовой консультационной фирме Big Short Equity (BSE). BSE хочет начать вести обработку и анализ по-новому, и этот план включает использование Kafka. Переход на новые рельсы уже частично выполнен, и вам поставлена задача модернизировать используемые методы анализа. Цель состоит в отображении последних фондовых транзакций и сопутствующей информации в реальном времени, и Kafka Streams подходит для этого идеально.

BSE работает с ценными бумагами из разных областей финансового рынка. Они записывают фондовые транзакции в режиме реального времени в реляционную базу данных. Постепенно BSE планирует перейти на запись транзакций напрямую в Kafka, но пока системой учета служит база данных.

Как же сократить разрыв между базой данных, в которую отправляются поступающие данные, и создаваемым приложением Kafka Streams? Ответ: воспользоваться Kafka Connect (https://kafka.apache.org/documentation/#connect) — фреймворком, включенным в дистрибутив Apache Kafka и служащим для интеграции Kafka с другими системами. Достаточно, чтобы данные попали в Kafka, после чего вы можете не волноваться о расположении их источника; нужно просто указать вашему приложению Kafka Streams топик-источник, как вы уже делали с другими приложениями Kafka Streams.

ПРИМЕЧАНИЕ

> При использовании Kafka Connect для получения данных из других источников точкой сопряжения является топик Kafka. Это значит, что применять импортированные данные может *любое* приложение посредством интерфейса KafkaConsumer. Поскольку данная книга посвящена Kafka Streams, основное внимание я сосредоточу на интеграции с приложениями Kafka Streams.

Рисунок 9.1 иллюстрирует интеграцию между базой данных и Kafka. В этом случае мы используем Kafka Connect для мониторинга таблицы базы данных и потоковой отправки обновлений в топик Kafka, который далее будет служить источником для приложения финансовой аналитики.

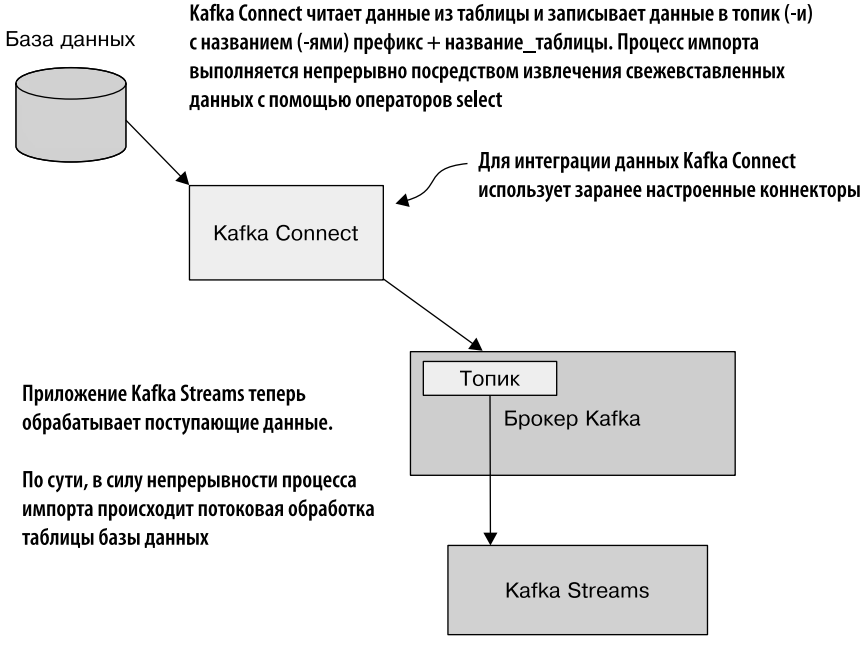


Рис. 9.1. Интеграция таблицы базы данных и Kafka Streams с помощью Kafka Connect

СОВЕТ

Поскольку эта книга посвящена Kafka Streams, мы лишь вкратце пробежимся по Kafka Connect в данном разделе. Более подробную информацию вы можете найти в документации Kafka (https://kafka.apache.org/documentation/#connect) и кратком руководстве по Kafka Connect (https://docs.confluent.io/current/connect/quickstart.html).

9.1.1. Интеграция данных с помощью Kafka Connect

Фреймворк Kafka Connect специально разработан для потоковой передачи данных из других систем в Kafka и из Kafka в другие системы хранения данных, например MongoDB (http://www.mongodb.com/) и Elasticsearch (http://www.elastic.co/). С помощью Kafka Connect можно импортировать в Kafka целые базы данных или другую информацию, например метрики производительности.

Для взаимодействия с внешними источниками данных Kafka Connect использует специальные *коннекторы* (connectors). Несколько коннекторов было доступно изначально (http://www.confluent.io/product/connectors), и множество других создано сообществом разработчиков коннекторов, что делает возможной интеграцию Kafka

практически с любой другой системой хранения. Если же нужного вам коннектора не существует, вы можете реализовать свой собственный (https://docs.confluent.io/current/connect/devguide.html).

9.1.2. Настройка Kafka Connect

Kafka Connect может работать в одном из двух режимов: распределенном и автономном. Для большинства сред промышленной эксплуатации имеет смысл запустить Kafka Connect в распределенном режиме, ведь при запуске нескольких экземпляров Kafka Connect можно воспользоваться выгодами параллелизма и отказоустойчивости. Я предполагаю, что вы запускаете примеры на локальной машине, так что применяю автономный режим.

Коннекторы Kafka Connect для взаимодействия с внешними источниками данных бывают двух типов: коннекторы источников и коннекторы стоков. На рис. 9.2 показано применение каждого из этих типов. Как вы можете видеть, коннекторы источников поставляют данные в Kafka, а коннекторы стоков получают их из Kafka (для использования в другой системе).

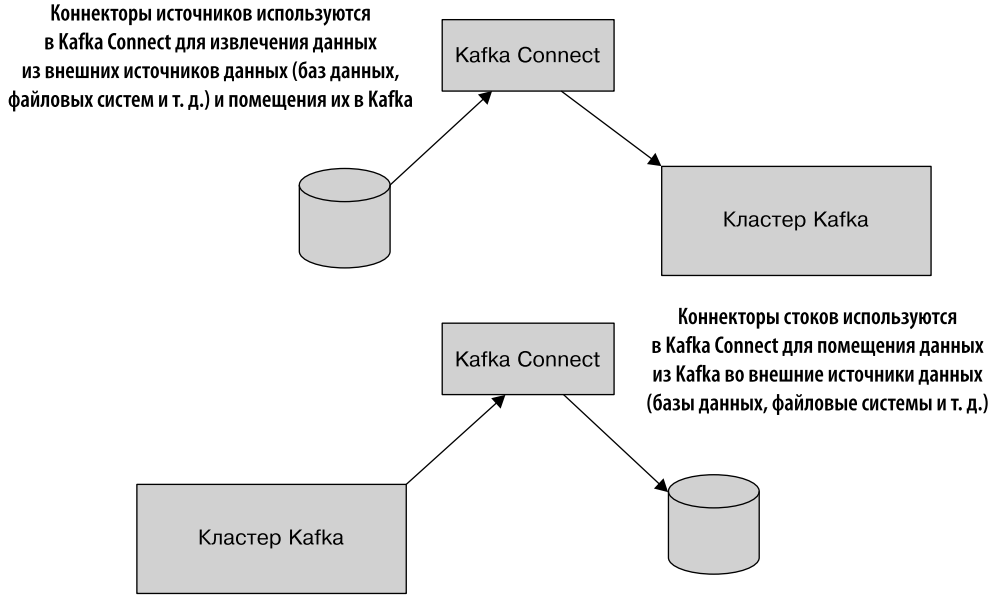


Рис. 9.2. Коннекторы источников и коннекторы стоков Kafka Connect

В этом примере мы воспользуемся JDBC-коннектором Kafka (https://github.com/confluentinc/kafka-connect-jdbc). Его можно найти на GitHub, кроме того, для удобства мы включили его в дистрибутив исходного кода для данной книги (https://manning.com/books/kafka-streams-in-action).

При использовании Kafka Connect вам потребуется выполнить небольшие настройки самого Kafka Connect и каждого из коннекторов, применяемых для импорта/экспорта данных. Во-первых, взглянем на параметры конфигурации, которые понадобятся нам для Kafka Connect:

- `bootstrap.servers` — разделенный запятыми список используемых Connect брокеров Kafka;
- `key.converter` — класс преобразователя, отвечающего за сериализацию ключей из формата Connect в формат, в котором данные записываются в Kafka;
- `value.converter` — класс преобразователя, отвечающего за сериализацию значений из формата Connect в формат, в котором данные записываются в Kafka. В нашем примере будет применяться встроенный класс `org.apache.kafka.connect.json.JsonConverter`;
- `value.converter.schemas.enable` — может быть равен `true` или `false`. Указывает, должен ли Connect включать схему для значения. В этом примере мы установим его равным `false`. В следующем разделе я объясню почему;
- `plugin.path` — содержит местоположение коннектора для Connect и его зависимостей. Можно указать один каталог верхнего уровня, содержащий один или несколько файлов JAR. Можно также указать несколько путей в виде разделенного запятыми списка;
- `offset.storage.file.filename` — файл, содержащий сохраненные потребителем Connect смещения.

Вам придется также указать несколько параметров конфигурации для JDBC-коннектора. Вот их список:

- `name` — название коннектора;
- `connector.class` — класс коннектора;
- `tasks.max` — максимальное число задач этого коннектора;
- `connection.url` — URL подключения к базе данных;
- `mode` — метод отслеживания изменений JDBC-коннектором для источника;
- `incrementing.column.name` — название столбца, по которому отслеживаются изменения;
- `topic.prefix` — Kafka Connect записывает все таблицы в топики с названиями `топик.префикс + название_таблицы`.

Большинство из этих настроек довольно просты, но две из них — `mode` и `incrementing.column.name` — нам придется обсудить подробнее, поскольку они играют важную роль в работе коннектора. JDBC-коннектор для источника определяет на основании параметра `mode`, какие строки загружать. В примере используется также параметр `incrementing.column.name`, ссылающийся на столбец, значение которого автоматически увеличивается на единицу при каждой вставке в таблицу. Благодаря отслеживанию этого столбца мы подтягиваем только *новые* записи, обновления

игнорируются. Нашему приложению Kafka Streams нужно подтягивать только самые свежие фондовые сделки, так что такое значение данной настройки оптимально.

СОВЕТ

В исходный код для данной книги включена почти полная конфигурация как для Kafka Connect, так и для JDBC-коннектора. Файлы настроек располагаются в каталоге src/main/resources/conf дистрибутива исходного кода (https://manning.com/books/kafka-streams-in-action). Вам остается только указать путь, куда вы разархивировали репозиторий исходного кода. Не забудьте прочитать полные инструкции в файле README.md.

На этом мы завершаем обзор Kafka Connect и JDBC-коннектора для источника. Нам осталось обсудить еще одну точку сопряжения, чем мы и займемся в следующем разделе.

ПРИМЕЧАНИЕ

Более подробную информацию о JDBC-коннекторе для источника вы можете найти в документации Confluent (http://mng.bz/01vh). Кроме того, существуют и другие заслуживающие внимания инкрементные режимы выполнения запросов (http://mng.bz/0pjP).

9.1.3. Преобразование данных

До получения этого нового задания вы уже разработали приложение Kafka Streams, работающее с аналогичными данными. В результате у вас уже есть модель и объекты `Serde` (где внутри для сериализации и десериализации используется Gson). Для ускорения разработки вам не хотелось бы писать дополнительный код для работы с Kafka Connect. Как вы увидите в следующем разделе, мы сможем «бесшовно» импортировать нужные данные из Kafka Connect.

СОВЕТ

Gson (https://github.com/google/gson) — разработанная компанией Google под лицензией Apache библиотека, предназначенная для сериализации и десериализации Java-объектов в формат JSON и из него. Узнать больше можно из следующего руководства пользователя: http://mng.bz/JqV2.

Чтобы сделать возможной такую бесшовную интеграцию, нам понадобится внести в свойства JDBC-коннектора еще несколько мелких изменений. Но прежде, чем сделать это, вернемся на минуту к подразделу 9.1.2, где мы обсуждали параметры конфигурации. А именно: я говорил, что мы будем использовать класс `org.apache.kafka.connect.json.JsonConverter` с отключенными схемами, а значит, значение преобразуется в простой формат JSON.

Хотя в нашем приложении Kafka Streams мы хотели бы потреблять данные именно в формате JSON, есть две проблемы. Во-первых, при преобразовании данных в формат JSON в качестве названий полей в итоговом JSON используются названия столбцов, которые заданы в сокращенном формате компании BSE, вне которой смысла не несут. Так что при преобразовании Gson-объекта `Serde` в ожидаемый объект модели все поля окажутся пустыми, поскольку названия не совпадут.

Во-вторых, время и дата хранятся в базе данных в виде метки даты/времени, как и следовало ожидать. Но в передаваемом Gson-объекте `Serde` не описан пользовательский `TypeAdapter` (http://mng.bz/inzB) для типа `Date`, так что все даты необходимо представить как `String` следующего вида: `yyyy-MM-dd'T'HH:mm:ss.SSS-0400`. К счастью, в Kafka Connect есть механизм, с помощью которого можно легко решить обе эти проблемы.

В Kafka Connect есть понятие преобразований (`Transformation`), благодаря которому можно выполнять *простые* преобразования перед тем, как Connect запишет данные в Kafka. На рис. 9.3 показано, где именно происходит этот процесс преобразования.

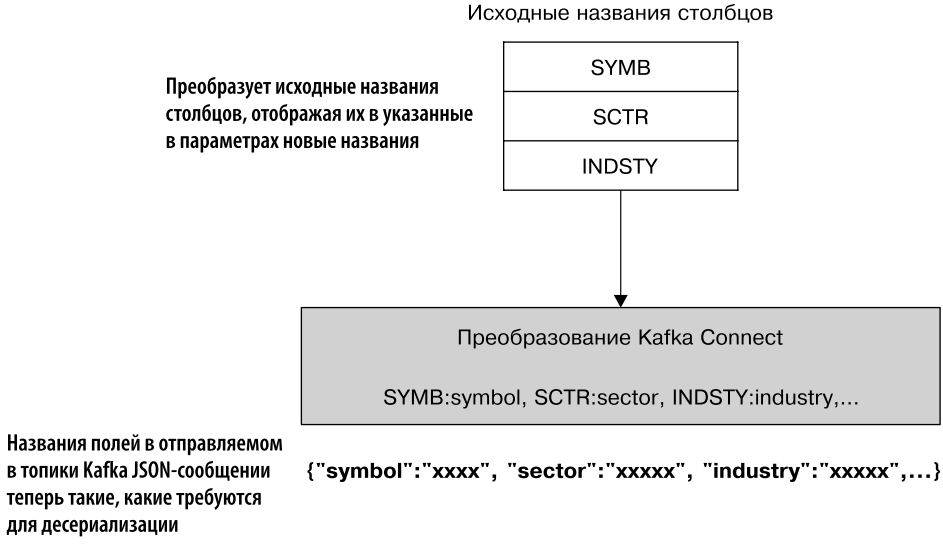


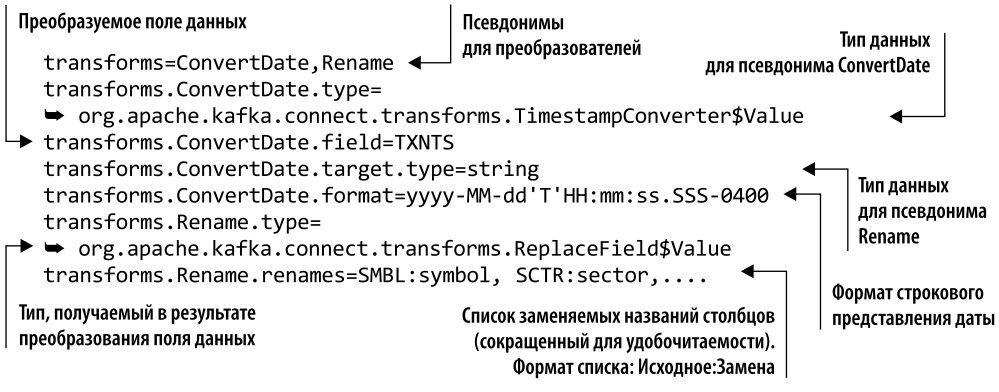
Рис. 9.3. Преобразование названий столбцов так, чтобы они соответствовали ожидаемым именам полей

В этом примере мы воспользуемся двумя встроенными преобразованиями: `TimestampConverter` и `ReplaceField`. Как уже упоминалось ранее, для их использования необходимо добавить следующие строки в файл конфигурации connector-jdbc.properties (см. src/main/resources/conf/connector-jdbc.properties) (листинг 9.1).

Эти свойства относительно понятны и не требуют дополнительного описания, так что мы не станем тратить на них время. Как вы можете видеть, благодаря им у вашего приложения Kafka Streams есть все, что требуется для успешной де-

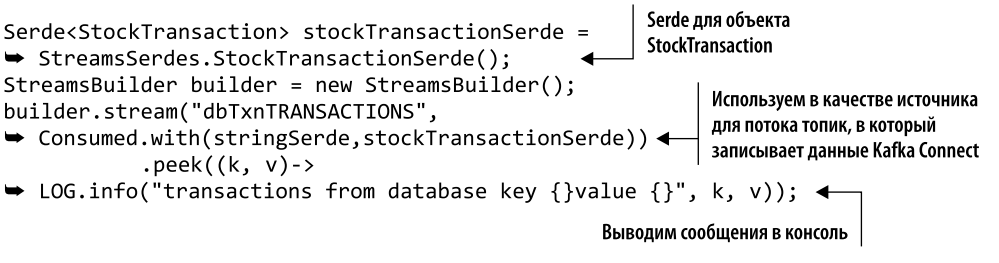
сериализации сообщений, импортированных в Kafka с помощью Kafka Connect и JDBC-коннектора.

Листинг 9.1. Свойства JDBC-коннектора



После подготовки Kafka Connect для завершения интеграции таблицы базы данных и вашего приложения Kafka Streams достаточно воспользоваться топиком с указанным в файле connector-jdbc.properties префиксом в названии (см.: src/main/java/bbejeck/chapter_9/StockCountsStreamsConnectIntegrationApplication.java) (листинг 9.2).

Листинг 9.2. Заполнение топика источника Kafka Streams данными из Connect



Пока мы производим потоковую обработку записей из таблицы базы данных в Kafka Streams, но это еще не все. Мы передаем в потоке данные о биржевых транзакциях, так что для анализа нам понадобится сгруппировать транзакции по символам акций.

Вы уже знаете, как выбрать ключ и повторно секционировать записи, но в смысле производительности лучше, если записи попадут в Kafka уже с ключами; в этом случае приложению Kafka Streams не потребуется шаг повторного секционирования, за счет отказа от которого можно сэкономить время и пространство на диске. Давайте снова взглянем на конфигурацию Kafka Connect.

Во-первых, можно добавить преобразователь `ValueToKey`, который бы принимал на входе список названий полей, подлежащих извлечению из значения и использованию в качестве ключа. Поменяйте файл `connector-jdbc.properties` так, как показано в листинге 9.3 (см.: src/main/resources/conf/connector-jdbc.properties).

Листинг 9.3. Модифицированные свойства JDBC-коннектора

```
transforms=ConvertDate,Rename,ExtractKey      ← Добавляем преобразование ExtractKey
transforms.ConvertDate.type=
➥ org.apache.kafka.connect.transforms.TimestampConverter$Value
transforms.ConvertDate.field=TXNTS
transforms.ConvertDate.target.type=string
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400
transforms.Rename.type=
➥ org.apache.kafka.connect.transforms.ReplaceField$Value
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
transforms.ExtractKey.type=                   ← Задаем имя класса
➥ org.apache.kafka.connect.transforms.ValueToKey   преобразования ExtractKey
transforms.ExtractKey.fields=symbol           ← Список полей, извлекаемых
                                                в качестве ключа
```

Мы добавили преобразование `ExtractKey` и указали Kafka Connect имя класса преобразователя: `ValueToKey`. Мы также указали название поля, которое будет служить ключом: `symbol`. Это свойство может состоять из нескольких разделенных запятыми значений, но в данном случае нам нужно только одно. Обратите внимание, что используется переименованная версия поля, поскольку данный преобразователь выполняется *после* преобразователя `Rename`.

Результат преобразования `ExtractKey` представляет собой структуру из одного значения. Но нам нужно только содержащееся в этой структуре значение для ключа — символ акции. Поэтому мы добавим преобразование `FlattenStruct` для извлечения только символа акции (см.: src/main/resources/conf/connector-jdbc.properties) (листинг 9.4).

Листинг 9.4. Добавляем еще одно преобразование

```
transforms=ConvertDate,Rename,ExtractKey,FlattenStruct   ← Добавляем
transforms.ConvertDate.type=                               последнее
➥ org.apache.kafka.connect.transforms.TimestampConverter$Value   преобразование
transforms.ConvertDate.field=TXNTS
transforms.ConvertDate.target.type=string
transforms.ConvertDate.format=yyyy-MM-dd'T'HH:mm:ss.SSS-0400
transforms.Rename.type=
➥ org.apache.kafka.connect.transforms.ReplaceField$Value
transforms.Rename.renames=SMBL:symbol, SCTR:sector,....
transforms.ExtractKey.type=org.apache.kafka.connect.transforms.ValueToKey
transforms.ExtractKey.fields=symbol
transforms.FlattenStruct.type=                             ← Задаем имя класса
➥ org.apache.kafka.connect.transforms.ExtractField$Key      преобразования
transforms.FlattenStruct.field=symbol     ← Название         (ExtractField$Key)
                                            извлекаемого поля
```

Мы добавили в топологию завершающее преобразование (`FlattenStruct`) и указали имя класса преобразования `ExtractField$Key`, используемого Kafka Connect для извлечения заданного поля и включения в результаты (в данном случае в ключ) только этого поля. Наконец, мы указали название поля (`symbol`) — такое же, как и в предыдущем преобразовании; это логично, поскольку именно данное поле использовалось при создании структуры для ключа.

С помощью всего нескольких строк настроек наше приложение Kafka Streams может теперь выполнять более продвинутые операции без необходимости выбора ключа и повторного секционирования (код можно найти в файле src/main/java/bbejeck/chapter_9/StockCountsStreamsConnectIntegrationApplication.java) (листинг 9.5).

Листинг 9.5. Обработка в Kafka Streams с помощью Connect транзакций из таблицы

```
Serde<StockTransaction> stockTransactionSerde =
➥ StreamsSerdes.StockTransactionSerde();
   StreamsBuilder builder = new StreamsBuilder();
   builder.stream("dbTxnTRANSACTIONS",
➥ Consumed.with(stringSerde, stockTransactionSerde))
          .peek((k, v)->
➥ LOG.info("transactions from database key {}value {}", k, v))
      .groupByKey(
➥ Serialized.with(stringSerde,stockTransactionSerde))
            .aggregate(()-> 0L,(symb, stockTxn, numShares) ->
➥ numShares + stockTxn.getShares(),
     Materialized.with(stringSerde, longSerde)).toStream()
        .peek((k,v) -> LOG.info("Aggregated stock sales for {} {}",k, v))
        .to( "stock-counts", Produced.with(stringSerde, longSerde));
```

Группировка по ключу

Выполняем агрегирование общего числа проданных акций

Благодаря тому что поступающие данные снабжены ключами, мы можем воспользоваться методом `groupByKey`, который не устанавливает флаг автоматического повторного секционирования. От операции группировки можно напрямую перейти к агрегированию без шага повторного секционирования, что важно по соображениям производительности. Включенный в исходный код для книги файл `README.md` содержит инструкции по запуску встраиваемой базы данных H2 (http://www.h2database.com/html/main.html) и Kafka Connect так, чтобы генерировать данные в необходимый для работы потокового приложения топик `dbTxnTRANSACTIONS`.

СОВЕТ

Хотя может показаться заманчивым применить преобразования повсеместно, импортируя данные в Kafka через Kafka Connect, помните: преобразования должны быть *простыми*. Для любых преобразований, кроме самых простых, показанных в наших примерах, лучше извлечь данные в Kafka и воспользоваться Kafka Streams для вычислительно сложных преобразований.

Мы разобрались, как с помощью Kafka Connect отправить данные в Kafka для дальнейшей обработки с помощью Kafka Streams. Время заняться вопросом визуализации состояния данных в режиме реального времени.

9.2. Выбрасываем базу данных за борт

Из главы 4 вы узнали, как добавить сохранение локального состояния в приложение Kafka Streams. Состояние необходимо потоковым приложениям для таких операций, как агрегирование, свертка и соединение. Локальное состояние необходимо всегда, разве что приложение работает исключительно с отдельными записями.

Возвращаясь к требованиям компании BSE, допустим, что вы разработали приложение Kafka Streams, захватывающее следующие данные по трем категориям фондовых операций:

- всего транзакций по сектору рынка;
- покупки акций клиентами, агрегированные по сеансам;
- общее число проданных акций по символам акций в разрезе «кувыркающихся» окон длительностью 0 секунд.

В предыдущих примерах мы или просматривали результаты в консоли, или читали их из топика-стока. Просмотр данных в консоли подходит для разработки, но консоль не лучшая среда для отображения результатов. Для нужд аналитики и для того, чтобы быстро разобраться, что к чему, гораздо лучше подойдет приложение — информационная панель.

Из этого раздела вы узнаете, как с помощью интерактивных запросов в Kafka Streams разработать приложение — информационную панель для просмотра аналитики *без* использования базы данных для хранения состояния. Приложение — информационная панель будет заполняться данными непосредственно из Kafka Streams по мере их потоковой обработки. Благодаря этому приложение будет непрерывно обновляться естественным образом.

В типичной архитектуре захватываемые/обрабатываемые данные помещаются в реляционную базу данных для просмотра. Такая архитектура показана на рис. 9.4: до появления Kafka Streams приходилось выполнять ввод и обработку данных с помощью Kafka, подавать их на вход аналитического движка, а затем помещать в таблицу реляционной базы данных, используемую приложением — информационной панелью.

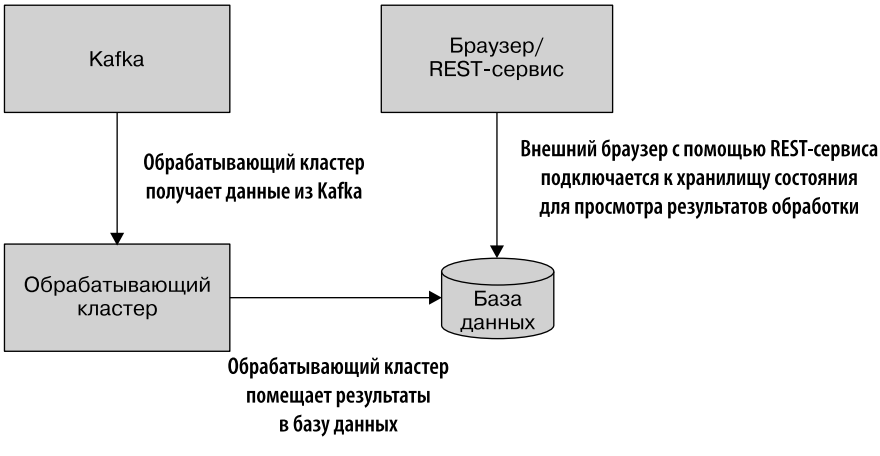


Рис. 9.4. Архитектура приложений для просмотра обработанных данных до появления Kafka Streams

Если добавить Kafka Streams с сохранением состояния, то архитектура слегка изменится, как показано на рис. 9.5. Структура существенно упрощается за счет исключения кластера (не говоря уже про упрощение развертывания). Kafka Streams

по-прежнему записывает данные в Kafka, а база данных остается основным потребителем преобразованных данных.

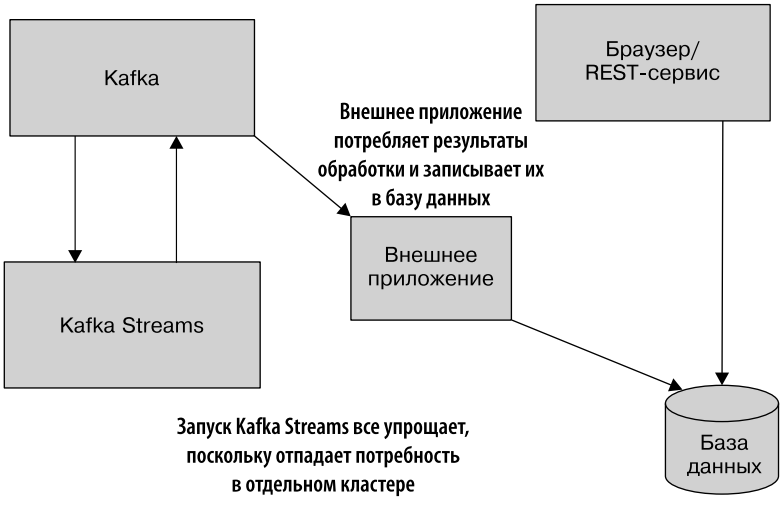


Рис. 9.5. Архитектура после добавления Kafka Streams с сохранением состояния

В главе 5 я говорил про интерактивные запросы. Напомню вам вкратце их определение: с помощью интерактивных запросов можно напрямую просматривать данные из хранилищ состояния, *без потребления данных из Kafka*. Другими словами, поток данных становится своего рода базой данных.

А поскольку один рисунок стоит тысячи слов, посмотрим еще раз на рис. 9.5, но уже с изменениями для использования интерактивных запросов (рис. 9.6).

Рис. 9.6. Та же архитектура, но уже с использованием интерактивных запросов

Показанная на этом рисунке идея — простая, но весьма многообещающая. В хранилище содержится состояние потока данных, а Kafka Streams предоставляет к нему доступ *только для чтения* извне потокового приложения посредством воплощающего REST интерфейса. Важно еще раз отметить широкие возможности такого подхода: для просмотра текущего состояния потока данных больше не требуется база данных.

Теперь, когда вы понимаете, насколько важны интерактивные запросы, посмотрим на них в действии.

9.2.1. Как работают интерактивные запросы

Чтобы интерактивные запросы могли работать, Kafka Streams предоставляет доступ (только для чтения) к хранилищам состояния. Важно понимать, что хранилища состояния, хотя для чтения и доступны, никак не обновляются и не модифицируются. Kafka Streams предоставляет доступ к хранилищам состояния с помощью метода `KafkaStreams.store`.

Вот как этот метод работает:

```
ReadOnlyWindowStore readOnlyStore =
➥ kafkaStreams.store(storeName, QueryableStoreTypes.windowStore());
```

В данном примере извлекается ссылка на объект `WindowStore`. Кроме того, в классе `QueryableStoreTypes` есть два других метода:

- `QueryableStoreTypes.sessionStore()`;
- `QueryableStoreTypes.keyValueStore()`.

После получения ссылки на (доступное только для чтения) хранилище достаточно предоставить сервису (например, воплощающему REST сервису) доступ к хранилищу, чтобы пользователи могли запрашивать информацию относительно состояния потоковых данных. Но получение ссылки на хранилище состояния — лишь часть задачи. Извлеченное выше хранилище состояния содержит ключи только из локального хранилища.

ПРИМЕЧАНИЕ

Не забывайте, что Kafka назначает по хранилищу состояния для каждой задачи и приложение Kafka Streams может состоять из нескольких экземпляров, лишь бы у них был один идентификатор приложения. Кроме того, эти экземпляры вовсе не обязаны располагаться на одной машине. Следовательно, может так случиться, что хранилище состояния, к которому производится запрос, содержит только некое подмножество ключей; а другие хранилища состояния (с тем же названием, расположенные на других машинах) содержат другое подмножество ключей.

Проясню эту концепцию на примере вышеприведенных аналитических операций.

9.2.2. Распределение хранилищ состояния

Рассмотрим первую из этих аналитических операций: агрегирование данных о биржевых сделках по секторам рынка. Поскольку производится агрегирование, нужны хранилища состояния. Для просмотра числа сделок по секторам рынка необходимо предоставить доступ к хранилищам состояния, чтобы узнать, в каком секторе в данный момент самая высокая активность.

Деятельность фондового рынка приводит к генерации колоссальных объемов данных, но в нашем примере для простоты будут использоваться только две секции. Кроме того, предположим, что у вас запущено два однопоточных экземпляра приложения на двух отдельных машинах, расположенных в одном ЦОД. Благодаря автоматической балансировке нагрузки Kafka Streams в каждом из экземпляров приложения одна задача будет обрабатывать данные из одной секции входного топика.

Рисунок 9.7 демонстрирует распределение задач и хранилищ состояния. Как вы можете видеть, экземпляр А обрабатывает записи из секции 0, а экземпляр Б — записи из секции 1.

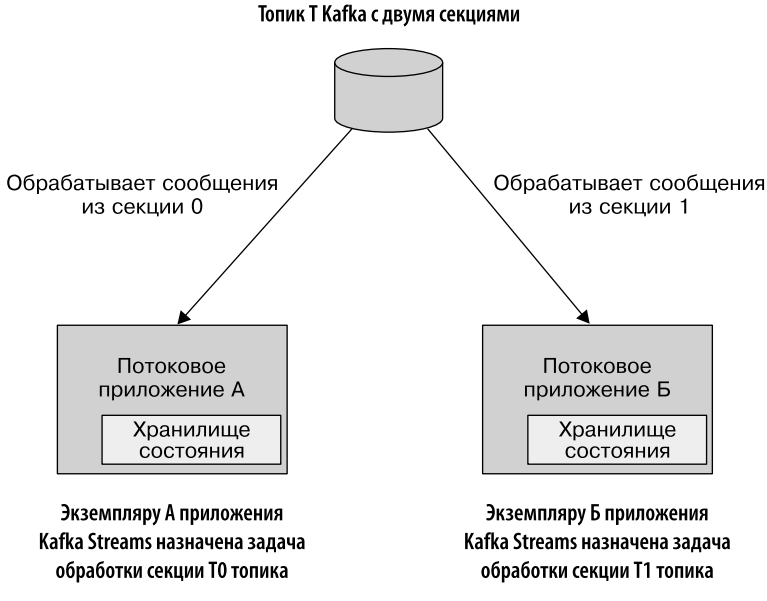


Рис. 9.7. Распределение задач и хранилищ состояния

Рисунок 9.8 иллюстрирует, что произойдет с двумя записями с ключами `"Energy"` и `"Finance"`.

`"Energy":"100000"` оказывается в хранилище состояния в экземпляре А приложения, а `"Finance":"110000"` попадает в хранилище состояния в экземпляре Б. Возвращаясь к примеру предоставления доступа к хранилищам состояния для выполнения запросов, можно видеть, что если сделать хранилище состояния на

экземпляре А доступным веб-сервису или другой внешней утилите для запросов, то из него можно будет извлечь только значение для ключа "Energy".

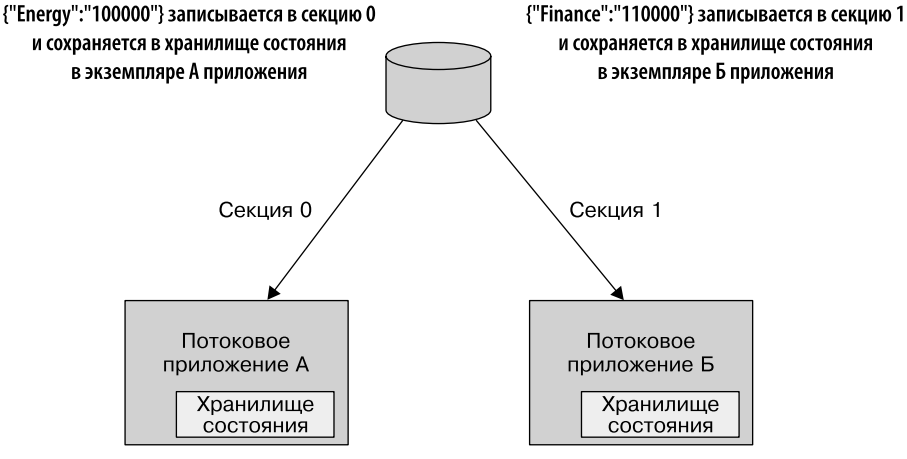


Рис. 9.8. Распределение ключей и значений в хранилищах состояния

Каково же решение этой проблемы? Безусловно, не имеет смысла использовать отдельный веб-сервис для запросов к каждому из экземпляров — подобный подход плохо масштабируется. К счастью, этого и не требуется: решение состоит просто в указании правильных параметров конфигурации.

9.2.3. Настройка и обнаружение распределенного хранилища состояния

Для активизации возможности выполнения интерактивных запросов необходимо задать параметр `StreamsConfig.APPLICATION_SERVER_CONFIG`. Его значение должно состоять из имени хоста приложения Kafka Streams и порта, на котором прослушивается сервис запросов, в формате `имя_хоста:порт`.

При получении экземпляром Kafka Streams запроса для заданного ключа необходимо выяснить, присутствует ли этот ключ в локальном хранилище. А главное, если ключ там отсутствует (не локальный), то необходимо узнать, в каком экземпляре находится данный ключ, и выполнить запрос к соответствующему хранилищу.

В классе `KafkaStreams` есть несколько методов, с помощью которых можно извлекать информацию из всех *работающих* в данный момент экземпляров с *одинаковым* идентификатором приложения и задавать значение параметра `APPLICATION_SERVER_CONFIG`. В табл. 9.1 приведены названия и описания этих методов.

Для получения информации обо всех экземплярах, доступных для интерактивных запросов, можно использовать метод `KafkaStreams.allMetadata`. При написании интерактивных запросов я чаще всего применяю метод `KafkaStreams.allMetadataForKey`.

Таблица 9.1. Методы для извлечения метаданных хранилища

Название	Параметр (-ы)	Использование
allMetadata	—	Все экземпляры, часть из которых, возможно, удаленные
allMetadataForStore	Название хранилища	Все экземпляры (часть из которых удаленные), содержащие указанное хранилище
allMetadataForKey	Key, Serializer	Все экземпляры (часть из которых удаленные), хранилище которых содержит заданный ключ
allMetadataForKey	Key, StreamPartitioner	Все экземпляры (часть из которых удаленные), хранилище которых содержит заданный ключ

Далее еще раз взглянем на распределение ключей/значений по экземплярам Kafka Streams, добавив последовательность действий проверки для ключа `"Finance"`, найденного в другом экземпляре и возвращенного из него (рис. 9.9). У каждого экземпляра Kafka Streams есть облегченный встраиваемый сервер, прослушивающий заданный в параметре `APPLICATION_SERVER_CONFIG` порт.

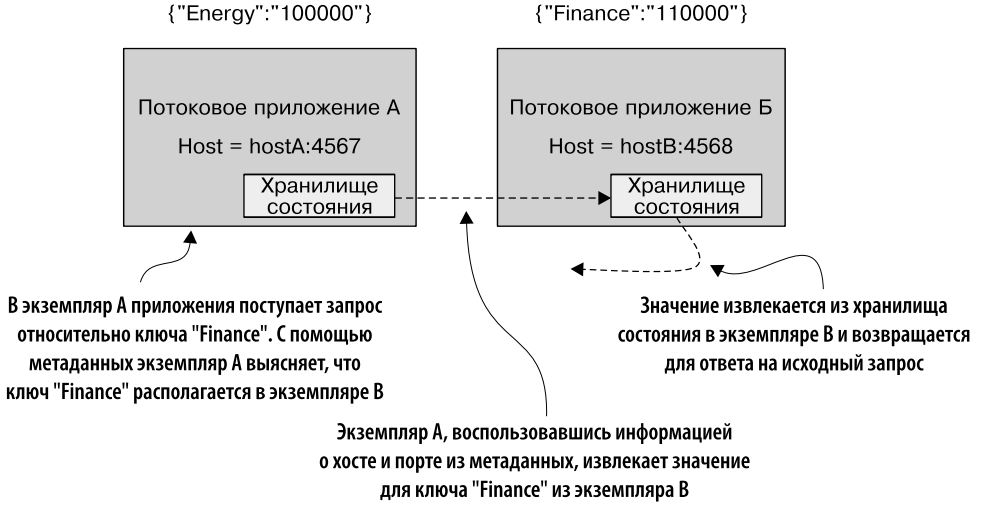


Рис. 9.9. Процесс запроса и обнаружения ключа и значения

Важно отметить, что достаточно отправить запрос только *одному* из экземпляров приложения Kafka Streams, причем какой экземпляр выбрать — неважно (конечно, если вы настроили приложение правильно). Если экземпляр, к которому вы обратились с запросом, не содержит искомых данных, то с помощью механизма RPC и методов извлечения метаданных он найдет, где они расположены, извлечет их и вернет результаты в ответ на исходный запрос.

Увидеть все это в действии можно посредством отслеживания последовательности вызовов на рис. 9.9. Экземпляр А не содержит ключа `"Finance"`, но он обнаруживает,

что искомый ключ есть в экземпляре Б. Поэтому экземпляр А обращается к встраиваемому серверу на экземпляре Б, который извлекает нужные данные и возвращает результат исходной запрашивающей стороне.

ПРИМЕЧАНИЕ

На отдельном узле интерактивные запросы будут работать без каких-либо дополнительных усилий, но готовый механизм RPC отсутствует — вам придется реализовать свой собственный. В этом разделе приведено одно из возможных решений, но вы можете реализовать свое, и я уверен, что многие из вас сумеют найти нечто получше моего варианта. Отличный пример еще одной реализации RPC можно найти в GitHub-репозитории Confluent kafka-streams-examples: http://mng.bz/Ogo3.

Взглянем теперь на интерактивные запросы в действии.

9.2.4. Написание кода для интерактивных запросов

Наше приложение для интерактивных запросов будет работать практически так же, как и все наши предыдущие приложения, но с несколькими небольшими отличиями. Во-первых, необходимо передавать два аргумента при запуске приложения Kafka Streams: имя хоста и порт, на котором будет слушать встраиваемый сервис (следующий код можно найти в файле src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java) (листинг 9.6).

Листинг 9.6. Передача параметров для имени хоста и порта

```
public static void main(String[] args) throws Exception {

  if(args.length < 2){
    LOG.error("Need to specify host and port");
    System.exit(1);
  }

  String host = args[0];
  int port = Integer.parseInt(args[1]);
  final HostInfo hostInfo = new HostInfo(host, port);

Properties properties = getProperties();
properties.put(
➥ StreamsConfig.APPLICATION_SERVER_CONFIG,host+":"+port);

// Остальные детали опущены для простоты
```

Создаем объект HostInfo для дальнейшего использования в приложении

Задаем значение параметра конфигурации, чтобы включить возможность применения интерактивных запросов

До сих пор мы запускали приложение не задумываясь. Теперь же нам нужно указать два параметра (хост и порт), но это лишь незначительное изменение.

Мы также встраиваем локальный сервер для собственно выполнения запросов: для данной реализации я выбрал веб-сервер Spark (http://sparkjava.com/). (Не *mom* Spark — это книга про Kafka Streams, в конце концов!) Основания для такого выбора — занимаемое им небольшое место на диске, подход «соглашения важнее

конфигурации», а также тот факт, что этот веб-сервер специально создан в расчете на микросервисы — а именно микросервисы удобно предоставлять с помощью интерактивных запросов. Если веб-сервер Spark вам не подходит, вы можете спокойно заменить его другим веб-сервером.

ПРИМЕЧАНИЕ

Полагаю, большинство читателей знакомо с понятием *«микросервис»* (microservice), но приведу все же лучшее из виденных мной определений — с сайта http://microservices.io/: «Микросервисы — известные также под названием микросервисной архитектуры — архитектурный стиль структурирования приложения в виде набора слабо сцепленных сервисов, реализующих бизнес-возможности. Микросервисная архитектура обеспечивает возможность непрерывной поставки/развертывания больших, сложных приложений, а также помогает организации расширять свой стек технологий».

Теперь взглянем на место в коде, где происходит встраивание сервера Spark, и вспомогательный код, служащий для управления им (его можно найти в файле src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java) (листинг 9.7).

Листинг 9.7. Инициализация веб-сервера и установка его состояния

```
// Подробности для простоты опущены

KafkaStreams kafkaStreams = new KafkaStreams(builder.build(), streamsConfig);
InteractiveQueryServer queryServer =
➥ new InteractiveQueryServer(kafkaStreams, hostInfo);
queryServer.init();

kafkaStreams.setStateListener(((newState, oldState) -> {
  if (newState == KafkaStreams.State.RUNNING && oldState ==
➥ KafkaStreams.State.REBALANCING) {
      LOG.info("Setting the query server to ready");
      queryServer.setReady(true);
   } else if (newState != KafkaStreams.State.RUNNING) {
      LOG.info("State not RUNNING, disabling the query server");
      queryServer.setReady(false);
   }
}));

kafkaStreams.setUncaughtExceptionHandler((t, e) -> {
     LOG.error("Thread {} had a fatal error {}", t, e, e);
     shutdown(kafkaStreams, queryServer);
});

Runtime.getRuntime().addShutdownHook(new Thread(() -> {
  shutdown(kafkaStreams, queryServer);
}));
```

Добавляем StateListener, чтобы разрешить запросы только к хранилищам состояний до готовности

Создаем встраиваемый веб-сервер (фактически класс-адаптер)

Активизируем возможность выполнения запросов к хранилищам состояния после перехода приложения Kafka Streams в состояние выполнения (RUNNING). В случае другого состояния приложения возможность выполнения запросов отключена

Описываем обработчик неперехваченных исключений для журналирования непредвиденных ошибок и завершения работы всех компонентов

Добавляем точку подключения для останова всех компонентов в случае штатного завершения работы приложения

В этом коде мы создали экземпляр `InteractiveQueryServer` — класса-адаптера для веб-сервера Spark и написали код для управления вызовами веб-сервисов, а также запуска и останова веб-сервера.

В главе 7 мы обсуждали использование интерфейса `StateListener` для уведомления о различных состояниях приложения Kafka Streams. А здесь демонстрируется его эффективное применение. Напомню, что при выполнении интерактивного запроса необходимо с помощью экземпляра `StreamsMetadata` выяснить, являются ли данные локальными по отношению к обрабатывающему запрос экземпляру приложения. Мы установили состояние сервера запросов в `true`, тем самым разрешив доступ к метаданным только в том случае, если приложение находится в состоянии выполнения.

Главное — помнить, что возвращаемые метаданные отображают мгновенное состояние структуры приложения Kafka Streams. В любой момент вам может понадобиться масштабировать приложение в сторону расширения или наоборот. В подобном случае (или в случае любого другого подходящего под требования события, например добавления топиков с узлом-источником на основе регулярного выражения) приложение Kafka Streams проходит через процесс перебалансировки, в результате чего распределение секций может поменяться. В данном случае применение запросов разрешается только в состоянии выполнения, но вы можете воспользоваться другой стратегией, если сочтете ее подходящей для своих целей.

Далее следует еще один пример идеи, описанной в главе 7: описание обработчика для неперехваченных исключений (`UncaughtExceptionHandler`). В данном случае мы заносим ошибку в журнал, а затем останавливаем приложение и сервер запросов. Поскольку это приложение работает в течение неограниченного времени, мы добавили еще точку подключения для останова всех компонентов после завершения вами работы демонстрационного приложения.

Мы разобрались, как инициализировать наш сервис и запустить его. Займемся теперь кодом, необходимым для работы сервера запросов.

9.2.5. Внутри сервера запросов

Первый шаг при реализации воплощающего REST-сервиса — поставить в соответствие URL-путям правильные методы для выполнения (см.: src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java) (листинг 9.8).

Этот код иллюстрирует наше решение использовать веб-сервер Spark: для обработки запроса можно лаконично отобразить URL в лямбда-выражения Java 8. Данные отображения довольно просты, но обратите внимание, что соответствие операции извлечения из оконного хранилища задается дважды. Для извлечения значений из оконного хранилища необходимо указать, *от* какого момента времени и *до* какого.

Обратите внимание на проверку булева значения `ready` при задании соответствий. Указанное значение задается в `StateListener`. Если значение `ready` равно

false, то программа не будет пытаться обработать запрос и вернет сообщение, что хранилища в настоящий момент недоступны. Это логично, поскольку оконное хранилище сегментировано по времени, причем размер сегментов задается при создании хранилища (мы обсуждали оконные операции в подразделе 5.3.2). Но я здесь немного жульничаю и предлагаю вам взглянуть в следующем примере на метод, принимающий в качестве параметров только ключ и хранилище со значениями по умолчанию для временны*х* параметров *«от»* и *«до»*.

Листинг 9.8. Ставим URL-путям в соответствие методы

```
public void init() {
  LOG.info("Started the Interactive Query Web server");

  get("/kv/:store", (req, res) -> ready ?
  ➥ fetchAllFromKeyValueStore(req.params()) :
  ➥ STORES_NOT_ACCESSIBLE);
  get("/session/:store/:key", (req, res) -> ready ?
  ➥ fetchFromSessionStore(req.params()) :
  ➥ STORES_NOT_ACCESSIBLE);
  get("/window/:store/:key", (req, res) -> ready ?
  ➥ fetchFromWindowStore(req.params()) :
  ➥ STORES_NOT_ACCESSIBLE);
  get("/window/:store/:key/:from/:to",(req, res) -> ready ?
  ➥ fetchFromWindowStore(req.params()) :
  ➥ STORES_NOT_ACCESSIBLE);
}
```

Соответствие пути методу для извлечения всех значений из обычного хранилища пар «ключ/значение»

Соответствие пути методу для возврата всех сеансов (из хранилища сеансов) для данного ключа

Соответствие пути методу для оконного хранилища без указания моментов времени

Соответствие пути методу для оконного хранилища с указанием, от какого момента времени и до какого

ПРИМЕЧАНИЕ

Существует предложение (KIP-205, http://mng.bz/lI9Y) по расширению класса ReadOnlyWindowStore — добавлению в него метода all(), который бы извлекал все временные сегменты по ключу, делая таким образом ненужным указание временны*х* параметров «от» и «до».

В качестве примера работы сервиса интерактивных запросов посмотрим на извлечение данных из оконного хранилища. Хотя я приведу только один пример, в исходном коде вы можете найти инструкции по выполнению всех типов запросов.

Проверка местоположения хранилища состояния

Вам нужно собрать различные метрики по ценным бумагам BSE для анализа данных о биржевых транзакциях. Вы решаете сначала отслеживать продажи отдельных типов акций, подсчитывая промежуточные итоги по 10-секундным окнам для обнаружения акций, демонстрирующих тенденцию к росту или падению.

Мы воспользуемся следующим отображением для нашего примера:

```
get("/window/:store/:key", (req, res) -> ready ?
➥ fetchFromWindowStore(req.params()) : STORES_NOT_ACCESSIBLE);
```

Чтобы не заблудиться в процессе выполнения запроса, задействуйте рис. 9.9 в качестве «дорожной карты». Мы начнем с отправки HTTP-запроса типа `GET`: `http://localhost:4567/window/NumberSharesPerPeriod/XXXX`, где `XXXX` соответствует тикерному символу для данных акций (см.: src/main/java/bbejeck/webserver/InteractiveQueryServer.java) (листинг 9.9).

Листинг 9.9. Отображение запроса и проверка местонахождения ключа

```
private String fetchFromWindowStore(Map<String, String> params) {
  String store = params.get(STORE_PARAM);
  String key = params.get(KEY_PARAM);
  String fromStr = params.get(FROM_PARAM);
  String toStr = params.get(TO_PARAM);

  HostInfo storeHostInfo = getHostInfo(store, key);

  if(storeHostInfo.host().equals("unknown")){
    return STORES_NOT_ACCESSIBLE;
  }

  if(dataNotLocal(storeHostInfo)){
    LOG.info("{} located in state store on another instance", key);
    return fetchRemote(storeHostInfo,"window", params);
  }
```

Извлекаем параметры запроса

Получаем объект HostInfo для ключа

Если имя хоста равно "unknown" («неизвестно»), возвращаем соответствующее сообщение

Проверяем, совпадает ли возвращенное имя хоста с именем хоста текущего экземпляра

Данный запрос отображается на метод `fetchFromWindowStore`. Прежде всего необходимо извлечь название хранилища и ключ (символ акции) из ассоциативного массива параметров запроса. Мы получаем объект `HostInfo` для запрашиваемого ключа и на основе содержащегося в нем имени хоста определяем, находится данный ключ в этом экземпляре или в каком-то удаленном.

Далее мы проверяем, происходит ли (повторная) инициализация экземпляра Kafka Streams, на которую указывает возврат методом `host()` значения `"unknown"`. Если да, то прекращаем обработку запроса и возвращаем сообщение `"not accessible"` («недоступен»).

Наконец, мы проверяем, совпадает ли возвращенное имя хоста с именем хоста текущего экземпляра. Если нет, получаем данные с содержащего ключ экземпляра и возвращаем результаты.

Далее посмотрим на извлечение и форматирование результатов (соответствующий код можно найти в файле src/main/java/bbejeck/webserver/InteractiveQueryServer.java) (листинг 9.10).

Я уже упоминал ранее, что мы сжульничаем немного с запросом к оконному хранилищу, если параметры *«от»* и *«до»* в запросе отсутствуют. Если пользователь не указал промежуток времени, то по умолчанию мы вернем результаты за последнюю минуту из оконного хранилища. А поскольку длительность окна у нас равна 10 секундам, то мы вернем результаты за шесть окон. После извлечения сегментов окон из хранилища мы проходим по ним в цикле, формируем ответ, включающий число акций, приобретенных за каждый 10-секундный интервал времени в течение последней минуты.

Листинг 9.10. Извлечение и форматирование результатов

```
Instant instant = Instant.now();
long now = instant.toEpochMilli();
long from = fromStr !=
➥ null ? Long.parseLong(fromStr) : now - 60000;
long to = toStr != null ? Long.parseLong(toStr) : now;

List<Integer> results = new ArrayList<>();

ReadOnlyWindowStore<String, Integer> readOnlyWindowStore =
➥ kafkaStreams.store(store,
➥ QueryableStoreTypes.windowStore());
try(WindowStoreIterator<Integer> iterator =
➥ readOnlyWindowStore.fetch(key, from , to)){
    while (iterator.hasNext()) {
      results.add(iterator.next().value);
    }
}
return gson.toJson(results);
```

Получаем текущее время в миллисекундах

Переменная, содержащая время начала оконного сегмента или, если оно не задано, время по состоянию на 1 минуту назад от текущего

Переменная, содержащая время окончания оконного сегмента или, если оно не задано, текущее время

Извлекаем объект ReadOnlyWindowStore

Получаем оконные сегменты

Формируем ответ

Преобразуем результаты в формат JSON и возвращаем запрашивающей стороне

Выполнение примера интерактивного запроса

Чтобы увидеть результаты работы указанного примера, необходимо выполнить три команды:

- `./gradlew runProducerInteractiveQueries` генерирует необходимые для примеров данные;
- `./gradlew runInteractiveQueryApplicationOne` запускает приложение Kafka Streams с портом 4567 в `HostInfo`;
- `./gradlew runInteractiveQueryApplicationTwo` запускает приложение Kafka Streams с портом 4568 в `HostInfo`.

Далее перейдите в браузере по адресу http://localhost:4568/window/NumberSharesPerPeriod/AEBB. Несколько раз нажмите кнопку Обновить — и увидите различные результаты. Вот статический список символов компаний для этого примера: `AEBB`, `VABC`, `ALBC`, `EABC`, `BWBC`, `BNBC`, `MASH`, `BARX`, `WNBC`, `WKRP`.

Запуск приложения — информационной панели для интерактивных запросов

Можно предложить лучший пример: миниатюрное веб-приложение — информационную панель, которое автоматически обновляется (с помощью Ajax) и отображает результаты четырех различных операций агрегирования Kafka Streams. Если вы запустили перечисленные в предыдущем разделе команды, то у вас все готово: для запуска приложения — информационной панели просто перейдите в браузере по адресу localhost:4568/iq или localhost:4567/iq. Перейдя в любой из экземпляров, вы

увидите, как интерактивные запросы Kafka Streams получают результаты из всех экземпляров приложения с одним идентификатором приложения. Полные инструкции по настройке и запуску приложения — информационной панели вы найдете в файле README в дистрибутиве исходного кода.

Как видите, вы можете просматривать результаты потока данных в реальном времени в приложении типа информационной панели. Ранее подобные приложения требовали реляционной базы данных, но теперь всю нужную информацию можно получить с помощью Kafka Streams.

На этом мы завершаем обсуждение интерактивных запросов. Перейдем к KSQL — потрясающей новой утилите, недавно выпущенной компанией Confluent (основанной первоначальными разработчиками Kafka из LinkedIn), с помощью которой можно выполнять «долгоиграющие» запросы к поступающим в Kafka записям посредством SQL, без написания кода.

9.3. KSQL

Представьте, что вы сотрудничаете с бизнес-аналитиками в BSE. Аналитиков заинтересовало ваше умение быстро писать приложения на Kafka Streams для анализа данных в режиме реального времени. Это ставит вас в затруднительное положение: с одной стороны, вы хотели бы сотрудничать с аналитиками и писать приложения для них, но у вас есть и своя работа — при дополнительной нагрузке вы не успеете ничего. Аналитики понимают, что создают для вас дополнительные проблемы, но они код писать не умеют, так что всецело зависят от вас.

Аналитики в совершенстве умеют работать с реляционными базами данных, так что SQL-запросы для них привычное дело. Если бы можно было создать SQL-слой поверх Kafka Streams, то производительность труда бы возросла — как аналитиков, так и ваша. Что ж, теперь такой слой есть.

В августе 2017 года компания Confluent представила новую мощную утилиту для потоковой обработки: KSQL (https://github.com/confluentinc/ksql#-ksql). KSQL представляет собой потоковый SQL-движок для Apache Kafka, предоставляющий интерактивный SQL-интерфейс, с помощью которого можно создавать полнофункциональные потоковые запросы *без* написания кода. KSQL особенно удобна для обнаружения мошенничества и для приложений, работающих в режиме реального времени.

ПРИМЕЧАНИЕ

KSQL — обширная тема для обсуждения, заслуживающая главы или двух, а может, и целой отдельной книги. Поэтому тут мы рассмотрим ее только вкратце. К счастью, вы уже знакомы с основными понятиями, лежащими в основе KSQL, поскольку ее внутренние механизмы используют Kafka Streams. Дополнительную информацию вы можете найти в документации KSQL (http://mng.bz/zw3F).

KSQL обеспечивает масштабируемую, распределенную потоковую обработку данных, включая агрегирование, соединения, оконные операции и др. Кроме того, в отличие от SQL-запросов к базе данных или системе пакетной обработки, результаты KSQL-запроса *непрерывны*. А сейчас мы займемся написанием потоковых запросов, но сначала остановимся на минуту и обсудим некоторые базовые понятия KSQL.

9.3.1. Потоки и таблицы KSQL

В подразделе 5.1.3 мы сравнивали понятия *потока событий* (event stream) и *потока обновлений* (update stream). Поток событий — неограниченный поток отдельных *независимых* событий, а поток обновлений — поток обновлений предыдущих записей с тем же ключом.

В KSQL есть аналогичное понятие запроса к `Stream` или `Table`. Объект `Stream` представляет собой бесконечную последовательность неизменяемых событий или фактов, а в случае запроса к `Table` факты могут обновляться или даже удаляться.

Хотя часть терминологии отличается, общая концепция остается неизменной. Если вы хорошо знакомы с Kafka Streams, то KSQL никаких трудностей у вас не вызовет.

9.3.2. Архитектура KSQL

KSQL применяет «под капотом» Kafka Streams для формирования и извлечения результатов запросов. KSQL состоит из двух компонентов: утилиты командной строки и сервера. Пользователи обычных SQL-утилит, например MySQL, Oracle и даже Hive, будут чувствовать себя как дома при написании KSQL-запросов в интерфейсе командной строки. А что самое лучшее, KSQL распространяется с открытым исходным кодом (лицензия Apache 2.0).

В KSQL есть также клиент командной строки для соединения с сервером KSQL. KSQL-сервер отвечает за обработку запросов и извлечение данных из Kafka, а также за запись результатов в Kafka.

KSQL может работать в двух режимах: *автономном* (standalone), удобном при разработке и создании предварительных версий, и *распределенном* (distributed), который вы, конечно, будете использовать при работе с данными в более реалистических масштабах. На рис. 9.10 показано функционирование KSQL в локальном режиме. Как вы можете видеть, интерфейс командной строки KSQL, сервер REST и движок KSQL — все расположены на одной виртуальной машине, что оптимально для работы на вашем ноутбуке.

Теперь взглянем на KSQL в распределенном режиме (рис. 9.11). Интерфейс командной строки KSQL располагается отдельно и подключается к одному из удаленных серверов KSQL (мы обсудим запуск и подключение в следующем разделе). Главное то, что, хотя вы подключаетесь явным образом к одному конкретному удаленному серверу KSQL, все относящиеся к одному кластеру Kafka серверы возьмут на себя часть работы по выполнению полученного запроса.

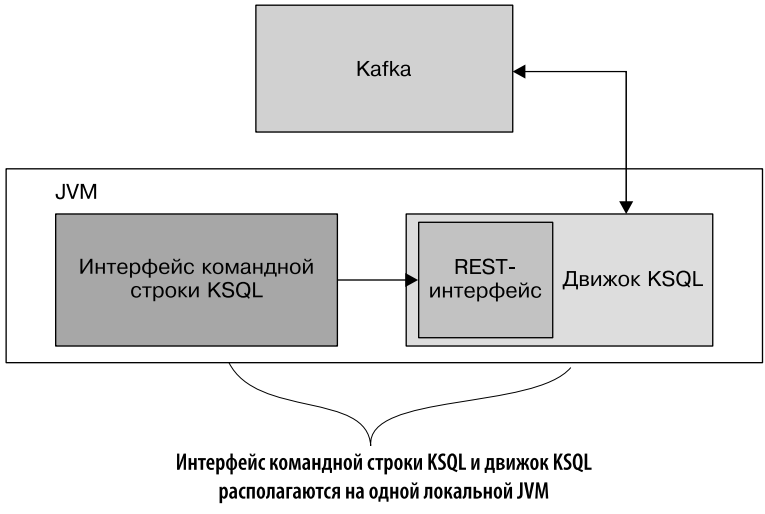


Рис. 9.10. KSQL в локальном режиме

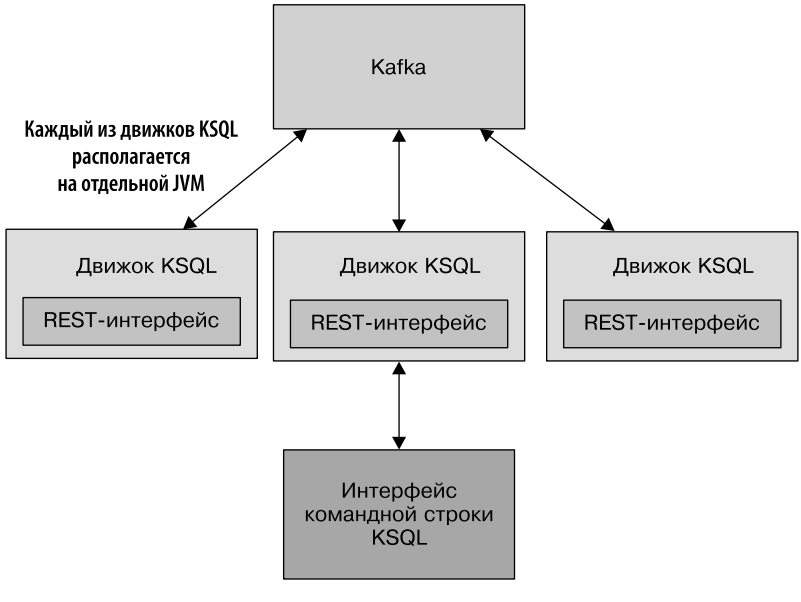


Рис. 9.11. KSQL в распределенном режиме

Заметим, что для выполнения запросов сервера KSQL используют Kafka Streams. Это значит, что при необходимости в дополнительных вычислительных мощностях можно просто запустить еще один сервер KSQL, даже и прямо во время работы (подобно запуску дополнительных экземпляров приложения Kafka Streams). Обратное также справедливо: в случае лишних вычислительных мощностей можно остановить

произвольное число серверов KSQL при условии, что хотя бы один останется в рабочем состоянии (иначе ваши запросы перестанут выполняться!).

А теперь посмотрим, как установить и запустить KSQL.

9.3.3. Установка и запуск KSQL

Для установки KSQL мы клонируем репозиторий KSQL с помощью команды `git clone git@github.com:confluentinc/ksql.git`, после чего перейдем (`cd`) в каталог ksql и выполним команду `mvn clean package` для сборки всего проекта KSQL. Если `git` у вас не установлен или вы не хотите выполнять сборку из исходного кода, можете скачать дистрибутив KSQL по адресу http://mng.bz/765U.

СОВЕТ

> KSQL представляет собой проект на основе Apache Maven, так что для его сборки вам понадобится Maven. Если он у вас не установлен, но вы работаете на компьютере под управлением операционной системы Mac OS и у вас установлена система управления пакетами Homebrew, то просто выполните команду brew install maven. В противном случае вы можете перейти по адресу https://maven.apache.org/download.cgi и скачать Maven; инструкции по установке можно найти на странице https://maven.apache.org/install.html.

Прежде чем продолжать, убедитесь, что вы находитесь в корневом каталоге проекта KSQL. Следующий шаг — запуск KSQL в локальном режиме:

```
./bin/ksql-cli local
```

Отмечу, что во всех примерах мы будем использовать KSQL в локальном режиме, но опишем и как запустить его в режиме распределенном.

После выполнения предыдущей команды вы увидите в своей консоли нечто напоминающее рис. 9.12. Поздравляю — вы успешно установили и запустили KSQL! Приступим к написанию запросов.

Рис. 9.12. Результаты успешного запуска KSQL

9.3.4. Создание потока данных KSQL

Вернемся к вашей деятельности на благо компании BSE. К вам обратился аналитик, заинтересовавшийся одним из написанных вами приложений и хотевший внести в него несколько небольших изменений. Но это пожелание не оказывается для вас источником дополнительной работы: вы просто запускаете консоль KSQL и говорите аналитику переделать ваше приложение в виде KSQL-запроса!

Пример, который мы собираемся переделать таким образом, представляет собой последний оконный поток данных из примера интерактивных запросов. Его можно найти в файле src/main/java/bbejeck/chapter_9/StockPerformanceInteractiveQueryApplication.java, строки 96–103. В этом приложении мы отслеживали число акций, проданных за каждый 10-секундный промежуток времени, сгруппированных по тикерному символу компании.

У нас уже описан топик (топик, соответствующий таблице базы данных) и объект модели, `StockTransaction`, где поля объекта соответствуют столбцам таблицы. Хотя топик уже описан, нам нужно зарегистрировать его в KSQL с помощью оператора `CREATE STREAM` в src/main/resources/ksql/create_stream.txt (листинг 9.11).

Листинг 9.11. Создание потока данных

```
CREATE STREAM stock_txn_stream (symbol VARCHAR, sector VARCHAR, \
  industry VARCHAR, shares BIGINT, sharePrice DOUBLE, \
  customerId VARCHAR, transactionTimestamp STRING, purchase BOOLEAN) \
WITH (VALUE_FORMAT = 'JSON', KAFKA_TOPIC = 'stock-transactions');
```

Оператор CREATE STREAM для создания потока данных с названием stock_txn_stream

Регистрируем в качестве названий столбцов поля объекта StockTransaction

Указываем формат данных и топик Kafka, служащий источником потока данных (оба параметра обязательны)

С помощью одного этого оператора мы создали экземпляр KSQL Streams, к которому сможем далее направлять свои запросы. У предложения `WITH` есть два обязательных параметра: `VALUE_FORMAT`, сообщающий KSQL формат данных, и `KAFKA_TOPIC`, указывающий KSQL, откуда извлекать данные. Можно также использовать в предложении `WITH` при создании потока данных два дополнительных параметра. Первый из них — `TIMESTAMP`, связывающий метку даты/времени сообщения со столбцом в потоке KSQL. Этот столбец будет использоваться при обработке записи в операциях, для которых требуется метка даты/времени, например в оконных операциях. Второй параметр — `KEY`, связывающий ключ сообщения со столбцом в заданном потоке. В таком случае ключ сообщения для топика `stock-transactions` соответствует полю `symbol` в JSON-значении, так что указывать ключ не нужно. В ином случае понадобилось бы задать соответствие ключа поименованному столбцу, поскольку для операций группировки всегда нужен ключ. Мы еще увидим это, когда будем выполнять потоковый SQL.

СОВЕТ

Команда KSQL list topics; выводит список топиков на брокере, на который указывает интерфейс командной строки KSQL, включая информацию о том, зарегистрированы ли они.

Можно также просмотреть список потоков данных и убедиться, что KSQL создал наш новый поток с помощью следующих команд:

```
show streams;
describe stock_txn_stream;
```

Результаты их выполнения приведены на рис. 9.13. Обратите внимание, что KSQL вставил в поток два дополнительных столбца: `ROWTIME` и `ROWKEY`. Столбец `ROWTIME` представляет собой метку даты/времени сообщения (добавляемую генератором или брокером), а `ROWKEY` — ключ сообщения (если таковой имеется).

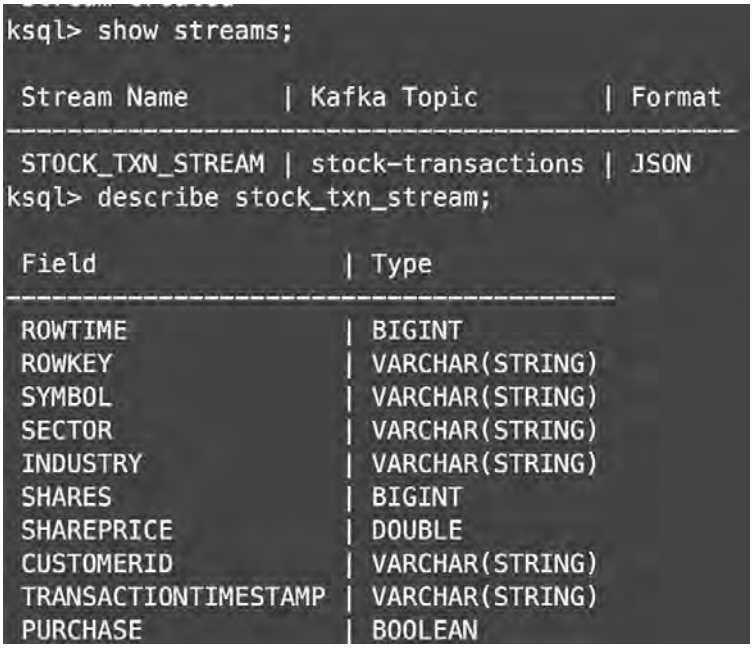


Рис. 9.13. Вывод списка потоков данных и вывод описания только что созданного потока

А теперь выполним запрос к этому потоку.

ПРИМЕЧАНИЕ

Чтобы подготовить данные для примеров KSQL, необходимо выполнить команду ./gradlew runProducerInteractiveQueries.

9.3.5. Написание KSQL-запроса

KSQL-запрос для биржевого анализа выглядит следующим образом:

```
SELECT symbol, sum(shares) FROM stock_txn_stream
➥ WINDOW TUMBLING (SIZE 10 SECONDS) GROUP BY symbol;
```

Выполните этот запрос, и вы увидите примерно такие результаты, какие показаны на рис. 9.14. Столбец слева представляет собой тикерный символ акций, а число справа — количество проданных акций с таким символом за последние 10 секунд. В этом запросе мы воспользовались «кувыркающимся» окном, но KSQL поддерживает также сеансовые и «прыгающие» окна, которые мы обсуждали в подразделе 5.3.2.

Рис. 9.14. Результаты запроса с «кувыркающимся» окном

Мы создали потоковое приложение без единой строки кода — чем не достижение? Для сравнения посмотрим на аналогичное приложение, написанное с помощью API Kafka Streams (листинг 9.12).

Листинг 9.12. Приложение для биржевого анализа, написанное на Kafka Streams

```
KStream<String, StockTransaction> stockTransactionKStream =
➥ builder.stream(MockDataProducer.STOCK_TRANSACTIONS_TOPIC,
                  Consumed.with(stringSerde, stockTransactionSerde)
         .withOffsetResetPolicy(Topology.AutoOffsetReset.EARLIEST));

Aggregator<String, StockTransaction, Integer> sharesAggregator =
➥ (k, v, i) -> v.getShares() + i;

stockTransactionKStream.groupByKey()
    .windowedBy(TimeWindows.of(10000))
    .aggregate(() -> 0, sharesAggregator,
         Materialized.<String, Integer,
         WindowStore<Bytes,
         byte[]>>as("NumberSharesPerPeriod")
             .withKeySerde(stringSerde)
             .withValueSerde(Serdes.Integer()))
    .toStream().
➥ peek((k,v)->LOG.info("key is {} value is{}", k, v));
```

Хотя код с использованием API Kafka Streams достаточно лаконичен, но его эквивалент в KSQL занимает *одну* строку. Прежде чем завершить обсуждение утилиты KSQL, взглянем еще на несколько ее возможностей.

9.3.6. Создание таблицы KSQL

Ранее мы продемонстрировали создание потока данных KSQL. Теперь посмотрим, как создать таблицу KSQL с топиком `stock-transactions` в качестве источника, чтобы вам было привычнее (этот запрос можно найти в файле src/main/resources/ksql/create_table.txt) (листинг 9.13).

Листинг 9.13. Создание KSQL-таблицы

```
CREATE TABLE stock_txn_table (symbol VARCHAR, sector VARCHAR, \
                              industry VARCHAR, shares BIGINT, \
                              sharePrice DOUBLE, \
                              customerId VARCHAR, transactionTimestamp \
                              STRING, purchase BOOLEAN) \
                              WITH (KEY='symbol', VALUE_FORMAT = 'JSON', \
                              KAFKA_TOPIC = 'stock-transactions');
```

После создания таблицы можно сразу выполнять к ней запросы. Имейте в виду, что таблица будет содержать обновления для транзакций по `symbol`, поскольку ключами в топике `stock-transactions` служат тикерные символы акций.

В качестве интересного эксперимента выберем какой-нибудь символ акций из потокового запроса показателей акций, после чего выполним следующие запросы в консоли KSQL и посмотрим на разницу результатов:

```
select * from stock_txn_stream where symbol='CCLU';
select * from stock_txn_table where symbol='CCLU';
```

Первый запрос — к потоку отдельных событий — возвращает несколько результатов. А запрос к таблице возвращает намного меньше результатов (одну запись в моем случае). Именно такое поведение и ожидается, поскольку таблица представляет обновления фактов, а поток данных — последовательность неограниченных событий.

9.3.7 Настройка KSQL

KSQL предоставляет пользователю знакомый синтаксис SQL и возможность быстрого написания полнофункциональных потоковых приложений, но вы могли заметить отсутствие конфигурации. Это не значит, что возможностей настройки KSQL нет. Вы можете переопределять любые нужные вам настройки, а также применять любые настройки потока данных, потребителя и генератора, которые только можно задавать для приложений Kafka Streams. Для просмотра текущих настроек выполните команду `show properties;`.

В качестве примера задания свойства конфигурации рассмотрим, как изменить значение параметра `auto.offset.reset` на `earliest`:

```
SET 'auto.offset.reset'='earliest';
```

Аналогичным образом можно задать в командной оболочке KSQL значение любого свойства. Но если нужно задать несколько параметров конфигурации, вводить каждый из них в консоли не слишком удобно. Вместо этого можно указать при запуске файл конфигурации:

```
./bin/ksql-cli local --properties-file /path/to/configs.properties
```

Это был лишь краткий обзор утилиты KSQL, но я надеюсь, вы оценили возможности и гибкость, предоставляемые ею для создания потоковых приложений на Kafka.

Резюме

- С помощью Kafka Connect можно использовать в своих приложениях Kafka Streams другие источники данных.
- Интерактивные запросы — мощный инструмент: с их помощью можно просматривать данные потока в процессе их прохождения через приложение Kafka Streams без применения реляционной базы данных.
- Язык KSQL помогает быстро создавать полнофункциональные потоковые приложения без написания кода. KSQL позволяет обычным сотрудникам, не программистам, использовать мощь и гибкость Kafka Streams.

Приложения

A Дополнительная информация о настройках

Данное приложение охватывает часто и не очень часто используемые настройки конфигурации приложений Kafka Streams. При чтении книги вы неоднократно встречали примеры описания конфигурации приложений Kafka Streams, но эта конфигурация включала обязательные (идентификатор приложения, серверы начальной загрузки) настройки и лишь несколько других (объекты `Serde` для ключей и значений). В этом приложении я расскажу вам об остальных настройках, хотя и необязательных, но весьма полезных для бесперебойной работы приложений Kafka Streams. Настройки будут описаны в стиле справочника с примерами применения.

A.1. Ограничение количества перебалансировок при запуске приложения

`GroupCoordinator` брокера распределяет все секции топиков первому запускаемому экземпляру приложения Kafka Streams. При запуске следующего экземпляра происходит перебалансировка, в результате которой текущие назначения секций топиков сбрасываются и секции топиков распределяются заново по обоим экземплярам Kafka Streams. Этот процесс повторяется до тех пор, пока не запустятся все экземпляры приложения Kafka Streams с одним идентификатором приложения.

Для приложения Kafka Streams такое поведение — норма. Но обработка записей приостанавливается на время перебалансировки, до ее завершения; следовательно, желательно при запуске приложения ограничить по возможности число перебалансировок.

В версии Kafka 0.11.0 появилась новая настройка брокеров — `group.initial.rebalance.delay.ms`. Эта настройка откладывает начальную перебалансировку потребителя при его присоединении к группе на указанное в `group.initial.rebalance.delay.ms` время (в миллисекундах). Значение данной настройки по умолчанию равно 3 секундам. По мере присоединения к группе других потребителей перебалансировка откладывается на заданный промежуток времени (вплоть до достижения предела,

задаваемого параметром `max.poll.interval.ms`). Это полезно для Kafka Streams, поскольку перебалансировка при запуске новых экземпляров откладывается до момента включения их всех в работу (если они запускаются по очереди). Например, если запустить четыре экземпляра приложения, задав подходящую задержку перебалансировки, то будет выполнена одна перебалансировка после запуска всех четырех экземпляров, а значит, вы сможете быстрее начать обрабатывать данные.

А.2. Устойчивость к отказам брокеров

Для обеспечения отказоустойчивости приложения Kafka Streams на случай сбоев брокеров рекомендуется использовать следующие настройки (листинг А.1):

- установить параметр `Producer.NUM_RETRIES` равным `Integer.MAX_VALUE`;
- установить параметр `Producer.REQUEST_TIMEOUT` равным `305000` (чуть более 5 минут);
- установить параметр `Producer.BLOCK_MS_CONFIG` равным `Integer.MAX_VALUE`;
- установить параметр `Consumer.MAX_POLL_CONFIG` равным `Integer.MAX_VALUE`.

Листинг А.1. Задание настроек для обеспечения устойчивости к выходу брокеров из строя

```
Properties props = new Properties();
props.put(StreamsConfig.producerPrefix(
➥ ProducerConfig.RETRIES_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.producerPrefix(
➥ ProducerConfig.MAX_BLOCK_MS_CONFIG), Integer.MAX_VALUE);
props.put(StreamsConfig.REQUEST_TIMEOUT_MS_CONFIG, 305000);
props.put(StreamsConfig.consumerPrefix(
➥ ConsumerConfig.MAX_POLL_INTERVAL_MS_CONFIG), Integer.MAX_VALUE);
```

Благодаря заданию таких настроек в случае аварийного останова всех брокеров кластера Kafka ваше приложение Kafka Streams останется работоспособным и будет готово возобновить работу после восстановления брокеров.

А.3. Обработка ошибок десериализации

Kafka работает с байтовыми массивами ключей и значений, и для их использования ключи и значения нужно десериализовать. Именно поэтому для всех узлов-источников и узлов-стоков требуются объекты `Serde`. Появление испорченных данных во время обработки записей вполне возможно. Поэтому для указания, как поступать с этими ошибками десериализации, в Kafka Streams существуют параметры `default.deserialization.exception.handler` и `StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_CONFIG`.

Значение по умолчанию для этих параметров — класс `org.apache.kafka.streams.errors.LogAndFailExceptionHandler`, выполняющий, как можно догадаться из

названия, журналирование ошибки. При такой настройке генерация исключения десериализации приведет к сбою (аварийному останову) приложения Kafka Streams. Другой класс, `org.apache.kafka.streams.errors.LogAndContinueExceptionHandler`, заносит ошибку в журнал, но приложение Kafka Streams при этом продолжает работать.

Вы можете реализовать свои собственные обработчики исключений десериализации путем создания класса, реализующего интерфейс `DeserializationExceptionHandler` (листинг А.2).

Листинг А.2. Задание обработчика исключений десериализации

```
Properties props = new Properties();
props.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HANDLER_CLASS_
➥ CONFIG, LogAndContinueExceptionHandler.class);
```

Я показал тут только задание обработчика `LogAndContinueExceptionHandler`, поскольку вариант журналирования с последующим остановом используется по умолчанию.

А.4. Масштабирование приложения

Во всех примерах этой книги приложения Kafka Streams работают с одним потоком выполнения. Для разработки такой режим вполне подходит, но при реальной эксплуатации, вероятно, вам понадобится более одного потока. Вопрос в том, сколько нужно потоков выполнения и сколько — экземпляров Kafka Streams. Однозначных ответов на эти вопросы не существует, поскольку только вы сами достаточно хорошо знаете свои обстоятельства для ответа на них, но мы можем привести тут простые выкладки, которые помогут вам в выборе.

Как вы помните из главы 3, Kafka Streams создает объекты `StreamTask` по числу секций входного топика (-ов). В нашем первом примере для простоты мы рассмотрим входной топик из 12 секций.

В случае 12 входных секций Kafka Streams создаст 12 задач. Допустим, нам нужно по одной задаче на поток выполнения. Можно использовать один экземпляр с 12 потоками выполнения, но у такого подхода есть недостаток: в случае сбоя машины, на которой запущено приложение Kafka Streams, вся потоковая обработка прекратится.

Но если запускать экземпляры с четырьмя потоками выполнения каждый, то каждый экземпляр будет обрабатывать четыре входные секции. Преимущество такого подхода: в случае сбоя одного из экземпляров Kafka Streams будет запущена перебалансировка и четыре задачи неработающего экземпляра будут перераспределены по двум остальным, таким образом, оставшиеся экземпляры будут обрабатывать по шесть задач каждый. Кроме того, после возобновления работы остановленного экземпляра произойдет еще одна перебалансировка и все три экземпляра снова будут обрабатывать по четыре задачи каждый.

Важно учесть, что Kafka Streams определяет количество создаваемых задач на основе максимального количества секций из *всех* входных топиков. В случае одного топика с 12 секциями число задач будет равно 12; но если входных топиков у вас четыре, по три секции каждый, то задач окажется три, каждая из которых будет отвечать за обработку четырех секций.

Помните, что если потоков выполнения больше, чем число задач, то оставшиеся потоки будут простаивать. Если запустить в вышеописанном примере с тремя экземплярами Kafka Streams четвертый экземпляр с четырьмя потоками выполнения, то после перебалансировки у вас окажется четыре простаивающих потока выполнения (16 потоков выполнения, но лишь 12 задач).

Это важнейшая особенность Kafka Streams, как я уже упоминал ранее в книге. Подобное динамическое масштабирование не требует останова приложения — оно происходит автоматически. Эта возможность удобна в случае неравномерного поступления данных в приложение — при необходимости можно запустить дополнительные экземпляры, чтобы справиться с возросшей нагрузкой, а затем отключить часть из них, когда объемы поступающих данных снизятся.

Всегда ли нужно использовать один поток выполнения на задачу? Возможно, хотя точно сказать трудно, это зависит от потребностей вашего приложения.

A.5. Конфигурация RocksDB

Для операций с сохранением состояния Kafka Streams применяет в качестве механизма постоянного хранения RocksDB (http://rocksdb.org/). RocksDB — быстрое хранилище пар «ключ/значение» с гибко изменяемой конфигурацией. Число его опций слишком велико, чтобы приводить тут подробные рекомендации, но упомяну, что Kafka Streams позволяет перекрыть его настройки по умолчанию с помощью интерфейса `RocksDBConfigSetter`.

Для задания пользовательских настроек RocksDB создайте класс, реализующий интерфейс `RocksDBConfigSetter`, и укажите название этого класса в конфигурации своего приложения Kafka Streams, в параметре `StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG`. Узнать больше о возможностях настройки RocksDB вы можете из руководства по настройке RocksDB (RocksDB Tuning Guide, http://mng.bz/I88k).

A.6. Заблаговременное создание топиков повторного секционирования

В Kafka Streams при всякой операции, способной потенциально изменить ключ `map` — например, `transform` или `groupBy`, — в классе `StreamsBuilder` устанавливается внутренний флаг, указывающий на необходимость повторного секционирования. При этом выполнение операций `map` или `transform` не приведет автоматически к созданию временного топика повторного секционирования; но операция повторного

секционирования будет запущена, как только вы добавите любую операцию, использующую обновленный ключ.

Хотя этот шаг является обязательным (см. главу 4), в некоторых случаях лучше самостоятельно выполнить повторное секционирование заблаговременно. Взгляните на следующий (сокращенный) пример:

```
KStream<String, String> mappedStream =
➥ streamsBuilder.stream("inputTopic").map(....);
KTable<Windowed<String>, Long> ktable1 =
➥ mappedStream.groupByKey().windowedBy...count()
KTable<Windowed<String>, Long> ktable2 =
➥ mappedStream.groupByKey().windowedBy...count()
KTable<Windowed<String>, Long> ktable3 =
➥ mappedStream.groupByKey().windowedBy...count()
```

Отображение исходного входного потока данных для создания нового ключа

Оконный подсчет количества, вариант 1

Оконный подсчет количества, вариант 2

Оконный подсчет количества, вариант 3

В этом коде мы выполняем отображение исходного потока данных, чтобы создать новый ключ группировки. Нам нужно осуществить три подсчета количества при трех различных вариантах оконных операций — вполне допустимый сценарий использования. Но вследствие отображения в новый ключ *каждая* оконная операция подсчета создает новый топик повторного секционирования. Опять же, поскольку ключ поменялся, топик повторного секционирования необходим, но три топика повторного секционирования вместо одного означают дублирование данных.

Решение этой проблемы очень простое: после вызова метода `map` необходимо сразу же вызвать операцию `through` для секционирования данных. При этом последующие вызовы `groupByKey` не запустят повторное секционирование данных, поскольку оператор `groupByKey` *не* устанавливает флаг потребности в повторном секционировании. Вот исправленный код:

```
KStream<String, String> mappedStream =
➥ streamsBuilder.stream("inputTopic").map(....).through(...);
```

Отображение исходного входного потока данных для создания нового ключа и повторное секционирование

Благодаря добавлению узла-обработчика `through` и повторному секционированию вручную у нас окажется один топик повторного секционирования вместо трех.

A.7. Настройка внутренних топиков

При создании топологии Kafka Streams может создавать многочисленные внутренние топики в зависимости от добавляемых вами узлов-обработчиков. Эти внутренние топики могут представлять собой журналы изменений для резервного копирования хранилищ состояния, или топики повторного секционирования. В зависимости от объемов данных они могут занимать значительное пространство на диске. Кроме того, хотя по умолчанию журналы изменений создаются со стратегией

очистки `compact`, в случае большого числа уникальных ключей их размеры будут расти. С учетом этого имеет смысл настроить их так, чтобы их размер не превышал разумных пределов.

Есть два варианта управления внутренними топиками. Во-первых, можно задавать настройки непосредственно при создании хранилищ состояния с помощью метода `StoreBuilder.withLoggingEnabled` или `Materialized.withLoggingEnabled`. Какой метод использовать — зависит от способа создания хранилища состояния. Оба метода принимают в качестве параметра объект `Map<String, String>` со свойствами топика. Пример их использования можно найти в файле src/main/java/bbejeck/chapter_7/CoGroupingListeningExampleApplication.

Второй вариант управления внутренними топиками — указание настроек для них в процессе настройки приложения Kafka Streams:

```
Properties props = new Properties();
// остальные свойства
props.put(StreamsConfig.topicPrefix("retention.bytes"), 1024 * 1024);
props.put(StreamsConfig.topicPrefix("retention.ms"), 3600000);
```

В случае подхода со `StreamsConfig.topicPrefix` указанные настройки используются глобально, для всех внутренних топиков. При этом настройки, задаваемые при создании хранилища состояния, имеют приоритет над настройками, указываемыми с помощью класса `StreamsConfig`.

Я не могу посоветовать вам, какие настройки использовать, поскольку это зависит от конкретного сценария использования. Но помните, что размер топика по умолчанию не ограничен, а время сохранения по умолчанию равно одной неделе, так что имеет смысл установить подходящие значения параметров `retention.bytes` и `retention.ms`. Помимо этого, для журналов изменений, служащих в качестве резервной копии хранилищ состояния с большим количеством уникальных ключей, можно установить параметр `cleanup.policy` равным `compact, delete`, чтобы размер топика не слишком рос.

A.8. Перезапуск приложения Kafka Streams

В какой-то момент может возникнуть необходимость перезапуска приложения Kafka Streams и повторной обработки данных или в процессе разработки, или после установки обновления кода. Для этой цели Kafka Streams предоставляет сценарий `kafka-streams-application-reset.sh`, расположенный в подкаталоге `bin` каталога, в который установлен Kafka.

У данного сценария есть один *обязательный* параметр: идентификатор приложения Kafka Streams. Он предлагает несколько возможностей (если вкратце): сбрасывать входящие топики до самого раннего из имеющихся смещений, сбрасывать промежуточные топики до последнего смещения и удалять любые внутренние топики. Отмечу, что при первом запуске приложения необходимо вызвать метод `KafkaStreams.cleanUp`, чтобы удалить локальное состояние, сохраненное с предыдущих запусков.

A.9. Очистка локального состояния

В главе 4 мы обсуждали вопрос сохранения Kafka Streams локального состояния задач в локальной файловой системе. При разработке, тестировании или миграции в новый экземпляр может понадобиться очистить все сохраненные локальные состояния.

Для полной очистки предыдущего состояния можно воспользоваться методом `KafkaStreams.cleanUp` либо перед вызовом метода `KafkaStreams.start`, либо после вызова `KafkaStreams.stop`. Вызов метода `KafkaStreams.cleanUp` в любой другой момент приведет к ошибке.

Б Строго однократная доставка

Важнейшим промежуточным этапом эволюции Kafka, относящимся к версии 0.11.0, стала *семантика строго однократной доставки* (exactly once semantics). До этой версии семантику доставки Kafka можно было описать как *«как минимум однократную»* (at-least-once) или *«как максимум однократную»* (at-most-once), в зависимости от генератора.

В случае как минимум однократной доставки возможна ситуация, когда брокер сохранил сообщение, но перед отправкой подтверждения обратно генератору в брокере возникла ошибка (мы предполагаем тут, что у генератора установлен параметр `asks="all"` и время ожидания подтверждения истекло). Если в настройках генератора указано ненулевое число повторов, он отправит сообщение повторно, будучи в неведении, что предыдущее сообщение было успешно сохранено. При таком (хотя и довольно редком) сценарии потребителям будет доставлено дублирующее сообщение — отсюда и фраза *«как минимум однократная»*.

Для обсуждения как максимум однократной доставки рассмотрим случай, когда в настройках генератора число повторов равно нулю. В сценарии предыдущего примера упомянутое сообщение будет доставлено ровно один раз, поскольку повторов отправки не производится. Но если ошибка в брокере произошла до того, как он сумел сохранить сообщение, оно вообще не будет отправлено. В этом случае мы считаем, что лучше получить не все сообщения, чем получить дубликаты.

В случае же семантики строго однократной доставки, даже при повторной отправке генератором уже сохраненного в топике сообщения, потребители получат сообщение ровно один раз. Для активизации транзакций (строго однократной обработки) при использовании `KafkaProducer` необходимо добавить параметр конфигурации `transactional.id` и несколько вызовов методов, как показано в следующем примере. В остальном транзакционная генерация сообщений выглядит вполне знакомо. Замечу, что этот фрагмент кода несамодостаточен — он предназначен лишь

для иллюстрации того, что необходимо для генерации и потребления сообщений с помощью транзакционного API:

```
Properties props = new Properties();
props.put("bootstrap.servers", "localhost:9092");
props.put("transactional.id", "transactional-id");

Producer<String, String> producer =
➥ new KafkaProducer<>(props, new StringSerializer(), new StringSerializer());

producer.initTransactions();  <-- При задании transactional.id необходимо вызвать этот метод до всех остальных

try {
  // Вызывается непосредственно перед отправкой каких-либо записей
  producer.beginTransaction();

  ...отправляем сообщения

  // По завершении отправки фиксируем транзакцию
  producer.commitTransaction();
} catch (ProducerFencedException | OutOfOrderSequenceException |
➥ AuthorizationException e) {

    producer.close();  <-- Единственный вариант в случае любого неустранимого исключения — завершить работу генератора
} catch (KafkaException e) {

    producer.abortTransaction();  <-- В случае любого другого исключения — прервать транзакцию и повторить попытку отправки
}
```

Для транзакционного использования `KafkaConsumer` необходимо добавить ровно один параметр конфигурации:

```
props.put("isolation.level", "read_committed");
```

В режиме `read_committed` `KafkaConsumer` читает только успешно зафиксированные транзакционные сообщения. Настройка по умолчанию — `read_uncommitted`, при которой возвращаются все сообщения. Нетранзакционные сообщения извлекаются всегда, в любом из режимов.

Строго однократная семантика стала очень важным достижением для Kafka Streams. Благодаря ей (транзакциям) гарантируется строго однократная обработка записей в топологии.

Для активизации строго однократной обработки в Kafka Streams установите значение параметра `StreamsConfig.PROCESSING_GUARANTEE_CONFIG` равным `exactly_once`. Значение по умолчанию для `PROCESSING_GUARANTEE_CONFIG` равно `at_least_once`, что соответствует нетранзакционной обработке. При указании данного простого параметра Kafka Streams берет на себя все необходимые для транзакционной обработки действия.

Это был краткий обзор транзакционного API фреймворка Kafka. Дополнительную информацию вы можете найти в следующих источниках:

- *Scott D.* Kafka in Action (Manning), www.manning.com/books/kafka-in-action;
- *Narkhede N.* Exactly-once Semantics Are Possible: Here's How Kafka Does It (Ния Нархид, «Строго однократная семантика возможна — и вот как ее осуществляет Kafka»), Confluent, 30 июня 2017, http://mng.bz/t9rO;
- *Mehta A., Gustafson J.* Transactions in Apache Kafka (Апурва Мехта и Джейсон Густавсон, «Транзакции в Apache Kafka»), Confluent, 17 ноября 2017, http://mng.bz/YKqf;
- *Wang G.* Enabling Exactly-Once in Kafka Streams (Гочжэн Ян, «Включение строго однократной доставки в Kafka Streams»), Confluent, 13 декабря 2017, http://mng.bz/2A32.

Билл Беджек

Kafka Streams в действии. Приложения и микросервисы для работы в реальном времени

Перевел с английского И. Пальти

Заведующая редакцией	*Ю. Сергиенко*
Руководитель проекта	*О. Сивченко*
Ведущий редактор	*Н. Гринчик*
Литературный редактор	*Е. Рафалюк-Бузовская*
Художественный редактор	*В. Мостипан*
Корректоры	*Е. Павлович, Т. Радецкая*
Верстка	*Г. Блинов*

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: 03.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —
Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 04.03.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 700. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

---

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*