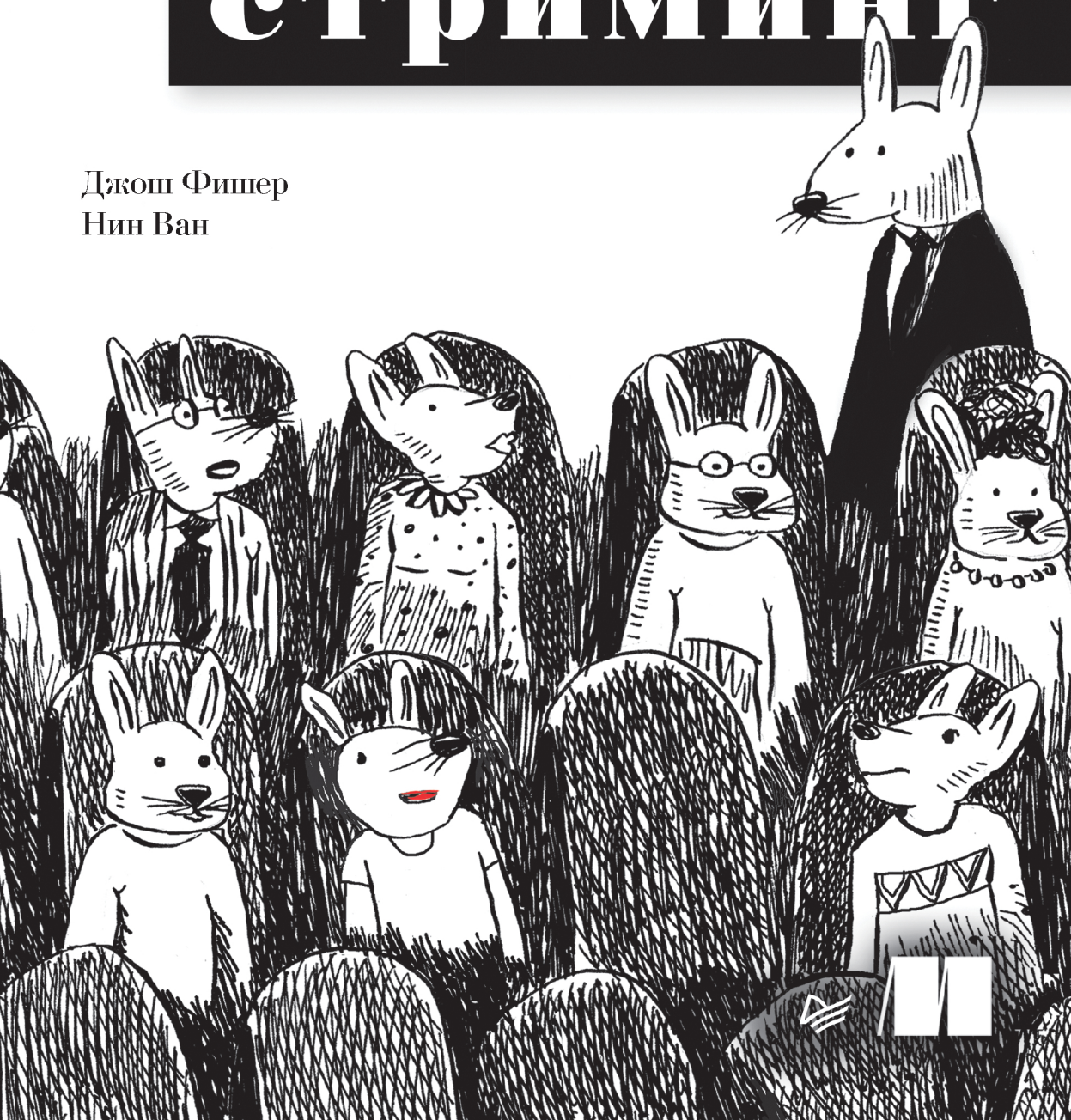
Обработка событий в реальном времени

грокаем стриминг

Джош Фишер
Нин Ван



grokking Streaming Systems

Real-time event processing

Josh Fischer
Ning Wang

MANNING
SHELTER ISLAND

грокаем стриминг

Джош Фишер, Нин Ван

ПИТЕР®

Санкт-Петербург · Москва · Минск

2023

Джош Фишер, Нин Ван

Грокаем стриминг

Серия «Библиотека программиста»

Перевел с английского Е. Матвеев

ББК 32.973.233-02
УДК 004.031.43

Фишер Джош, Ван Нин

Ф68 Грокаем стриминг. — СПб.: Питер, 2023. — 288 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-2306-3

Стриминговые системы позволяют сократить до минимума время между событием и обработкой информации, чтобы вы получали результаты в реальном времени. В приложениях для финансовой сферы, в вопросах безопасности и IoT важна каждая миллисекунда, поэтому без стриминговых систем не обойтись. А еще — это модно и приносит деньги ;) ! Неслучайно специалисты в Spark, Heron и Kafka так востребованы.

Наконец, вы можете познакомиться с созданием стриминговых приложений и обработкой событий в реальном времени не продираясь через технические подробности конкретных фреймворков, головоломные термины и сложные формулировки. Простой язык и яркие примеры позволят вам познакомиться с базовыми концепциями, а чтобы усвоить описанные идеи и приемы, вы построите собственную простую стриминговую программу с нуля.

От читателя не требуется опыт работы со стриминговыми системами. Примеры написаны на языке Java.

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

ISBN 978-1617297304 англ.
ISBN 978-5-4461-2306-3

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: 10.2022. 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 12.08.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 23,220. Тираж 1300. Заказ 0000.

Оглавление

Предисловие

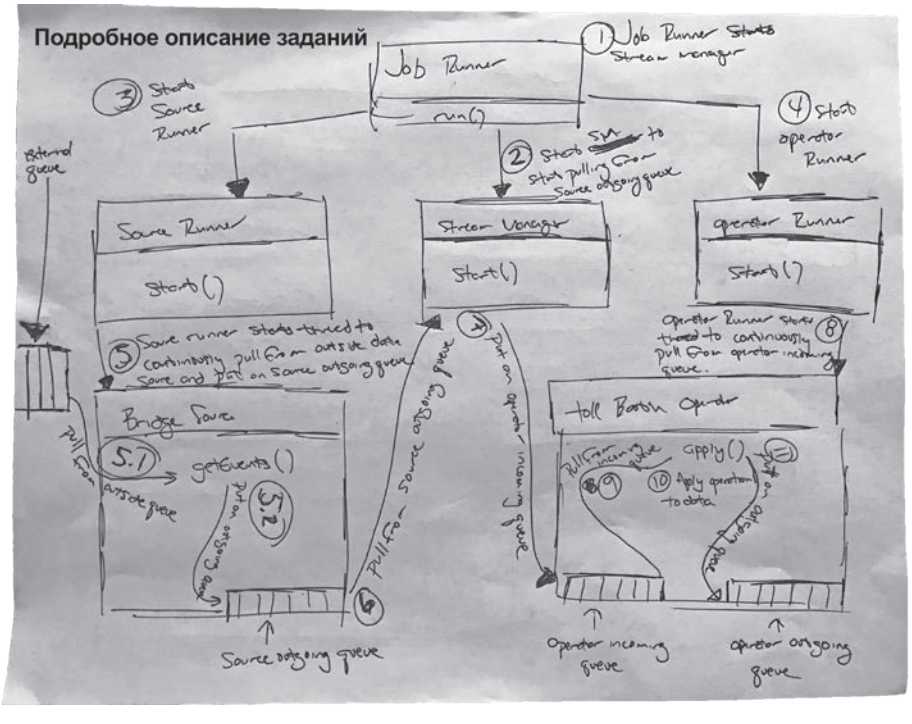
Когда я только начинал свой путь технического специалиста, мой наставник сказал мне: «Если ты хочешь выбрать что-то одно для развития карьеры, участвуй в проектах с открытым исходным кодом». Все годы я хранил эту мысль где-то на задворках памяти, но никогда не пытался следовать ей. Я думал: «Что я могу создать такого, что будет полезно другим?». Во время работы в 1904labs я разработал ECO API для (на то время) Twitter Heron. Я исходил из потребностей заказчика — и небольшой доли эгоизма; я действительно хотел написать этот код и сделать его своим творческим вкладом в проект. Со временем компания Twitter передала Heron в Apache Foundation, и меня пригласили войти в комитет управления проектом Heron. Я заинтересовался этой возможностью, потому что это был первый проект с открытым кодом, которым я занялся действительно глубоко.

Год спустя, после сохранения главной ветви Heron, около 16 часов в понедельник, я получил письмо от Элеонор Гарднер (Eleonor Gardner) с темой «Книга об Apache Heron, или Учебный курс». Пробежавшись по письму, я едва не удалил его, приняв за розыгрыш. В конце концов, с чего бы кому-то желать, чтобы я написал книгу или провел учебный курс? Но после обсуждения с Майком Стивенсом из Manning, и переписки с его заместителем Элеонорой я понял, что без помощи мне не обойтись. Я обратился к своему другу и коллеге по Apache Heron Нину Вану в надежде, что он захочет написать книгу вместе со мной. К счастью, он согласился — и это стало началом нашего долгого и плодотворного сотрудничества.

Сначала нам предлагали написать книгу, посвященную Heron. Но у Нина были свои идеи относительно того, как сделать эту книгу лучше. В конце концов, технологии быстро меняются и из-за качественных изменений в программном обеспечении материал быстро устареет. Нам хотелось взять тему, выходящую за рамки конкретных стриминговых фреймворков. Мы согласились написать

книгу, не привязанную к конкретному фреймворку, и изложить базовые концепции, чтобы читатель мог перейти к документации любого стримингового фреймворка и сразу начать работу.

Итак, мы приступили к работе, используя только текст, после чего нам с Нином «деликатно» предложили попробовать другой подход. Потом еще раз. И еще. И еще. Мы поняли, что с иллюстрациями материал книги усваивается намного быстрее. Первые схемы были нарисованы карандашом на бумаге и выглядели просто ужасно:



В ходе работы наши примитивные, нацарапанные от руки творения превратились в диаграммы, которые вы видите в книге. Мы с Нином разработали их сами. Мы очень довольны тем, что у нас получилось, и надеемся, что и вы оцените книгу по достоинству.

Джош Фишер, ноябрь 2021 г.

Благодарности

Прежде всего хочу поблагодарить своих детей и мою замечательную супругу Мелиссу. Она самая терпеливая и прекрасная спутница, которую я мог только пожелать. Она помогла справиться со всеми сложностями во время работы над книгой. Мои дети — Эйден, Уэс, Холлин, Оливер, Деклан и Дилан — проявили терпение и нередко обходились без меня по вечерам или утрам, когда я сидел над книгой.

Спасибо тебе, Нин, за то, что ты оставался со мной в ходе работы. Возможность учиться у тебя была одним из самых положительных моментов в работе над книгой.

Есть еще немало людей, которых я должен поблагодарить: Дэн Тамминелло (Dan Tumminello), Дэйв Лодес (Dave Lodes), Лора Стоби (Laura Stobie), Джим Тауи (Jim Towey), Стив Уиллис (Steve Willis), Майк Бэноси (Mike Banocy), Шон Уолш (Sean Walsh), Паван Виерамакинени (Pavan Veeramachineni), Роберт Макмиллан (Robert McMillan), Чед Сторм (Chad Storm), Картик Рамасами (Karthik Ramasamy) и Чандра Шекар (Chandra Shekar). Все они значительно повлияли на меня — как в личном, так и в профессиональном плане.

Напоследок хочу поблагодарить Берта Бэйтса (Bert Bates). Несомненно, это самый терпеливый, великодушный и вообще фантастический учитель из всех, кого я знаю. Беки Уитни (Becky Whitney) всегда участвовала в обсуждениях, которые бывали непростыми, но неизменно помогала нам не сбиться с пути, чтобы у нас получилось то, что нужно Manning. Спасибо Майку Стивенсу (Mike Stephens) за предоставленную возможность. Элеонор Гарднер (Eleonor Gardner) организовала наше общение на этапе подготовки, и наконец, Энди Маринкович (Andy Marinkovich) и Кери Хэйлз (Keri Hales) внесли завершающие штрихи.

Рецензенты: Андрес Сакко (Andres Sacco), Анто Аравинт (Anto Aravinth), Анупам Сенгупта (Anupam Sengupta), Апурв Гупта (Apoorv Gupta), Бо Бендер (Beau Bender), Брент Хонадел (Brent Honadel), Бриньяр Смари Бьярнасон (Brynjar Smári Bjarnason), Крис Лундберг (Chris Lundberg), Чичеро Зандона

(Cicero Zandona), Дэмиен Эстебан (Damian Esteban), Дипика Фернандес (Deepika Fernandez), Фернандо Антонио да Сильва Бернардино (Fernando Antonio da Silva Bernardino), Иоханнес Лохманн (Johannes Lochmann), Кент Р. Спиллнер (Kent R. Spillner), Кумар Унникришнан (Kumar Unnikrishnan), Лев Андельман (Lev Andelman), Марк Рулло (Marc Roulleau), Массимо Сиани (Massimo Siani), Матиас Буш (Matthias Busch), Мигель Монтальво (Miguel Montalvo), Себастиан Пальма (Sebastián Palma), Симеон Лейзерсон (Simeon Leyzerzon), Саймон Сейаг (Simon Seyag) и Саймон Верховен (Simon Verhoeven): ваши комментарии, вопросы и замечания сделали эту книгу лучше. Спасибо всем.

Джош Фишер, ноябрь 2021 г.

Два года! Я уже сбился со счета, кого мне следует благодарить. Эта книга никогда бы не увидела свет без тех людей, имена которых названы ниже, а также многих других.

Прежде всего, я бы не смог завершить книгу без понимания и поддержки моей дочери. Я задолжал тебе выходные за два года, Синьи! Уже больше двух лет я не навещал своих родителей Джили Ван (Jili Wang) и Сюцзюнь Лю (Shujun Liu) и свою сестру Фен Ван (Feng Wang) в Китае. Я по ним очень скучаю.

Большое спасибо моему соавтору Джошу. Какое же это было приключение! Оно было бы невозможно без твоего творческого подхода и замечательных идей.

Я верю в потенциал обработки данных и благодарен за то, что мне довелось работать со многими талантливыми инженерами. Многое из того, что я узнал от этих людей, стало важным для книги: это Маосун Фу (Maosong Fu), Нэн Лу (Neng Lu), Хуэйцзюнь У (Huijun Wu), Дмитрий Русаков (Dmitry Rusakov), Сяояо Цянь (Xiaoyao Qian), Яо Ли (Yao Li), Чжэньсяо Ло (Zhenxiao Luo), Хао Ло (Hao Luo), Майнак Гош (Mainak Ghosh), Да Чэн (Da Cheng), Фред Дай (Fred Dai), Бэйнань Ван (Beinan Wang), Чуньсю Тан (Chunxu Tang), Жуньхан Ли (Runhang Li), Ялян Ван (Yaliang Wang), Томас Купер (Thomas Cooper) и Фария Калим (Faria Kalim) из команды Real-Time Compute в Twitter; Паван Патибандла (Pavan Patibandla), Фаршад Ростамабади (Farshad Rostamabadi), Курт Норвуд (Kurt Norwood), Жюльен Дубо (Julien Dubeau), Кэти Нам (Cathy Nam), Лео Чжан (Leo Zhang), Неха Бамбхани (Neha Bhambhani), Ник У (Nick Wu), Робин Нейсон (Robyn Nason), Закери Миранда (Zachery Miranda), Джеффри Ван (Jeffrey Wang) и Нирмал Утвани (Nirmal Utwani) из команды Data Pipeline в Amplitude; а также многие другие участники сообщества Apache Heron.

Так как я впервые пишу книгу (да еще на английском!), я бы ни за что не справился без той помощи, которую я получал от трудолюбивых редакторов Manning. Огромное спасибо Берту Бэйтсу (Bert Bates), Беки Уитни (Becky Whitney), Дженнифер Хаул (Jennifer Houle), Мэтью Спору (Matthew Spaur) и многим другим редакторам и рецензентам, принимавшим участие в работе. Я очень много узнал от вас!

Нин Ван, ноябрь 2021 г.

О книге

Книга «Грокаем стриминг» поможет вам разобраться в том, что такое стриминговые системы, как они работают и подходят ли они для ваших задач. Так как книга написана без привязки к конкретному инструментарию, вы сможете применить полученные знания независимо от того, какой фреймворк выберете. Мы начнем с основных концепций, а затем постепенно перейдем к более сложным примерам, включая отслеживание в реальном времени счетчиков событий от датчиков IoT и выявление мошеннических действий с кредитными картами. Вы даже сможете поэкспериментировать с собственной стриминговой системой, для чего достаточно будет загрузить специально разработанный для этой книги и предельно упрощенный стриминговый фреймворк. К концу последней главы вы сможете пользоваться функциональностью стриминговых фреймворков и решать типичные задачи, возникающие при построении стриминговых систем.

Для кого написана эта книга?

Мы написали эту книгу для разработчиков, имеющих хотя бы пару лет опыта практического программирования и желающих расширить свои знания. Если вы разрабатывали веб-клиенты, API, пакетные задания и т. д. и теперь хотите двигаться дальше — эта книга для вас.

Структура книги

Книга имеет простую структуру: 11 глав разделены на две части; после того как вы прочитаете по порядку главы с 1 по 5, изучайте остальные главы в любом порядке на свое усмотрение. Краткое содержание глав:

- В главе 1 дан общий обзор стриминговых систем и их сравнение с другими типичными разновидностями компьютерных систем.

- В главе 2 рассматриваются фундаментальные принципы работы стриминговых систем.
- В главе 3 обсуждается параллелизация, группировка данных и возможности масштабирования стриминговых заданий.
- Глава 4 посвящена потоковым графам и способам представления стриминговых заданий.
- В главе 5 рассматривается семантика доставки — например, возможности использования стриминговых систем для надежной (и не очень) доставки событий.
- Глава 6 содержит обзор основных концепций, а также краткое содержание дальнейших глав.
- В главе 7 рассматриваются окна — способы сегментации бесконечного потока данных.
- В главе 8 обсуждаются стриминговые соединения, то есть слияние данных в реальном времени.
- В главе 9 рассказано о том, как происходит обработка сбоев в стриминговых системах.
- Из главы 10 вы узнаете, как стриминговые системы выполняют операции с состоянием в реальном времени.
- Глава 11 подводит итог второй части и содержит наши рекомендации о том, как найти применение вашему интересу к стриминговым системам.

О коде

Главы 2, 3, 4, 5, 7 и 8 содержат примеры кода. Вы можете загрузить их по адресу *https://github.com/nwangtw/GrokkingStreamingSystems*. Кроме того, исходный код можно загрузить бесплатно на сайте издательства Manning: *https://www.manning.com/books/grokking-streaming-systems*. Для запуска примеров вам понадобится Java 11, Apache Maven 3.8.1 и программы командной строки Netcat или NMap.

Книга содержит множество примеров исходного кода — как в пронумерованных листингах, так и в строках с обычным текстом. В обоих случаях исходный код выделяется `моноширинным шрифтом`, чтобы он отличался от обычного текста. Иногда код также выделяется жирным шрифтом, которым обозначаются изменения по сравнению с предыдущими шагами, например при добавлении новых функций в существующую строку кода. Во многих случаях исходный код был переформатирован; мы добавили разрывы строк и изменили величину отступов по ширине страниц книги.

Кроме того, мы часто опускали комментарии к коду в листинге, если код был описан в тексте. Многие листинги сопровождаются аннотациями, выделяющими важные понятия.

Форум liveBook

Приобретая книгу «Грокаем стриминг», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от авторов и других пользователей. Чтобы получить доступ к форуму, откройте страницу *https://livebook.manning.com/book/grokking-streaming-systems/discussion/*. Информацию о форумах Manning и правилах поведения на них см. на *https://livebook.manning.com/#!/discussion*.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия авторов, которое остается добровольным (и неоплачиваемым). Задавайте авторам интересные вопросы, чтобы они не теряли интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

От издательства

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

Мы будем рады узнать ваше мнение!

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

Об авторах

Джош Фишер в настоящее время руководит командой разработчиков в 1904labs. Он работал над перемещением больших наборов данных в реальном времени для сторонних организаций, включая Monsanto и Bayer.

Нин Ван — разработчик из Amplitude, занимающийся построением конвейеров данных реального времени. Он являлся одним из ключевых участников проекта Apache Heron в команде вычислений реального времени Twitter.

Оба автора участвуют в проекте Apache и входят в комитет управления проектом для ядра распределенной потоковой обработки Apache Heron.

Часть I
Знакомство

В первой части книги вы с головой погрузитесь в мир стриминговых систем и найдете ответы на такие вопросы, как «Почему стриминговые системы работают именно так, а не иначе?» и «Зачем вообще их использовать?». В главе 1 рассказано, чем стриминговые системы отличаются от любых других на высоком уровне. Глава 2 знакомит с азами стриминговых систем; в ней представлены фундаментальные концепции их работы. В главе 3 рассматривается проблема масштабирования таких систем, а глава 4 повествует о том, как данные перемещаются в стриминговых заданиях. Глава 5 рассказывает, как эти системы обеспечивают надежную доставку данных в реальном времени, а глава 6 содержит обзор ключевых моментов каждой предыдущей главы. К концу части I вы будете знать все необходимое, чтобы выбрать понравившийся фреймворк и без промедлений приступить к работе.

1 Знакомство со стриминговыми системами

В этой главе

- ✓ Общие сведения о потоковой обработке
- ✓ Отличия стриминговых систем от других систем

«Если бы не камни на дне, то не звучала бы песня потока».

Карл Перкинс

В этой главе мы попытаемся ответить на несколько основных вопросов, касающихся стриминговых систем: от вопроса «Что такое потоковая обработка?» до «Для чего используются системы потоковой обработки, или стриминговые системы?». Мы представим некоторые базовые концепции, которые рассмотрим подробнее в следующих главах.

Что такое потоковая обработка?

В последние годы потоковая обработка стала одной из самых популярных технологий работы с большими данными. К стриминговым системам относятся компьютерные системы, обрабатывающие непрерывные потоки событий.

Важнейшая особенность потоковой обработки заключается в том, что *события* обрабатываются сразу (или почти сразу) после того, как они станут доступными. Это делается для того, чтобы свести к минимуму задержку между поступлением в систему исходного события и конечным результатом обработки этого события. В большинстве случаев задержка составляет от нескольких миллисекунд до нескольких секунд, что можно считать реальным временем или почти реальным временем; поэтому потоковая обработка также называется *обработкой в реальном времени*. Потоковая обработка обычно используется для анализа разных типов событий. В результате стриминговые системы в разных сценариях также могут называться «аналитика реального времени», «потоковая аналитика» и «обработка событий». В этой книге мы используем популярный в отрасли термин *потоковая обработка*.

Примеры событий

Вот несколько примеров того, что считается событием:

- Щелчки мышью при работе на компьютере.
- Касания и смахивания на экране мобильного телефона.
- Прибытие и отправление поездов на станции.
- Отправка сообщений и электронных писем пользователем.
- Температурные данные, поступающие с датчиков в лаборатории.
- Взаимодействия пользователей с веб-сайтом (просмотры страниц, вход в систему и т. д.).
- Журналы серверов в центрах обработки данных.
- Транзакции по всем счетам в банке.

Следует отметить, что для событий, обрабатываемых в стриминговых системах, обычно не существует заранее установленного времени окончания. Их можно рассматривать как непрекращающиеся; таким образом, поток событий часто считается *непрерывным* и *бесконечным*. События встречаются на каждом шагу — в буквальном смысле. Мы живем в информационную эпоху. В мире постоянно генерируются, собираются и обрабатываются огромные объемы данных.

Подумайте об этом

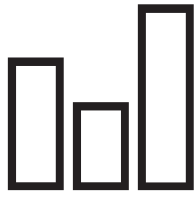
Стриминговые системы — компьютерные системы, предназначенные для обработки непрерывных потоков событий.

Примеры стриминговых систем

Рассмотрим пару примеров:

- Первый пример — система контроля температуры в лаборатории. Многочисленные датчики, установленные в разных местах, собирают данные о температуре каждую секунду. Стриминговая система обрабатывает собранные данные и выводит информацию на панель в реальном времени. Также система может подавать сигналы тревоги при обнаружении аномалий. Администраторы лаборатории используют систему для наблюдения за всеми помещениями и проверки того, что температура находится в заданном диапазоне.

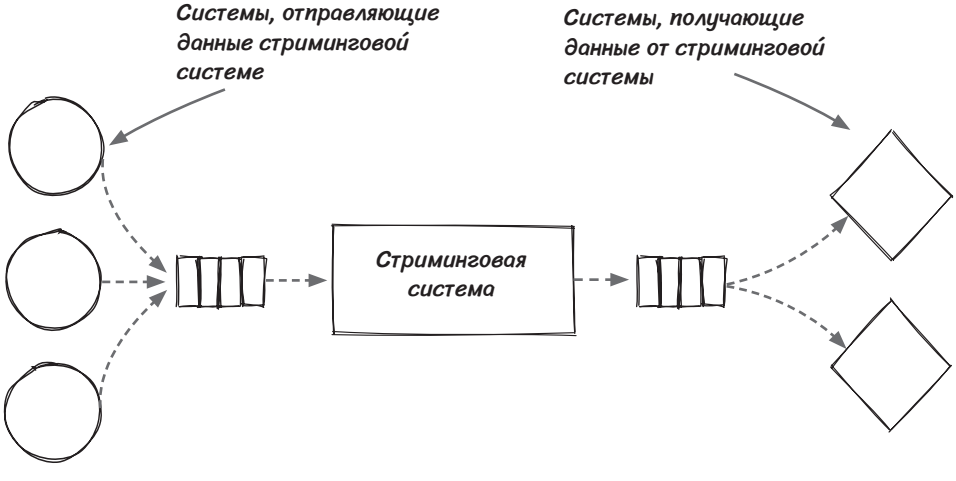
- Второй пример — системы мониторинга и анализа, которые обрабатывают взаимодействия пользователя с сайтом: просмотры страниц, авторизацию или нажатия кнопок. Когда вы заходите на веб-сайт, в логе обычно регистрируется множество событий. Эти низкоуровневые события часто содержат множество полей данных, поэтому сохранять их в исходном виде неэффективно. Кроме того, некоторые поля непонятны для человека, и перед тем, как их представить, их необходимо преобразовать. Стриминговые системы чрезвычайно полезны для преобразования данных низкоуровневых событий в более ценную информацию — например, в данном контексте это может быть количество запросов, количество активных пользователей, просмотры каждой страницы или подозрительное поведение пользователей.



В этих примерах стриминговая система может обрабатывать в реальном времени огромное количество событий и извлекать из них полезные данные. Стриминговые системы очень актуальны, потому что события содержат массу ценной информации и для многих сценариев важно обрабатывать ее в реальном времени.

Стриминговые системы и реальное время

Термином «стриминговая система» обозначается система, которая извлекает полезную информацию из непрерывного потока событий. А конкретнее, как упоминалось в начале раздела, стриминговая система должна обрабатывать события и генерировать результат после сбора данных как можно быстрее. И это важно, потому что результаты становятся доступными с минимальной задержкой и своевременной реакцией. Благодаря тому что они работают в реальном времени, стриминговые системы очень полезны во многих ситуациях — например, в лабораторных условиях и на сайтах, где желательно получать результаты с минимальной задержкой.



В лаборатории система наблюдения может подавать сигналы тревоги, автоматически запускать резервные устройства и при необходимости оповещать администраторов. Если неисправное оборудование вовремя не отремонтировать или не заменить, а температура не контролируется, это может привести к изменению состояния или повреждению температурно-чувствительных образцов. Также могут быть прерваны некоторые проводимые эксперименты. Для вебсайтов, помимо контроля, диаграммы и дашборды, генерируемые стриминговой системой, помогут разработчикам понять типичные сценарии взаимодействия с пользователями и соответствующим образом улучшить свои продукты.

Как работает стриминговая система

Приведенные примеры событий и стриминговых систем дают некоторое представление о том, что такое стриминговые системы. Далее мы опишем, как эти системы работают на очень высоком уровне, сравнив их с другими разновидностями систем.

Сравнение четырех типичных компьютерных систем

У стриминговых систем много общего с другими компьютерными системами. В конце концов, стриминговая система не перестает быть компьютерной. Вот некоторые типичные системы, которые мы выбрали для сравнения:

- Приложения.
- Серверные службы.
- Службы пакетной обработки.
- Службы потоковой обработки.

Приложения

Приложение — компьютерная программа, напрямую взаимодействующая с пользователем. Программы, установленные на компьютере или смартфоне, являются приложениями. Калькулятор, текстовый редактор, видео- и аудиоплееры, мессенджеры, браузер и игры — все это примеры приложений. Приложения повсюду! Пользователи взаимодействуют с компьютерами при помощи разнообразных приложений.

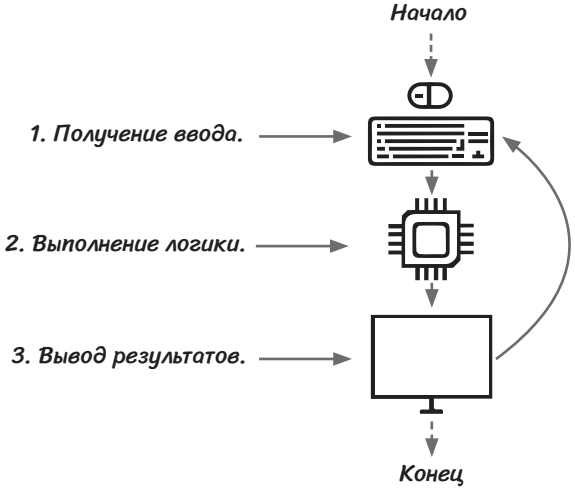
Пользователи используют приложения, чтобы выполнять задачи. Вы можете создать заметку или книгу в текстовом редакторе и сохранить ее в файле. Если у вас есть видеофайл, его можно открыть и воспроизвести в приложении-проигрывателе. Браузер может использоваться для поиска информации, просмотра видео и совершения покупок в интернете.

Что внутри

Приложения сильно отличаются друг от друга. Командная строка, текстовый редактор, калькулятор, фоторедактор, браузер и видеоигра не похожи как по внешнему виду, так и по поведению. Вы когда-нибудь рассматривали их как программные продукты одной категории? По своему внутреннему строению они различаются еще сильнее. Простой калькулятор можно реализовать в нескольких строках кода, а код веб-браузера или видеоигры может состоять из миллионов строк.

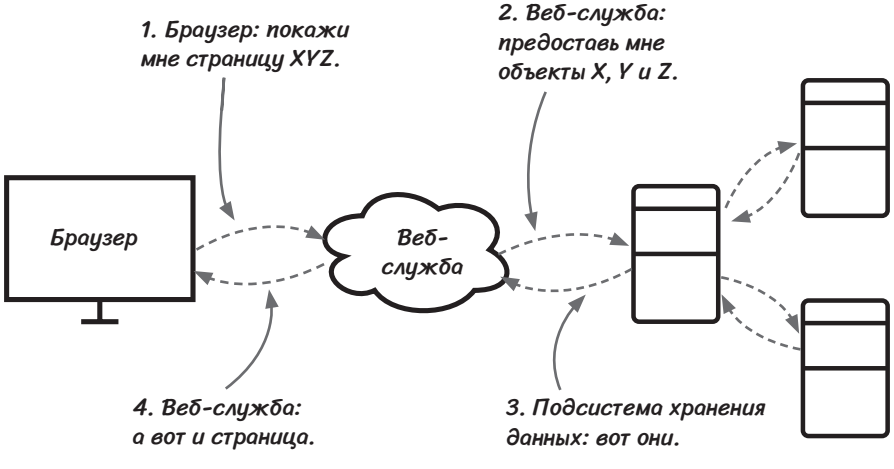
Несмотря на все различия, основной процесс для большинства приложений похож: начальная точка (в которой приложение запускается), конечная точка (в которой приложение закрывается) и цикл (главный цикл) из трех шагов:

1. Получение пользовательского ввода.
2. Выполнение логики.
3. Вывод результатов.



Серверные службы

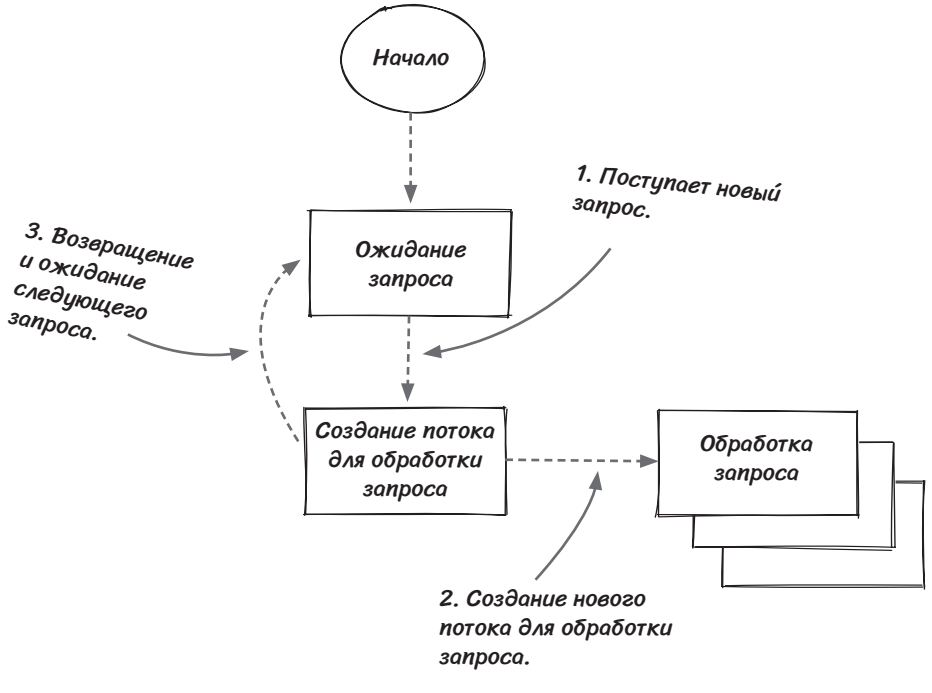
Серверная служба — компьютерная программа, выполняемая *скрыто от глаз пользователя*. В отличие от приложения серверная служба не взаимодействует с пользователями напрямую. Вместо этого она реагирует на запросы и выполняет соответствующие задачи. Служба обычно является процессом с длительным временем выполнения, и она постоянно ожидает входных запросов.



Для примера рассмотрим простую веб-службу. При получении запроса программа интерпретирует его, последовательно обрабатывает и, наконец, отвечает на запрос. После этого программа ожидает следующего запроса. Веб-службы часто работают не сами по себе, а совместно с другими службами. Службы могут обрабатывать запросы, получаемые друг от друга, и каждая служба отвечает за конкретную операцию. На следующей диаграмме изображена схема совместной работы веб-службы и службы хранения данных для обслуживания запроса страницы.

Внутри серверной службы

Внутри серверной службы также есть основной цикл, но он работает по-другому, потому что обрабатываемые службой запросы сильно отличаются от пользовательского ввода приложения. Так как с приложением чаще всего работает один пользователь, проверки пользовательского ввода в начале основного цикла обычно оказывается достаточно, но в серверную службу может в любой момент поступить сразу несколько запросов. Чтобы запросы обрабатывались быстро, в этом сценарии использования важно применять *многопоточную* обработку. *Программный поток* представляет собой подзадачу, выполняемую внутри процесса; в контексте одного процесса могут существовать несколько потоков. Эти потоки совместно используют ресурсы процесса (например, память) и могут выполняться одновременно.



Типичная служба работает по схеме, представленной на рисунке. При получении запроса обработчик создает новый поток для выполнения реальной логики и немедленно возвращает управление, не дожидаясь результатов. Вычисления, занимающие много времени (реальная логика), выполняются параллельно в отдельном потоке. При такой схеме основной цикл выполняется очень быстро, так что новые входящие запросы будут приниматься с максимальной скоростью.

Системы пакетной обработки

Как приложения, так и серверные службы спроектированы для обслуживания клиентов (пользователей-людей или удаленных запросов) с максимальной быстротой. С системами пакетной обработки дело обстоит иначе. Они не должны реагировать на любой ввод. Вместо этого системы пакетной обработки проектируются так, чтобы задачи выполнялись в запланированное время или когда позволяет наличие ресурсов.

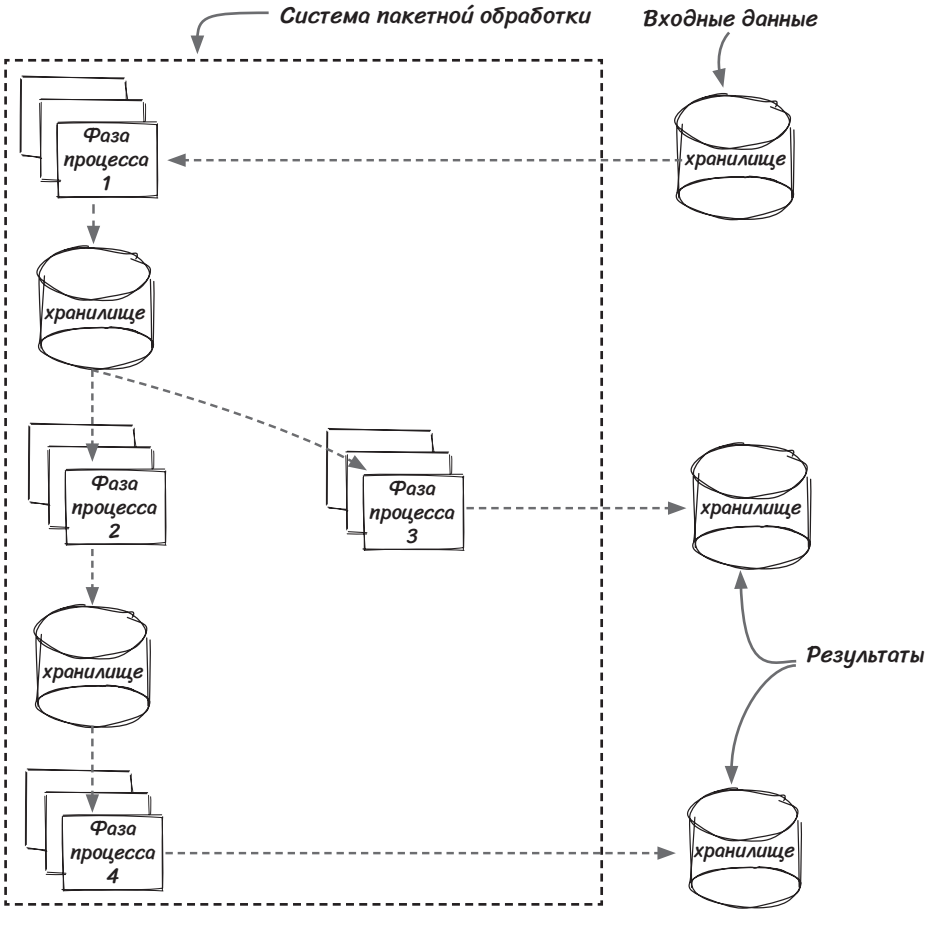
В реальной жизни системы пакетной обработки встречаются достаточно часто. Например, на почте корреспонденцию собирают, сортируют, перевозят и доставляют в запланированное время, потому что такая схема наиболее эффективна. Трудно представить систему, в которой сотрудник принимает ваше письмо к отправке, выбегает из отделения и мчится доставлять его получателю. Вообще говоря, это возможно, но крайне неэффективно, и для такого понадобятся очень веские причины.

В наши дни каждую секунду формируются все новые огромные объемы данных: статьи, электронные письма, взаимодействия с пользователями и данные служб и устройств. Обработка таких данных и поиск полезной информации — очень важная, хотя и достаточно сложная задача. Системы пакетной обработки предназначены именно для этого.

> ***Внимание!***
>
> Системы пакетной обработки предназначены для эффективной обработки огромных объемов данных.

Внутри системы пакетной обработки данных

В типичной системе пакетной обработки весь процесс делится на несколько этапов, или *фаз*. Фазы связаны хранилищами, в которых содержатся промежуточные данные.



В нашем примере входные данные обрабатываются группами, или *пакетами* (пример — данные пользовательских взаимодействий на веб-сайте за каждый час). При появлении новых данных (когда весь пакет получен и готов к обработке) запускается фаза 1 для загрузки данных и выполнения логики. Результаты сохраняются в промежуточном хранилище, откуда они переходят на следующие фазы. После того как данные пакета будут обработаны, фаза завершается и запускается следующая фаза (фаза 2 на приведенной выше диаграмме), на которой ведется работа с промежуточными результатами, сгенерированными на фазе 1. Обработка завершается после того, как пакет пройдет все фазы.

Системы потоковой обработки

Архитектура пакетной обработки — исключительно мощный инструмент в мире больших данных. Тем не менее у систем пакетной обработки есть один серьезный недостаток: *задержка*.

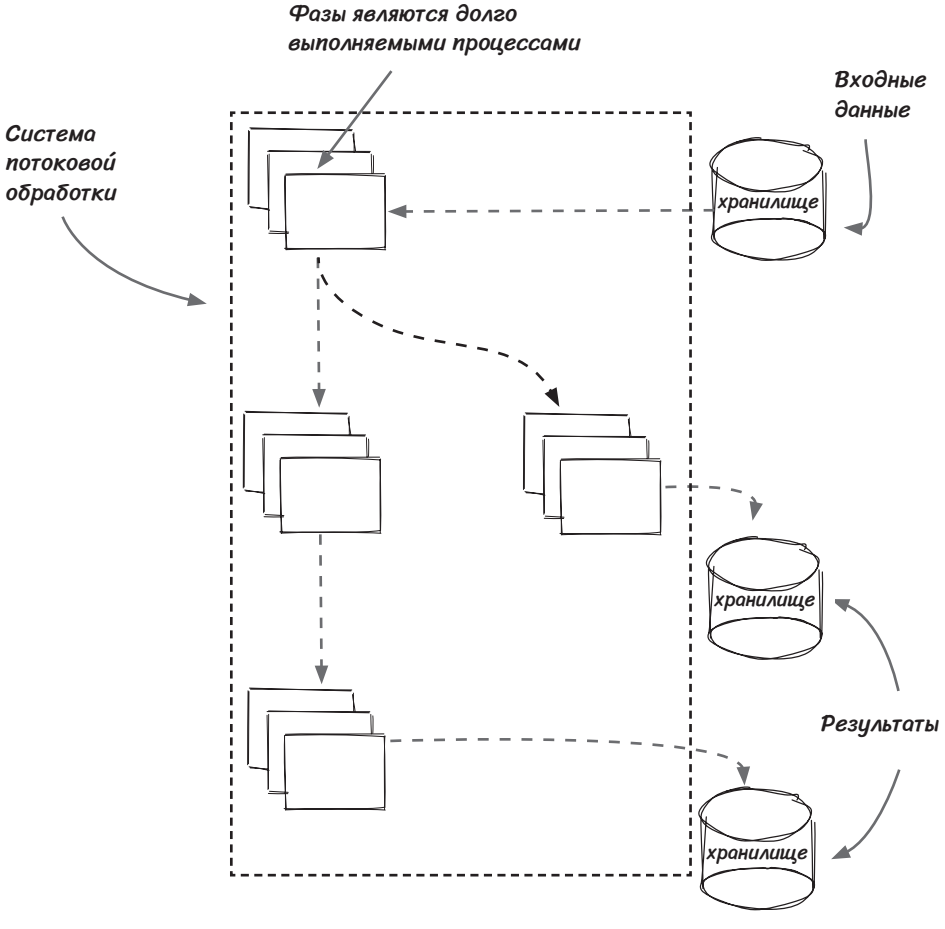
Системы пакетной обработки требуют, чтобы перед запуском данные собирались и сохранялись в пакетах с регулярными интервалами, например ежечасно или ежедневно. Любые события, собранные в определенном временном окне, будут обработаны только после закрытия этого окна. В некоторых случаях это может быть неприемлемо — например, если в системе наблюдения в лаборатории сигнал тревоги сработает только через час. В таких случаях предпочтительнее обрабатывать данные сразу же после их получения — чтобы получать результаты в реальном времени. Системы потоковой обработки предназначены для сценариев, ориентированных на обработку в реальном времени. В системах потоковой обработки события обрабатываются после их получения как можно быстрее.

В качестве примера реальной системы пакетной обработки мы рассматривали почту. В этой системе корреспонденцию собирают, сортируют, перевозят и доставляют несколько раз в день в запланированное время. Реальным примером системы потоковой обработки может быть сборочный конвейер на фабрике. Процесс сборки тоже состоит из нескольких фаз, и конвейер постоянно работает и получает новые детали. В каждой фазе операция применяется к одному продукту за другим. В конце со сборочной линии одно за другим выходят готовые изделия.

> ***Внимание!***
>
> Системы потоковой обработки предназначены для обработки огромных объемов данных с малой задержкой.

Внутри системы потоковой обработки

Архитектура типичной системы потоковой обработки близка к архитектуре систем пакетной обработки. Весь процесс делится на отдельные этапы, называемые *компонентами*, а данные переходят от компонента к компоненту, пока не будут завершены все фазы обработки.

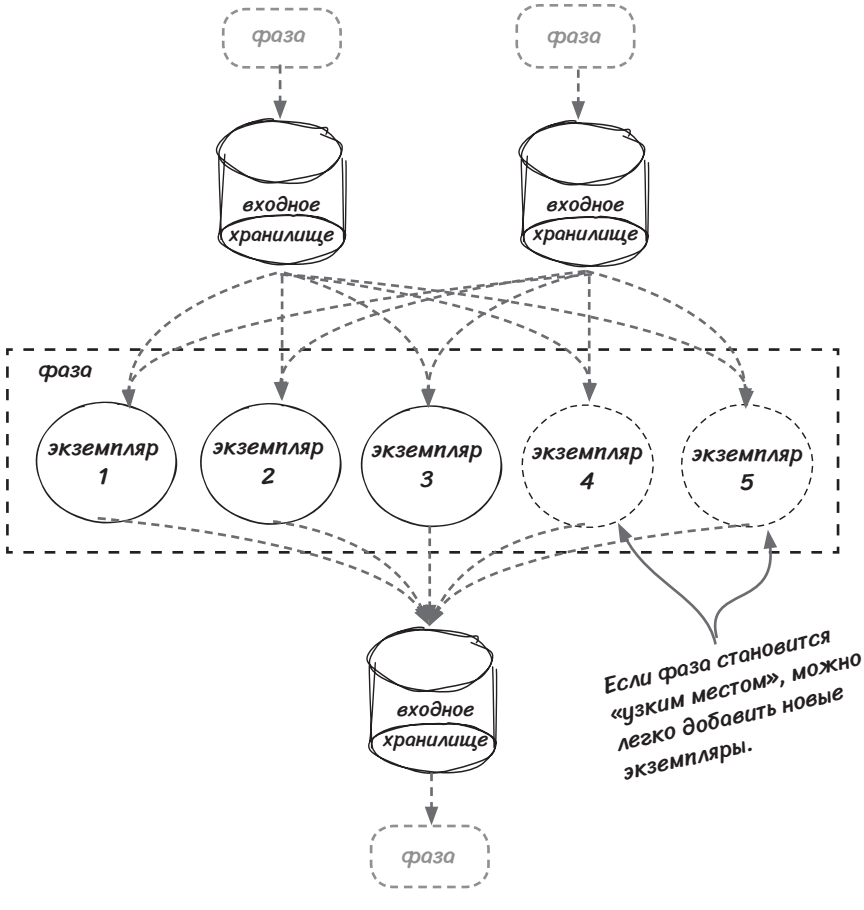


Основное отличие систем потоковой обработки от систем пакетной обработки заключается в том, что компоненты являются длительными процессами. Они продолжают работать и получать новые данные. Каждое событие начинает обрабатываться следующим компонентом сразу же после обработки предшествующим компонентом. Следовательно, окончательный результат генерируется вскоре после того, как событие поступает в стриминговую систему.

Преимущества многофазной архитектуры

Системы как пакетной, так и потоковой обработки используют многофазную архитектуру. Она обладает рядом преимуществ, благодаря которым хорошо подходит для сценариев обработки данных:

- *Гибкость* — разработчики могут добавлять или исключать фазы по своему усмотрению.
- *Масштабируемость* — фазы соединены, но каждая из них существует независимо от других. Если одна фаза становится «узким местом» всего процесса с существующими экземплярами (с 1-го по 3-й на следующей диаграмме), проще подключить дополнительные экземпляры (с 4-го по 5-й) для повышения пропускной способности.
- *Сопровождаемость* — сложные процессы могут состоять из простых операций, которые создают меньше проблем с реализацией и сопровождением.



Многофазная архитектура в системах пакетной и потоковой обработки

Системы пакетной обработки

В системах пакетной обработки фазы выполняются независимо друг от друга, как и экземпляры одной фазы. Это означает, что они не выполняются одновременно. Все экземпляры в системе могут выполняться один за одним или же пакет за пакетом — при условии, что порядок выполнения правилен. Это позволяет построить систему пакетной обработки огромных (действительно огромных!) объемов данных при весьма ограниченных ресурсах (хотя при недостатке ресурсов это потребует больше времени). Для компенсации затрат на хранение промежуточных данных события обычно более эффективно обрабатываются пакетами большего размера. Например, на практике часто применяются часовые или суточные пакетные окна. Событиям, происходящим в начале окна, для обработки приходится целый час или сутки дожидаться его закрытия. Этим и объясняется высокая задержка.

Одним из важных преимуществ систем пакетной обработки является простота преодоления сбоев. Если возникнет проблема (например, сбой компьютера или ошибка при чтении или записи данных), фазу со сбоем можно просто запланировать на другой машине и выполнить заново.

Системы потоковой обработки

При потоковой обработке все фазы представляют собой процессы с длительным временем выполнения. События непрерывно передаются между фазами. В результате возможность остановки фаз со сбоем теряется и обработка сбоев усложняется. С другой стороны, события обрабатываются с максимально возможной скоростью и мы можем получать результаты в реальном времени.

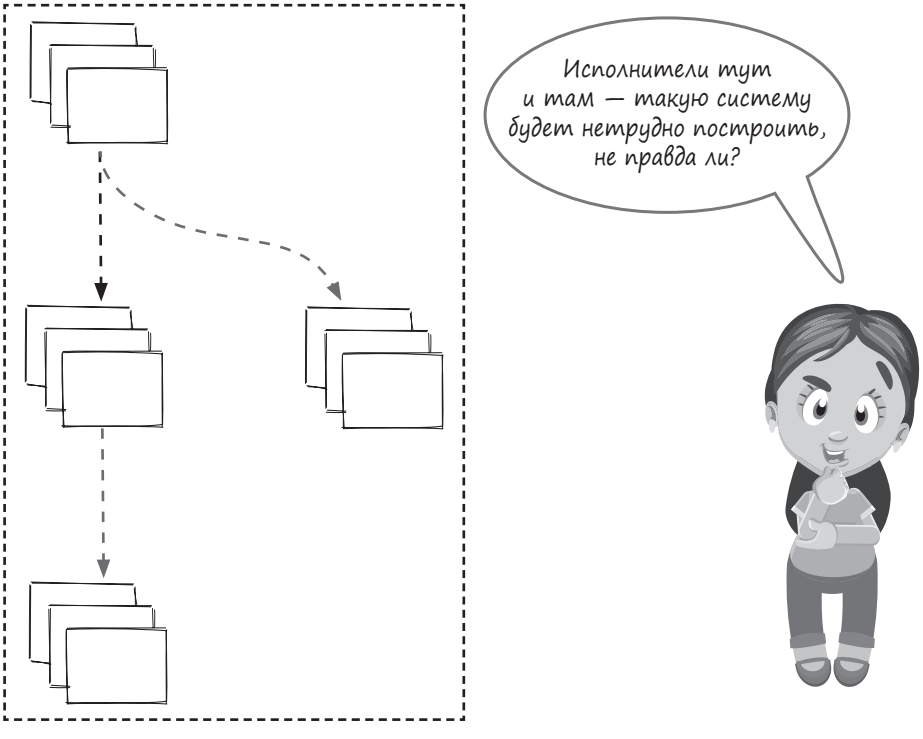
Сравнение систем

Сравнительная таблица поможет лучше понять, как работают разные компьютерные системы, представленные в этом разделе.

Приложение	Серверная служба	Система пакетной обработки	Система потоковой обработки
Обработка пользовательского ввода	Обработка запросов	Обработка данных	Обработка данных
Прямое взаимодействие с пользователем	Прямое взаимодействие с клиентами и другими службами. Косвенное взаимодействие с пользователем	Применение операций к данным. Результаты предназначены для пользователя прямо или косвенно	Применение операций к данным. Результаты предназначены для пользователя прямо или косвенно
Приложения запускаются и останавливаются пользователями	Экземпляры службы представляют собой долго выполняемые процессы	Планирование запуска и остановки экземпляров служб	Экземпляры службы представляют собой долго выполняемые процессы
Один основной цикл	Один основной цикл с потоками	Многофазный процесс	Многофазный процесс

Эталонная система обработки событий

Рассмотрев в общих чертах несколько разных систем, сосредоточимся на системах обработки событий. Из предыдущего раздела вы узнали, что стриминговая система состоит из нескольких долго выполняемых процессов-компонентов.



Ответ на этот вопрос зависит от того, какая система вам нужна. Что вы хотите сделать? Каков объем трафика? Сколько ресурсов у вас есть? Как вы будете управлять этими ресурсами? Как восстанавливаться при сбое? Как убедиться в том, что после восстановления получен правильный результат? Все эти вопросы приходится учитывать при построении системы потоковой обработки. Ну как, вы ответите «да»?

Стриминговые системы могут быть довольно сложными, но строить их *не так* сложно. В следующих главах вы узнаете, как строятся стриминговые системы и как работают их механизмы. Готовы?

Итоги

Из этой главы мы узнали, что потоковая обработка представляет собой технологию обработки непрерывно поступающих событий для получения результатов в реальном времени. Чтобы понять, чем системы потоковой обработки отличаются от других, мы изучили и сравнили типичные архитектуры четырех разных типов компьютерных систем.

- Приложения.
- Серверные службы.
- Системы пакетной обработки.
- Системы потоковой обработки.

Упражнение

Подумайте и приведите другие примеры приложений, служб, систем пакетной обработки и систем потоковой обработки.

2 Привет, стриминговые системы!

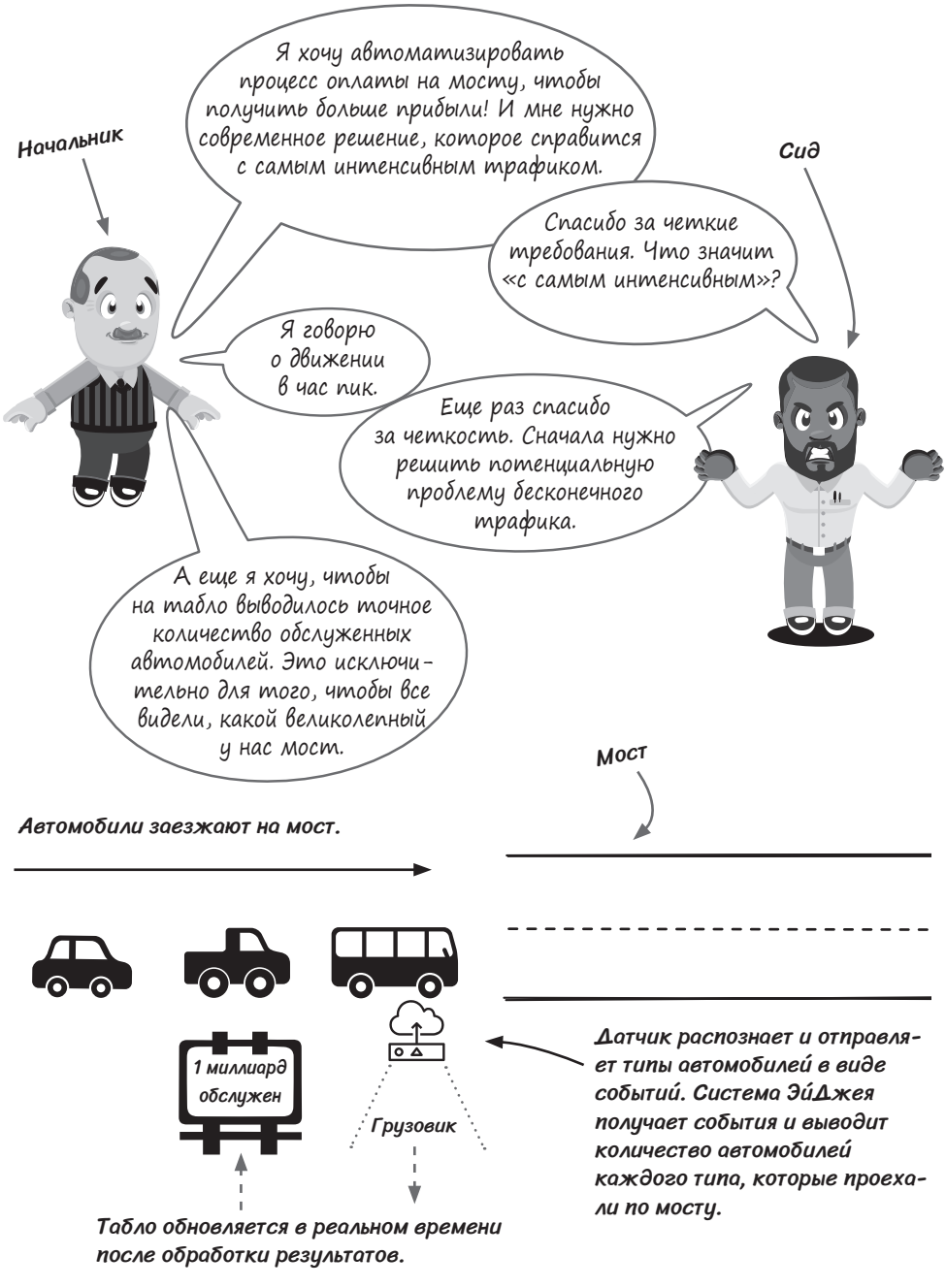
В этой главе

- ✓ События в стриминговых системах.
- ✓ Разные стриминговые компоненты.
- ✓ Построение задания из стриминговых компонентов.
- ✓ Выполнение кода.

«Сначала решите задачу. Потом пишите код».

Джон Джонсон

Начальнику нужен современный пункт оплаты



Вначале были запросы HTTP… и ничего не вышло

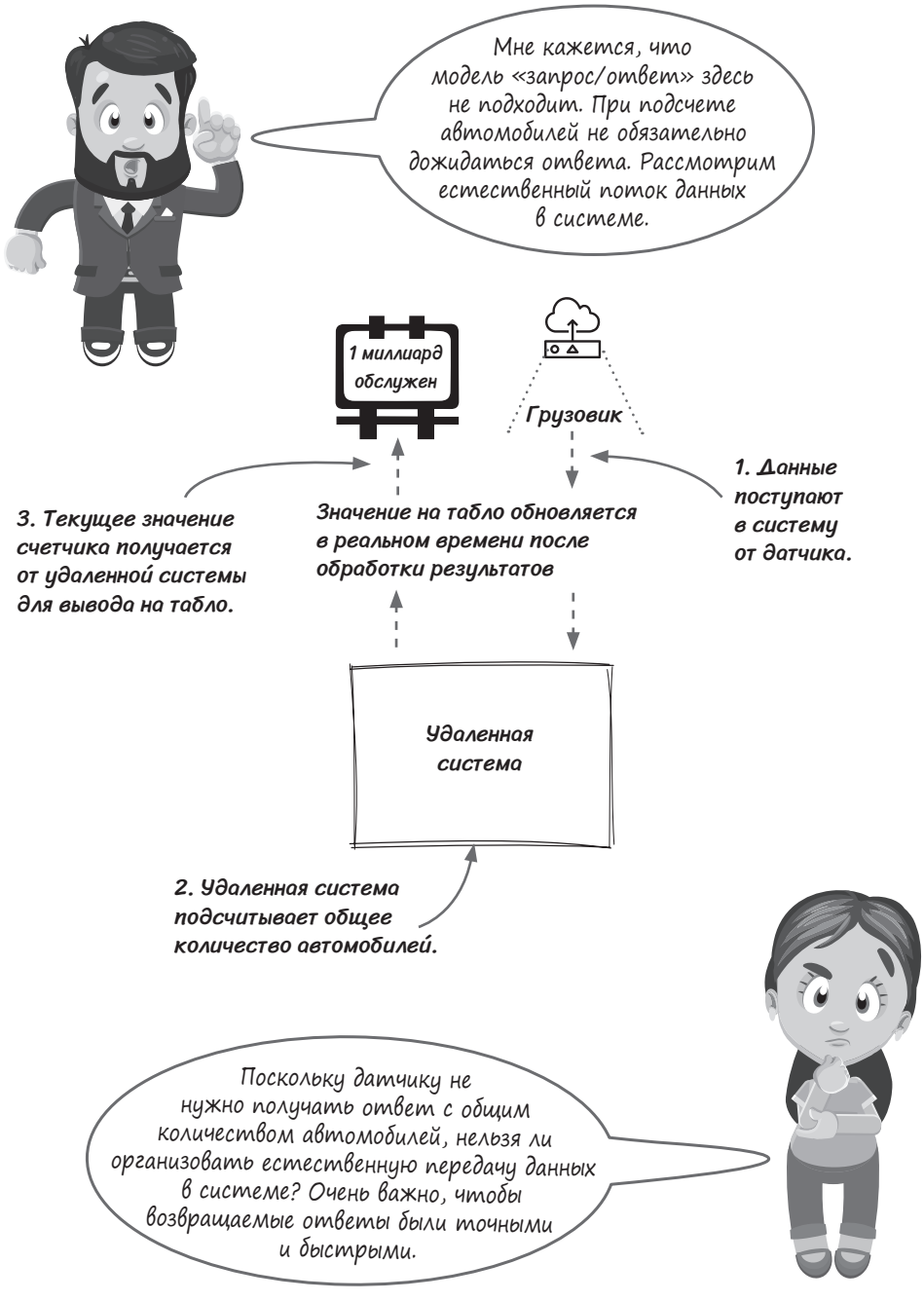
Со стремительным развитием технологий за последние годы большая часть оборудования пунктов оплаты, которые раньше обслуживались вручную, была заменена устройствами IoT («Интернет вещей»). Когда автомобиль въезжает на мост, система получает данные о его типе с датчика IoT. Первая версия системы должна была подсчитывать общее количество автомобилей разных типов (легковых автомобилей, грузовиков, микроавтобусов и т. д.), проехавших по мосту. Начальник хочет, чтобы результаты обновлялись в реальном времени, поэтому каждый раз, когда проезжает новый автомобиль, соответствующий счетчик должен немедленно обновляться.

ЭйДжей, Миранда и Сид, как обычно, начали с проверенной архитектуры серверной службы и использовали запросы HTTP для передачи данных.

Однако попытка завершилась неудачей.

В выходные трафик вырос, и система столкнулась с нагрузкой, с которой не справилась. Из-за задержки при обработке запросов система начала отставать, на табло выводились неточные результаты, а у ЭйДжея и Миранды возникли проблемы.

ЭйДжей и Миранда берут паузу, чтобы подумать



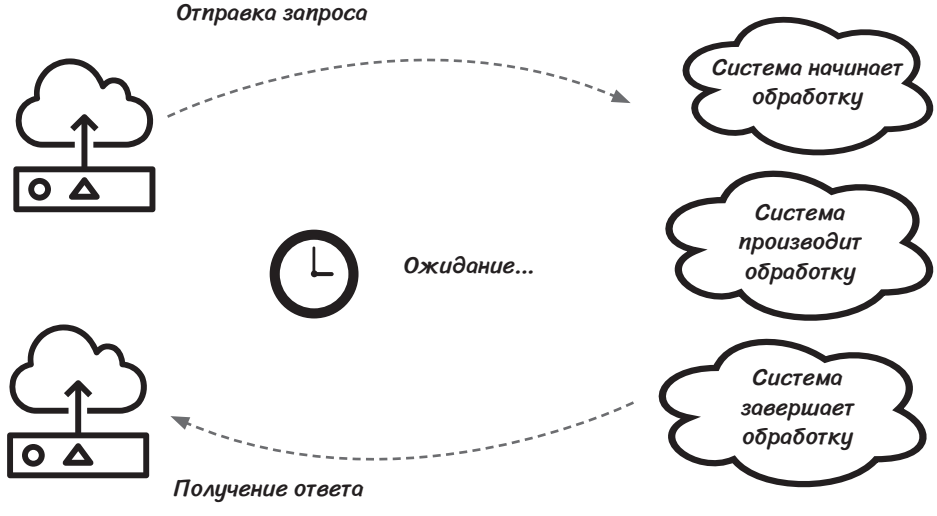
ЭйДжей размышляет о стриминговых системах

Если не углубляться в подробности сетевых взаимодействий и передачи пакетов, различие проявляется еще и в том, как организуется взаимодействие стриминговых систем, использующих архитектуру серверных служб http. Главное отличие архитектуры серверных служб заключается в том, что клиент отправляет запрос, ожидает, пока служба выполнит некие вычисления, а затем получает ответ. В стриминговых системах клиент отправляет запрос, но не дожидается его обработки, прежде чем отправлять следующий. Без необходимости ожидать обработки данных системы быстрее реагируют на события.

Все еще непонятно? Вскоре мы все подробно объясним.

Сравнение серверных служб и потоковой обработки

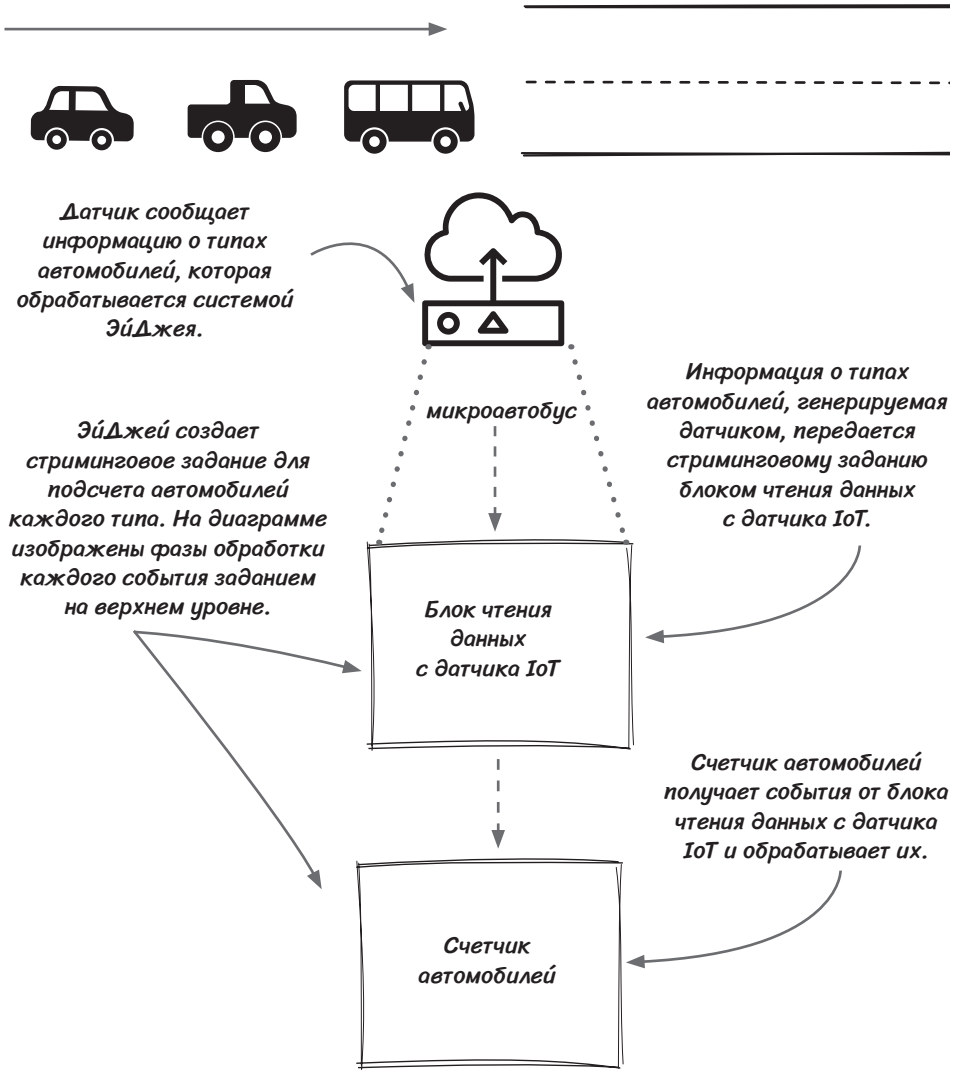
Серверная служба: синхронная модель



Потоковая обработка: асинхронная модель

Как использовать стриминговую систему в этой задаче

На верхнем уровне ЭйДжей отказывается от модели «запрос/ответ» и разделяет процесс на две фазы. Следующая диаграмма показывает, как стриминговая система может вписаться в сценарий с подсчетом автомобилей, проезжающих по мосту. Подробности — далее в этой главе.



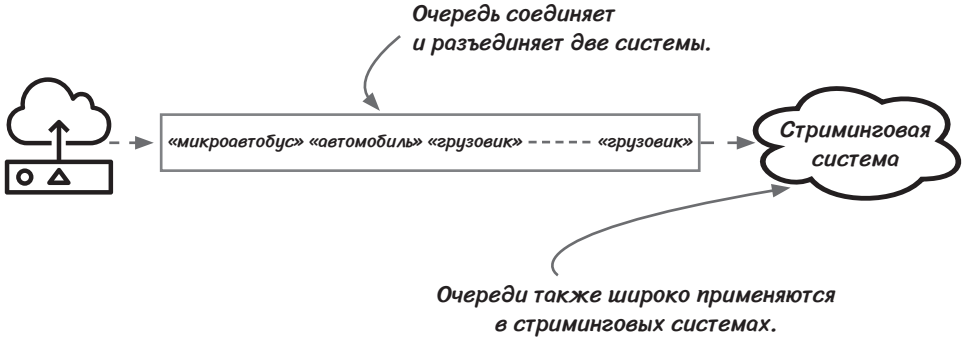
Очереди: фундаментальная концепция

Прежде чем идти дальше, стоит уделить особое внимание одной структуре данных: *очереди*. Эта структура широко используется во всех стриминговых системах.

В традиционных распределенных системах взаимодействие обычно происходит по модели «*запрос/ответ*», также называемой синхронной моделью. Это не относится к потоковым системам, так как модель «запрос/ответ» подразумевает ненужную задержку при работе с данными в реальном времени (формально правильнее говорить о «*системах квазиреального времени*» или «системах, близких к реальному времени», но стриминговые системы часто рассматриваются как системы *реального времени*). На верхнем уровне распределенные стриминговые системы поддерживают долгосрочное подключение к компонентам в пределах системы для сокращения времени передачи данных. Долгосрочное подключение предназначено для непрерывной передачи данных, которая позволяет стриминговым системам реагировать на события по мере их возникновения.

Во всех распределенных системах существует некоторая форма внутренних процессов передачи данных. Среди всех возможных вариантов очередь особенно полезна для упрощения архитектуры стриминговых сценариев использования.

- Очереди помогают организовать разделение модулей в системе, чтобы каждая часть могла работать в своем темпе, не беспокоясь о зависимостях и синхронизации.
- Очереди помогают организовать обработку событий по порядку, так как они являются структурой данных FIFO (First In First Out, то есть «первым вошел, первым вышел»).



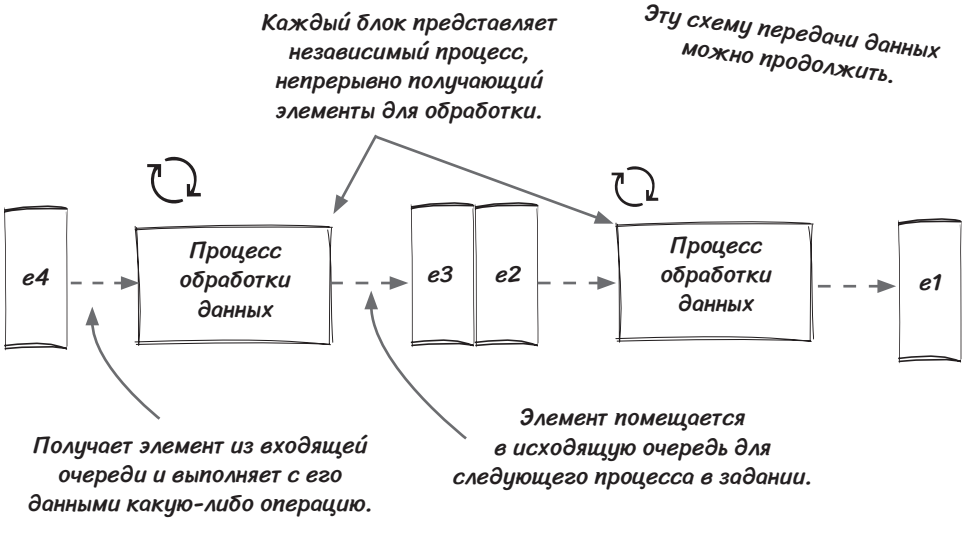
Однако использование очередей для упорядочения непрерывно передаваемых данных имеет свои недостатки. Если вы пытаетесь гарантировать последовательность обработки данных, вас поджидает множество неожиданностей. Эта тема рассматривается в главе 5.

Передача данных в очередях

Уделите пару минут и постарайтесь разобраться в приведенной ниже диаграмме. На ней изображены два компонента и промежуточная очередь событий между ними, а также очереди к предшествующему и последующему компонентам. Передача данных от одного компонента к другому создает концепцию *потока*, то есть непрерывно передаваемых данных.

Процессы и потоки

В компьютерной теории термин «процесс» относится к выполнению программы, а термином «поток выполнения» (thread) обозначается отдельная ветвь выполнения в процессе. Главное различие между ними заключается в том, что разные потоки выполнения одного процесса совместно используют одно пространство памяти, а процессы обладают собственными пространствами памяти. Как потоки выполнения, так и процессы могут использоваться для выполнения операций с данными на приведенной диаграмме. Стриминговые системы могут выбрать тот или иной вариант (или их комбинацию) с учетом своих требований. Чтобы избежать недоразумений, в этой книге термин «процесс» будет использоваться (если явно не указано иное) для обозначения независимой последовательности выполнения в любом варианте реализации.



Потоковый фреймворк (вернее, его начало)

Во время планирования этой книги не раз обсуждалось, как объяснять концепции потоковой обработки без тесной привязки к конкретной стриминговой технологии для примеров. Технологии непрерывно развиваются, и постоянно дополнять материал было бы очень трудно. Мы подумали, что упрощенный фреймворк, который мы назвали Streamwork, поможет изложить базовые концепции стриминговых систем без привязки к конкретному фреймворку.

Фреймворк Streamwork имеет сильно упрощенное ядро, которое работает на компьютере локально. Он может использоваться для построения и запуска простых стриминговых заданий, которые, как мы надеемся, помогут вам в изучении концепций. Функциональность, которая поддерживается в распространенных стриминговых фреймворках, таких как Apache Heron, Apache Storm или Apache Flink, осуществляющих потоковую передачу в реальном времени между физическими машинами, в данном фреймворке ограничена. С другой стороны, так ее проще понять.

Один из самых интересных аспектов работы с компьютерными системами (как нам кажется) заключается в том, что не существует единственно правильного способа решения всех проблем. В отношении функциональности стриминговые фреймворки, включая наш фреймворк Streamwork, имеют много общего, так как они используют общие концепции, но их внутренние механизмы реализации могут сильно различаться из-за различных ограничений и зависимостей.

Подумайте об этом

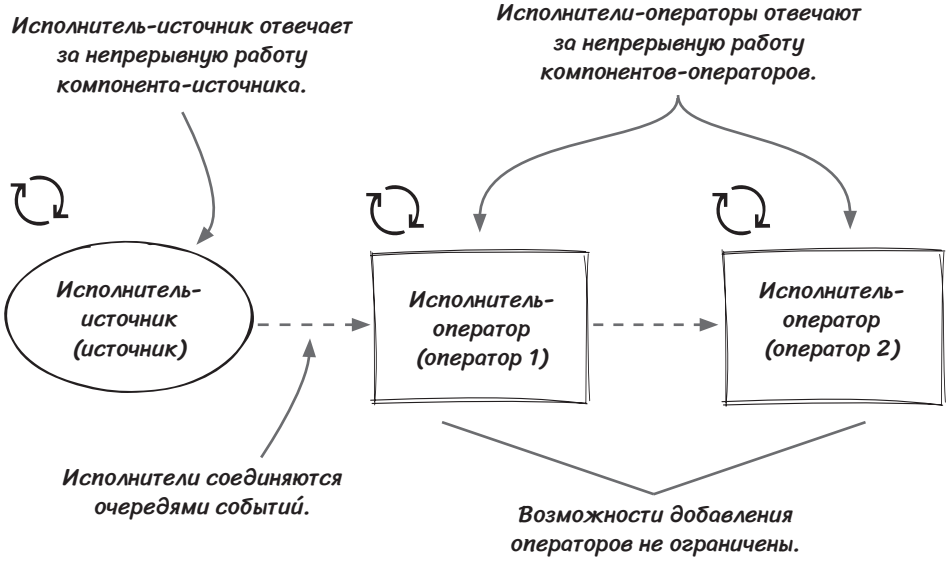
Построение стриминговых систем с нуля потребовало бы слишком больших усилий. Фреймворки берут на себя основную часть рутинной работы, что позволяет нам сосредоточиться на бизнес-логике. Тем не менее иногда важно знать, как работают внутренние механизмы фреймворка.

Обзор фреймворка Streamwork

В общем случае стриминговые фреймворки должны решать две задачи.

- Предоставлять программный интерфейс (API), который будет использоваться разработчиками для подключения специализированной логики и построения задания.
- Предоставлять ядро для выполнения стримингового задания.

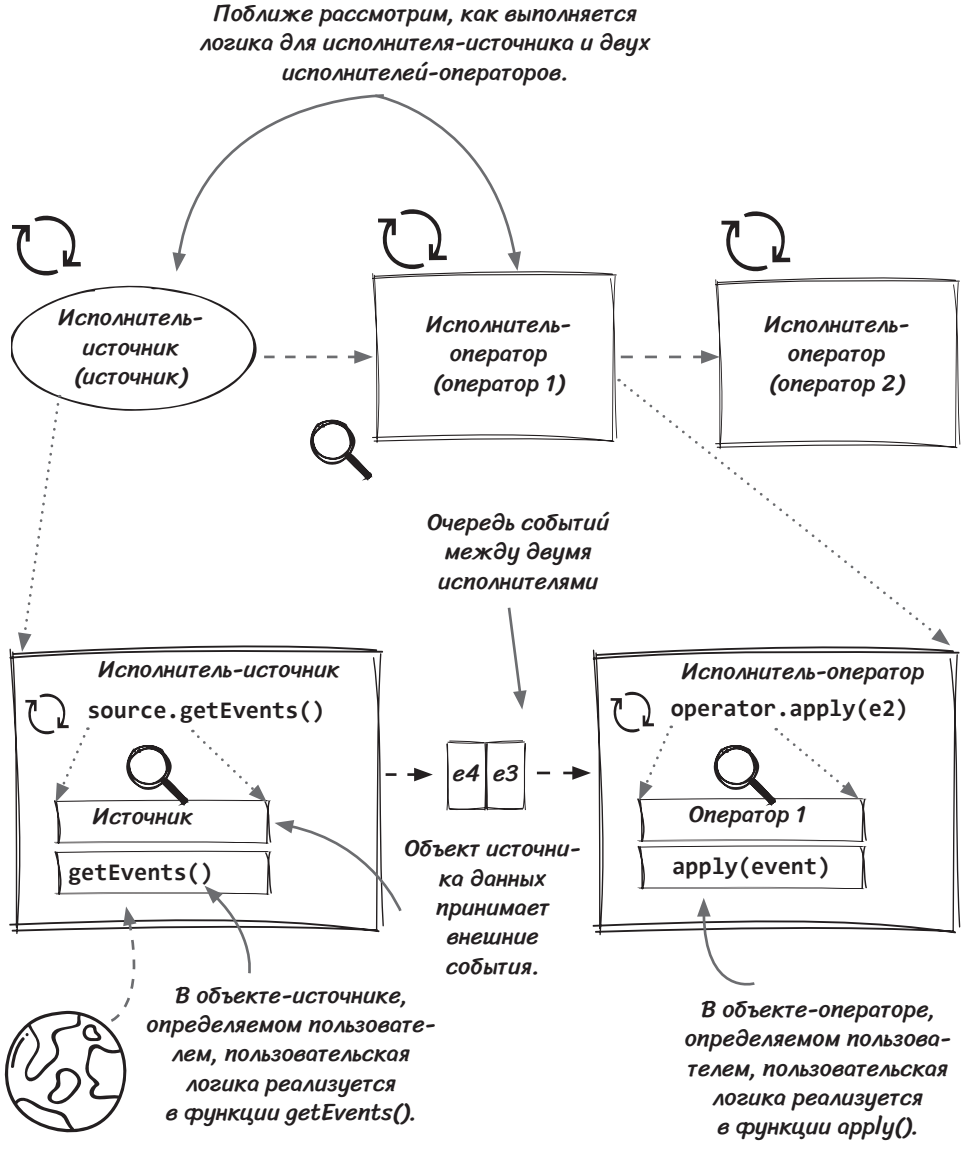
API мы рассмотрим далее. Следует понимать, что эта книга не ставит цель научить вас использовать API Streamwork. Мы используем этот фреймворк только для того, чтобы не привязывать пояснения к конкретному фреймворку. Начнем с ядра. Следующая диаграмма описывает на верхнем уровне все активные составляющие фреймворка Streamwork. Следует понимать, что существует еще один процесс, который запускает каждого исполнителя, а каждый исполнитель в свою очередь запускает источник данных или компонент. Каждый исполнитель существует автономно, он не останавливает и не запускает других исполнителей.



Фреймворк, представленный в этой главе, очень прост. Тем не менее все его обозначенные компоненты сравнимы с компонентами реальных стриминговых фреймворков. В следующих главах во фреймворк Streamwork будет добавляться новая функциональность.

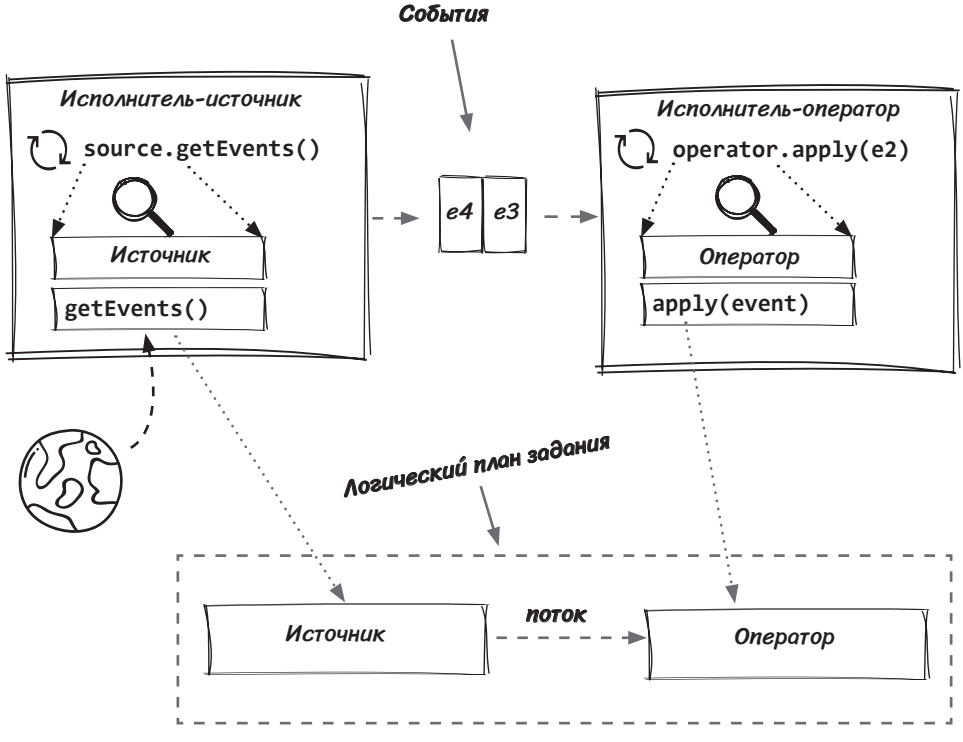
Подробнее о ядре Streamwork

Сейчас мы более подробно рассмотрим, как исполнители применяют пользовательскую логику к событиям.



Основные стриминговые концепции

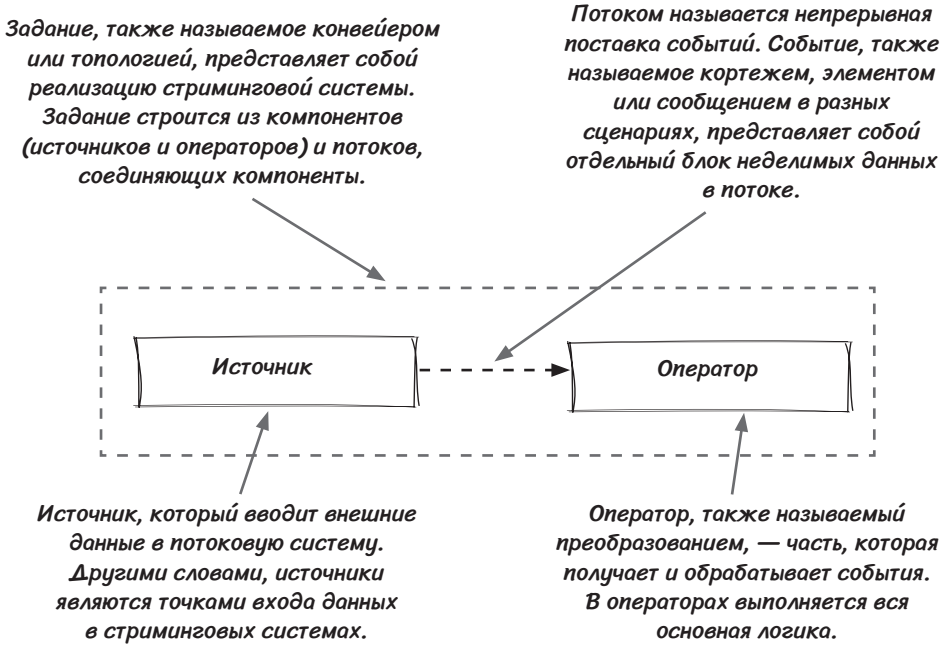
В большинстве стриминговых систем присутствуют пять ключевых концепций: *событие*, *задание*, *источник*, *оператор* и *поток*.



Если забыть об исполнителях и рассматривать только объекты, определяемые пользователем, мы получим новую диаграмму справа — более чистое (более абстрактное) краткое представление стриминговой системы. Эта диаграмма (назовем ее *логическим планом*) является высокоуровневой абстракцией, на которой изображены компоненты и структуры системы, а также логические потоки данных в них. Из этой диаграммы можно видеть, как объект-источник и объект-оператор соединяются через поток для формирования стримингового задания. Следует понимать, что поток — не что иное, как непрерывная передача данных от одного компонента к другому.

Подробнее о концепциях

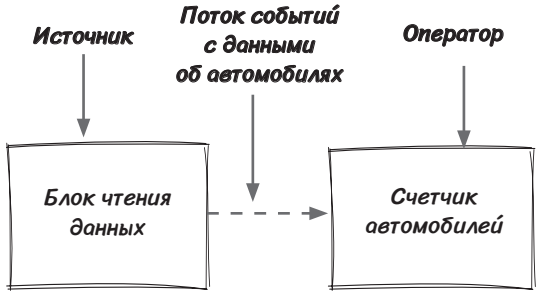
На следующей диаграмме более подробно представлены пять ключевых концепций: *событие*, *задание*, *источник*, *оператор* и *поток*.



Мы также рассмотрим применение концепций в потоковой системе при работе с первым стриминговым заданием. А пока убедитесь, что вы поняли суть пяти ключевых концепций.

Последовательность выполнения стримингового задания

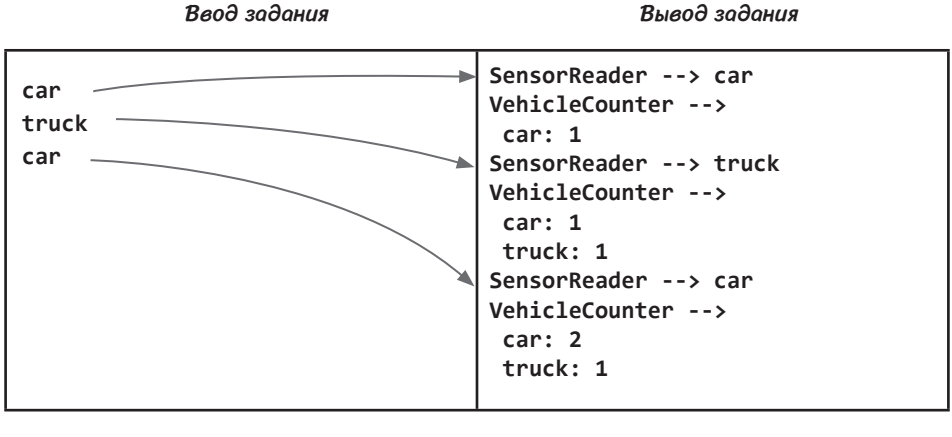
С концепциями, рассмотренными на двух предыдущих страницах, можно составить визуализацию стримингового задания для подсчета автомобилей, состоящего из двух компонентов и одного потока между ними, как показано справа.



- Блок чтения данных получает данные с датчика и сохраняет события в очереди. Это источник.
- Счетчик автомобилей отвечает за подсчет автомобилей в потоке. Это оператор.
- Непрерывное перемещение данных от источника к оператору — поток событий автомобилей.

Блок чтения данных с датчика становится началом задания, а счетчик автомобилей — его концом. Линия, соединяющая блок чтения (источник) со счетчиком автомобилей (оператором), представляет поток типов автомобилей (событий), проходящий от блока чтения данных к счетчику автомобилей.

В этой главе такая система будет описана более подробно. Она будет выполняться на ваших локальных компьютерах с двумя терминалами: один получает ввод пользователя (левый столбец), а другой — вывод задания (правый столбец).



Первое стриминговое задание

Стриминговое задание создается средствами Streamwork API и включает следующие шаги.

1. Создание класса задания.
2. Построение источника.
3. Построение оператора.
4. Соединение компонентов.

Первое стриминговое задание: создание класса события

Событие — один фрагмент данных в потоке, который должен обрабатываться заданием. Во фреймворке Streamwork класс API `Event` отвечает за хранение или инкапсуляцию пользовательских данных. Аналогичные концепции существуют и в других стриминговых системах.

В задании каждое событие представляет один тип автомобилей. Для простоты будем считать, что каждый тип представляет собой строку (например, `car` или `truck`). В нашем примере будет использоваться класс события `VehicleEvent`, производный от класса `Event` из API. Каждый объект `VehicleEvent` содержит информацию об автомобиле, которую можно получить вызовом функции `getData()`.

Внутренняя строка для обозначения типа автомобиля.

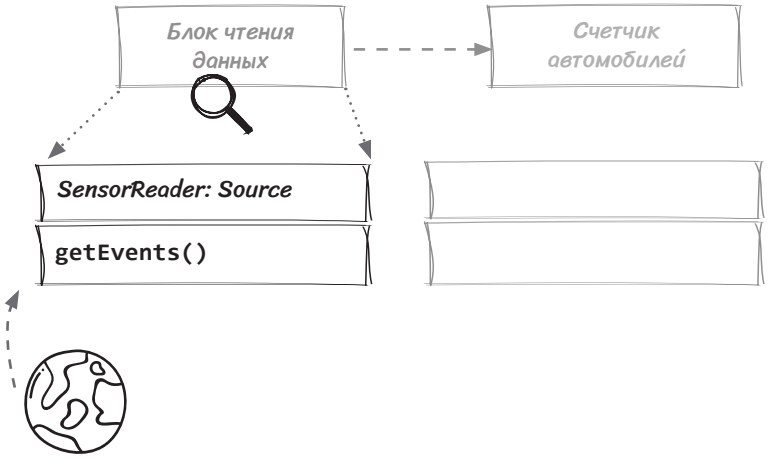
```
public class VehicleEvent extends Event {
  private final String vehicle;

  public VehicleEvent(String vehicle) {
    this.vehicle = vehicle;
  }

  @Override
  public String getData() {
    return vehicle;
  }
}
```

Конструктор получает vehicle в виде строки и сохраняет значение.

Получает данные, хранящиеся в событии.

Первое стриминговое задание: источник данных

Источником (source) называется компонент, который вводит внешние данные в стриминговую систему. Земной шар на следующей диаграмме обозначает данные, внешние по отношению к заданию. В стриминговом задании блок чтения данных от датчика вводит данные, полученные от локального порта, в систему.



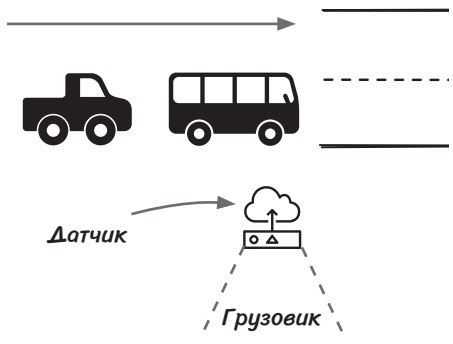
Во всех стриминговых фреймворках присутствует API, дающий возможность задавать для источников данных логику, которая представляет интерес только для вас. Во всех API источников данных присутствует некая разновидность *перехватчика жизненного цикла* (lifecycle hook), который будет вызываться для получения внешних данных. В этой точке код выполняется фреймворком.

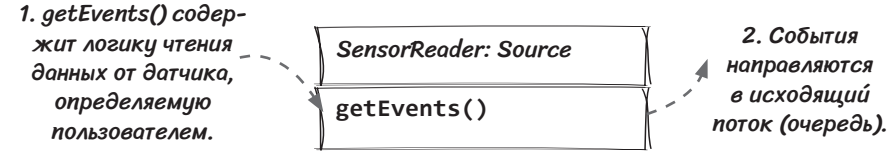
Что такое перехватчик жизненного цикла?

Перехватчиками жизненного цикла называются методы, которые вызываются по определенному повторяемому шаблону фреймворком, которому они принадлежат. Как правило, эти методы позволяют разработчикам настроить поведение приложения в фазах жизненного цикла фреймворка, на основе которого строится приложение. В случае фреймворка Streamwork имеется перехватчик жизненного цикла (или метод), называемый getEvents(). Он многократно вызывается фреймворком, чтобы вы могли получить внешние данные. Перехватчики жизненного цикла позволяют разработчикам писать логику, которая для них важна, и поручить рутинную работу фреймворку.

Первое стриминговое задание: источник данных (продолжение)

В этом задании блок чтения данных датчика будет читать события. В нашем упражнении вы будете моделировать датчик на мосту, самостоятельно создавая события и передавая их на открытый порт вашего компьютера, прослушиваемый стриминговым заданием. Блок чтения получает данные о типах автомобилей, отправленные в порт, и передает их потоковому заданию — так выглядит бесконечный (или неограниченный) поток событий.





Java-код класса `SensorReader` выглядит так:

Перехватчик жизненного цикла стримингового системы выполняет логику, определяемую пользователем.

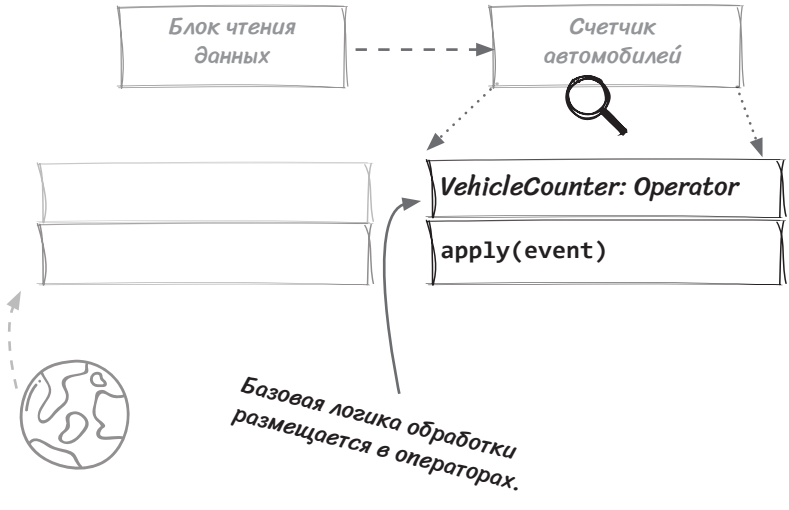
Чтение типа автомобиля из входных данных.

```
public class SensorReader extends Source {
  private final BufferedReader reader;
  public SensorReader(String name, int port) {
    super(name);
    reader = setupSocketReader(port);
  }

  @Override
  public void getEvents(List<Event> eventCollector) {
    String vehicle = reader.readLine();
    eventCollector.add(new VehicleEvent(vehicle));
    System.out.println("SensorReader --> " + vehicle);
  }
}
```

Передача строки получателю.

Первое стриминговое задание: оператор

Вся пользовательская логика обработки содержится в операторах. Они отвечают за получение входящих событий для обработки и генерирование исходящих событий; следовательно, у них есть как ввод, так и вывод. Вся логика обработки данных в стриминговых системах обычно размещается в компонентах операторов.



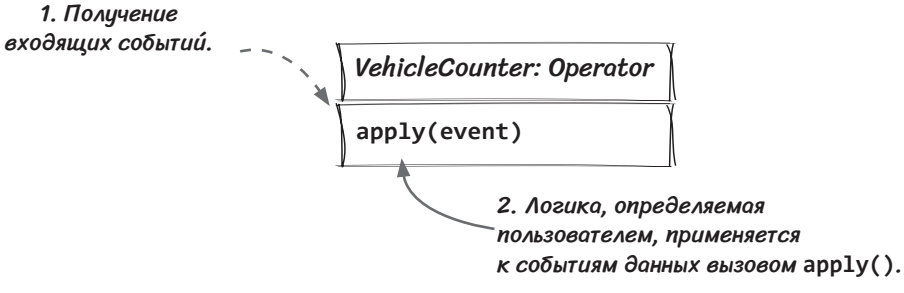
Для простоты мы ограничились одним источником и одним оператором. Рассматриваемая реализация счетчика автомобилей только подсчитывает автомобили, а затем регистрирует текущее значение счетчика в системе. Другой (возможно, лучший) способ реализации системы основан на направлении данных в новый поток. Тогда регистрация результатов может выполняться в дополнительном компоненте, который следует за счетчиком автомобилей. Как правило, в задании используются компоненты, которые имеют только одну функцию.

Кстати говоря, Сид занимает должность технического директора. Иногда он бывает старомодным, но он очень умен и интересуется новыми технологиями.

Первое стриминговое задание: оператор (продолжение)

В компоненте `VehicleCounter` карта `<vehicle, count>` используется для хранения счетчиков типов автомобилей в памяти. Она обновляется соответствующим образом при получении нового события. В стриминговом задании счетчик автомобилей представляет собой оператор, который подсчитывает события. Этот оператор завершает задание и не генерирует вывод для последующих операторов.

Ключ (Vehicle)	*Значение (Count)*
автомобиль	*2*
грузовик	*1*
микроавтобус	*1*




```
public class VehicleCounter extends Operator {
  private final Map<String, Integer> countMap =
    new HashMap<String, Integer>();

  public VehicleCounter(String name) {
    super(name);
  }

  @Override
  public void apply(Event event,List<Event> collector) {
    String vehicle = ((VehicleEvent)event).getData();
    Integer count = countMap.getOrDefault(vehicle, 0);
    count += 1;
    countMap.put(vehicle, count);
    System.out.println("VehicleCounter --> ");
    printCountMap();
  }
}
```

Получение счетчиков из карты

Увеличение счетчика

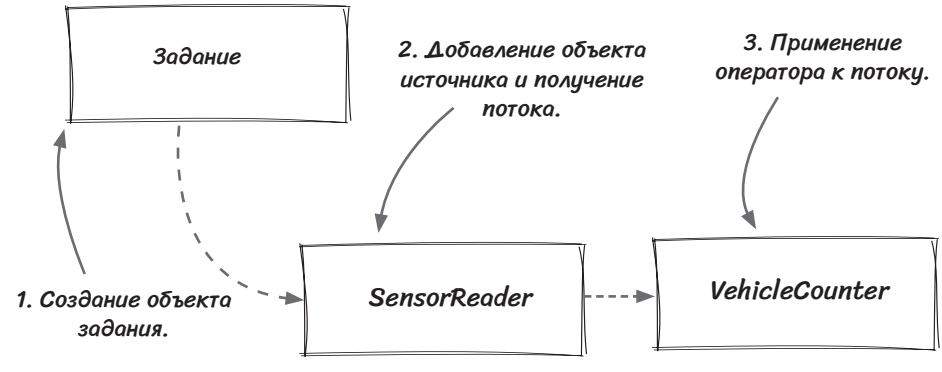
Сохранение счетчика обратно в карте

Вывод текущего значения счетчика

Первое стриминговое задание: сборка задания

Чтобы собрать стриминговое задание, необходимо добавить источник `SensorReader` и оператор `VehicleCounter` и соединить их. В классах `Job` и `Stream`, которые мы построили для вас, присутствуют перехватчики:

- `Job.addSource()` добавляет источник данных в задание.
- `Stream.applyOperator()` добавляет оператор в поток.



Следующий код выполняет описанные выше шаги:

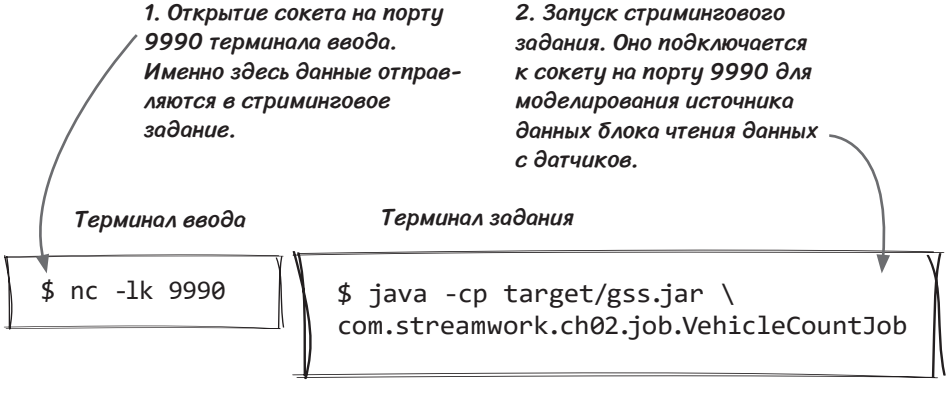
```
public static void main(String[] args) {
  Job job = new Job();                                    Создание объекта задания
  Stream bridgeOut=job.addSource(new SensorReader());
                                  Добавление объекта потока и получение потока

  bridgeOut.applyOperator(newVehicleCounter());           Применение оператора
  JobStarter starter = new JobStarter(job);               к потоку
  starter.start();                                        Запуск задания
}
```

Выполнение задания

Все, что вам понадобится для выполнения задания, — машина с операционной системой Mac, Linux или Windows с доступом к *терминалу* (*командной строке* в Windows). Вам также понадобятся инструменты для компиляции и выполнения кода: git, пакет Java Development Kit (JDK) 11, Apache Maven, Netcat (или Nmap в Windows). После установки всех инструментов вы сможете загрузить код и откомпилировать его:

```
$ git clone https://github.com/nwangtw/GrokkingStreamingSystems.git
$ cd GrokkingStreamingSystems
$ mvn package
```

Команда `mvn` генерирует файл target/gss.jar. Наконец, для запуска стримингового задания вам понадобятся два терминала: для запуска задания и для отправки данных, обрабатываемых заданием.



Откройте новый терминал (терминал ввода) и выполните следующую команду (обратите внимание: в Mac и Linux используется команда `nc`, а в Windows — команда `ncat`):

```
$ nc -lk 9990
```

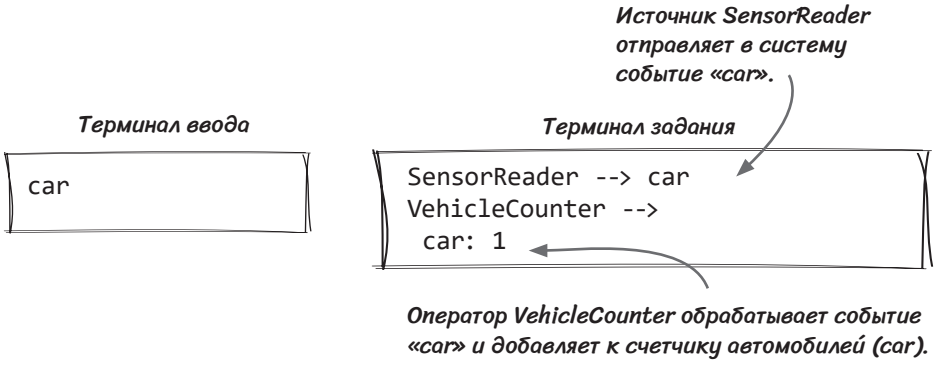
Команда запускает на порту 9990 небольшой сервер, к которому можно подключаться из других приложений. Весь пользовательский ввод в этом терминале будет направляться в порт.

Затем в исходном терминале (терминале задания), который использовался для компиляции задания, выполните задание следующей командой:

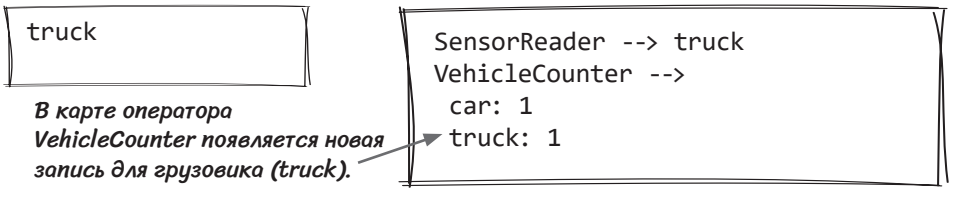
```
$ java -cp target/gss.jar com.streamwork.ch02.job.VehicleCountJob
```

Ход выполнения задания

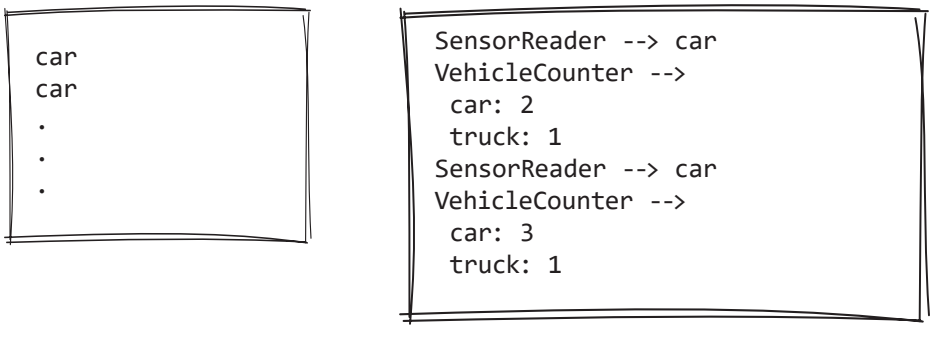
После запуска задания введите в *терминале ввода* строку `car` и нажмите Ввод. Счетчик выводится в *терминале задания*.



Если теперь ввести в терминале ввода строку `truck`, в терминале задания будут выведены счетчики для `car` и `truck`.

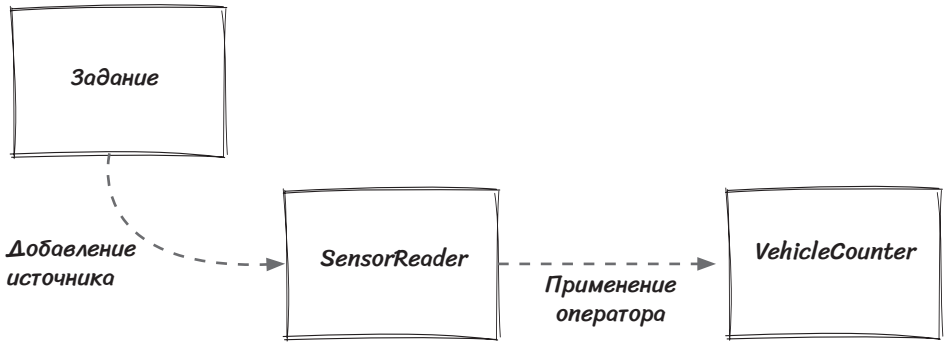


Вы можете и дальше вводить разные типы автомобилей (можно заранее подготовить набор данных в текстовом редакторе и скопировать их в терминал ввода); задание будет продолжать выводить счетчики, как в приведенном ниже примере, пока вы не остановите его. Как видно, после поступления данных в систему стриминговое задание немедленно приступает к их обработке.

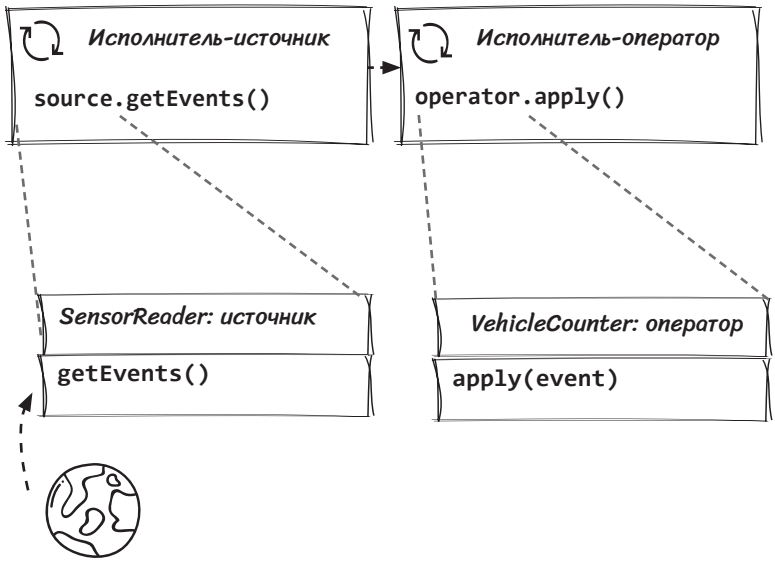


Внутри ядра

Итак, вы узнали, как создаются компоненты и задание, а также понаблюдали за ходом выполнения задания на компьютере. И вероятно, вы заметили, что при этом события автоматически перемещаются от объекта блока чтения данных к объекту счетчика автомобилей без реализации дополнительной логики. Интересно, не правда ли?

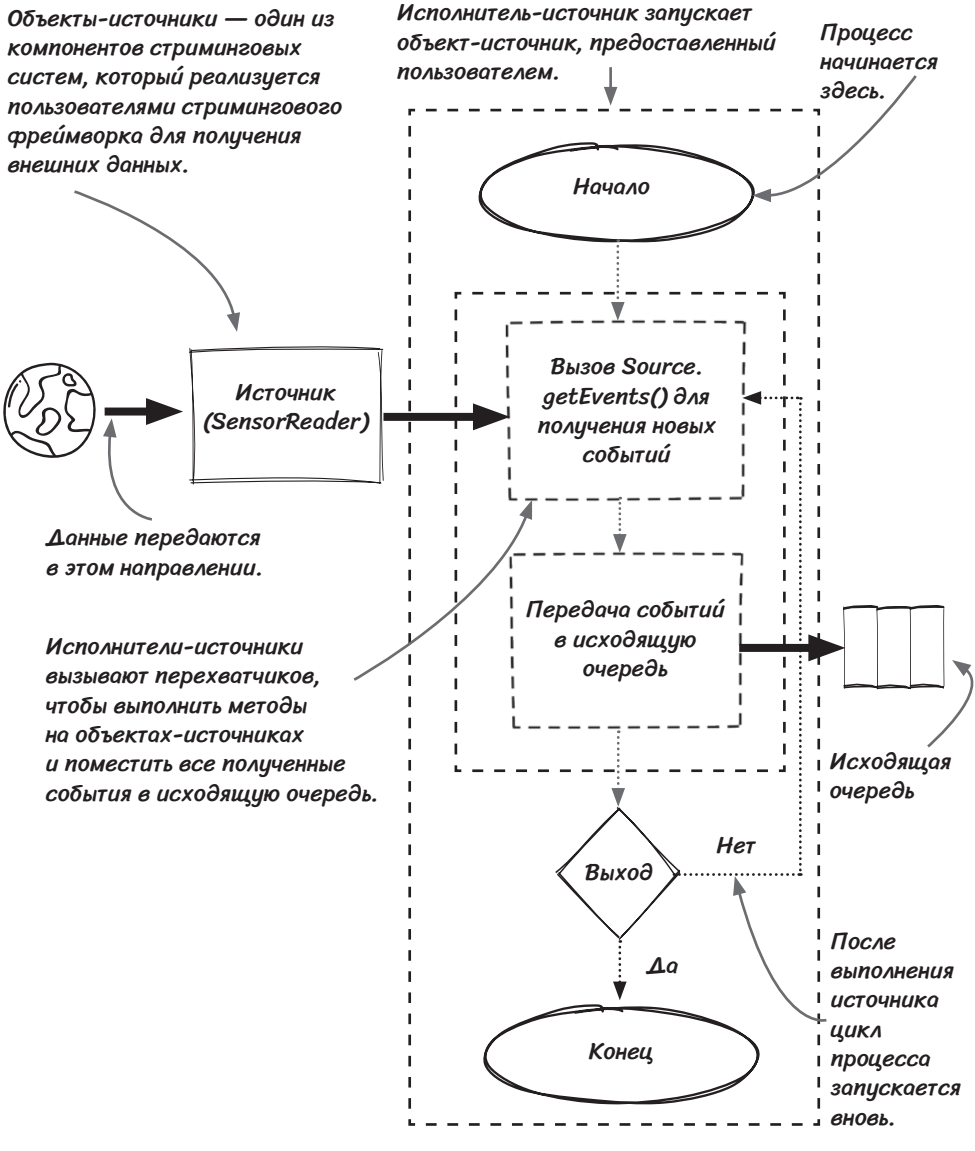


Задания и компоненты не выполняются сами по себе. Ими управляет стриминговое ядро. Заглянем под капот и посмотрим, как задание выполняется в ядре Streamwork. Всего существуют три активные части (на этом этапе), и мы рассмотрим их поочередно: *исполнитель-источник*, *исполнитель-оператор* и *блок запуска задания*.



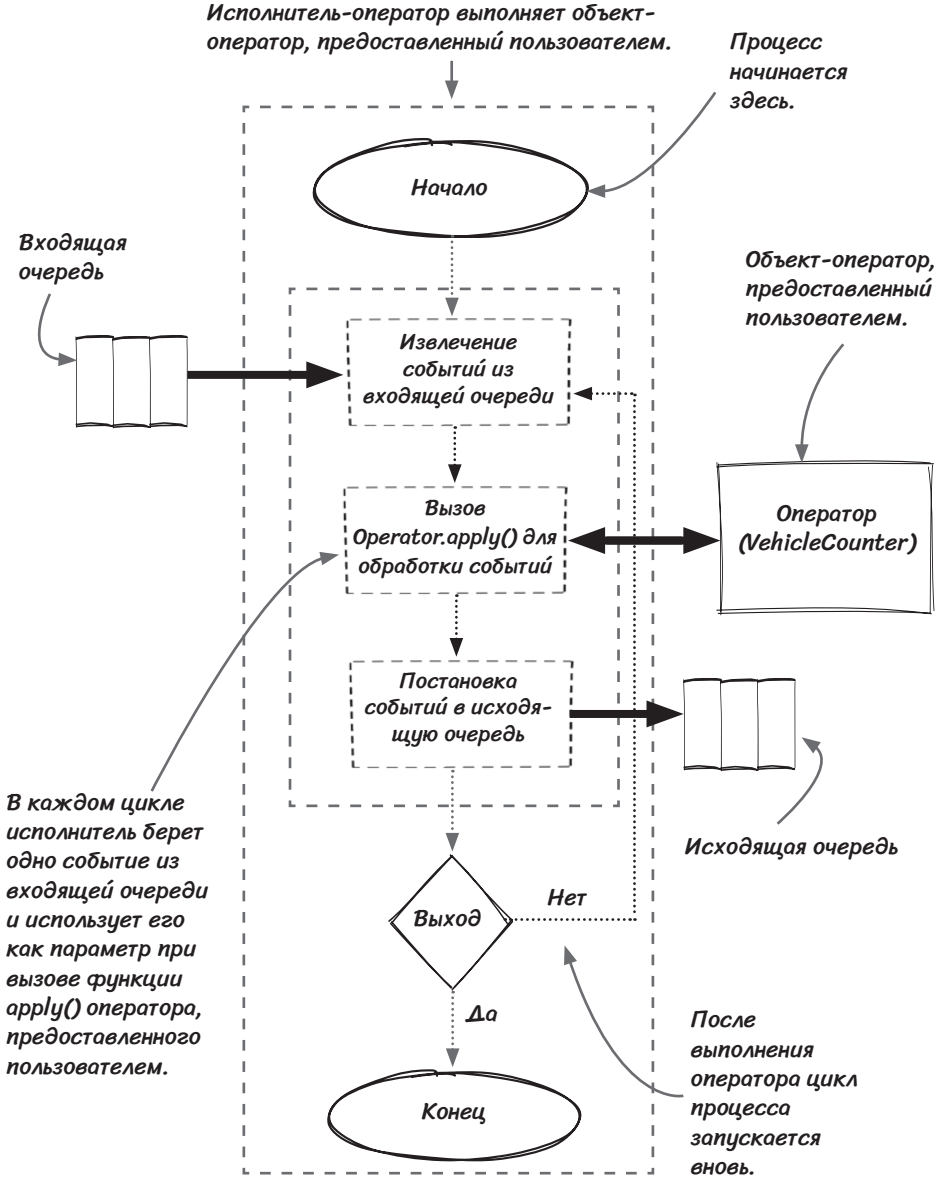
Внутри ядра: исполнители-источники

Во фреймворке Streamwork, который мы построили для вас, исполнитель-источник непрерывно управляет получением данных от источников. Для этого он выполняет бесконечные циклы, которые извлекают внешние данные для включения в исходящую очередь внутри стримингового задания. Несмотря на то что для блока *Выход* предусмотрен вариант *Да*, он никогда не будет реализован.



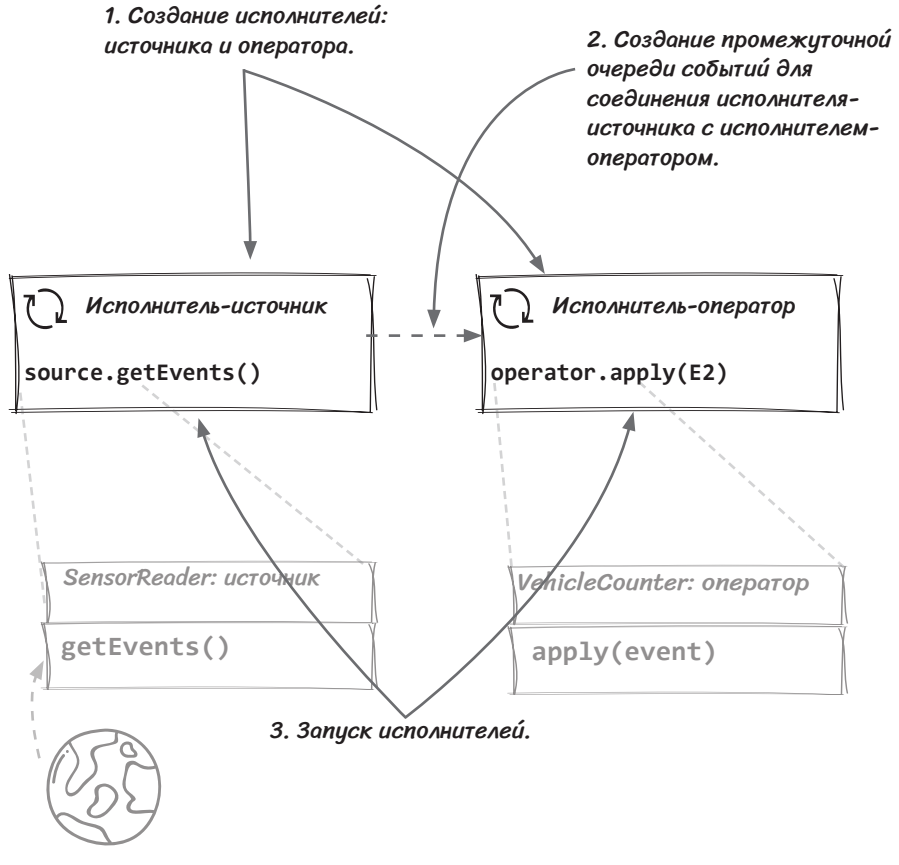
Внутри ядра: исполнители-операторы

В Streamwork исполнитель-оператор работает почти так же, как исполнитель-источник. Единственное отличие заключается в том, что ему приходится управлять очередью входящих событий. И хотя для блока *Выход* предусмотрен вариант *Да*, он никогда не будет реализован.



Внутри ядра: блок запуска задания

`JobStarter` отвечает за подготовку всех активных частей (исполнителей) задания, а также связей между ними. Наконец, он запускает исполнителей для обработки данных. После того как исполнители будут запущены, события начинают проходить через компоненты.

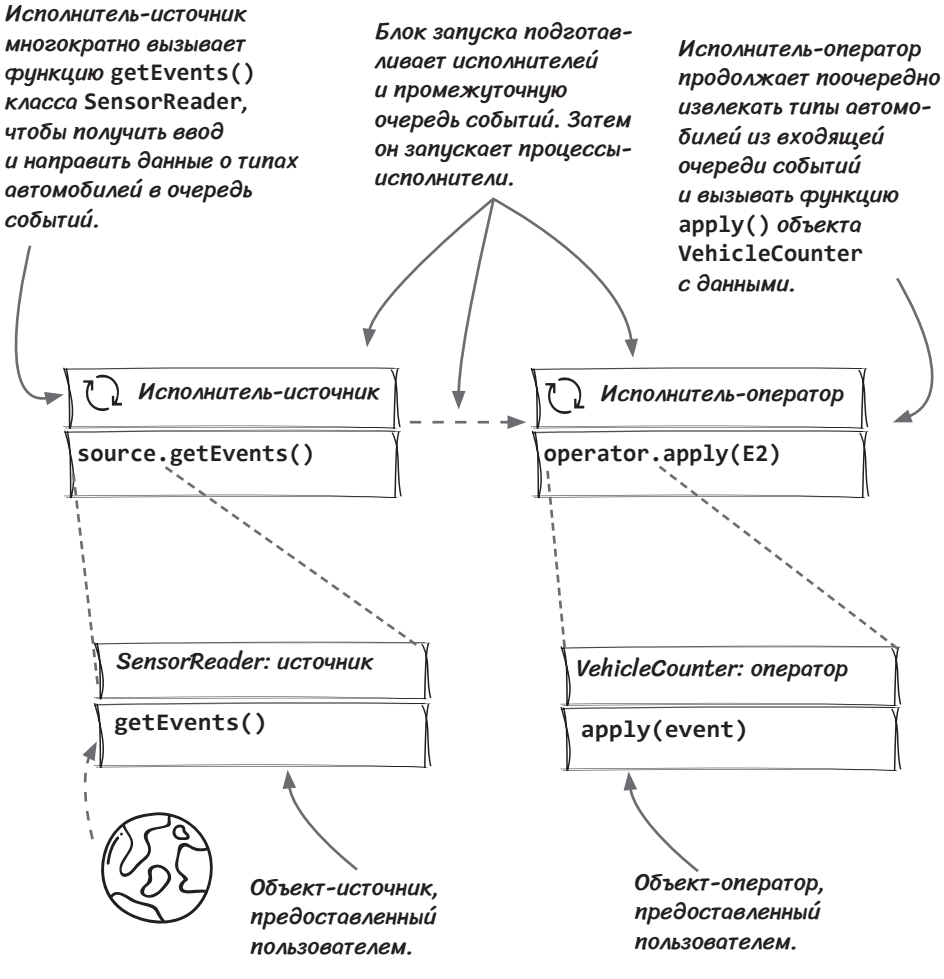


Помните!

Здесь описана архитектура типичного стримингового ядра, чтобы обобщить работу фреймворков на верхнем уровне. Разные стриминговые фреймворки могут работать по-разному.

Перемещение событий

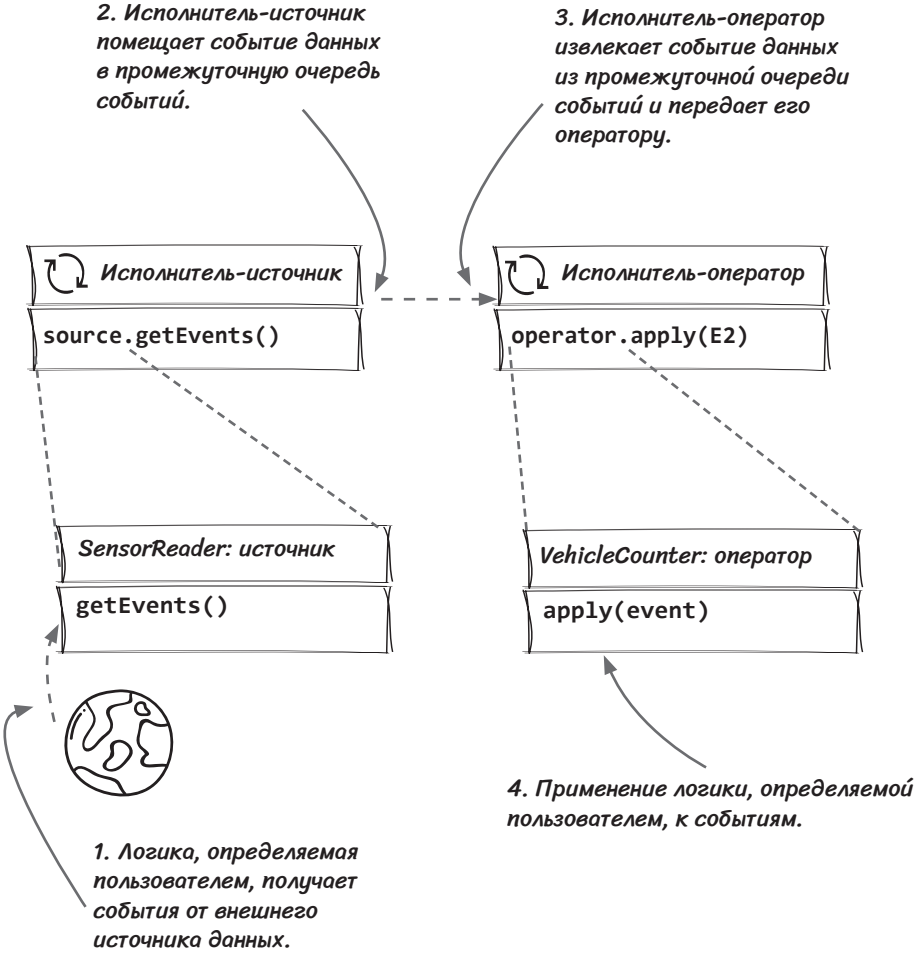
Рассмотрим подробнее ядро и его активные части, включая компоненты активного задания, определяемые пользователем.



После запуска задания все исполнители начинают выполняться параллельно — иначе говоря, одновременно!

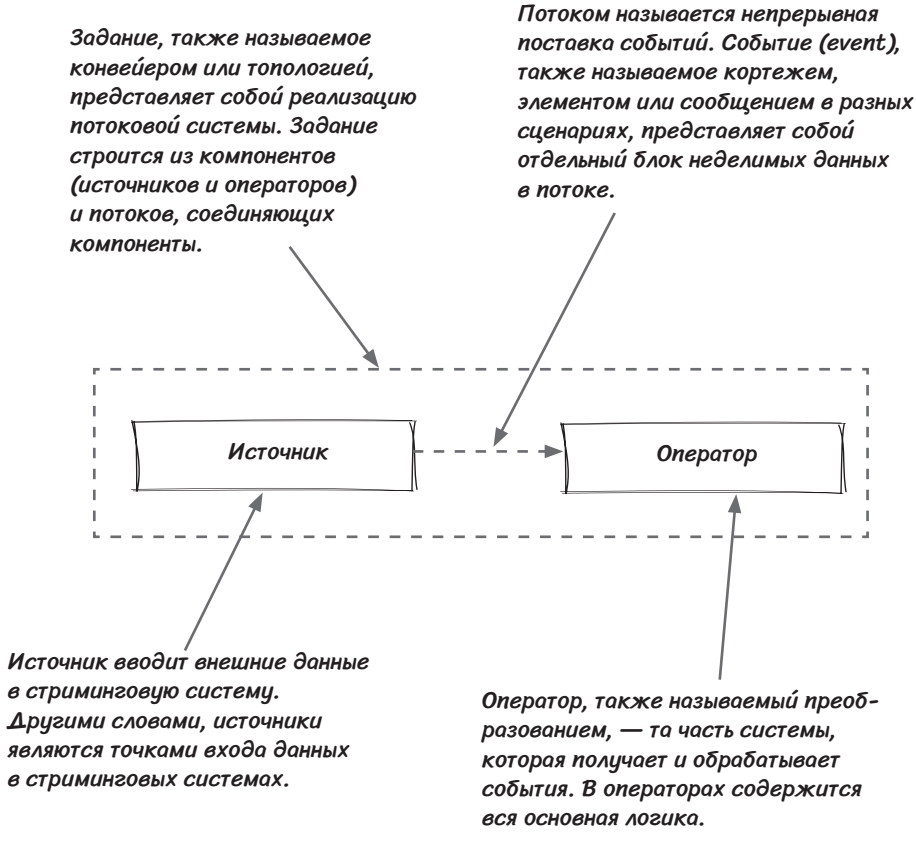
Жизненный цикл элемента данных

Рассмотрим другой аспект стриминговых систем — жизненный цикл одного события. Если добавить строку *car* и нажать клавишу Ввод в терминале ввода, событие пройдет через стриминговую систему, как объясняется на следующей диаграмме.



Краткий обзор концепций стриминга

Поздравляем, вы справились с первым стриминговым заданием! А теперь вернемся немного назад и сделаем обзор ключевых концепций стриминговых систем.



Итоги

Стриминговое задание представляет собой систему, обрабатывающую события в реальном времени. Каждый раз, когда происходит событие, задание принимает его в систему и обрабатывает его. В этой главе мы построили простое задание для подсчета автомобилей, въезжающих на мост, и рассмотрели следующие концепции.

- Потоки и события.
- Компоненты (источники и операторы).
- Потоковые задания.

Кроме того, мы рассмотрели простое стриминговое ядро, чтобы понять, как на самом деле выполняется задание. И хотя это ядро сильно упрощено и выполняется на локальном компьютере, а не в распределенной среде, оно включает типичные активные компоненты.

Упражнения

1. Чем источник отличается от оператора?
2. Найдите в реальной жизни три примера, которые можно смоделировать в виде стриминговых систем. (А если расскажете нам о них, возможно, мы используем их в следующем издании этой книги!)
3. Загрузите исходный код и измените реализацию `SensorReader` для автоматической генерации событий.
4. Измените логику `VehicleCounter` для вычисления накопленной суммы сбора в реальном времени. Размер сбора для каждого типа автомобилей задайте самостоятельно.
5. Оператор `VehicleCounter` в первом задании выполняет две функции: подсчет автомобилей и вывод результатов — и это не лучший вариант. Попробуйте изменить реализацию и переместить логику вывода в новый оператор.

Параллелизация и группировка данных

3

В этой главе

- ✓ Параллелизация.
- ✓ Параллелизм данных и параллелизм задач.
- ✓ Группировка событий.

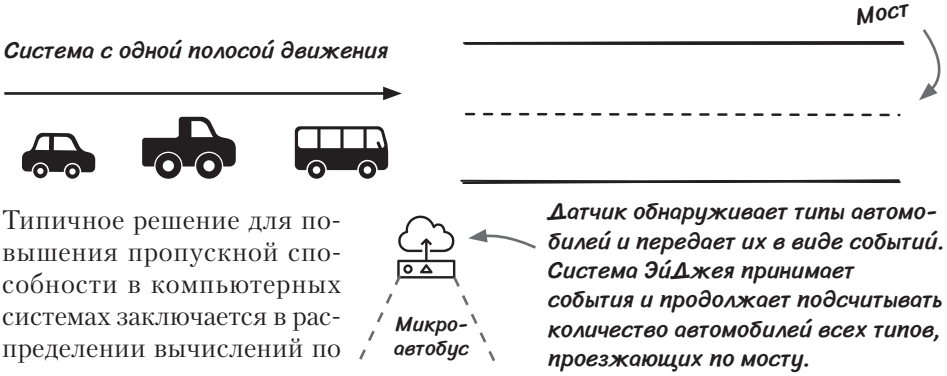
«Девять женщин не родят ребенка за один месяц».

Фредерик П. Брукс

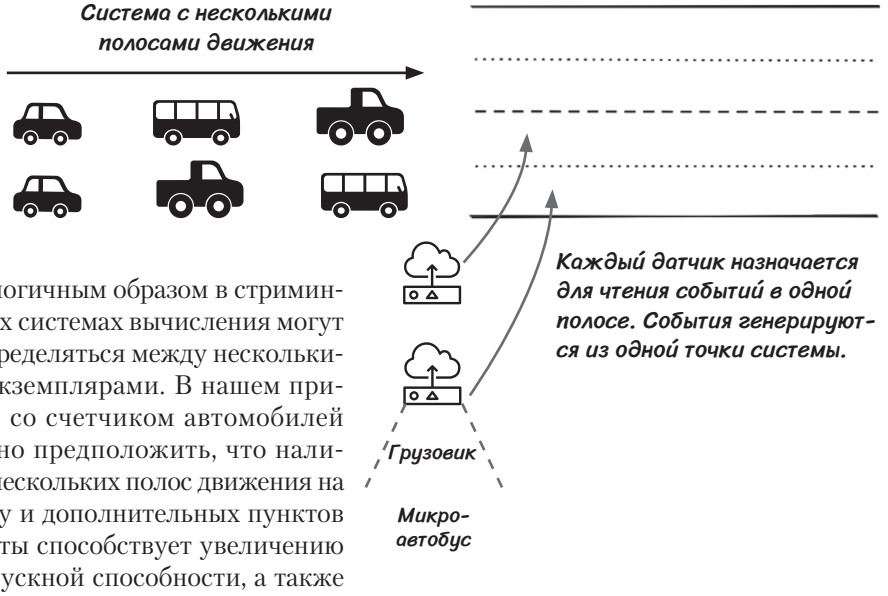
В предыдущей главе ЭйДжей и Миранда взялись за задачу подсчета в реальном времени количества проезжающих по мосту автомобилей с использованием стримингового задания. В системе, построенной ими, возможности обработки интенсивного трафика относительно ограничены. Представьте, что творится в пункте оплаты на мосту с одной полосой движения в час пик. Ужас! В этой главе мы изучим базовый прием для решения фундаментальной задачи в большинстве распределенных систем — масштабирования системы для повышения пропускной способности задания или, другими словами, обработки большего объема данных.

Датчик генерирует больше событий

В предыдущей главе ЭйДжей и Миранда взялись за задачу подсчета в реальном времени количества проезжающих по мосту автомобилей с использованием стримингового задания. Был принят вариант сбора данных о движении одним датчиком, сообщающим о событиях. Естественно, начальник хочет заработать больше денег, поэтому он решает построить больше полос на мосту. По сути, он требует, чтобы стриминговое задание масштабировалось по количеству событий трафика, которые могут обрабатываться за один раз.



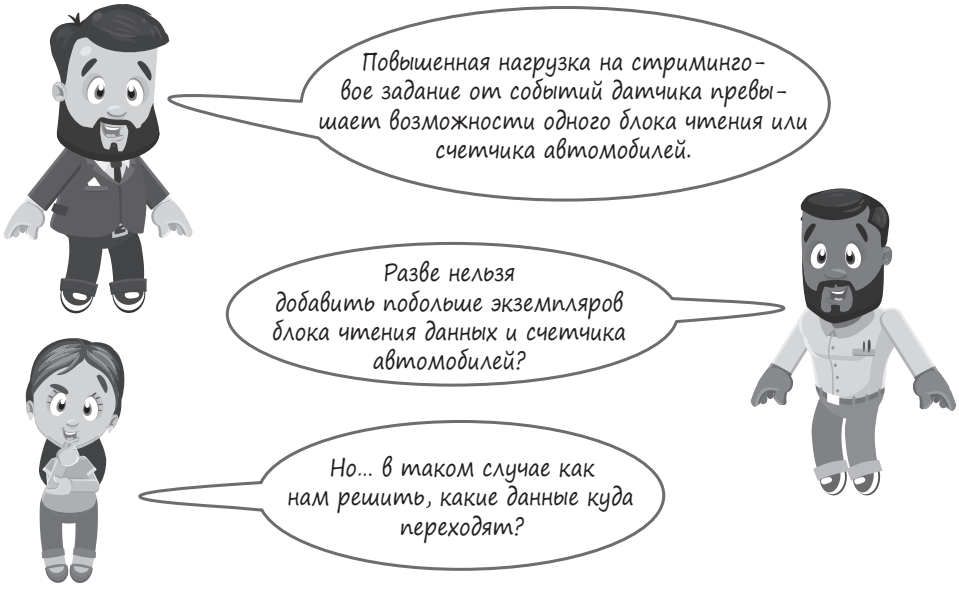
Типичное решение для повышения пропускной способности в компьютерных системах заключается в распределении вычислений по нескольким процессам; это называется *параллелизацией*.

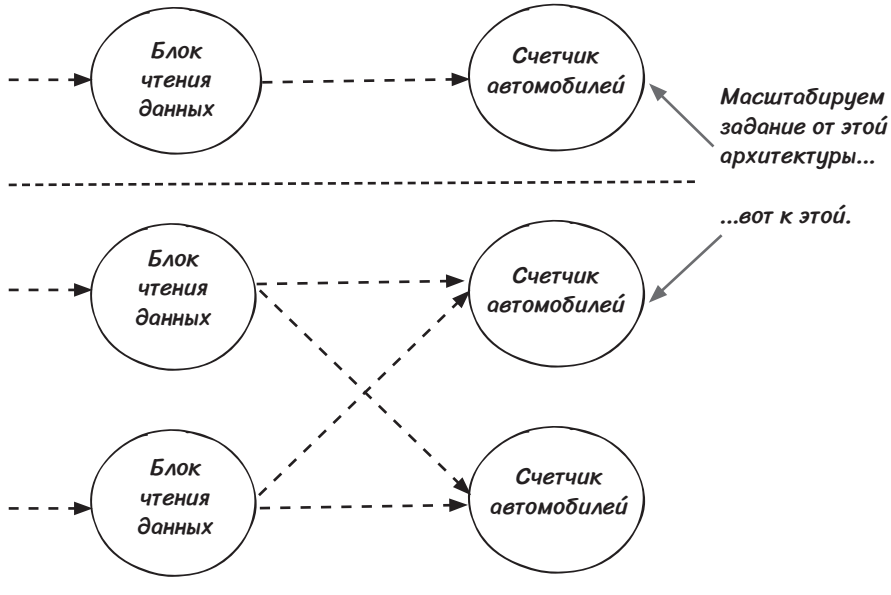


Аналогичным образом в стриминговых системах вычисления могут распределяться между несколькими экземплярами. В нашем примере со счетчиком автомобилей можно предположить, что наличие нескольких полос движения на мосту и дополнительных пунктов оплаты способствует увеличению пропускной способности, а также сокращению времени ожидания.

Даже в потоковых системах непросто добиться обработки в реальном времени

При увеличении количества полос задание перестало справляться с нагрузкой



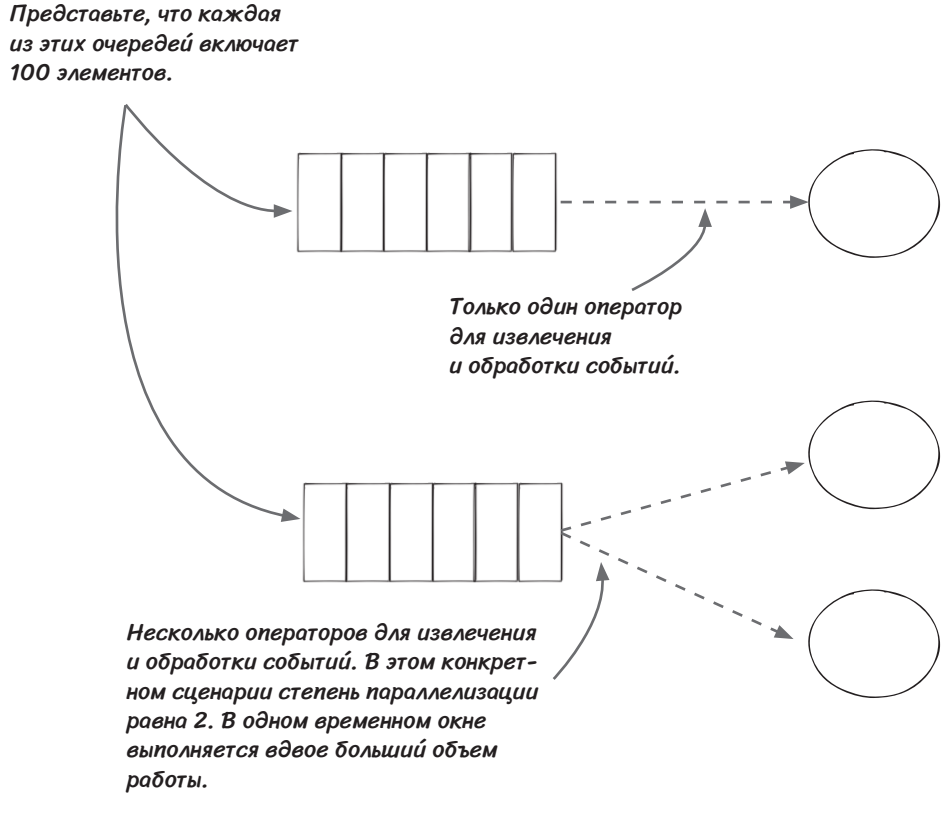


Новые концепции: параллелизм важен

Параллелизация часто применяется в компьютерных системах. Ее идея заключается в том, что задачу, отнимающую много времени, часто удается разбить на меньшие подзадачи, которые могут выполняться параллельно. Тогда другие компьютеры могут работать над задачей совместно, что приведет к существенному сокращению времени выполнения.

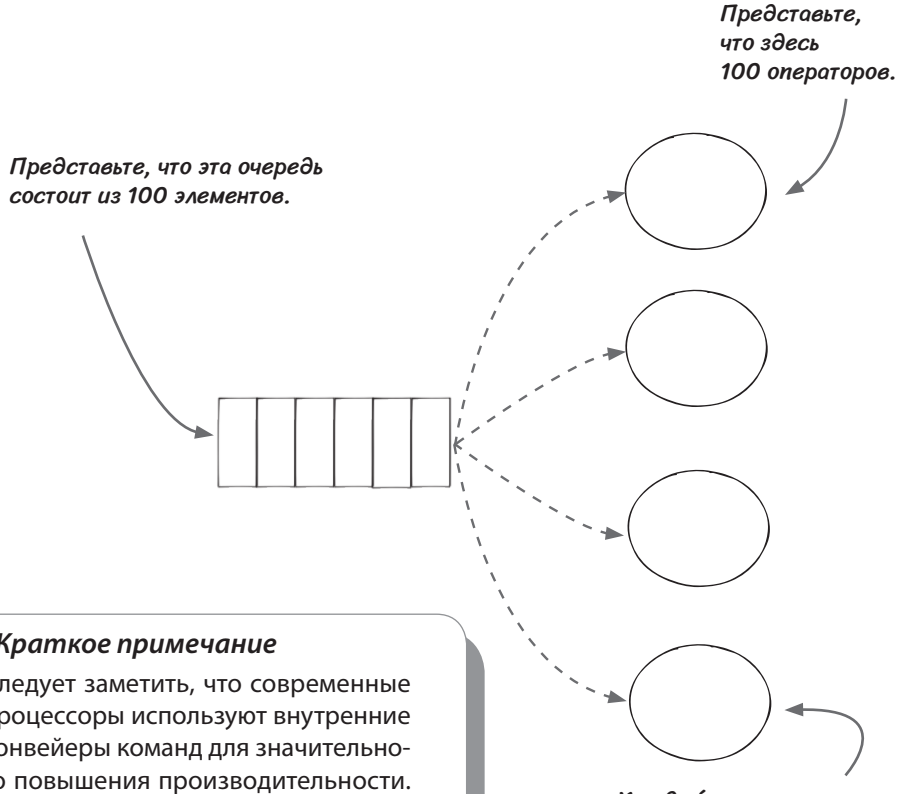
Почему это важно

Возьмем в качестве примера стриминговое задание из предыдущей главы. Если в очереди ожидают обработки 100 событий, один счетчик автомобилей должен обработать их все одно за другим. В реальном мире в стриминговую систему могут ежесекундно поступать миллионы событий. Последовательная обработка этих событий во многих случаях неприемлема, и параллелизация критична для решения крупномасштабных задач.



Новые концепции: параллелизм данных

Для решения задачи подсчета на одном компьютере не хватает скорости. К счастью, у начальника под рукой есть несколько компьютеров — да и в каком центре обработки данных их нет? Будет разумно поручать разные события разным компьютерам, чтобы все они работали над вычислениями параллельно. Тем самым вы обработаете данные всех автомобилей за один шаг вместо 100 шагов. Другими словами, производительность повышается в 100 раз. Если данных становится больше, распределение вычислений по нескольким компьютерам вместо одного более мощного компьютера позволит быстрее решить задачу. Такой подход называется *горизонтальным масштабированием*.

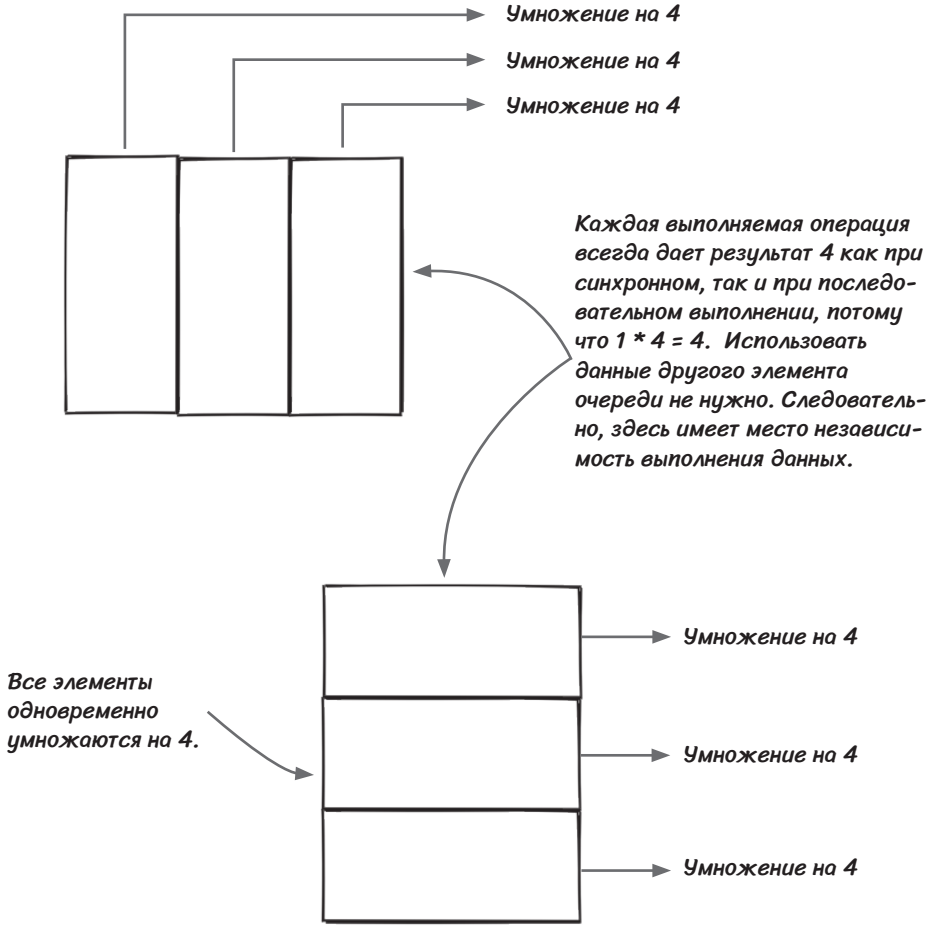


> ***Краткое примечание***
>
> Следует заметить, что современные процессоры используют внутренние конвейеры команд для значительного повышения производительности. В данном случае (и в остальных главах книги) мы ограничимся простыми вычислениями и опустим этот вариант оптимизации, говоря о параллелизации.

Новые концепции: независимость выполнения данных

Произнесите фразу «*независимость выполнения данных*» вслух и подумайте, что бы она значила. Замысловатый термин, но он не настолько сложен, как может показаться.

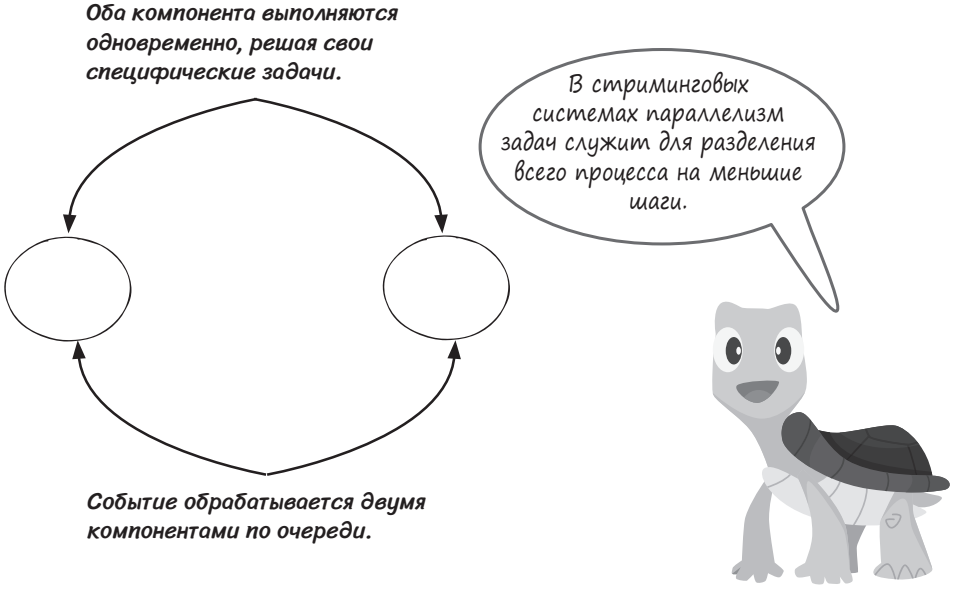
Независимость выполнения данных в контексте стриминга означает, что конечный результат остается неизменным независимо от порядка вычислений, выполняемых с элементами данных. Например, в случае умножения каждого элемента очереди на 4 результат останется постоянным, будут ли элементы обрабатываться одновременно либо поочередно. Независимость позволяет применить параллелизм данных.



Новые концепции: параллелизм задач

Параллелизм данных критичен для многих систем больших данных, а также для распределенных систем вообще, потому что он позволяет разработчикам решать задачи более эффективно, используя большее количество компьютеров. Наряду с параллелизмом данных существует другой тип параллелизации: *параллелизм задач*, также называемый параллелизмом функций. В отличие от параллелизма данных, который подразумевает выполнение одной задачи с разными данными, параллелизм задач ориентирован на выполнение разных задач с одними данными.

Чтобы понять суть параллелизма задач, можно взглянуть на стриминговое задание, которое мы рассматривали в главе 2. Компоненты блока чтения данных с датчика и счетчика автомобилей продолжают выполняться для обработки входящих событий. Когда компонент счетчика автомобилей обрабатывает событие (выполняет подсчет), компонент блока чтения данных с датчика одновременно получает другое, новое событие. Иначе говоря, две разные задачи выполняются одновременно. Это означает, что событие генерируется блоком чтения данных, а затем обрабатывается компонентом счетчика автомобилей.



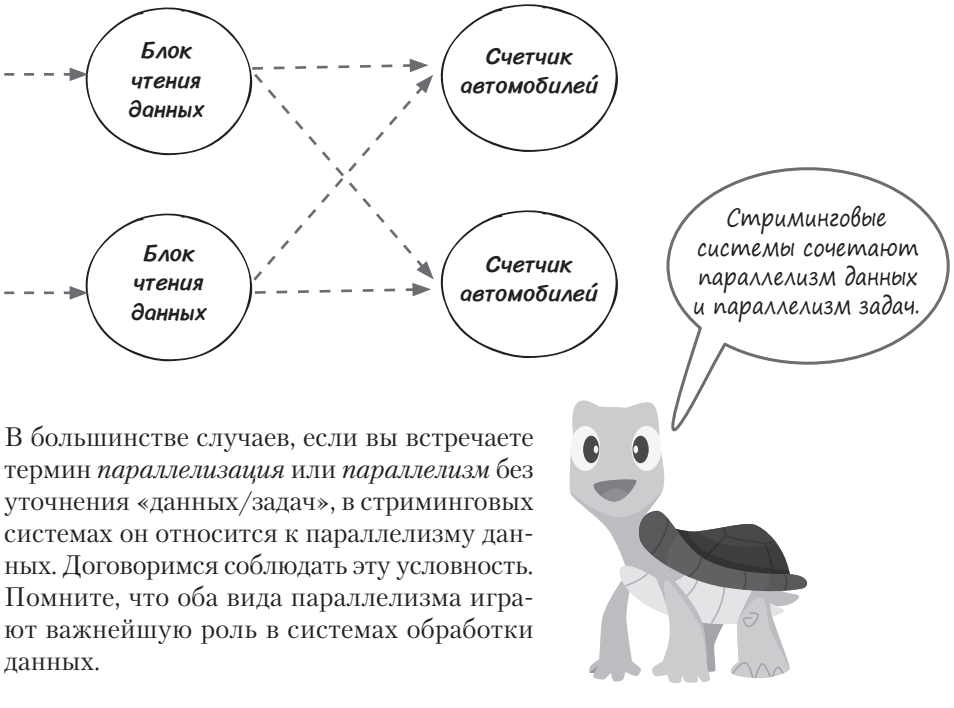
Параллелизм данных и параллелизм задач

Повторим:

- Параллелизм задач означает, что одна задача выполняется с разными наборами событий одновременно.
- Параллелизм задач означает, что разные задачи выполняются одновременно.

Параллелизм данных широко применяется в распределенных системах для достижения горизонтального масштабирования. В таких системах относительно легко повысить степень параллелизации, подключая дополнительные компьютеры. С другой стороны, при параллелизме данных обычно требуется делить существующие процессы вручную на несколько фаз для повышения степени параллелизации.

Стриминговые системы сочетают параллелизм данных с параллелизмом задач. В стриминговой системе под параллелизмом данных понимается создание нескольких экземпляров каждого компонента, а под параллелизмом задач — разбиение всего процесса на разные компоненты для решения задачи. В предыдущей главе мы применили метод параллелизма задач и разбили всю систему на два компонента. В этой главе вы узнаете, как применять метод параллелизма данных и создать несколько экземпляров каждого компонента.



В большинстве случаев, если вы встречаете термин *параллелизация* или *параллелизм* без уточнения «данных/задач», в стриминговых системах он относится к параллелизму данных. Договоримся соблюдать эту условность. Помните, что оба вида параллелизма играют важнейшую роль в системах обработки данных.

Параллелизм и многозадачность

В чем отличия?

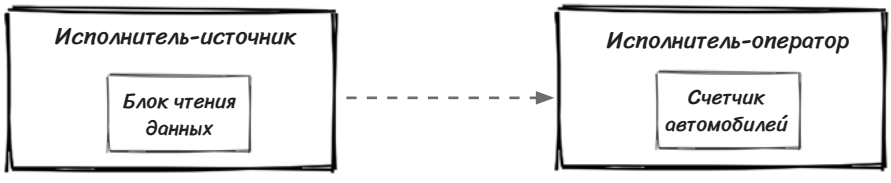
Этот абзац легко бы мог вызвать ожесточенные технические споры — пожалуй, так же легко, как абзац, оправдывающий использование табуляции вместо пробелов. В ходе планирования книги этот вопрос поднимался несколько раз. Как правило, наши встречи всегда завершались тем, что мы спрашивали себя, какой же термин использовать.

Параллелизм — термин, который мы решили использовать для объяснения того, как изменять стриминговые задания для улучшения производительности и масштабирования. Если говорить точнее, в контексте этой книги термин «параллелизм» относится к количеству экземпляров конкретного компонента. Также можно сказать, что параллелизм — количество экземпляров, выполняемых для завершения одной и той же задачи. С другой стороны, *многозадачность* — более общий термин, которым обозначается одновременное выполнение двух и более операций.

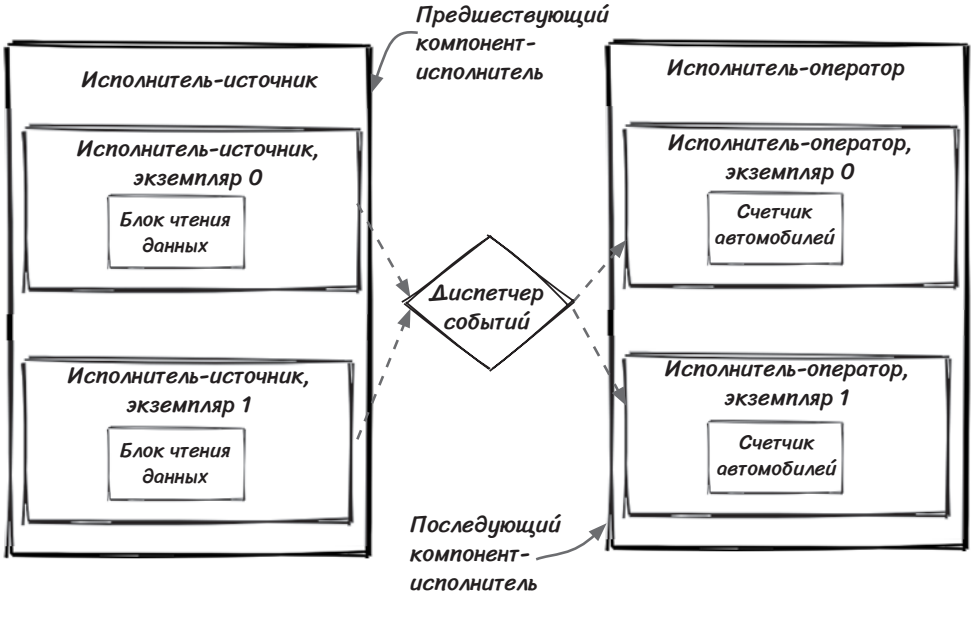
Следует заметить, что для выполнения разных задач в нашем стриминговом фреймворке используются потоки выполнения, но в реальных стриминговых заданиях обычно где-то работают несколько физических машин для поддержки задания. В этом случае можно использовать термин «параллельные вычисления». Некоторые читатели могут усомниться, насколько хорошо термин «параллелизация» подходит для кода, выполняемого на одной машине. Это еще один вопрос, который мы задавали себе. Правильно ли писать об этом? Мы решили не затрагивать этот вопрос. В конце концов, эта книга написана для того, чтобы вы хорошо разбирались в стриминге. Просто знайте, что параллелизация является важнейшим компонентом стриминговых систем, а для вас важно уверенно понимать концепции и различия.

Параллелизация задания

Самое время рассмотреть состояние последнего стримингового задания. Задание для анализа трафика состоит из двух компонентов: блока чтения данных с датчика и счетчика автомобилей. Напомним схему задания на следующей диаграмме.

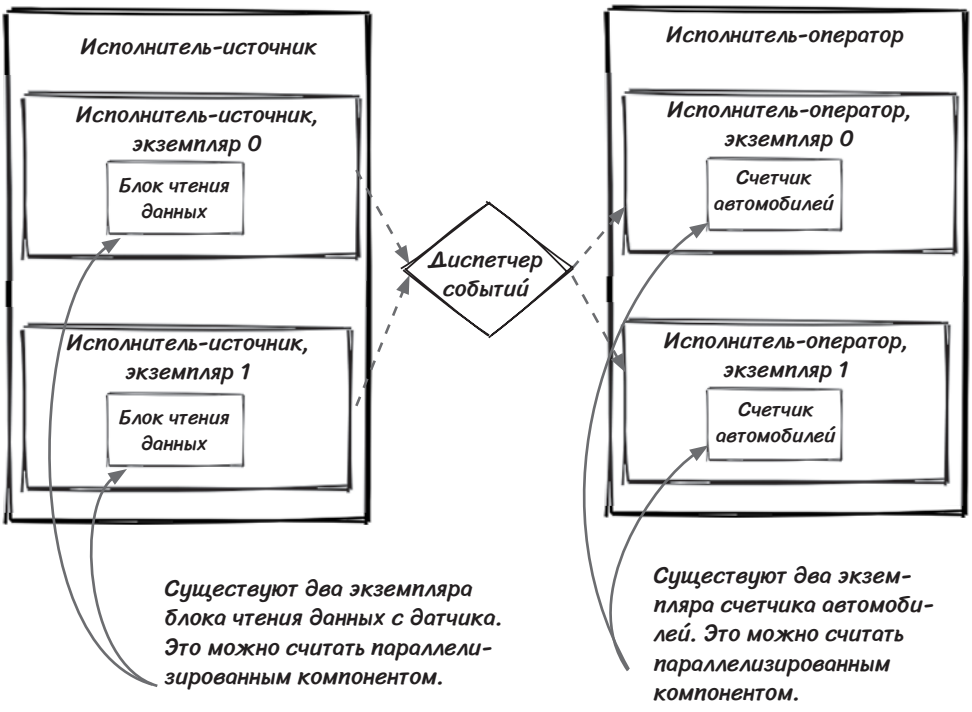


Эта реализация работала в предыдущей главе. Сейчас же мы добавим в схему новый компонент, который мы решили назвать *диспетчером событий* (event dispatcher). Он позволяет перенаправлять данные между разными экземплярами параллелизированного компонента. С компонентом `eventDispatcher` структура задания из главы 2 выглядит, как показано ниже. Следующая диаграмма обобщает результат чтения этой главы и повторения действий по построению задания. К концу главы вы добавите два экземпляра каждого компонента и будете понимать, как система будет принимать решения об отправке данных каждому экземпляру.



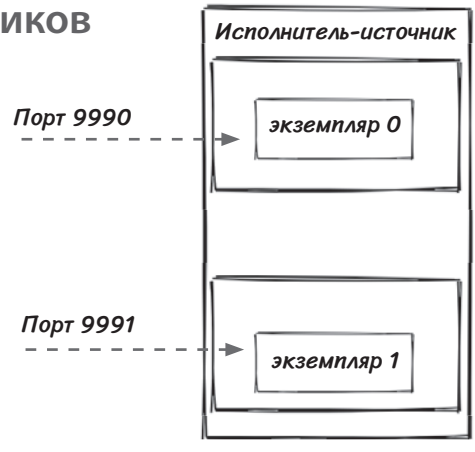
Параллелизация компонентов

На следующей диаграмме изображена конечная цель параллелизации компонентов в стриминговом задании. Диспетчер событий позволяет распределить нагрузку между последующими экземплярами.



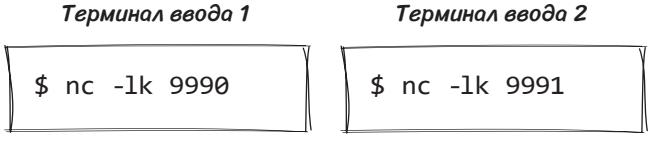
Параллелизация источников

Сначала мы собираемся параллелизировать только источники данных в стриминговом задании: два вместо одного. Для моделирования параллелизированного источника это новое задание должно прослушивать два разных порта для получения входных данных — порты 9990 и 9991. Мы обновили ядро для поддержки параллелизма, а изменения в коде задания достаточно просты:




```
Stream bridgeStream = job.addSource(
  new SensorReader("sensor-reader", 2, 9990)
);
```

Чтобы выполнить задание, необходимо сначала создать два терминала ввода и выполнить команду с разными портами:



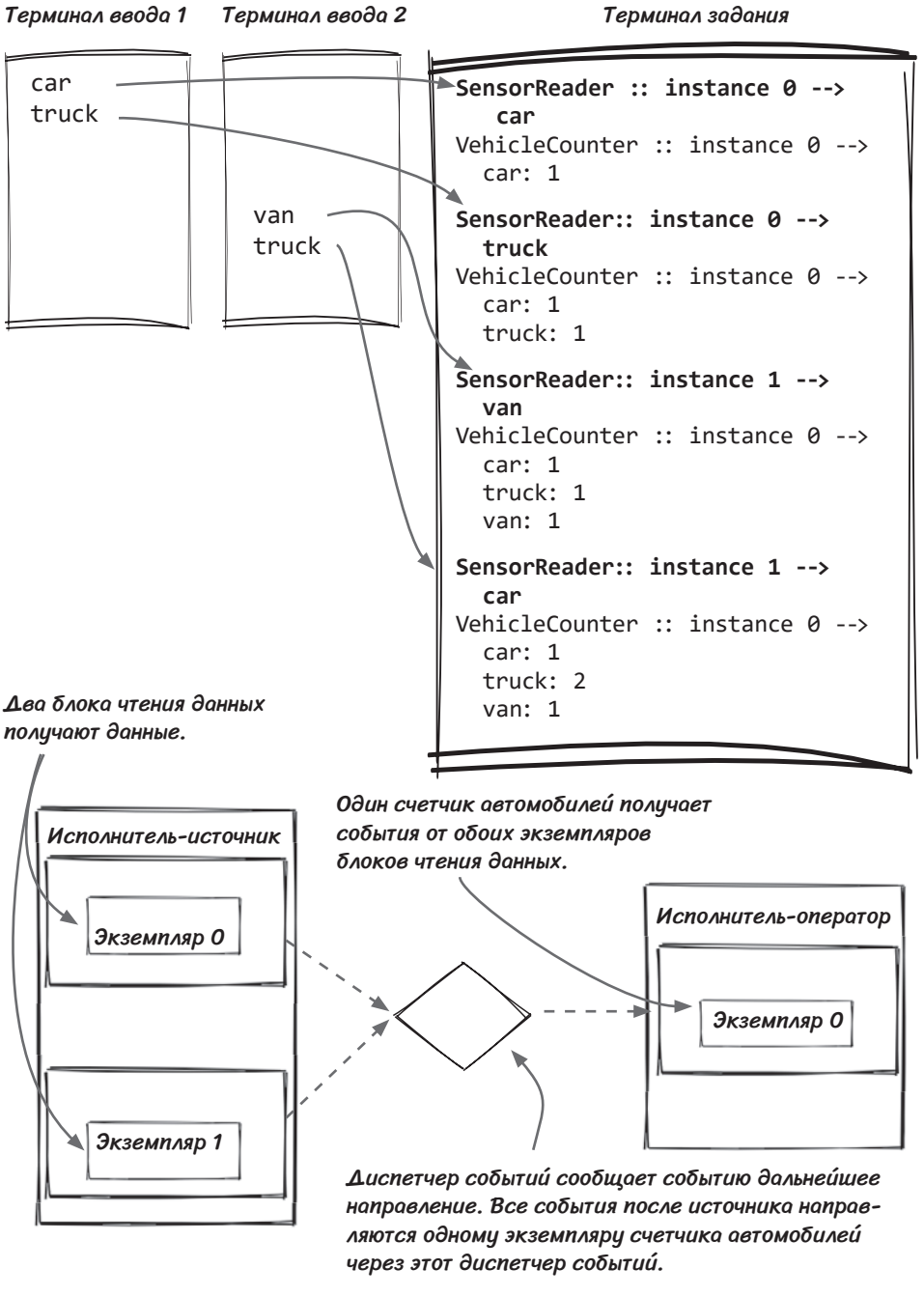
После этого код компилируется и выполняется в отдельном терминале *задания*:

```
$ mvn package
$ java -cp target/gss.jar \
  com.streamwork.ch03.job.ParallelizedVehicleCountJob1
```

К этому моменту у вас должны быть открыты три терминала для выполнения задания: терминал ввода 1, терминал ввода 2 и терминал задания. В терминалах ввода 1 и 2 вводятся события, которые будут извлекаться стриминговым заданием. Пример вывода приведен на следующей странице.

> ***Немного о сетях***
>
> Из-за ограничений сетевых технологий прослушивание одного порта не может осуществляться более чем одним процессом, нитью выполнения и т. д. Так как в нашем учебном примере два источника работают на одной машине, приходится запускать дополнительный экземпляр источника на другом порту.

Результат выполнения



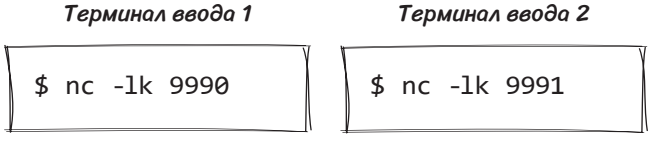
Параллелизация операторов

Запуск нового задания

А теперь параллелизируем оператор `VehicleCounter`:

```
bridgeStream.applyOperator(
  new VehicleCounter("vehicle-counter", 2));
```

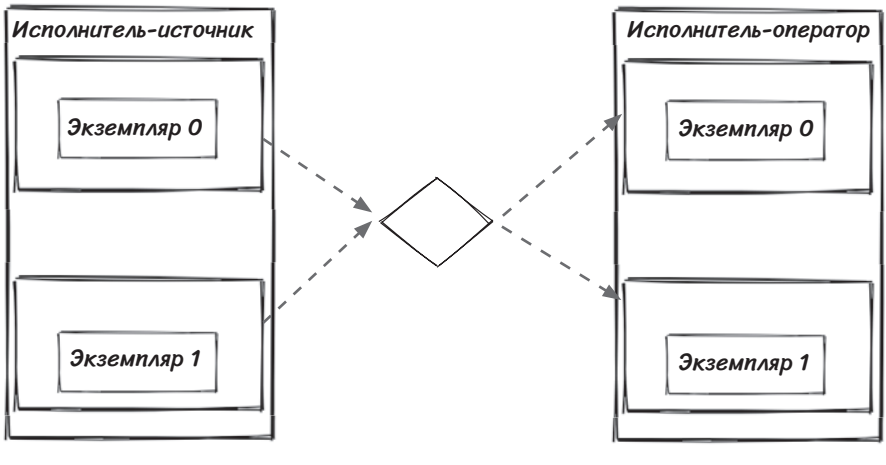
Помните, что мы используем два параллелизированных источника, поэтому необходимо использовать те же команды `netcat`, как ранее, в двух разных терминалах. Напомним, что каждая команда приказывает Netcat прослушивать подключения на порту, заданном в команде.



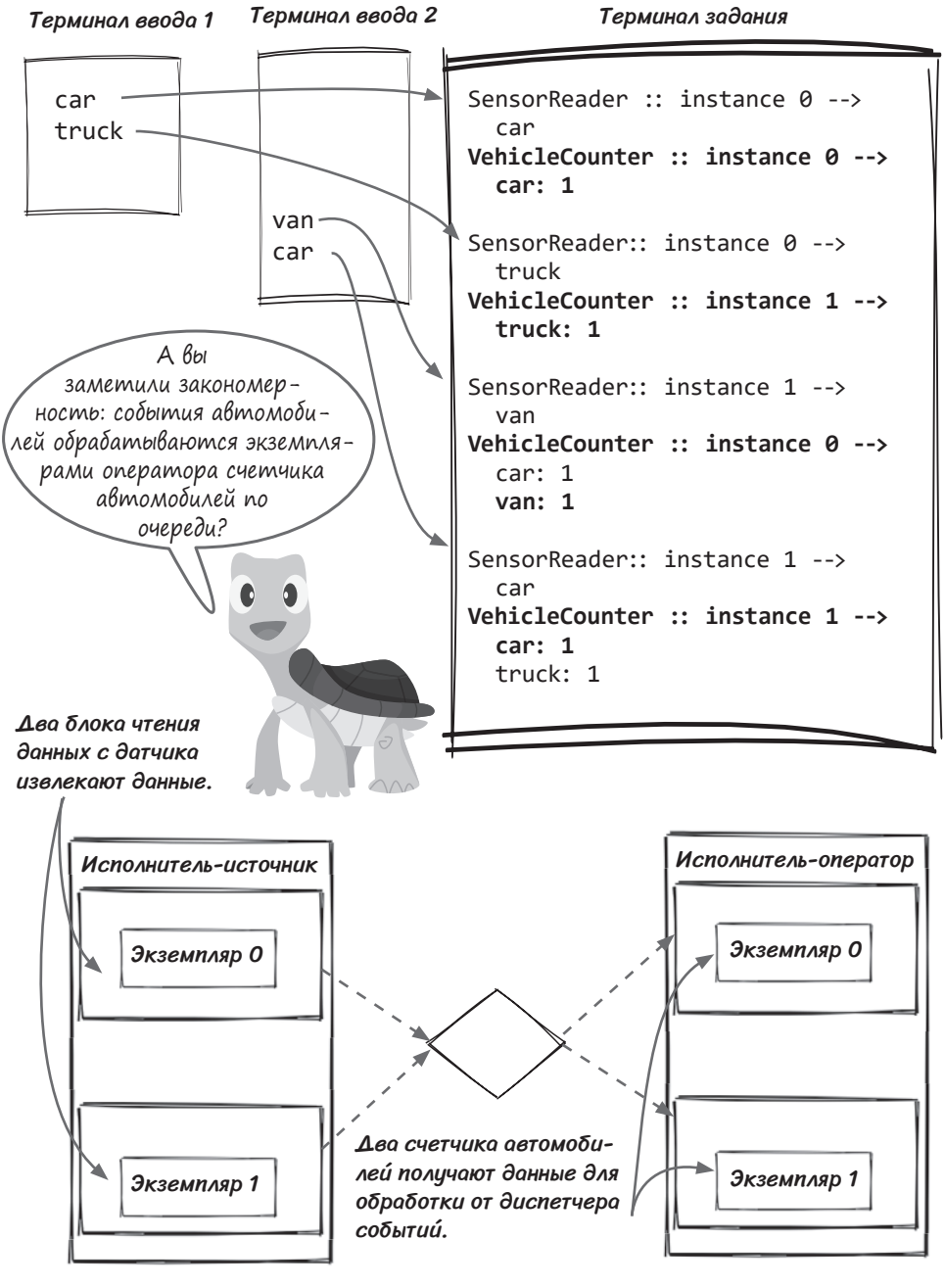
После этого код компилируется и выполняется в отдельном, третьем терминале *задания*:

```
$ mvn package
$ java -cp gss.jar \
  com.streamwork.ch03.job.ParallelizedVehicleCountJob2
```

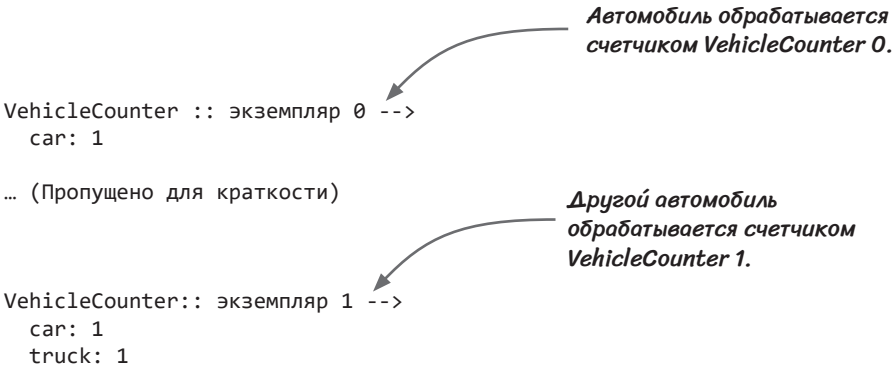
Выполняемое задание содержит два источника и два оператора, что наглядно представлено на следующей диаграмме. Результат выполнения будет показан далее.



Результат выполнения



События и экземпляры



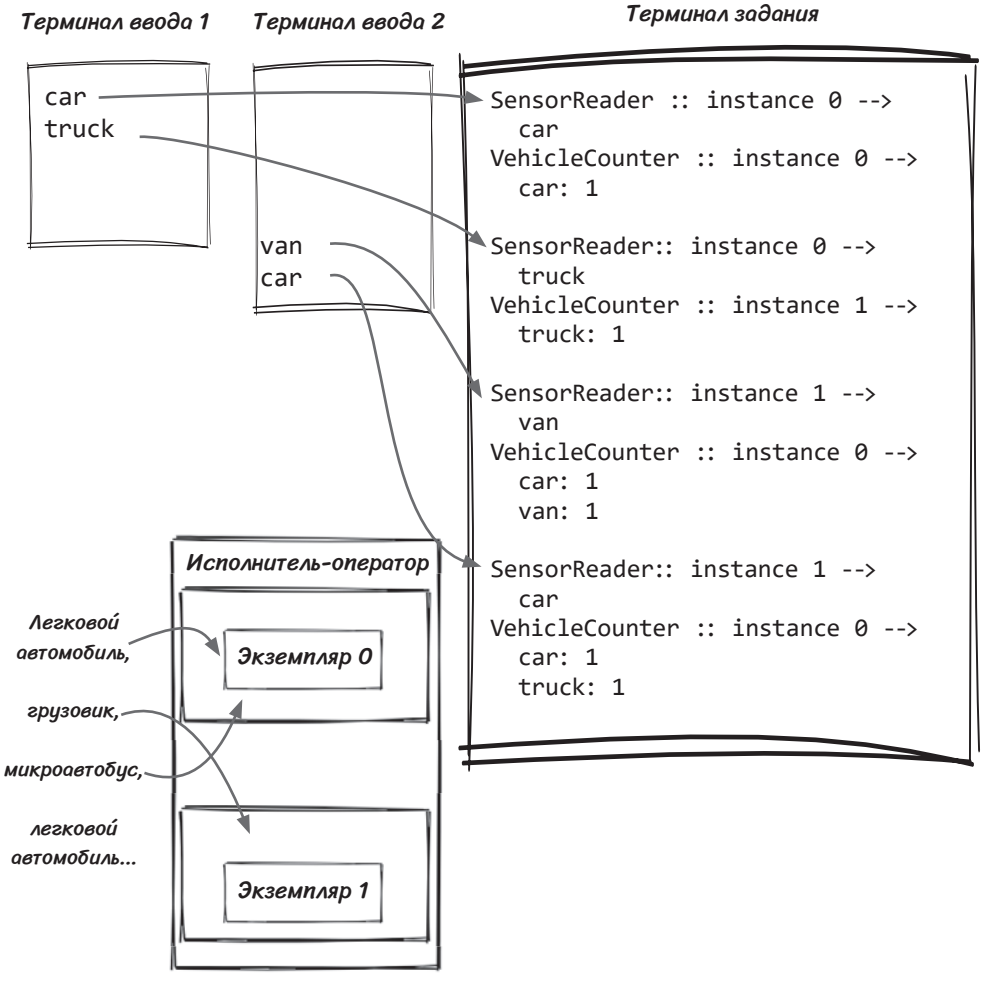
Внимательно присмотревшись к результатам экземпляров счетчика автомобилей, вы увидите, что счетчики получают разные события. В зависимости от того, как настроена система для реализации такого поведения, это может быть нежелательно для стримингового задания. Позднее мы представим концепцию *группировки событий*, которая поможет понять поведение и возможности для улучшения системы. А пока достаточно знать, что любой автомобиль может быть обработан любым из двух экземпляров пункта оплаты.

Еще одна важная концепция, которую необходимо понимать, — *упорядочение событий*. У событий в системе существует определенный порядок — в конце концов, все они обычно находятся в очередях. Как узнать, будет ли одно событие обработано раньше другого? В общем случае действуют два правила.

- Внутри экземпляра порядок обработки *гарантированно* соответствует исходному порядку (порядку во входящей очереди).
- Между экземплярами порядок обработки *не гарантирован*. Может оказаться, что более позднее событие будет обработано и/или завершено раньше другого события, которое поступило до него, если два события обрабатываются разными экземплярами.

Далее приводится более конкретный пример.

Упорядочение событий



Рассмотрим четыре события автомобилей, которые были введены в терминалах ввода. Первый и третий автомобили — легковой (`car`) и микроавтобус (`van`); они передаются экземпляру 0 `VehicleCounter`, тогда как второе и четвертое событие — грузовик (`truck`) и легковой автомобиль (`car`) — передаются экземпляру 1 `VehicleCounter`.

В ядре Streamwork два экземпляра оператора выполняются независимо. Ядра Streaming обычно гарантируют, что первый и третий автомобили обрабатываются в порядке поступления, поскольку это происходит в одном экземпляре. Тем не менее нет гарантии, что первый автомобиль (`car`) будет обработан до второго (`truck`) или второй автомобиль (`truck`) будет обработан до третьего (`van`), потому что процессы двух операторов не зависят друг от друга.

Группировка событий

До сих пор параллелизированное стриминговое задание содержало экземпляры (instances) счетчика автомобилей, которые получали события в результате *случайной* (вернее, псевдослучайной) диспетчеризации экземплярами счетчика автомобилей.

Псевдослучайная маршрутизация приемлема во многих случаях, но иногда предпочтительнее передавать события определенному последующему экземпляру. Эта концепция перенаправления событий экземплярам называется *группировкой событий*. Термин «группировка» может показаться несколько странным, поэтому попробуем пояснить: все события делятся на группы, и каждая группа закрепляется за определенным экземпляром для обработки. Существуют разные стратегии группировки событий. Наиболее часто применяются следующие:

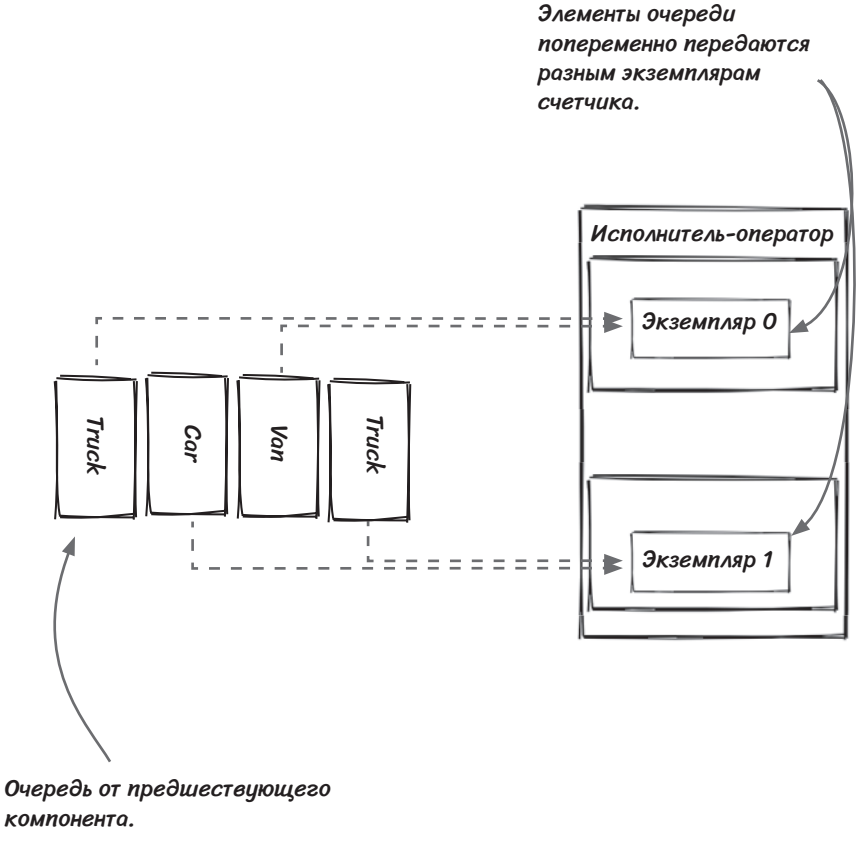
- *Случайная группировка* — события псевдослучайным образом распределяются между последующими компонентами.
- *Группировка по значениям полей* — события предсказуемым образом передаются одним и тем же последующим экземплярам на основании значений конкретных полей события.

Как правило, группировка событий представляет собой функциональность, интегрированную в стриминговые фреймворки для дальнейшего использования разработчиками. Сейчас мы подробнее разберем, как работают эти две стратегии группировки.

Случайная группировка

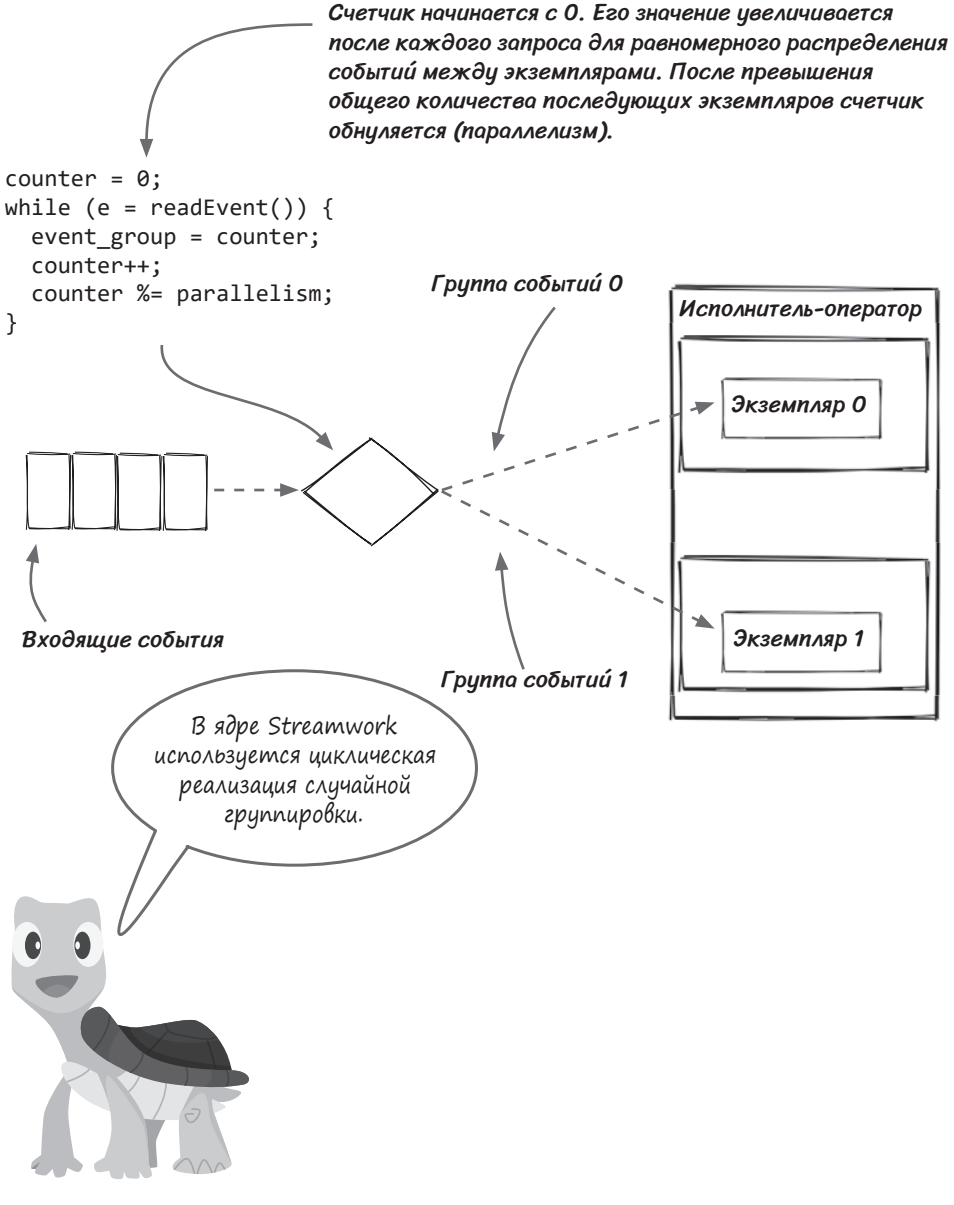
Если коротко, *случайная группировка* представляет собой случайное распределение элементов данных от компонента к последующему оператору. Она позволяет относительно равномерно распределять нагрузку между последующими операторами.

Для реализации случайной группировки во многих фреймворках используется циклический алгоритм. В этой стратегии группировки последующие экземпляры (то есть входящие очереди) выбираются равными частями и в циклическом порядке. По сравнению со случайной группировкой на основании случайных чисел распределение становится более равномерным, а вычисления — более эффективными. Реализация представлена на следующей диаграмме. Обратите внимание: на диаграмме два грузовика (`truck`) подсчитываются двумя разными экземплярами `VehicleCounter`.



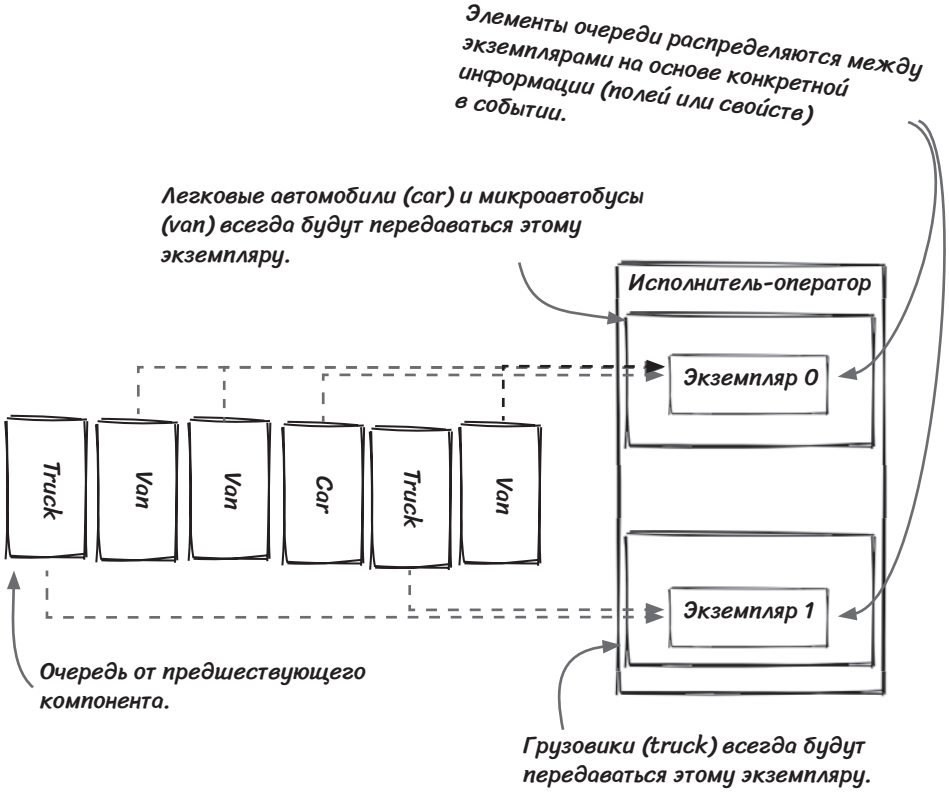
Случайная группировка: внутренний механизм

Чтобы убедиться, что события равномерно распределяются между экземплярами, во многих стриминговых системах используется циклический алгоритм для выбора следующего получателя события.



Группировка по значениям полей

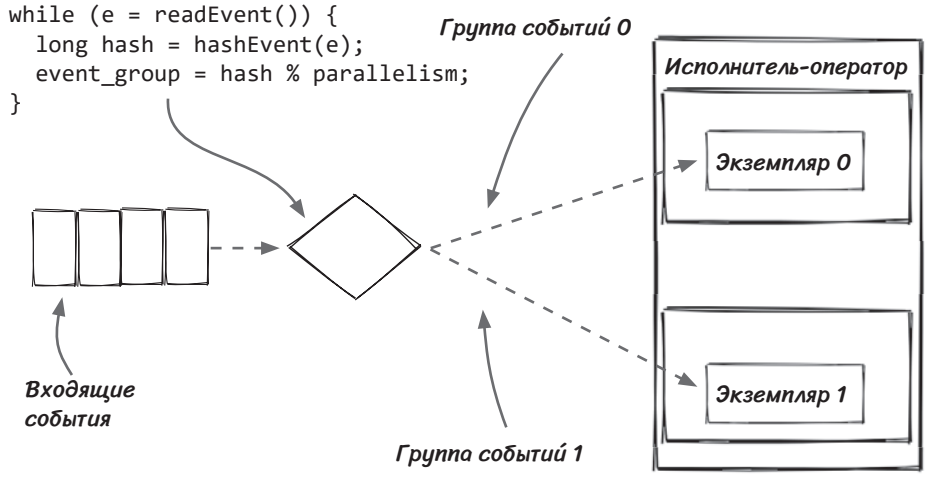
Случайная группировка хорошо работает во многих сценариях использования. Тем не менее если вам потребуется механизм прогнозируемой отправки элементов, случайная группировка не подойдет. Для прогнозируемого шаблона распределения, когда он потребуется для ваших целей, следует использовать *группировку по значениям полей*. В этой стратегии решение о том, куда передавать данные, принимается на основе полей передаваемого элемента событий (обычно эти поля выбирает разработчик). Во многих сценариях группировка по значениям полей также называется *группировкой по ключу*.



В стриминговом задании этой главы мы берем каждое событие автомобиля, въезжающего на мост, и отправляем их счетчику автомобилей 0 или счетчику автомобилей 1 в зависимости от типа автомобиля, так что события одного типа автомобилей всегда будут передаваться одному экземпляру счетчика. При таком подходе отдельные типы автомобилей подсчитываются на уровне экземпляров (и с большей точностью).

Группировка по значениям полей: внутренний механизм

Чтобы события одного типа всегда назначались в одну группу (которая передается одному экземпляру), применяется *хеширование*. Это часто используемый вычислительный метод, при котором большой диапазон значений (например, строки) переносится на меньший набор значений (например, целые числа).



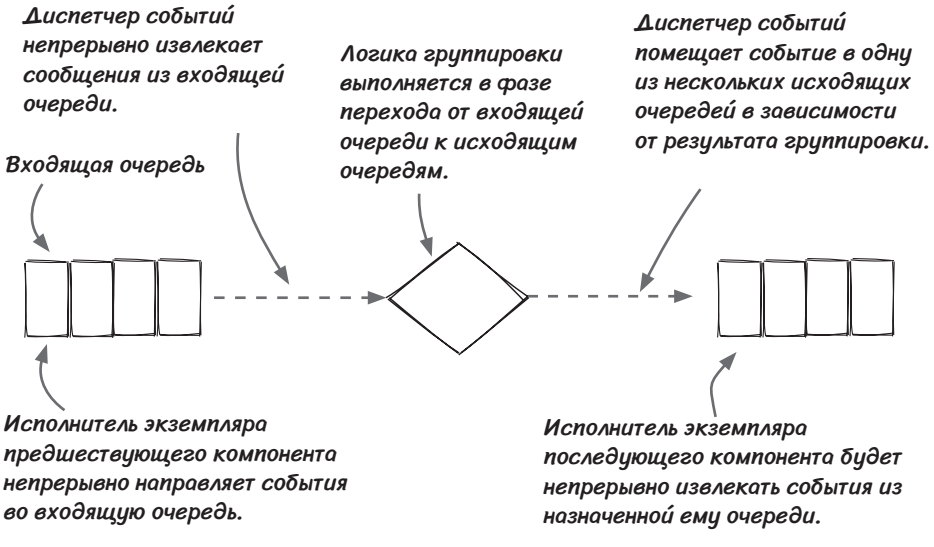
Важнейшая особенность хеширования заключается в том, что для постоянных входных данных результат всегда будет неизменным. После получения результата хеширования (обычно это большое число, называемое *ключом*, например 98216) выполняются следующие вычисления:

```
key % parallelism
```

Делит key на parallelism и возвращает остаток, на основании которого решается, какому экземпляру последующего оператора должно быть назначено событие. С двумя экземплярами событие с ключом 98216 будет направлено во входящую очередь экземпляра 0, потому что остаток от деления 98216/2 равен 0.

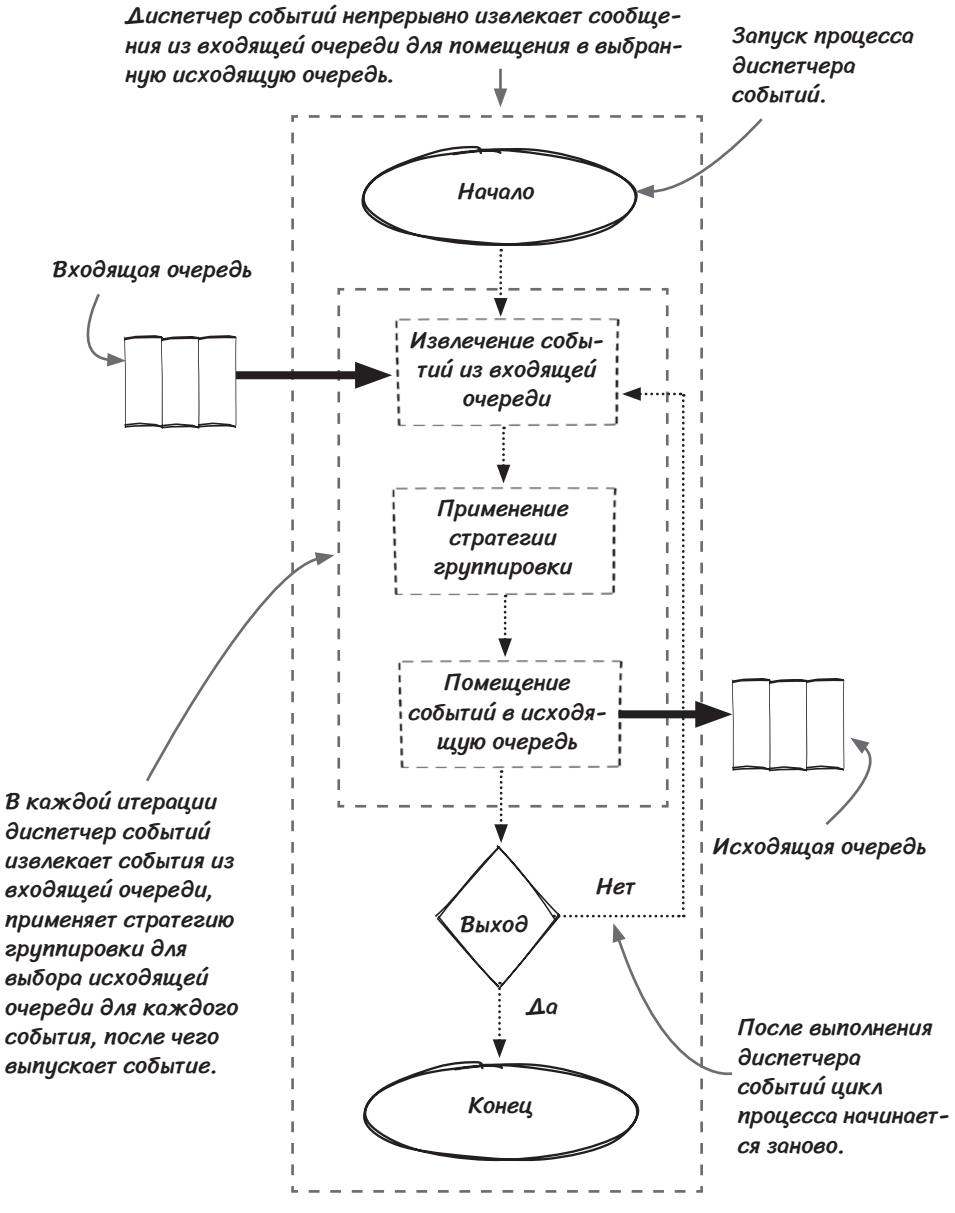
Выполнение группировки событий

Диспетчер событий — часть стриминговой системы, расположенная между исполнителями компонентов и выполняющая процесс группировки событий. Диспетчер непрерывно извлекает события из входящей очереди и помещает их в исходящие очереди на основании ключа, возвращаемого стратегией группировки. Следует помнить, что в каждой стриминговой системе используются свои решения. Это описание применимо только к нашему фреймворку Streamwork.



Заглянуть в ядро: диспетчер событий

Диспетчер событий отвечает за получение событий от предшествующего компонента, применение стратегии группировки и отправку событий последующему компоненту.



Применение группировки по значениям полей в задании

С применением группировки по значениям полей в задании намного проще поддерживать сводный счетчик для разных типов автомобилей, так как каждый тип всегда будет направляться одному экземпляру. С использованием Streamwork API группировку по значениям полей применить очень просто:

```
bridgeStream.applyOperator(
  new VehicleCounter("vehicle-counter", 2, new FieldsGrouping() )
);
```

Применение группировки по значениям полей

Единственное, что необходимо сделать, — добавить дополнительную переменную при вызове функции `applyOperator()`. Все остальное сделает ядро Streamwork.

Помните, что стриминговые фреймворки помогают сосредоточиться на бизнес-логике, не отвлекаясь на реализацию ядер. Разные ядра могут использовать разные способы применения группировки по значениям полей. Обычно стоит поискать функцию с именем вида `groupBy()` или `{операция}ByKey()`.

Чтобы выполнить пример кода, действуйте как раньше. Сначала выполните приведенные ниже команды и откройте два терминала ввода, чтобы вводить типы автомобилей. Затем компилируйте и выполняйте код в отдельном, третьем терминале задания:

Терминал ввода 1

```
$ nc -lk 9990
```

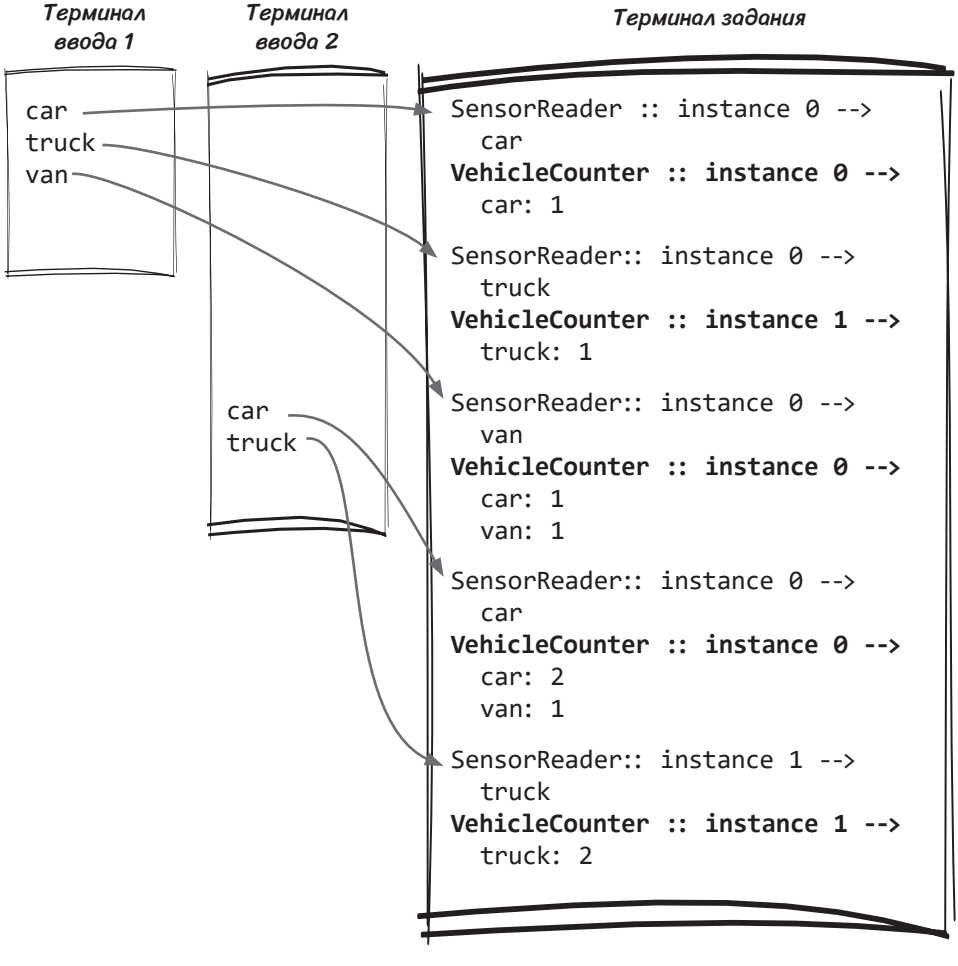
Терминал ввода 2

```
$ nc -lk 9991
```

```
$ mvn package
$ java -cp target/gss.jar \
  com.streamwork.ch03.job.ParallelizedVehicleCountJob3
```

Упорядочение событий

Результат, который выводится при выполнении этих команд на терминале задания, выглядит примерно так:



Сравнение поведения группировок

Сравним результаты выполнения заданий со случайной группировкой и группировкой по значениям полей и рассмотрим различия в поведении с одними и теми же входными данными. На самом деле неважно, от какого терминала поступил ввод, поэтому мы объединим их. Заметите ли вы, чем отличается результат в этих двух заданиях?

Ввод задания: `car truck van car truck …`

Результат выполнения задания со случайной группировкой

```
SensorReader :: instance 0 ->
  car
VehicleCounter :: instance 0 ->
  car: 1

SensorReader:: instance 0 ->
  truck
VehicleCounter :: instance 1 ->
  truck: 1

SensorReader:: instance 0 ->
  van
VehicleCounter :: instance 0 ->
  car: 1
  van: 1

SensorReader:: instance 0 ->
  car
VehicleCounter :: instance 1 ->
  car: 1
  truck: 1

SensorReader:: instance 1 ->
  truck
VehicleCounter:: instance 0 ->
  car: 1
  truck: 1
  van: 1
```

Результат выполнения задания с группировкой по значениям полей

```
SensorReader :: instance 0 ->
  car
VehicleCounter :: instance 0 ->
  car: 1

SensorReader:: instance 0 ->
  truck
VehicleCounter :: instance 1 ->
  truck: 1

SensorReader:: instance 0 ->
  van
VehicleCounter :: instance 0 ->
  car: 1
  van: 1

SensorReader:: instance 0 ->
  Car
VehicleCounter :: instance 0 ->
  car: 2
  van: 1

SensorReader:: instance 1 ->
  truck
VehicleCounter:: instance 1 ->
  truck: 2
```

Итоги

В этой главе были представлены основные концепции масштабирования стриминговых заданий. Масштабируемость — одна из главных сложностей для всех распределенных систем, а параллелизация — основной метод масштабирования. Вы узнали, как осуществлять параллелизацию компонентов в потоковом задании, а также познакомились с взаимосвязанными концепциями параллелизма данных и заданий. Если в стриминговых системах термин *параллелизм* используется без уточнения «*данных/задач*», обычно он относится к *параллелизму данных*.

При параллелизации компонентов также необходимо знать, как управлять маршрутизацией событий или прогнозировать маршрутизацию в стратегиях группировки событий для получения ожидаемых результатов. Для достижения прогнозируемости можно воспользоваться случайной группировкой или группировкой по значениям полей. Также мы рассмотрели стриминговое ядро Streamwork; вы увидели, как параллелизация и группировка событий реализуются с концептуальной точки зрения. Этот материал подготовит вас к следующим главам и использованию реальных стриминговых систем.

Параллелизм и группировка событий чрезвычайно важны, потому что они помогают решать один из важнейших вопросов распределенных систем: вопрос пропускной способности. Если в стриминговой системе обнаруживается узкое место, его можно горизонтально масштабировать, повышая степень параллелизма, и система сможет обрабатывать события быстрее.

Упражнения

1. Почему так важна параллелизация?
2. Придумайте другую стратегию группировки. Попробуйте реализовать ее в Streamwork.
3. В нашем примере для группировки по значениям полей используется результат хеширования строки. Попробуйте реализовать другую группировку по значениям полей, в которой вместо хеширования используется первый символ. Какими преимуществами и недостатками обладает новая стратегия группировки?

Граф потока | 4

В этой главе

- ✓ Разветвление потока по выходу.
- ✓ Объединение потока по входу.
- ✓ Графы и DAG (направленные ациклические графы).

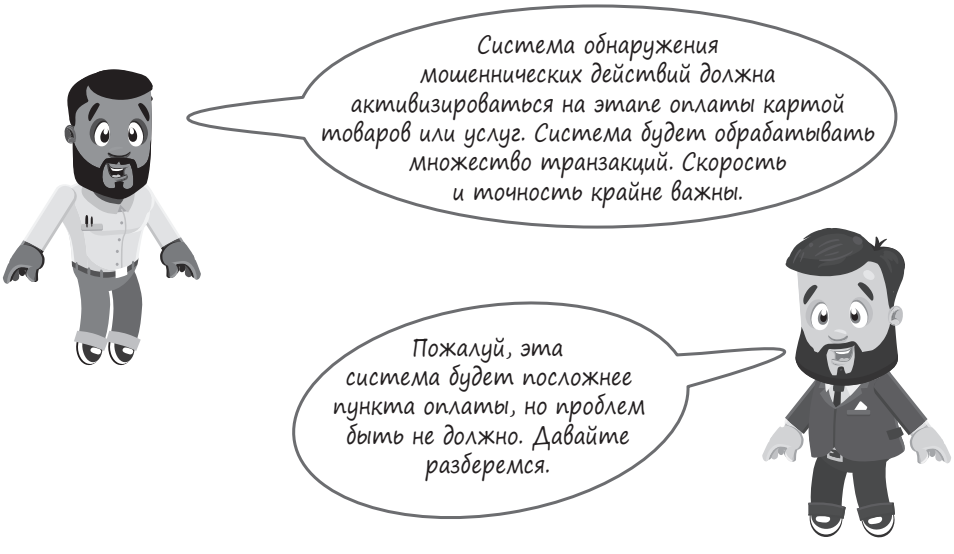
«Плохие программисты думают о коде. Хорошие программисты думают о структурах данных и их взаимосвязях».

Линус Торвальдс

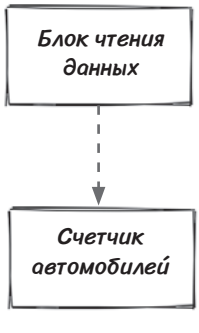
В предыдущих главах ЭйДжей построил стриминговое задание, а затем масштабировал его. Такое решение хорошо работает для контроля движения по мосту. Тем не менее структура задания относительно проста, так как задание в основном представляет собой список операторов. В этой главе вы узнаете, как строить более сложные стриминговые системы для решения других прикладных задач.

Система обнаружения мошеннических действий с кредитными картами

Система подсчета автомобилей, построенная ЭйДжеем, произвела впечатление на Сида, и он думает о новых задачах, которые можно решить с помощью технологий потоковой обработки. Сейчас его больше всего интересует задача обнаружения попыток мошенничества. Но его беспокоит одно обстоятельство: новая система будет более сложной и требует очень низкой задержки. Можно ли решить задачу при помощи стриминга?



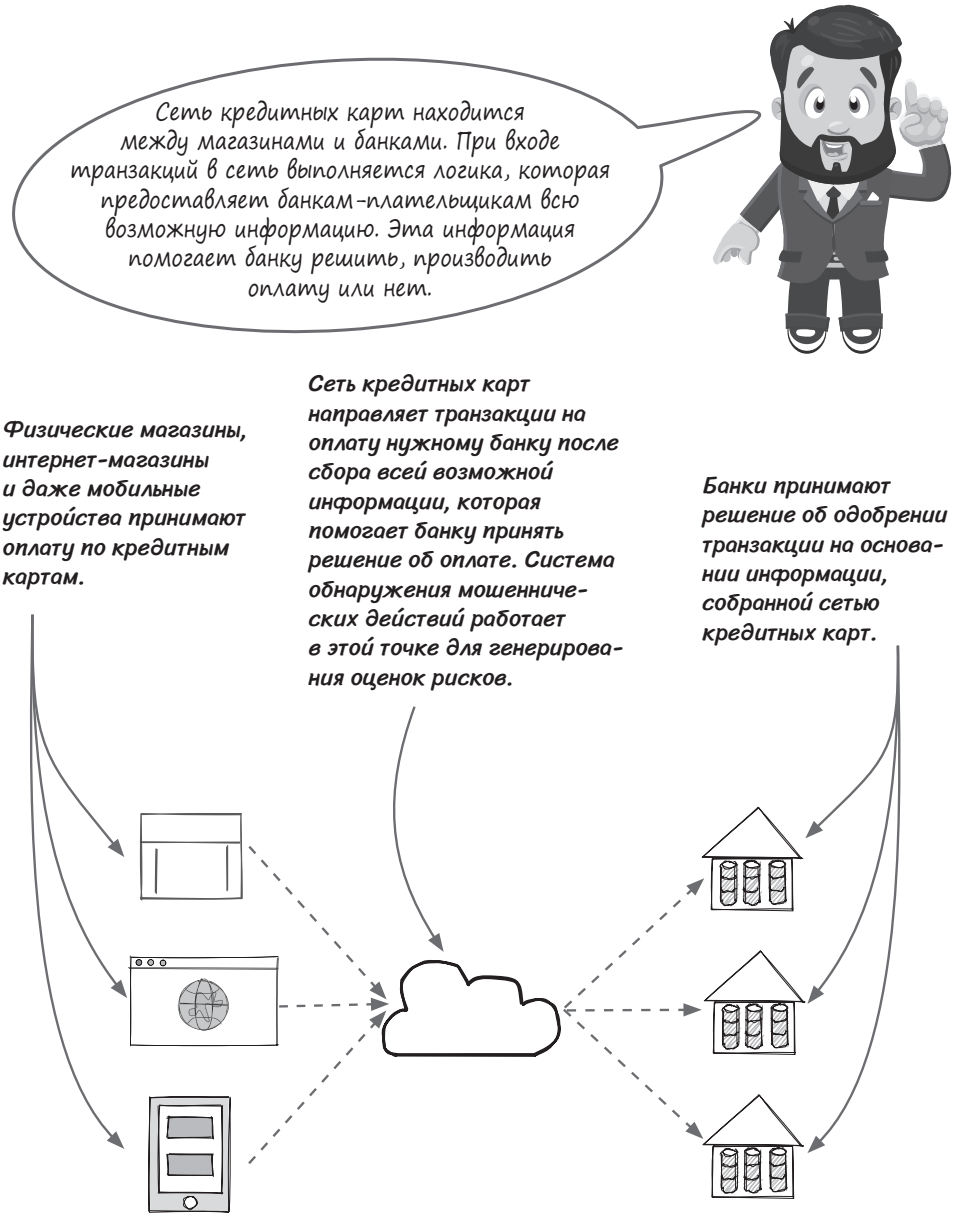
Стриминговое задание, с которым мы работали в двух предыдущих главах, ограничено по функциональности. Каждый элемент данных, входящий в задание, должен проходить оба компонента в фиксированном порядке: блок чтения данных с датчика, а затем счетчик автомобилей. В этой схеме отсутствует условная маршрутизация данных для граничных случаев или ошибок, которые могут возникать в стриминговых системах. Путь элементов данных в этом стриминговом задании можно наглядно представить в виде прямой линии.



Подробнее о системе обнаружения мошеннических действий с кредитными картами

В этой главе мы построим систему обнаружения мошеннических действий с кредитными картами. Она будет сложнее системы пункта оплаты, которой мы занимались ранее.

Процедура обнаружения мошеннических действий

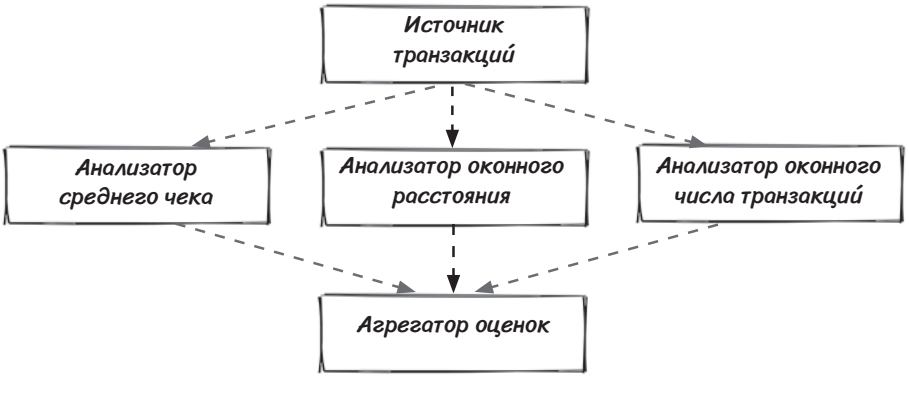


Потоковая обработка не всегда прямолинейна

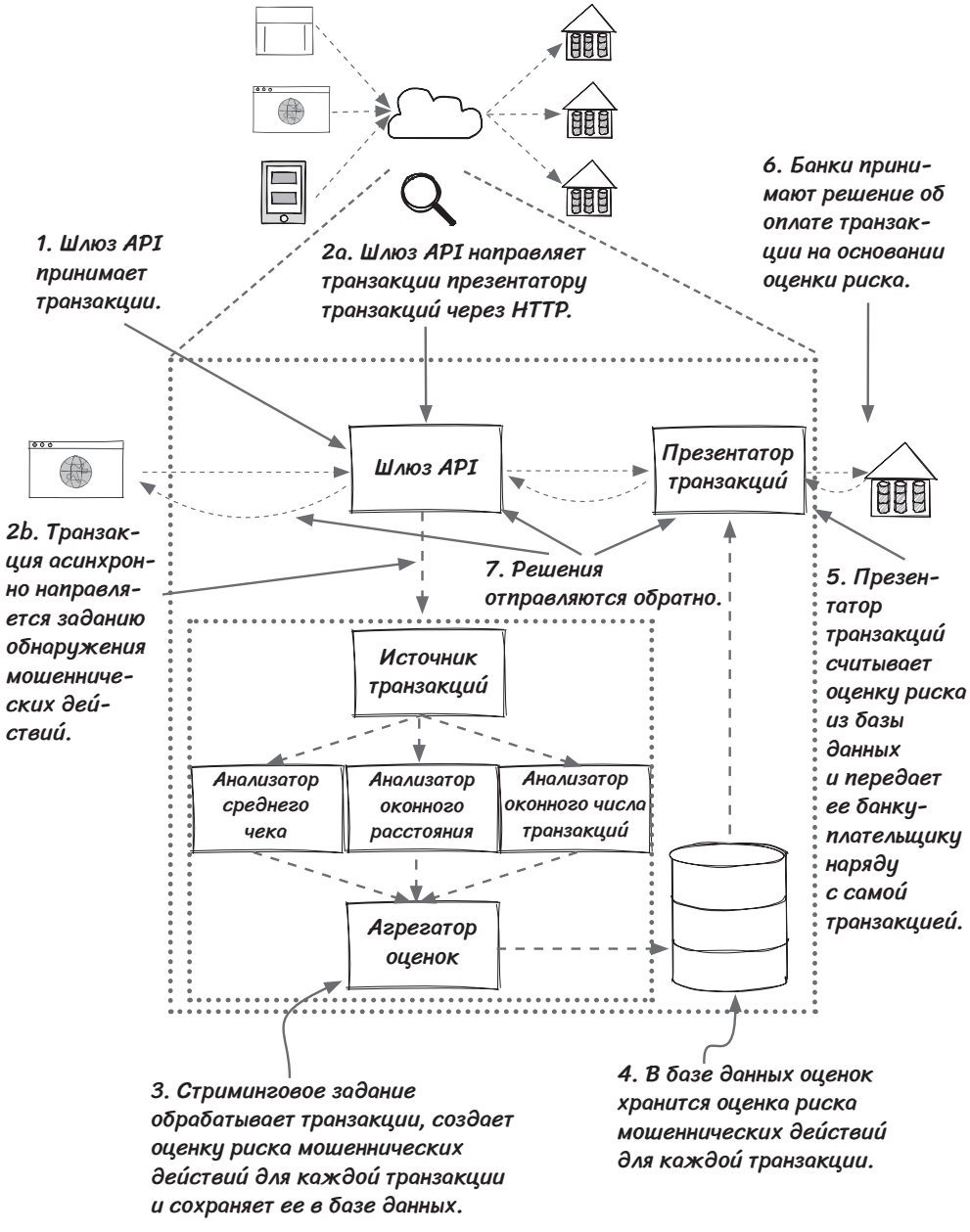
Систему можно построить по образцу системы пунктов оплаты. Сначала компонент — источник транзакций отвечает за получение событий транзакции от внешних систем. Затем анализаторы применяются по очереди, а в события добавляются оценки рисков. Наконец, агрегатор оценок принимает решение.

Такая схема работает, но она не идеальна. В дальнейшем в нее будут добавляться новые анализаторы, список будет расти, а время задержки между конечными точками будет возрастать. Кроме того, большое количество анализаторов затрудняет обслуживание системы.

Также можно построить систему, приведенную на диаграмме ниже. Все три анализатора соединяются с источником транзакции и выполняются независимо. Агрегатор оценок получает от них результаты и объединяет оценки для принятия окончательного решения. В этой системе добавление новых анализаторов не приведет к увеличению задержки между конечными точками.



Механизм работы системы



Подробнее о задании обнаружения мошеннических действий

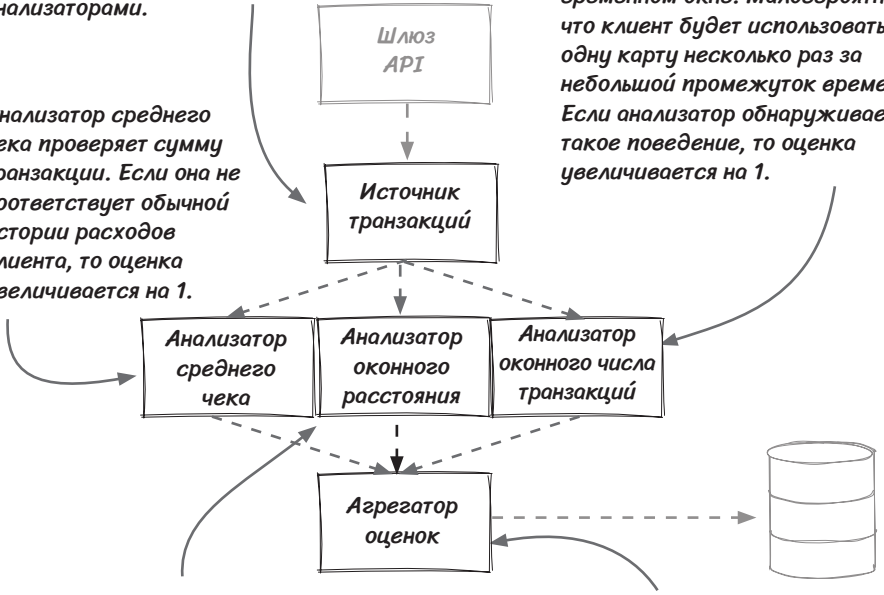
Рассмотрим подробнее задание обнаружения мошеннических действий и функции всех компонентов.

> ***Как распознать подозрительную транзакцию?***
>
> Оценки лежат в диапазоне 0–3. Оценка 0 означает, что анализатор не обнаружил признаков мошенничества, а оценка 3 — что признаки мошенничества обнаружены всеми тремя анализаторами. Каждый анализатор увеличивает оценку на 1 балл. Транзакция считается потенциально мошеннической, если ее оценка составляет 2 и выше.

Источник транзакций извлекает события при поступлении их в шлюз API системы кредитных карт. Он создает 3 разных экземпляра одной транзакции и распределяет их между анализаторами.

Анализатор среднего чека проверяет сумму транзакции. Если она не соответствует обычной истории расходов клиента, то оценка увеличивается на 1.

Анализатор оконного числа транзакций проверяет транзакции с одного счета в заданном временном окне. Маловероятно, что клиент будет использовать одну карту несколько раз за небольшой промежуток времени. Если анализатор обнаруживает такое поведение, то оценка увеличивается на 1.



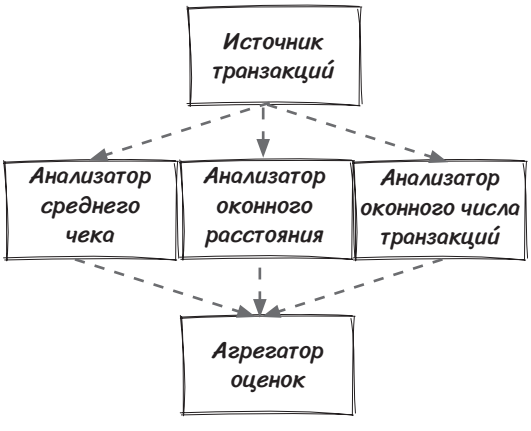
Анализатор оконного расстояния проверяет транзакции с одного счета в заданном временном окне. Маловероятно, что кто-то прокатает карту в одном считывателе, а вскоре в другом, расположенном за 200 километров от первого. Если анализатор обнаруживает такое поведение, то оценка увеличивается на 1.

Агрегатор оценок некоторое время ожидает данные от каждого предшествующего анализатора, после чего вычисляет общую итоговую оценку риска мошеннических действий. По завершении времени ожидания оценка записывается в базу данных.

Новые концепции

В главе 2 вы узнали об активных частях стриминговой системы, источниках данных и операторах, а также связях между ними. Кроме того, мы рассмотрели, как они реализуются внутри ядра. Все это очень важные концепции, которые будут использованы на протяжении всей книги.

В этой главе мы рассмотрим стриминговые задания, имеющие более сложную структуру. Новая диаграмма кажется более сложной по сравнению с прежней линейной диаграммой. Это действительно так, но не стоит беспокоиться.



Прежде чем продолжить, рассмотрим несколько новых концепций, представленных на диаграмме.

- Предшествующие и последующие компоненты.
- Разветвление потока по выходу
- Объединение потока по входу.
- Графы и DAG[1] (направленный ациклический граф).

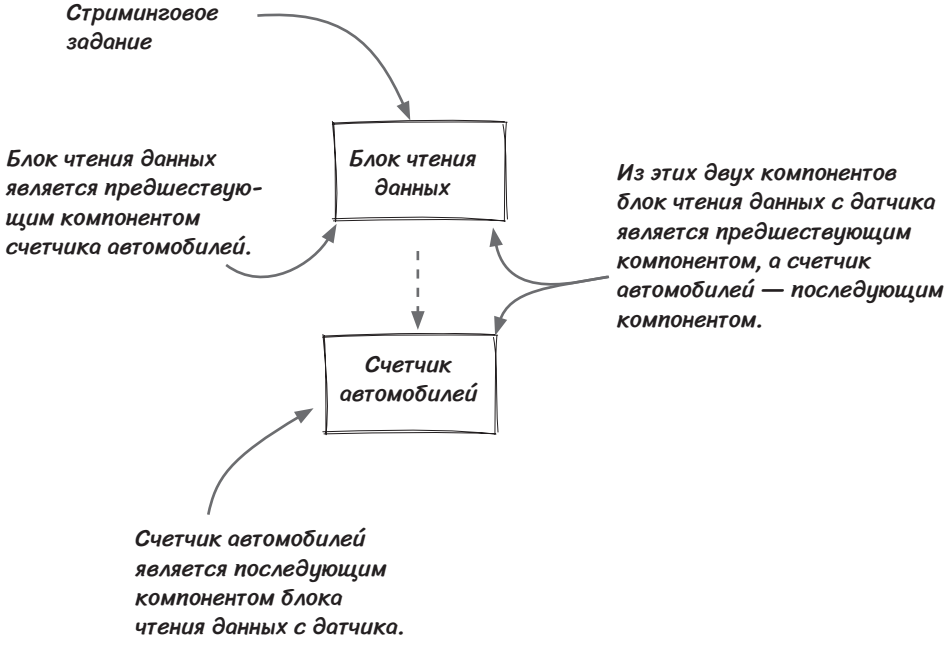
С новыми концепциями можно строить более сложные стриминговые системы для решения более общих задач.

[1] DAG — directed acyclic graph. — *Примеч. ред.*

Предшествующие и последующие компоненты

Начнем с двух новых концепций: *предшествующих* (upstream) и *последующих* (downstream) компонентов. Они довольно просты.

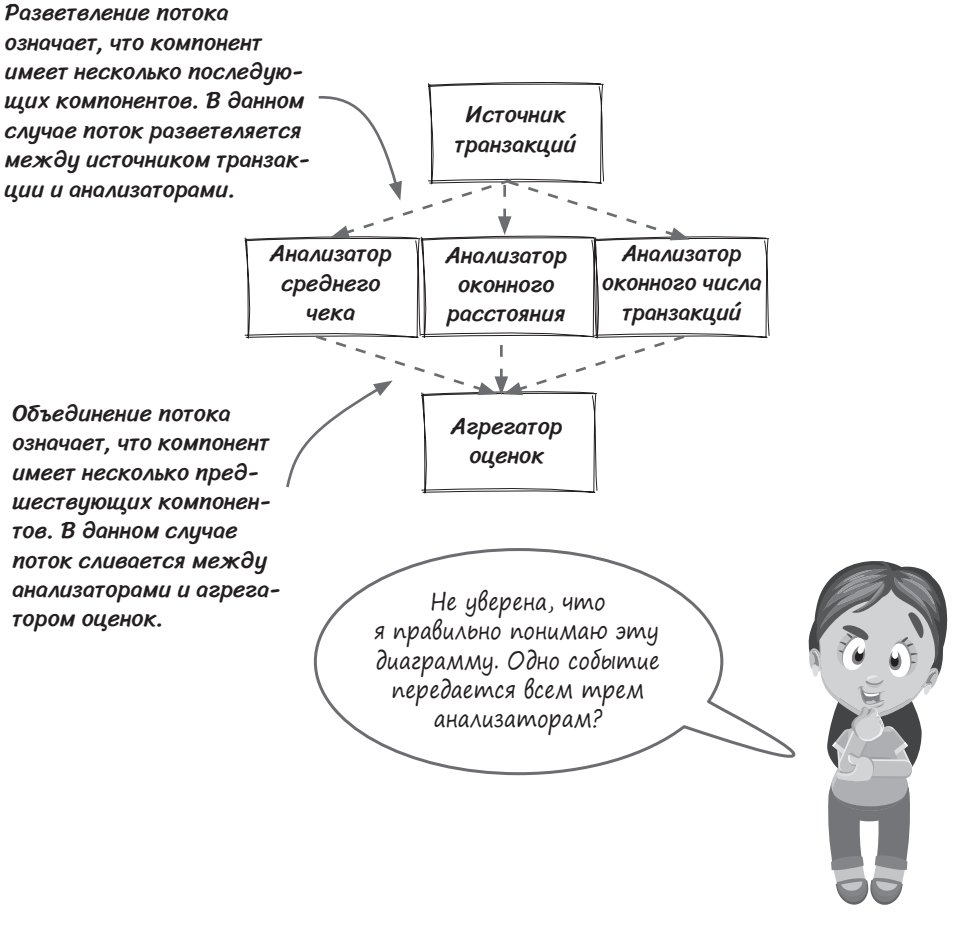
В целом стриминговое задание выглядит как серия событий, проходящих через компоненты. Для каждого компонента другой компонент (или компоненты), находящийся непосредственно перед ним, называется *предшествующим*, а компонент после него называется *последующим*. События проходят от предшествующих компонентов к последующим. Если взглянуть на диаграмму стримингового задания, построенного в предыдущей главе, мы увидим, что события проходят от блока чтения данных с датчика к счетчику автомобилей. Следовательно, блок чтения данных является предшествующим компонентом, а счетчик автомобилей — последующим.



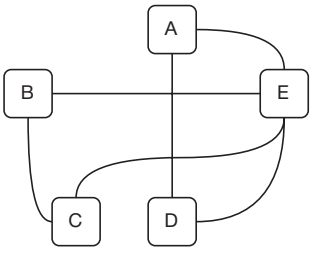
Разветвление и объединение потока

Рассмотрим новую диаграмму, которую предлагает ЭйДжей. В целом она сильно отличается от предыдущего задания. Главное отличие — в том, что у одного компонента может быть несколько предшествующих или последующих компонентов.

Компонент — источник транзакций соединен с тремя последующими компонентами; это называется *разветвлением потока* (stream fan-out). Аналогичным образом агрегатор оценок имеет три предшествующих компонента (также можно сказать, что у трех анализаторов один последующий компонент). Это называется *объединением потока* (stream fan-in).

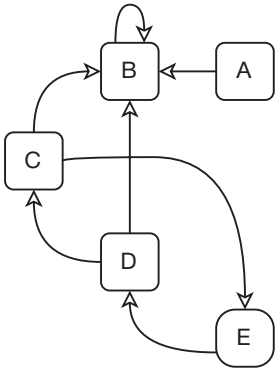


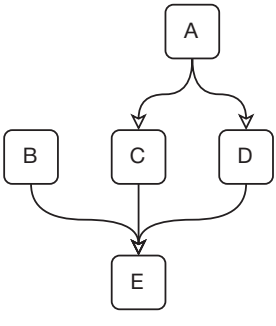
Графы, направленные графы и DAG



Последние три концепции, которые будут рассмотрены в этой главе, — графы, направленные графы и направленные ациклические графы (DAG). *Граф* представляет собой структуру данных, состоящую из *вершин* (узлов) и *ребер* (сторон). Две структуры данных, часто используемые разработчиками, — дерево и список — являются примерами графов.

Если каждое ребро графа имеет определенное направление (от одной вершины к другой), такой граф называется *направленным*. Диаграмма справа — пример направленного графа с пятью вершинами и семью направленными ребрами.





Особую разновидность направленных графов составляют направленные ациклические графы (DAG). DAG представляет собой направленный граф, в котором нет ни одного направленного цикла; это означает, что в таком графе невозможно начать с некоторой вершины и вернуться к ней по направленным ребрам.

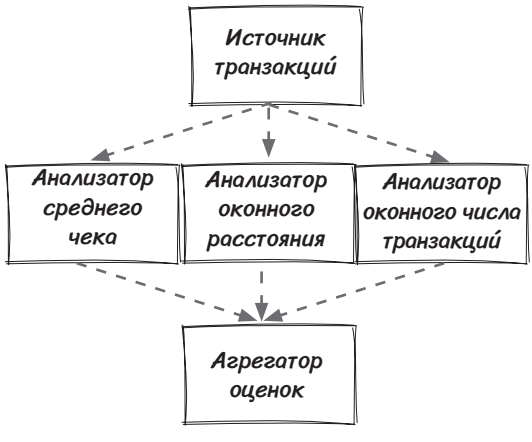
Граф, изображенный на диаграмме слева, является DAG, потому что ни от одной вершины невозможно провести путь, который бы возвращался к ней самой. На диаграмме с направленным графом вершины C, D и E образуют цикл; следовательно, этот граф не является DAG. Обратите внимание: в графе существует еще один цикл, потому что от вершины B отходит ребро, которое возвращается к этой же вершине.

DAG в системах потоковой обработки

Структура данных DAG играет важную роль в компьютерной теории и системах потоковой обработки. Мы не будем вдаваться в математические подробности, но важно знать, что термин DAG часто встречается в области стриминга.

Прохождение событий в системе удобно представлять в виде направленного графа. Цикл в направленном графе означает, что события могут возвращаться и снова обрабатываться тем же компонентом. К таким ситуациям следует подходить очень осторожно из-за дополнительной сложности и рисков. Иногда циклы бывают необходимы, но такие случаи встречаются относительно редко.

В большинстве систем потоковой обработки отсутствуют циклы; следовательно, такие системы могут быть представлены в виде DAG.



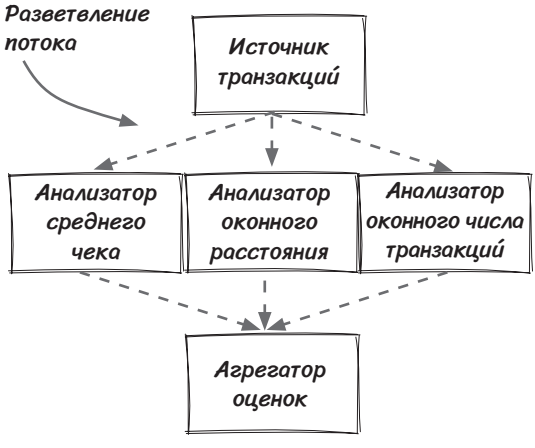
Начиная с этой главы, каждый раз, когда мы будем рисовать диаграмму задания, это будет DAG. Он будет включать только логические компоненты задания без объектов ядра — таких как исполнители и диспетчеры событий (если только они не являются необходимыми) — см. приведенную выше диаграмму. Таким образом, мы можем сосредоточиться на бизнес-логике, не беспокоясь о подробностях уровня ядра. Параллелизм сюда не включен, потому что он не связан с бизнес-логикой.

Все новые концепции на одной странице

В этой главе было представлено несколько новых концепций. Давайте объединим их на одной странице, чтобы было проще понять, как они связаны друг с другом.

Разветвление потока к анализаторам

Пришло время внимательнее присмотреться к нашей системе, начиная с места разветвления потока. В системе обнаружения мошеннических действий разветвление потока происходит между компонентом-источником и операторами-анализаторами. Средства Streamwork API позволяют просто связать поток, поступающий от компонента-источника, с анализаторами. Схема связывания выглядит, как показано ниже.




```
  Job job = new Job();
  Stream transactionOut = job.addSource(new TransactionSource());
  Stream evalResults1 = transactionOut.applyOperator(new
AvgTicketAnalyzer());
  Stream evalResults2 = transactionOut.applyOperator(new
WindowedProximityAnalyzer());
  Stream evalResults3 = transactionOut.applyOperator(new
WindowedTransactionAnalyzer());
```

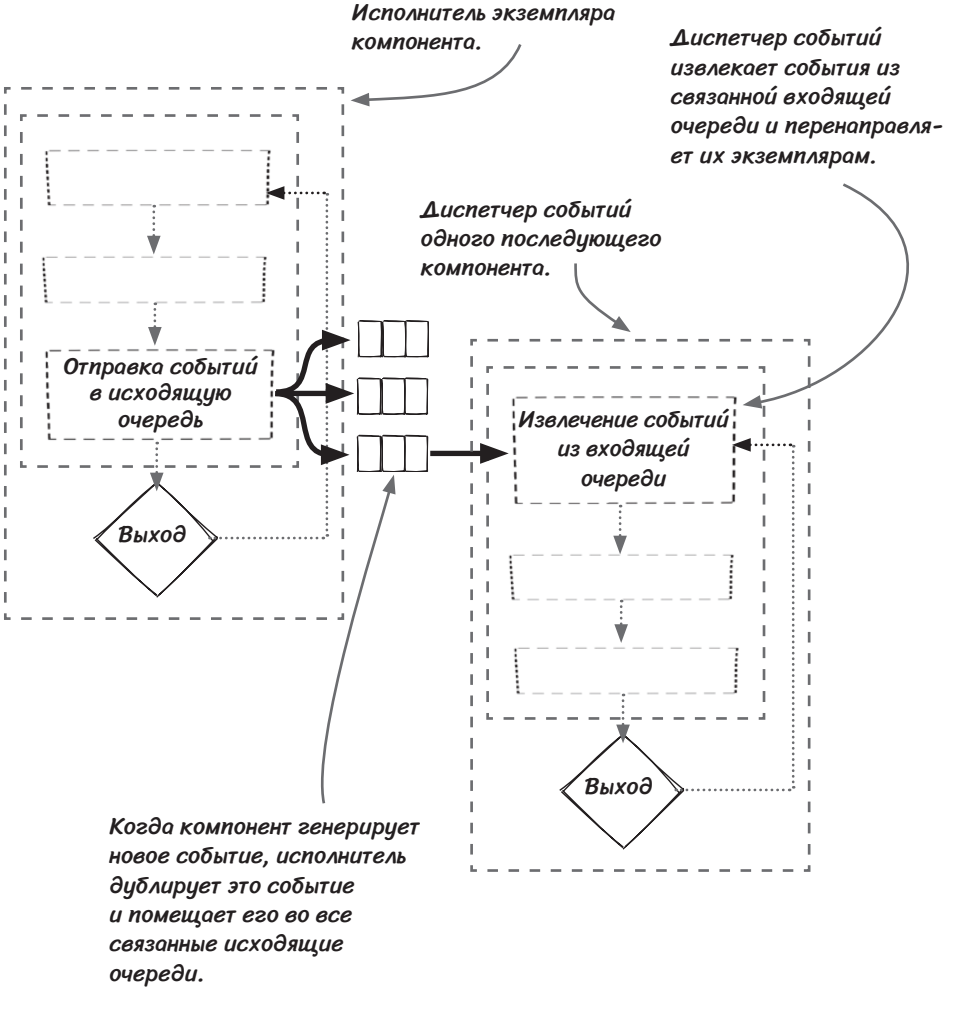
Применение нескольких операторов к одному потоку.

По сути, несколько операторов (в данном случае анализаторов) могут применяться к одному потоку транзакций от компонента-источника. Во время выполнения каждое событие, генерируемое компонентом-источником, будет продублировано трижды и отправлено трем анализаторам.

Разветвление потока — один компонент с несколькими последующими компонентами.

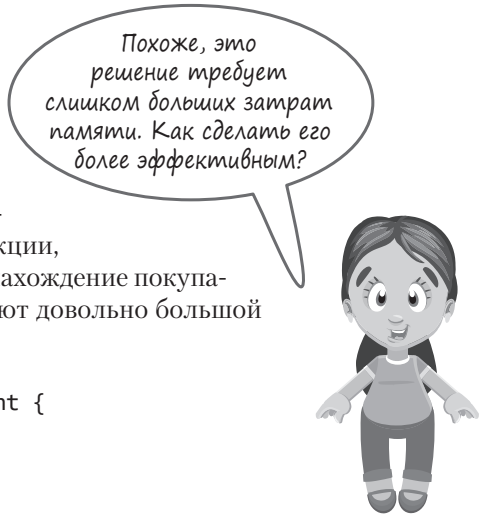
Что происходит внутри ядра

Настоящая работа выполняется внутри ядра. В ядре Streamwork при подключении к потоку нового оператора создается новая очередь между диспетчером событий оператора и исполнителями экземпляров компонента, генерирующего поток. Иначе говоря, один исполнитель экземпляра может отправлять события нескольким исходящим очередям.



Проблема эффективности

Теперь у каждого анализатора имеется своя копия событий транзакции, к которой он может применять свою логику оценки. Тем не менее такое решение не очень эффективно.



Каждое событие является записью транзакции. Оно содержит большой объем информации о транзакции: идентификатор товара, идентификатор транзакции, время транзакции, сумму, счет, категории товаров, местонахождение покупателя и т. д. В результате события имеют довольно большой размер:

```
class TransactionEvent extends Event {
  long transactionId;
  float amount;
  Date transactionTime;
  long merchandiseId;
  long userAccount;
  ……
}
```

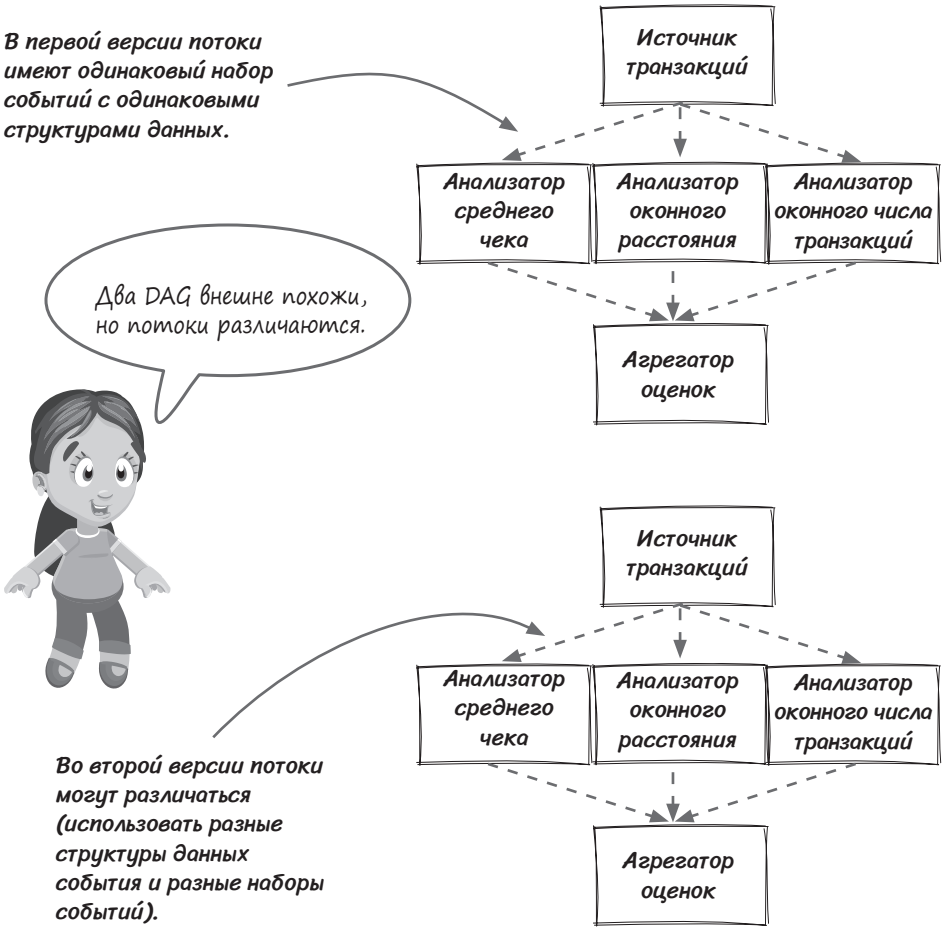
В текущем варианте каждое событие многократно дублируется, поскольку дубликаты направляются в разные очереди. Это позволяет разным анализаторам обрабатывать каждое событие асинхронно. Эти «расширенные» события передаются по сети, загружаются и обрабатываются анализаторами. Кроме того, некоторые анализаторы не собираются (или не могут) обрабатывать некоторые события, но эти события все равно передаются и обрабатываются. В результате память и ресурсы сети используются неэффективно. Схему можно усовершенствовать, что важно при высоком уровне трафика.

Разветвление с несколькими потоками

При разветвлении потока исходящие очереди могут различаться. Слово *различаться* здесь имеет два смысла.

- Сгенерированное событие может направляться в некоторые исходящие очереди и пропускать другие.
- Кроме того, события в разных исходящих очередях, направленных к разным последующим компонентам, могут иметь разные структуры данных.

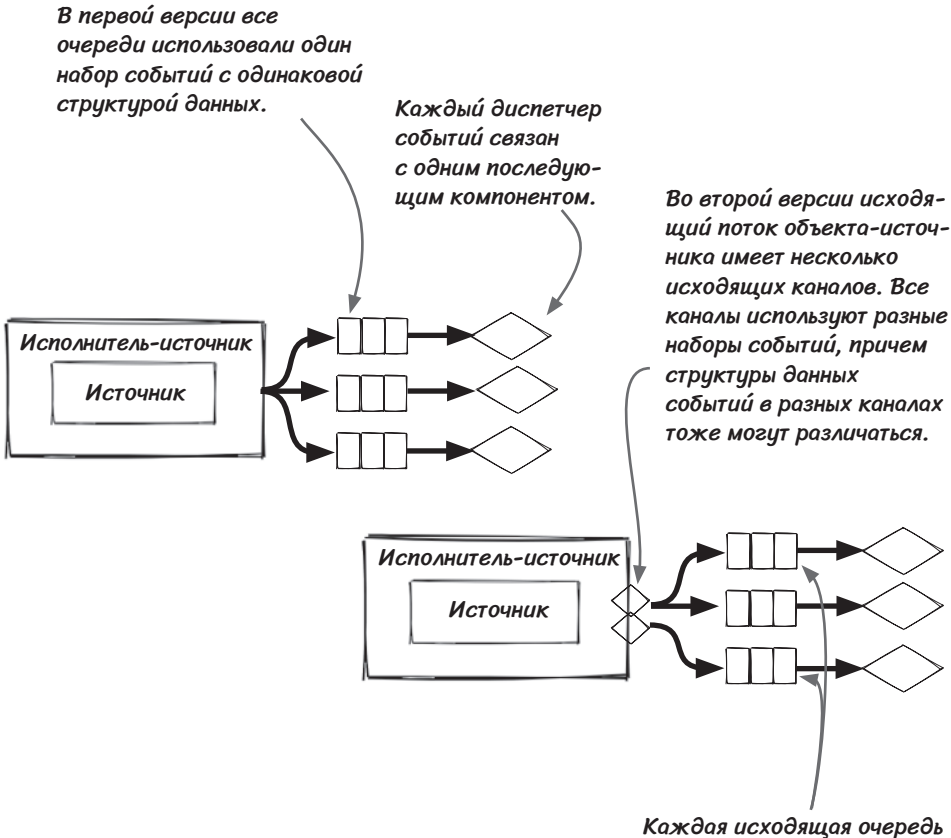
В результате к каждому анализатору направляются только необходимые события с необходимыми полями.



Что происходит внутри ядра (еще раз)

Вы узнали, что один исполнитель компонента может иметь несколько исходящих очередей. Ранее исполнитель просто направлял одно событие во все исходящие очереди, связанные с диспетчерами событий последующих компонентов. Теперь для поддержки нескольких потоков исполнитель должен взять события, сгенерированные каждым компонентом, и поместить их в соответствующие исходящие очереди.

Объект-компонент предоставляет эту информацию по *каналам* (channels). Разные события направляются в разные каналы, и последующие компоненты могут выбрать, из какого канала получать события.



Это многоканальное разветвление обеспечивает большую гибкость. Она предоставляет дополнительные возможности повысить эффективность задания.

Коммуникации между компонентами по каналам

Для поддержки нового типа разветвления потоков необходимо внести изменения в компонент и исполнитель.

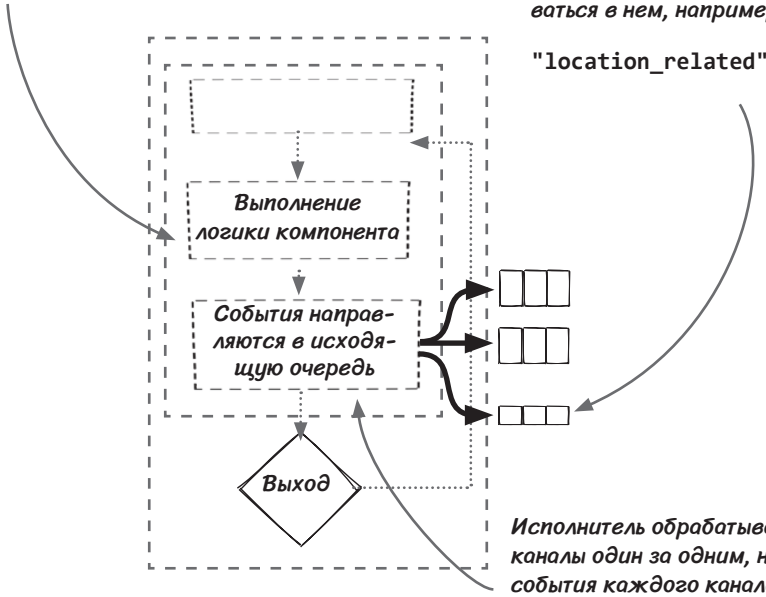
- Компонент должен быть способен направлять события в разные каналы.
- Исполнитель должен получать события из каждого канала и направлять их в правильные исходящие очереди.
- И наконец, последующий компонент должен иметь возможность выбрать нужный канал, подключаясь к нему вызовом `applyOperator()`.

Ранее на выходе компонента формировался список событий. Теперь это карта, связывающая имена каналов со списками событий:

```
default: [……]
amount_only: [……]
location_related: [……]
```

Когда компонент добавляется в задание (к исходящему потоку его предшествующего компонента), он может выбрать нужный канал и зарегистрироваться в нем, например:

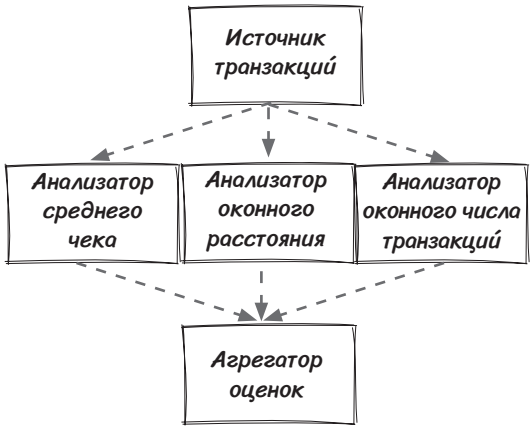
```
"location_related"
```



Исполнитель обрабатывает каналы один за одним, направляя события каждого канала в исходящие очереди, зарегистрированные в этом канале.

Несколько каналов

С поддержкой многоканальности разветвление в системе обнаружения мошеннических действий можно изменить так, чтобы анализаторам в событиях отправлялись только необходимые поля. В классе `TransactionSource` при отправке событий можно указать информацию канала. Следует заметить, что одно входящее событие может быть преобразовано в разные события для разных каналов.



Событие направляется в канал по умолчанию.

Выбирается другой канал для передачи событий.

```
eventCollector.add(new DefaultEvent(transactionEvent));
eventCollector.add("location _ based",
                   new LocationalEvent(transactionEvent);
```

События в канале используют разные структуры данных.

Тогда при добавлении анализатора в стриминговое задание при помощи функции `applyOperator()` можно сначала указать канал:

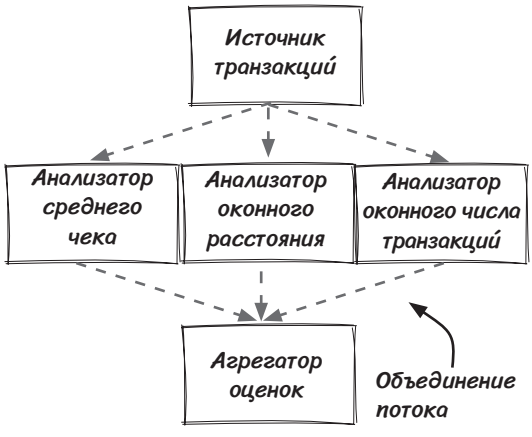
```
Job job = new Job();
Stream transactionOut = job.addSource(new TransactionSource());
```

Если канал не указан при применении оператора, используется канал по умолчанию.

```
Stream evalScores1 = transactionOut
    .applyOperator(new AvgTicketAnalyzer());
Stream evalScores2 = transactionOut
    .selectChannel("location _ based")
    .applyOperator(new WindowedProximityAnalyzer());
Stream evalScores3 = transactionOut
    .applyOperator(new WindowedTransactionAnalyzer());
```

Для применения оператора указывается конкретный канал.

Объединение потока в агрегаторе оценок



Анализаторы получают события транзакций и выполняют вычисления. Результат работы каждого анализатора представляет собой *оценку риска* (risk score) для каждой транзакции. В нашей системе оценки рисков для всех транзакций передаются компоненту — *агрегатору оценок* для принятия решения. Если обнаруживаются признаки мошенничества, в базу данных записывается информация о потенциальном нарушении.

Из диаграммы видно, что оператор *агрегатора оценок* получает ввод от нескольких предшествующих компонентов — анализаторов. Также можно рассмотреть эту структуру под другим углом: исходящие потоки анализаторов объединяются, а все их события передаются оператору агрегатора оценок. Это называется *объединением потока*.

Стоит упомянуть о том, что в операторе агрегатора оценок события от разных потоков обрабатываются одинаково. Также возможен другой случай: события разных входящих потоков могут содержать разные данные и должны использоваться по-разному. Во втором случае используется более сложное объединение потока, заслуживающее отдельной главы. Сейчас сосредоточимся на простом случае.

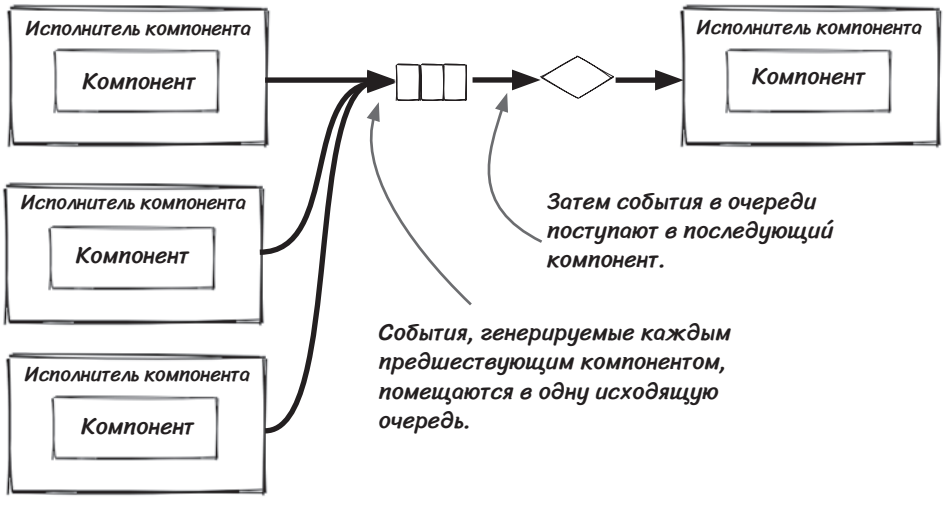
Несколько потоков объединяются в один объект Streams.

```
Stream evalScores1 = ……
Stream evalScores2 = ……
Stream evalScores3 = ……

Operator aggregator = new ScoreAggregator(
    "aggregator", 2, new GroupByTransactionId());
Streams.of(evalScores1, evalScores2, evalScores3)
    .applyOperator(aggregator);
```

Оператор `ScoreAggregator` *применяется к объекту* `Streams`*. Обратите внимание:* `GroupByTransactionId` *должен выступать подклассом* `FieldsGrouping`*, чтобы оценки конкретной транзакции гарантированно передавались одному экземпляру агрегатора.*

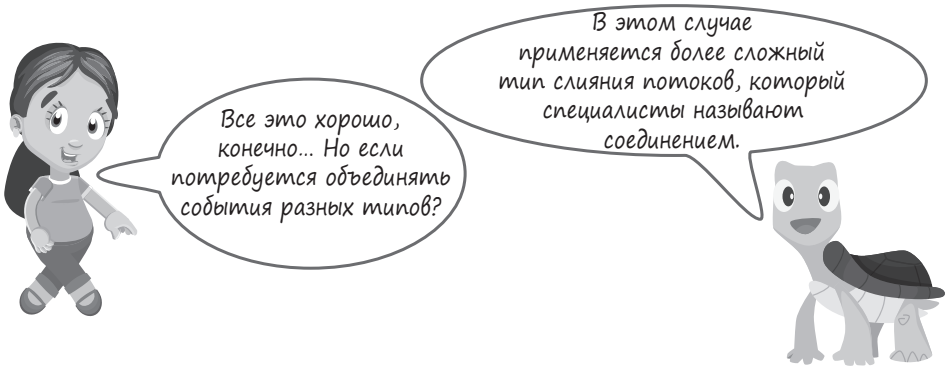
Объединение потоков в ядре

В ядре Streamwork объединение потоков реализовано достаточно просто. Входящая очередь компонента (связанная с его диспетчером событий) может использоваться несколькими предшествующими компонентами. Когда событие генерируется любым из его предшествующих компонентов (а на самом деле экземпляром самого компонента), событие помещается в очередь. Последующий компонент извлекает события из очереди и обрабатывает их. Он не различает, кто именно поместил события в очередь.



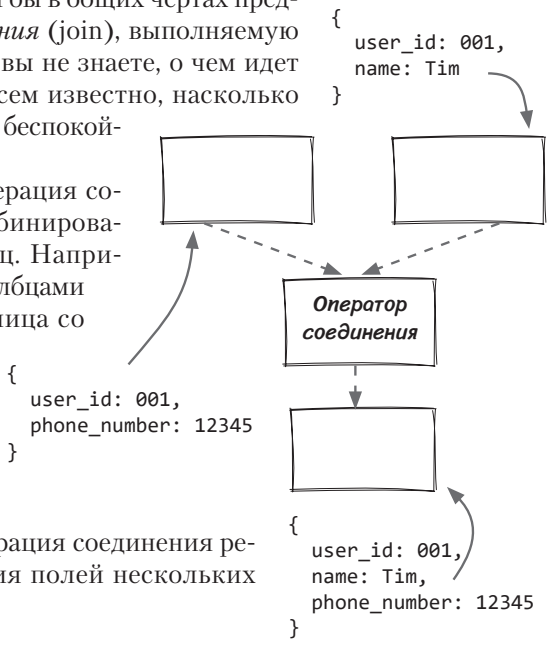
Как упоминалось ранее, очередь реализует логическую изоляцию между предшествующими и последующими компонентами.

Краткий обзор разновидности объединения потоков — соединения



Ранее говорилось о том, что кроме объединения потока, использованного в рассмотренном примере, существует более сложная разновидность объединения. Мы вкратце познакомим вас с ней, чтобы вы лучше понимали все типы объединений и разветвлений.

При простом объединении все входящие события имеют одну структуру данных и обрабатываются одинаково. Иначе говоря, входящие потоки одинаковы. А если они отличаются друг от друга и их необходимо объединить? Если вы работали с базами данных, то хотя бы в общих чертах представляете себе операцию *соединения* (join), выполняемую с несколькими таблицами. Если вы не знаете, о чем идет речь, или успели забыть (ведь всем известно, насколько надежна человеческая память), не беспокойтесь — это не обязательно.



В работе с базами данных операция соединения используется для комбинирования столбцов нескольких таблиц. Например, если у вас есть таблица со столбцами `user-id` и `name` и еще одна таблица со столбцами `user-id` и `phone-number`, эти таблицы можно соединить по столбцу `user-id` двух исходных таблиц и создать новую таблицу со столбцами `user-id`, `name` и `phone-number`. В мире потоковой обработки операция соединения решает похожую задачу соединения полей нескольких источников данных.

Тем не менее по сравнению с таблицами баз данных потоки намного динамичнее. События принимаются и обрабатываются непрерывно, а соответствию полей нескольких непрерывных источников данных приходится уделять намного больше внимания. Здесь мы объясним базовую концепцию соединения, а дальнейшее исследование этой темы будет вынесено в отдельную главу.

Система в целом

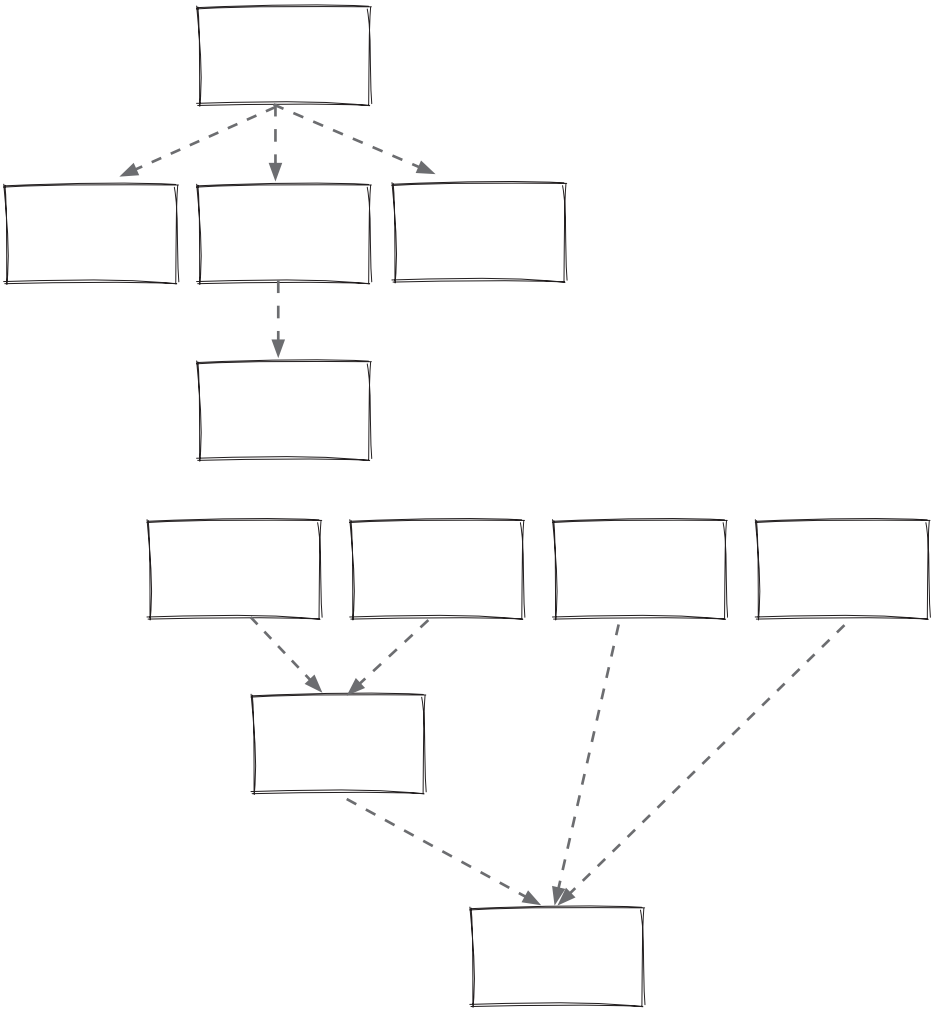
Мы рассмотрели разветвление и объединение потоков по отдельности, а сейчас взглянем на систему в целом. На высоком уровне задание можно представить в виде следующего графа; иногда такой граф называется *логическим планом*. Он представляет логическую структуру задания (компоненты и связи между ними).

Существующие системы обнаружения мошеннических действий постоянно развиваются, и в них добавляются новые анализаторы. Благодаря использованию Streamwork или других потоковых фреймворков добавлять, удалять или заменять анализаторы становится довольно просто.

Графы и стриминговые задания

Понимая объединение и разветвление потоков, мы можем строить потоковые системы на уровне более сложных и общих структур графов. Это очень важный шаг вперед, потому что эта новая структура позволяет решать прикладные задачи.

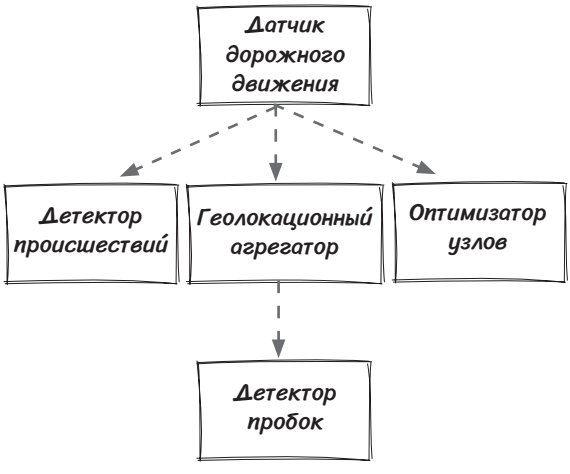
Ниже приведены DAG двух стриминговых систем. Подумайте, какие виды систем они могут представлять?

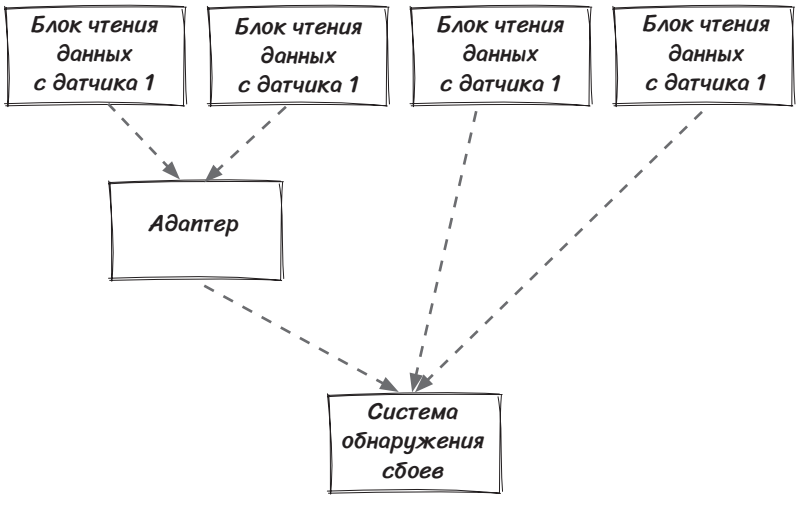


Примеры систем

Сказать по правде, эти графы могут представлять массу вещей! Назовем несколько возможных вариантов.

Первая диаграмма может представлять простую систему контроля и наблюдения за дорожным движением. События, поступающие от датчиков, передаются трем основным обработчикам: детектору происшествий, детектору пробок и оптимизатору узлов. У детектора пробок имеется геолокационный агрегатор, используемый в качестве препроцессора.



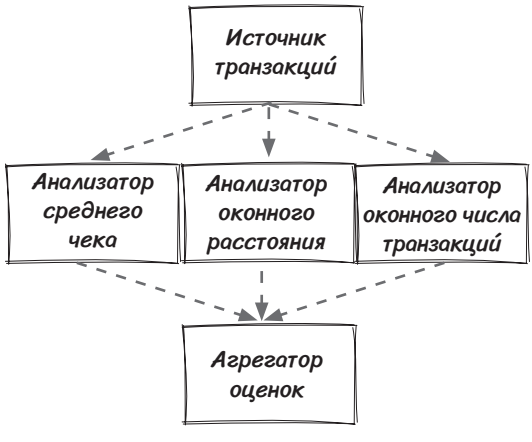


Вторая диаграмма может представлять систему обнаружения сбоев, которая обрабатывает события от блоков чтения данных с датчиков в нескольких версиях. События, генерируемые первыми двумя версиями, несовместимы с датчиком; следовательно, для них нужен адаптер. В такой системе все блоки чтения данных с датчиков могут работать совместно, в ней легко добавлять новые версии или убирать старые.

В общем и целом стриминговые задания не очень сложны. Приведенные примеры значительно упрощены по сравнению с реальными системами. Тем не менее хочется верить, что вы получили некоторое представление о возможностях стриминговых систем. В простейшем варианте стриминговые задания состоят из компонентов и связей между ними. После настройки и запуска стримингового задания события протекают через компоненты по связям *постоянно*.

Итоги

В этой главе мы перешли от списочной структуры систем, которые рассматривались в предыдущих главах, к более общему типу структуры системы — *графам*. Так как события в системах направляются от источников к операторам, в большинстве случаев стриминговое задание можно представить в виде направленного ациклического графа (DAG). Большинство прикладных заданий имеет архитектуру графа, следовательно, этот этап чрезвычайно важен.

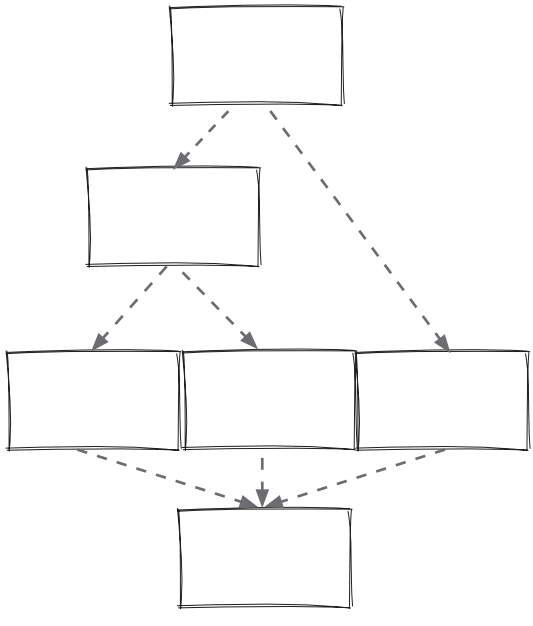


В отличие от компонентов в списочной структуре, в заданиях со структурой графа компонент может связываться с несколькими предшествующими и последующими компонентами. Такие типы соединений называются объединением и разветвлением потока. Потоки, входящие в компонент или выходящие из него, могут использовать одинаковые или разные типы событий.

Кроме того, мы подробнее рассмотрели фреймворк Streamwork, чтобы показать, как ядро обрабатывает связи между компонентами. Надеемся, это поможет вам понять общий принцип работы стриминговых систем.

Упражнения

1. Попробуйте добавить новый анализатор в задание обнаружения мошеннических действий.
2. Каждый анализатор получает событие транзакции от компонента — источника транзакций и создает оценку. Два анализатора используют одинаковые вычисления для формирования своих оценок. Попробуйте изменить задание для этого случая. Результат будет выглядеть так, как показано на схеме:



Семантика доставки | 5

В этой главе

- ✓ Семантика доставки и ее влияние.
- ✓ Семантика доставки «не более одного».
- ✓ Семантика доставки «не менее одного».
- ✓ Семантика доставки «ровно один».

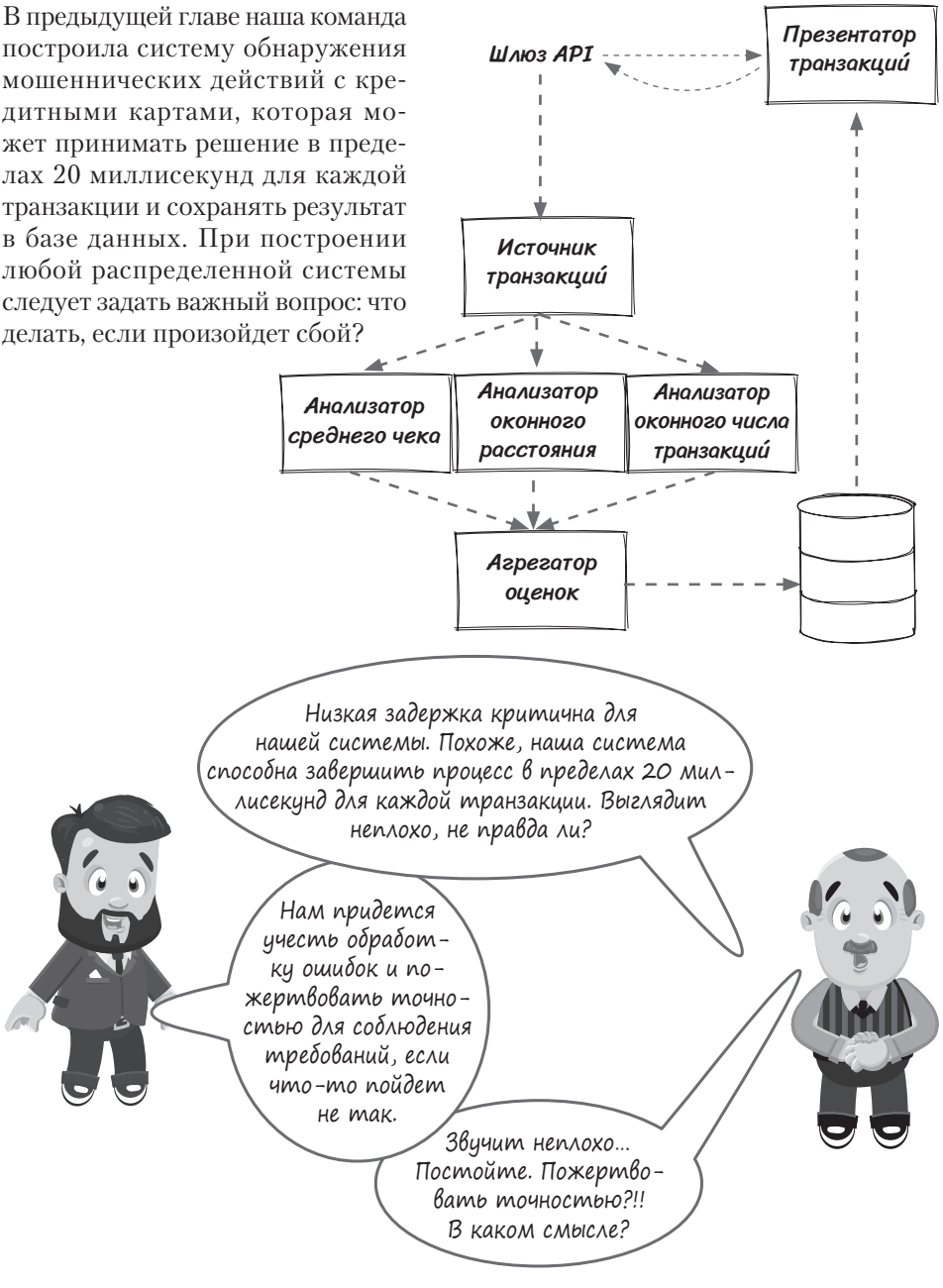
«Времени на то, чтобы сделать все как следует, никогда не хватает, зато всегда есть время на то, чтобы все переделать».

Джек Бергман

Компьютеры хорошо справляются с точными вычислениями. Тем не менее при совместной работе компьютеров в распределенных системах (каковыми являются многие стриминговые системы) проблема точности немного (а на самом деле очень заметно!) усложняется. Иногда стопроцентная точность не нужна, потому что необходимо выполнять более важные требования. Кто-то спросит: «Зачем нужны неправильные ответы?» Это отличный вопрос, и его необходимо задавать себе при проектировании стриминговой системы. В этой главе обсуждается важная тема, относящаяся к точности в потоковых системах, — *семантика доставки*.

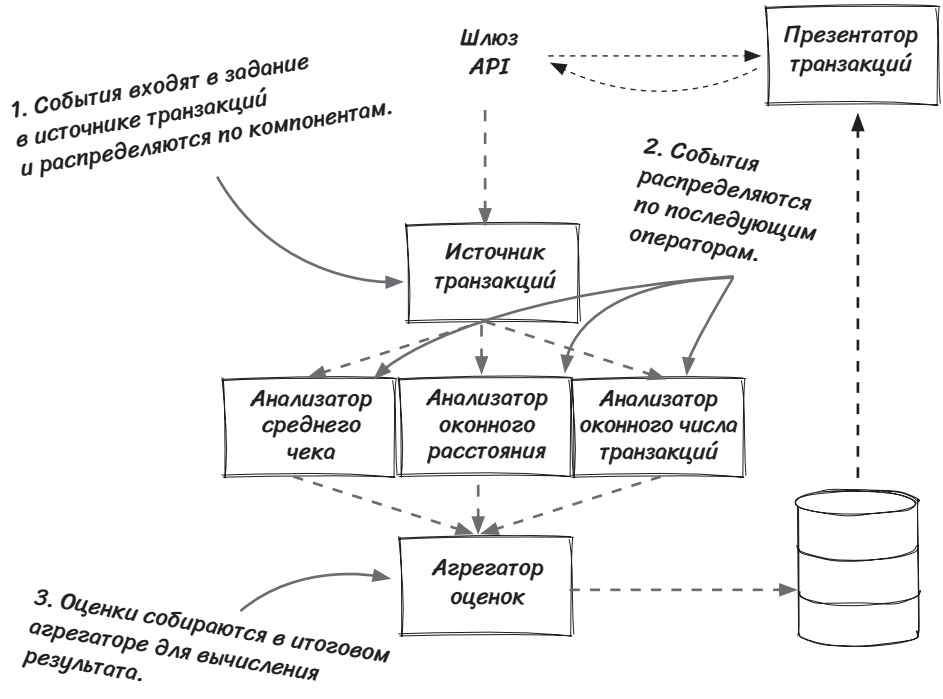
Требования к задержке в системе обнаружения мошеннических действий

В предыдущей главе наша команда построила систему обнаружения мошеннических действий с кредитными картами, которая может принимать решение в пределах 20 миллисекунд для каждой транзакции и сохранять результат в базе данных. При построении любой распределенной системы следует задать важный вопрос: что делать, если произойдет сбой?



Возвращаемся к заданию обнаружения мошеннических действий

Для обсуждения темы семантики доставки мы продолжим использовать систему обнаружения мошеннических действий из предыдущей главы. Кратко напомним структуру системы и задания.



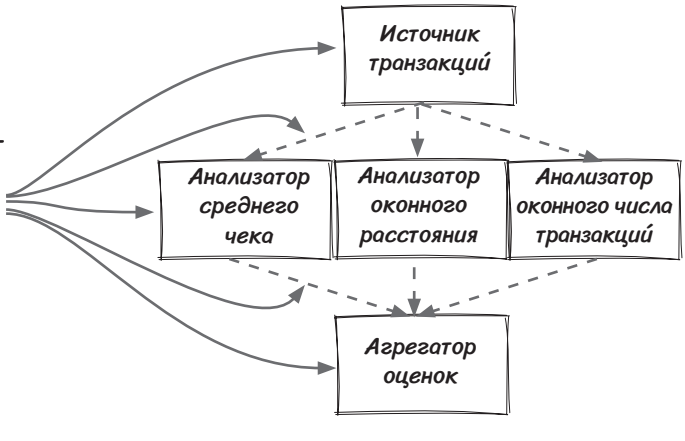
В задании обнаружения мошеннических действий несколько анализаторов работают параллельно для обработки транзакций, охватывающих сеть кредитных карт. Оценки риска от этих анализаторов передаются агрегатору, который вычисляет окончательный результат для каждой транзакции, а результаты записываются в базу данных для презентатора.

20-миллисекундный порог задержки критичен. Если решение не будет принято вовремя, то презентатор не сможет предоставить ответ по транзакции банку. В идеале нам хотелось бы, чтобы задание работало бесперебойно и постоянно соблюдало требования к задержке. Но в жизни бывает всякое.

О точности

В распределенных системах часто приходится идти на компромисс. Главной проблемой любой стриминговой системы становится надежная обработка событий. Стриминговые фреймворки помогают обеспечить надежное выполнение задания в большинстве случаев, но вам нужно правильно определить потребности. Мы привыкли, что компьютеры выдают точные результаты; а значит, очень важно понимать, что в стриминговых системах точность не абсолютна. Там, где это необходимо, ею приходится жертвовать.

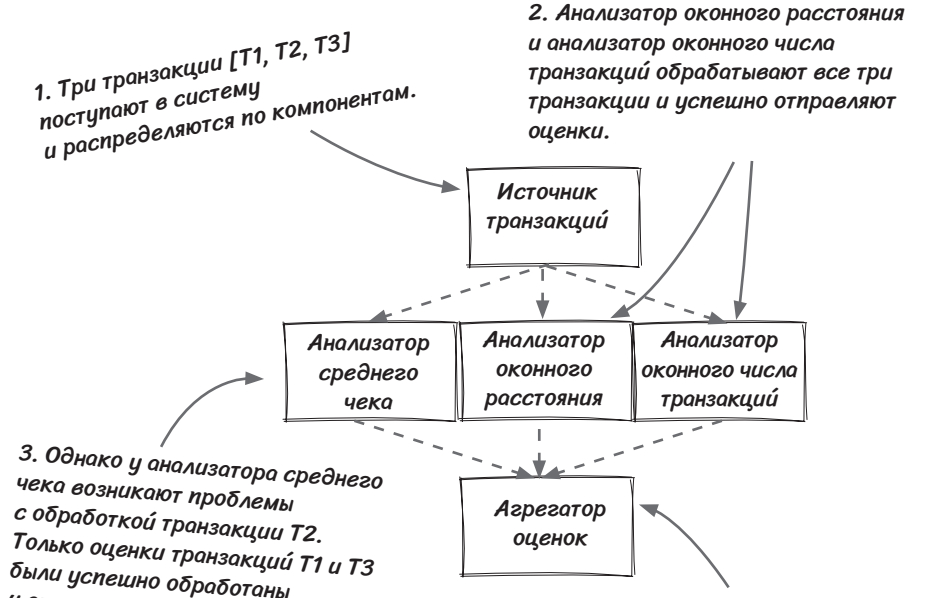
В системе обнаружения мошеннических действий может быть множество неполадок — например, сетевые сбои или нехватка ресурсов у отдельных экземпляров. Необходимо принять меры к тому, чтобы система работала надежно.



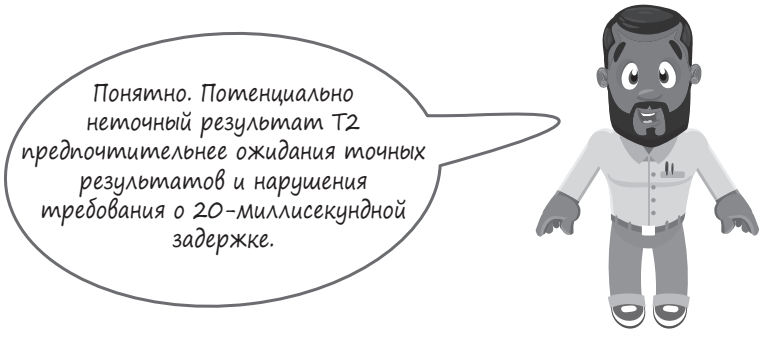
Без паники! Сейчас мы подробно рассмотрим вышесказанное.

Частичный результат

Частичный результат появляется из-за неполноты данных; его точность не гарантирована. На диаграмме ниже приведен пример частичного результата при возникновении проблемы в анализаторе среднего чека.

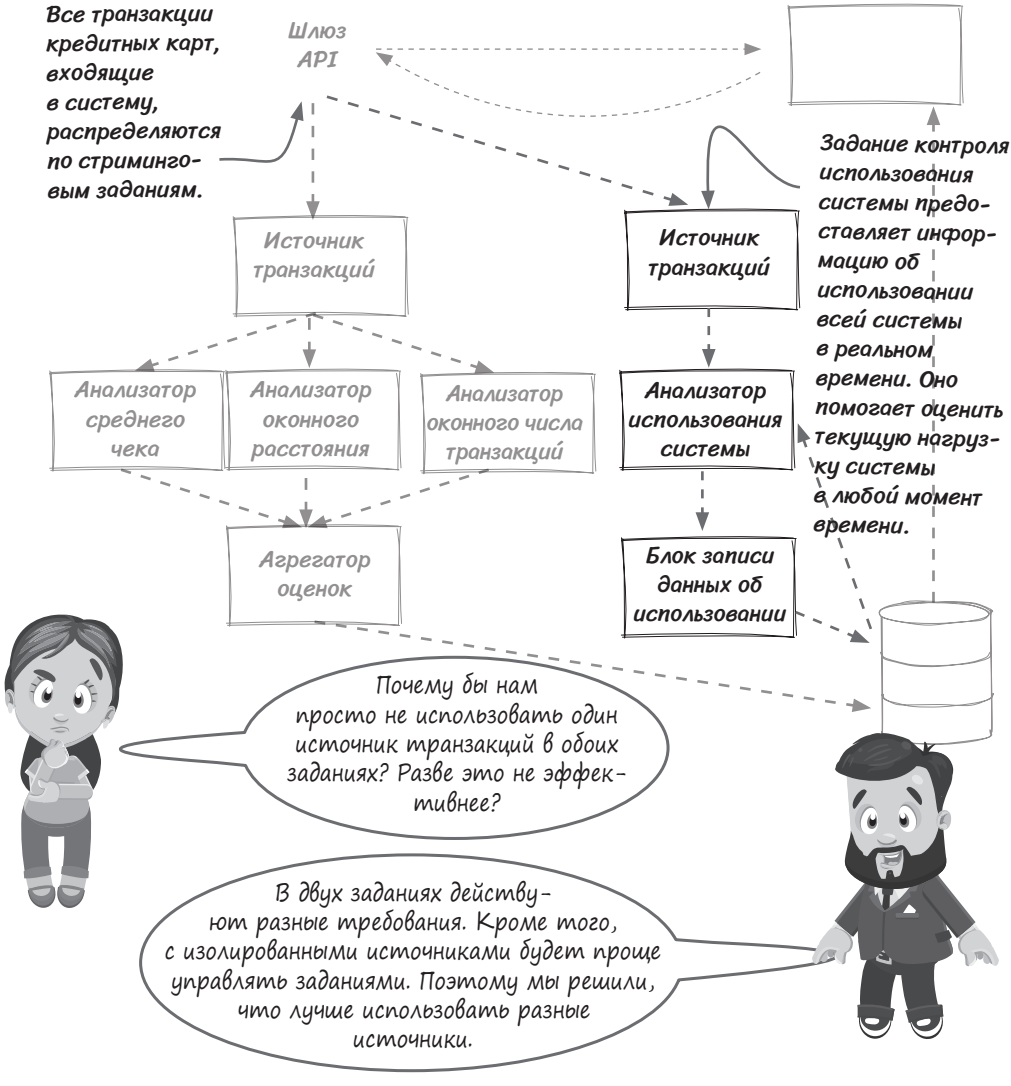


В стриминговых системах часто приходится принимать компромиссные решения.



Новое стриминговое задание для контроля за использованием системы

Теперь, когда вы узнали требования задания обнаружения мошеннических действий, мы введем для сравнения другое задание с другими требованиями, чтобы лучше понять разную семантику доставки. Система обнаружения мошеннических действий произвела фурор в обработке транзакций кредитных карт. Другие компании, выпускающие кредитные карты, начали интересоваться идеей, и с ростом интереса команда решила добавить в систему другое стриминговое задание для контроля за использованием системы. Задание отслеживает ключевую информацию, например количество обработанных транзакций.



Новое задание контроля использования системы

Новое задание используется внутренними механизмами для наблюдения за текущей нагрузкой на систему. Начнем с двух критических показателей, которые нас интересуют в первую очередь.

- Сколько транзакций было обработано? Это число важно для понимания приблизительного общего объема данных, обрабатываемого заданием обнаружения мошеннических действий.
- Сколько подозрительных транзакций было обнаружено? Это число помогает понять количество новых записей, созданных в базе данных результатов.

Логика подсчета сосредоточена в операторе `SystemUsageAnalyzer`:

```
class SystemUsageAnalyzer extends Operator {
  private int transactionCount = 0;
  private int fraudTransactionCount = 0;

  public void apply(Event event, EventCollector collector) {
    String id = ((TransactionEvent)event).getTransactionId();
    transactionCount++;

    Thread.sleep(20);

    boolean fraud = fraudStore.getItem(id);

    if (fraud) {
      fraudTransactionCount++;
    }

    collector.emit(new UsageEvent(
    transactionCount, fraudTransactionCount));
  }
}
```

Подсчет транзакций.

Приостановка на 20 миллисекунд для завершения обработки заданием обнаружения мошеннических действий.

Чтение результата обнаружения транзакции из базы данных. Эта операция может завершиться неудачей, если база данных недоступна; в этом случае выдается исключение.

Подсчет мошеннических транзакций, если результат равен true.

Оператор выглядит очень просто:

- Для каждой транзакции значение `transactionCount` увеличивается на 1.
- Если для транзакции обнаруживается попытка мошенничества, то значение `fraudTransactionCount` увеличивается на 1.

Однако вызов `getItem()` в функции может завершиться неудачей. Поведение задания при обнаружении ошибки — ключевое различие разных *семантик доставки*.

Требования к заданию контроля

Прежде чем беспокоиться об ошибках, необходимо обсудить ряд моментов. Начнем с требований к заданию. Так как задание контроля является внутренним инструментом, требования к задержке и точности могут сильно отличаться от аналогичных в задании обнаружения мошеннических действий.

- *Задержка* — требование о 20-миллисекундной задержке для задания обнаружения мошеннических действий не распространяется на задание контроля за использованием системы, так как результаты не используются презентатором для генерирования решений для банков. Люди не могут так быстро воспринимать результаты. Более того, небольшая задержка при возникновении проблем вполне допустима.
- *Точность* — с другой стороны, точные результаты могут быть важны для принятия правильных решений.

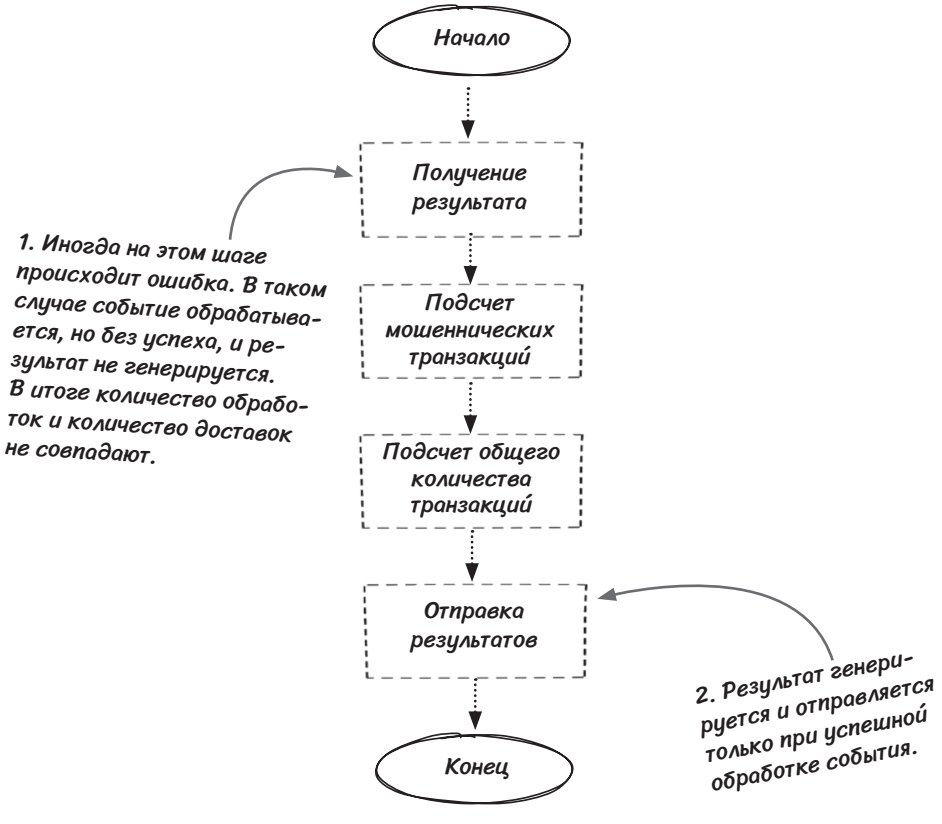
Ниже мы рассмотрим наиболее распространенные типы семантики доставки, чтобы вам было проще начинать знакомство с потоковой обработкой. Заодно опишем некоторые возможности использования стриминговых систем для гарантированной обработки транзакций и расскажем, в каких ситуациях это может быть полезно.

Новые концепции: количество доставок и обработок

Концепции *количества обработок* и *количества доставок* помогают понять реальный смысл семантики доставки.

- Количество обработок показывает, сколько раз событие было обработано компонентом.
- Количество доставок показывает, сколько раз результат был сгенерирован компонентом.

Эти два числа совпадают в большинстве случаев, но не всегда. Например, на приведенной ниже блок-схеме логики оператора `SystemUsageAnalyzer` может оказаться, что на шаге *получения результата* возникли проблемы из-за недоступности базы данных. Если возникает ошибка, событие обрабатывается один раз (но безуспешно) и результат не генерируется. В результате количество обработок будет равно 1, а количество доставок — 0. Также можно рассматривать количество доставок как количество *успешных обработок*.

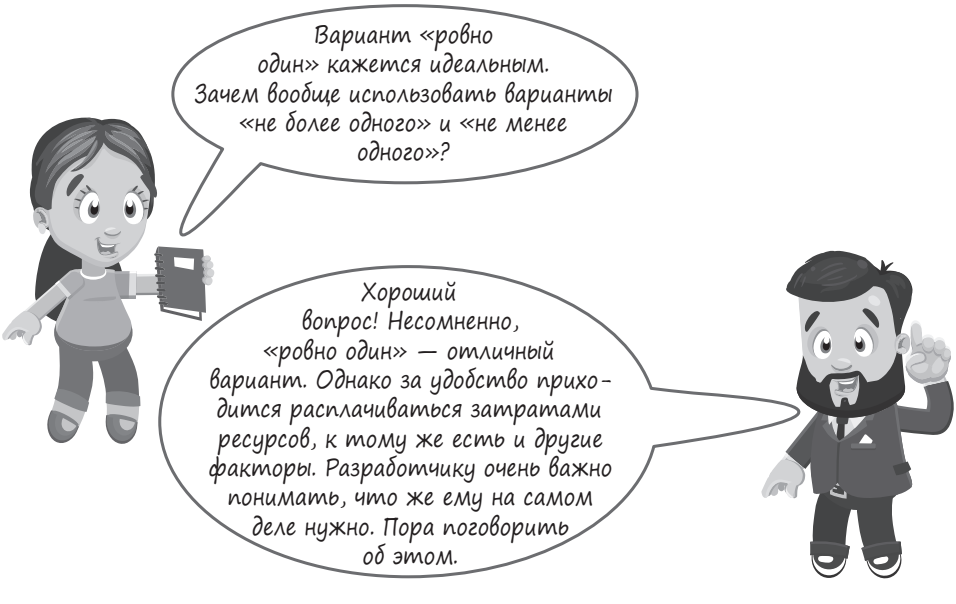


Новая концепция: семантика доставки

Мы подходим к ключевой концепции этой главы: *семантике доставки*, также называемой *гарантией доставки*. Это очень важная концепция стриминговых заданий, которую необходимо понять, прежде чем переходить к более сложным темам.

Семантика доставки определяет то, как стриминговые ядра гарантируют доставку (или успешную обработку) событий в стриминговых заданиях. Существуют три основные категории семантики доставки. Сейчас мы рассмотрим их кратко, а подробности разберем позже.

- «*Не более одного*» (at-most-once) — стриминговые задания гарантируют, что каждое событие будет обработано не более одного раза, но не гарантируют, что оно будет обработано успешно.
- «*Не менее одного*» (at-least-once) — стриминговые задания гарантируют, что каждое событие будет успешно обработано по крайней мере один раз, но не гарантируют количество его обработок.
- «*Ровно один*» (exactly once) — стриминговые задания гарантируют, что каждое событие будет успешно обработано один и только один раз (по крайней мере внешне). В некоторых фреймворках используется термин «*фактически один*» (effectively-once). Если вам кажется, что это слишком хорошо, чтобы быть правдой, поскольку условия «ровно один» чрезвычайно трудно добиться в распределенных системах, или что два термина противоречат друг другу, — вы определенно не одиноки. О том, что в действительности означает семантика «ровно один», мы расскажем позже в соответствующем разделе.



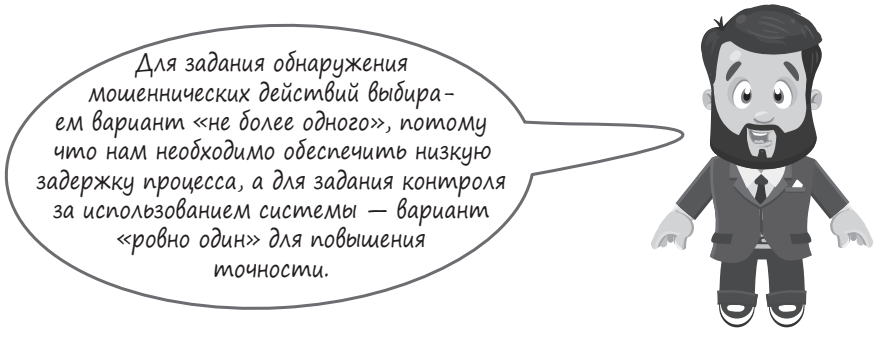
Выбор семантики

Правда ли, что семантика «ровно один» идеально подходит для всего? Ее преимущества очевидны: результаты гарантированно точны, а правильный ответ лучше неправильного.

С семантикой «ровно один» стриминговое ядро сделает все за вас, и беспокоиться не о чем. Для чего тогда два других варианта? Зачем нужно знать о них? Дело в том, что все варианты полезны, потому что в разных стриминговых системах действуют разные требования.

Ниже приведена простая таблица достоинств и недостатков каждого варианта. Мы еще вернемся к этой таблице позже.

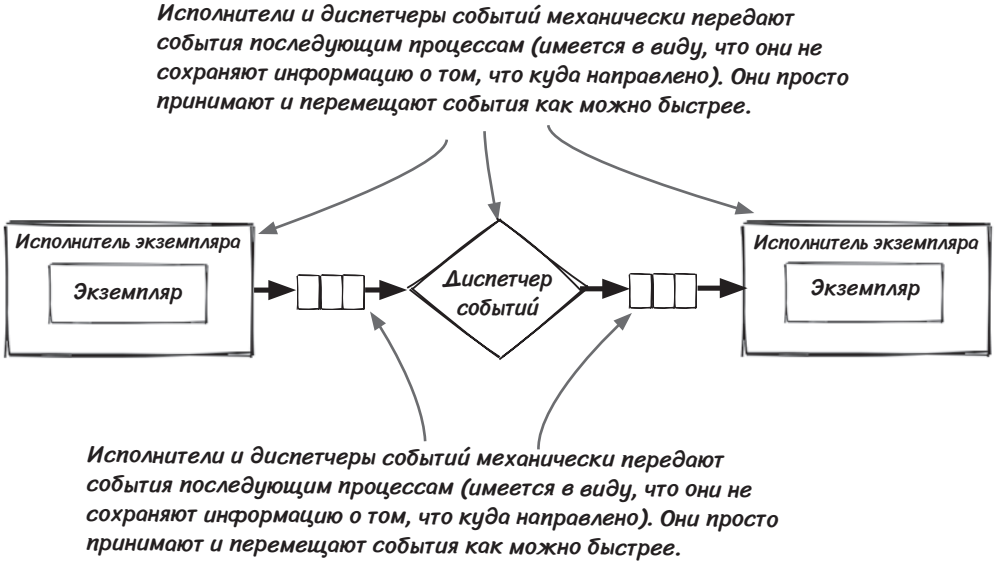
Семантика доставки	**Не более одного**	**Не менее одного**	**Ровно один**
Точность	Точность не гарантирована из-за пропуска событий	Точность не гарантирована из-за дублирования событий	(На первый взгляд) гарантирован точный результат
Задержка (при возникновении ошибки)	Устойчивость к сбоям; отсутствие задержки при возникновении ошибок	Чувствительность к сбоям; возможны задержки при возникновении ошибок	Чувствительность к сбоям; возможны задержки при возникновении ошибок
Сложность	Очень простая	Средняя (в зависимости от реализации)	Высокая



А теперь продолжим изучать применение семантики доставки в стриминговых системах: это позволит лучше понять достоинства и недостатки каждого варианта. Следует заметить, что в реальном мире каждый фреймворк имеет собственную архитектуру и по-разному подходит к семантике доставки. Мы постараемся излагать материал без привязки к конкретному фреймворку.

«Не более одного»

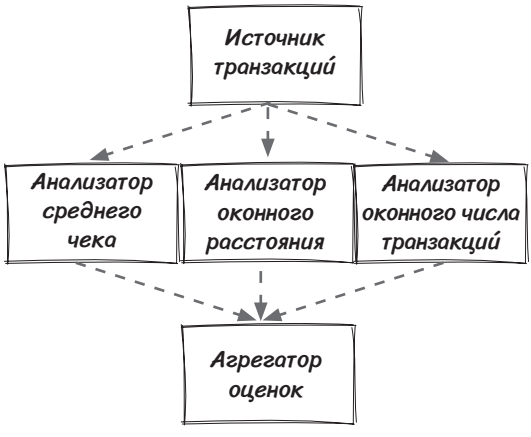
Начнем с простейшей семантики: «не более одного». В заданиях с такой семантикой события не отслеживаются. Ядро прикладывает максимум усилий к тому, чтобы обрабатывать каждое событие успешно, но если все же произойдет ошибка, ядро переключается на другое событие. На следующей диаграмме показана реализация обработки событий ядром Streamwork для заданий «не более одного».



Так как ядра не отслеживают события, все задание может выполняться очень эффективно без особых потерь. А поскольку задание при возникновении ошибки просто продолжает работать без необходимости восстанавливаться, ошибки не влияют на задержку и пропускную способность. Кроме того, такой подход облегчает обслуживание вследствие его простоты. С другой стороны, эффект потери событий при возникновении проблем заключается в том, что результаты могут оказаться временно неточными.

Задание обнаружения мошеннических действий

Рассмотрим задание обнаружения мошеннических действий с семантикой «не более одного». Задание отвечает за суммирование оценок риска для каждой транзакции в сети кредитных карт и должно выдавать результат в течение 20 миллисекунд.



Плюсы

С гарантией «не более одного» система упрощается, а транзакции обрабатываются с меньшей задержкой. Когда в системе что-то идет не так (например, происходит ошибка при обработке или передаче транзакции либо какой-то экземпляр временно становится недоступным), соответствующие события просто пропускаются, а агрегатор оценок обрабатывает доступные данные, так что критичное требование задержки не нарушается.

Низкое потребление ресурсов и затраты на сопровождение — другая причина для выбора семантики «не более одного». Например, если вам приходится обрабатывать огромный объем данных в реальном времени с ограниченными ресурсами, стоит присмотреться к семантике «не более одного».

Минусы

А теперь обсудим оборотную сторону: неточность. Она определенно является важным фактором при выборе семантики «не более одного». Эта семантика подходит для случаев, в которых временная неточность приемлема. При рассмотрении этого варианта важно задать себе вопрос: к каким последствиям приведет временная неточность результатов?

Надежды

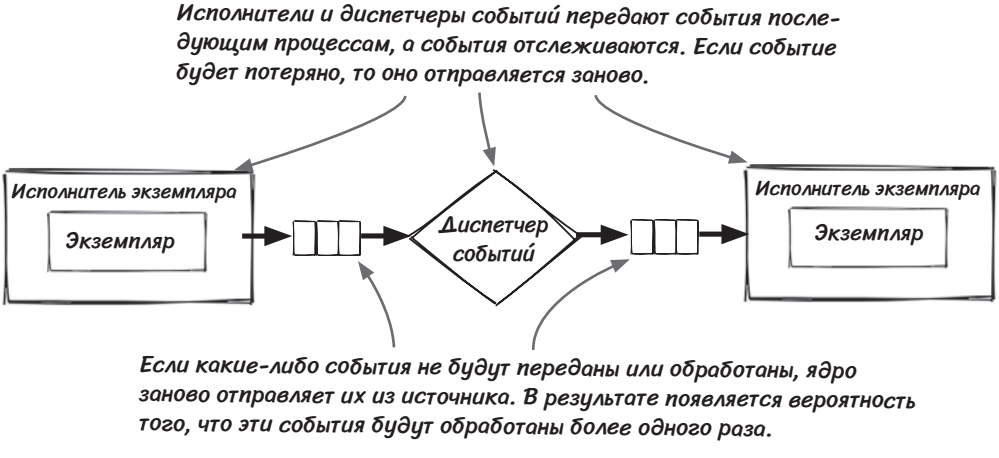
Если вам нужны преимущества семантики «не более одного» в сочетании с точными результатами, не отчаивайтесь. Хотя ожидать всего и сразу слишком оптимистично, это ограничение все же можно преодолеть (в некоторой степени). Полезные приемы для этого мы рассмотрим в конце главы, а пока перейдем к двум другим семантикам доставки.

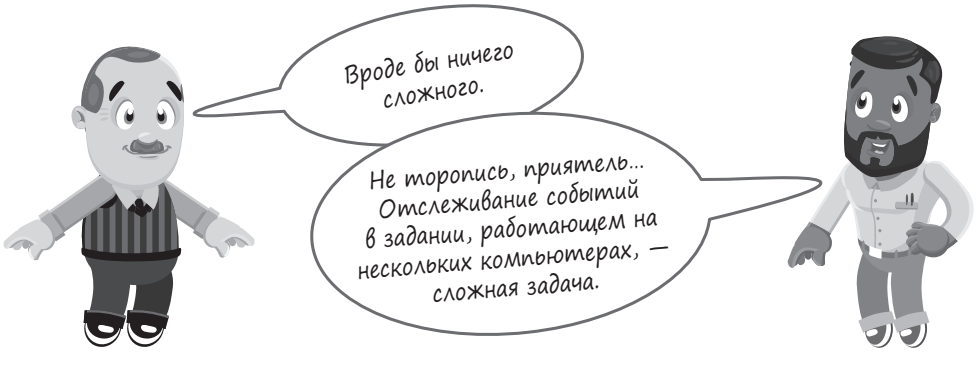
«Не менее одного»

Какой бы удобной ни была семантика «не более одного», у нее есть очевидный недостаток: отсутствие гарантий того, что каждое событие будет надежно обработано. Во многих ситуациях это попросту неприемлемо. Другой недостаток заключается в том, что события теряются и для повышения точности почти ничего нельзя сделать.

Следующая семантика доставки — «не менее одного» — может пригодиться для преодоления недостатков, упомянутых выше. С семантикой «не менее одного» стриминговые ядра гарантируют, что событие будет обработано хотя бы один раз. Побочный эффект семантики «не менее одного» заключается в том, что события могут обрабатываться более одного раза. На следующей диаграмме показано, как происходит обработка событий ядром Streamwork для заданий «не менее одного».

Обратите внимание: может показаться, что отслеживать события и обеспечивать успешную обработку каждого из них несложно, но в распределенных системах эта задача далеко не тривиальна. Ее мы рассмотрим далее.

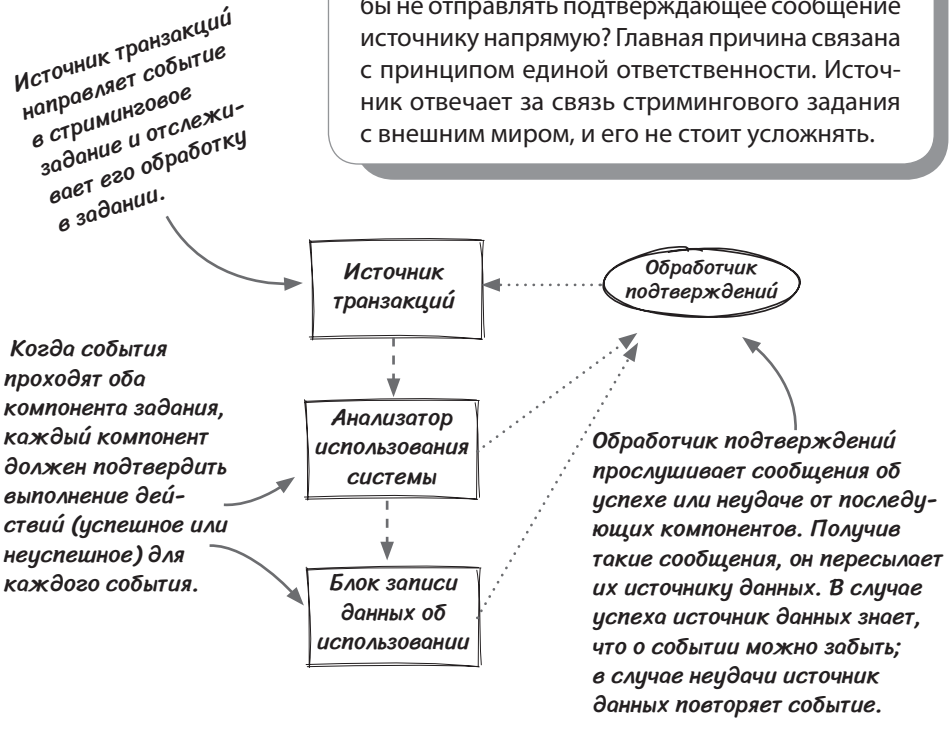




«Не менее одного» с подтверждением

Типичный подход к поддержке семантики доставки «не менее одного» заключается в том, что каждый компонент стримингового задания подтверждает, что он успешно обработал событие или потерпел неудачу. Стриминговые фреймворки обычно предоставляют механизм контроля в виде нового *обработчика подтверждений* (acknowledger). Этот обработчик подтверждений отвечает за отслеживание *текущих* и *завершенных* обработок для каждого события. Когда все обработки будут завершены и для события не останется ни одной *незавершенной* обработки, обработчик выдает сообщение об *успехе* или *неудаче* источнику данных. Рассмотрим задание контроля за использованием системы, выполняемое с семантикой «не менее одного».

> ***Обработчик подтверждений***
>
> Некоторые читатели могут спросить: почему бы не отправлять подтверждающее сообщение источнику напрямую? Главная причина связана с принципом единой ответственности. Источник отвечает за связь стримингового задания с внешним миром, и его не стоит усложнять.



После того как компонент-источник сгенерирует событие, он сохраняет его в буфере. При получении сообщения об успехе от обработчика подтверждений событие удаляется из буфера, так как оно было успешно обработано. Если компонент-источник получит сообщение об ошибке для события, он воспроизводит это событие, повторно отправляя его в задание.

Отслеживание событий

Рассмотрим отслеживание событий на примере. В тот момент, когда базовое событие покидает источник данных, ядро добавляет к нему метаданные, в том числе идентификатор события, по которому оно отслеживается в задании. Компоненты сообщают обработчику подтверждений о завершении обработки события.

Обратите внимание: последующие компоненты включаются в данные подтверждения, поэтому обработчик подтверждений знает, что ему нужно ожидать данных отслеживания от всех последующих компонентов, прежде чем пометить событие как *полностью обработанное*.

1. Источник транзакций получает транзакцию и отправляет ее для обработки, присвоив ей идентификатор 101. Транзакция остается готовой к повторной отправке, пока все компоненты не подтвердят, что событие обработано успешно. Подтверждение может выглядеть примерно так:

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: источник транзакций,
  Последующие компоненты: [
    Анализатор использования системы
  ]
}
}
```

2. Анализатор отправляет подтверждение для идентификатора после завершения обработки полученного события.

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: анализатор
  использования системы,
  Последующие компоненты: [
    блок записи данных
  ]
}
```

3. Блок записи данных отправляет другое подтверждение для идентификатора после завершения обработки.

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: блок записи данных,
}
```

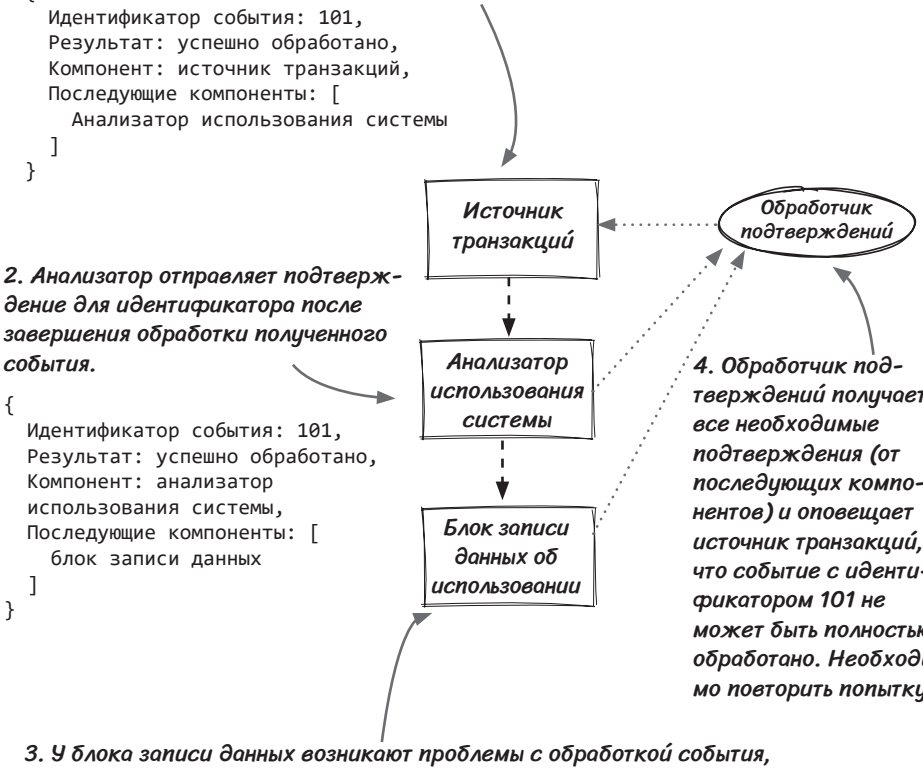
4. Обработчик подтверждений получает все необходимые подтверждения (от последующих компонентов) и оповещает источник транзакций о том, что транзакция с идентификатором 101 полностью обработана.

Управление сбоями при обработке событий

В противном случае, если событие не будет обработано в каком-либо компоненте, обработчик подтверждений приказывает компоненту-источнику отправить его заново.

Источник транзакций получает транзакцию и отправляет ее для обработки, присвоив ей идентификатор 101. Транзакция остается готовой к повторной отправке, пока все компоненты не подтвердят, что событие обработано успешно. При получении оповещения о сбое событие отправляется заново с новым присвоенным идентификатором. Подтверждение может выглядеть примерно так:

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: источник транзакций,
  Последующие компоненты: [
    Анализатор использования системы
  ]
}
```



2. Анализатор отправляет подтверждение для идентификатора после завершения обработки полученного события.

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: анализатор
  использования системы,
  Последующие компоненты: [
    блок записи данных
  ]
}
```

4. Обработчик подтверждений получает все необходимые подтверждения (от последующих компонентов) и оповещает источник транзакций, что событие с идентификатором 101 не может быть полностью обработано. Необходимо повторить попытку.

3. У блока записи данных возникают проблемы с обработкой события, и после обнаружения сбоя он отправляет другое подтверждение для идентификатора. Подтверждение может выглядеть примерно так:

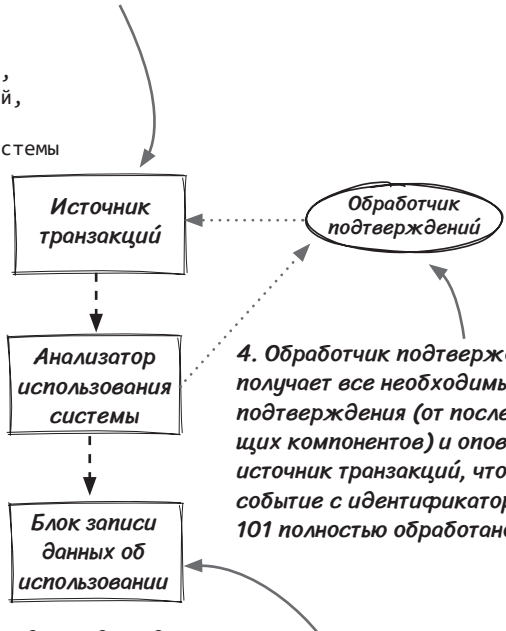
```
{
  Идентификатор события: 101,
  Результат: ошибка при обработке,
  Компонент: блок записи данных,
}
```

Раннее обнаружение потерянных событий

Последний случай, который необходимо рассмотреть, — случай, когда не все события проходят через все компоненты. Некоторые события могут завершить свое перемещение преждевременно. Вот почему так важна информация от последующих компонентов в подтверждающем сообщении. Например, если транзакция недействительна и не может быть записана в хранилище, анализатор использования системы будет последней остановкой на пути события и обработка на этом завершится.

1. Источник транзакций получает транзакцию и отправляет ее для обработки, присвоив ей идентификатор 101. Транзакция остается готовой к повторной отправке, пока все компоненты не подтвердят, что событие обработано успешно. Подтверждение может выглядеть примерно так:

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: источник транзакций,
  Последующие компоненты: [
    Анализатор использования системы
  ]
}
```



4. Обработчик подтверждений получает все необходимые подтверждения (от последующих компонентов) и оповещает источник транзакций, что событие с идентификатором 101 полностью обработано.

2. Анализатор отправляет подтверждение для идентификатора после завершения обработки полученного события. Если это последний компонент для события, в данных подтверждения последующий компонент отсутствует. Подтверждение может выглядеть примерно так:

```
{
  Идентификатор события: 101,
  Результат: успешно обработано,
  Компонент: анализатор использования системы
  ]
}
```

3. Компонент блока записи данных не входит в список последующих компонентов, поэтому обработчик подтверждения не будет ожидать подтверждения от него.

Код подтверждения в компонентах

У вас появился вопрос, как ядро узнает, успешно ли была выполнена обработка компонентом? Хорошо! Ниже приводятся фрагменты кода, которые будут реализованы в компонентах `SystemUsageAnalyzer` и `UsageWriter`.

```
class SystemUsageAnalyzer extends Operator {
  public void apply(Event event, EventCollector collector) {
    if (isValidEvent(event.data)) {
      if (analyze(event.data) == SUCCESSFUL) {
        collector.emit(event);

        collector.ack(event.id);
      } else {
        // Сигнал о сбое при обработке события
        collector.fail(event.id);
      }
    } else {
      // Сигнал об успешной обработке события
      collector.ack(event.id);
    }
  }
}
```

Подтверждение будет отправляться при отправке события для подтверждения того, что событие обработано успешно.

Анализ завершился неудачей. Событие подтверждается как неуспешное.

Событие должно быть пропущено. Событие подтверждается как успешное, чтобы компонент-источник не отправлял его заново.

```
class UsageWriter extends Operator {
  public void apply(Event event, EventCollector collector) {
    if (database.write(event) == SUCCESSFUL) {
      // Сигнал об успешной обработке события
      collector.ack(event.id);
    } else {
      // Сигнал о сбое при обработке события
      collector.fail(event.id);
    }
  }
}
```

Отправлять это событие не нужно. Вручную подтверждается успешная обработка события.

У базы данных возникли проблемы с записью. Событие подтверждается как неуспешное.

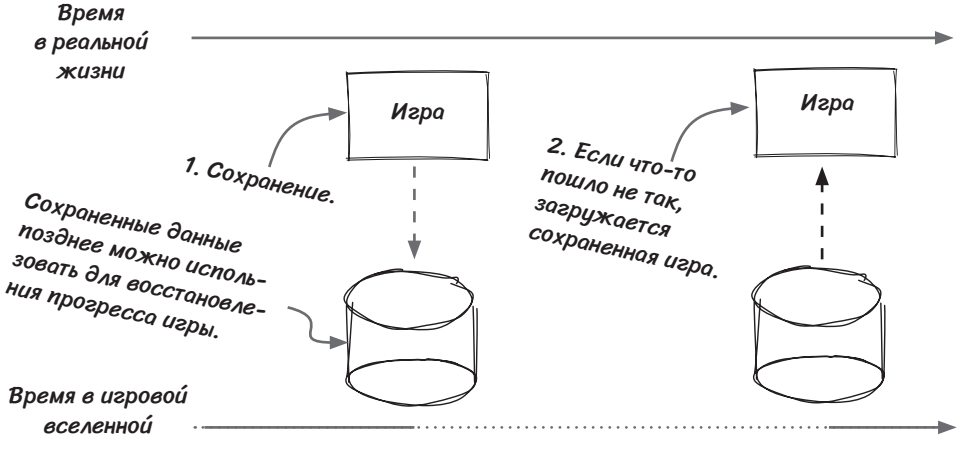
Новая концепция: контрольные точки

Подтверждение хорошо работает для семантики «не более одного», но у него есть свои недостатки.

- Для него нужна логика подтверждения (то есть изменения в коде).
- Порядок обработки событий может отличаться от порядка ввода, что создает проблемы. Например, если имеются три события [A, B, C] для обработки и в задании обработки произошел сбой при обработке события A, то позднее источник воспроизведет другую копию события A; таким образом, заданию будут переданы четыре события [A (сбой), B, C, A] и событие A успешно обрабатывается после B и C.

К счастью, существует другой вариант поддержки семантики «не менее одного» (со своими достоинствами и недостатками, как у всего остального в распределенных системах): *контрольные точки* (checkpointing). Это важный метод обеспечения *ошибкоустойчивости* (fault tolerance) в стриминговых системах, то есть продолжения нормального функционирования системы после сбоев. Так как в схеме задействовано много составляющих, подробно рассматривать контрольные точки в стриминговых системах будет непросто. Попробуем подойти к этому немного иначе. Хотя концепция контрольных точек может показаться довольно сложной, на самом деле вы наверняка сталкивались с ней в жизни, если играли в видеоигры. Впрочем, даже если не играли, ничего страшного — вспомните любой текстовый редактор (или попробуйте сыграть в видеоигру).

Допустим, в видеоигре вы сражаетесь с ордами зомби и спасаете мир. Вряд ли вам удастся завершить игру от начала до конца без перерыва; с этим справится только супергерой, никогда не совершающий ошибок. Большинство из нас время от времени ошибается (а некоторые даже чаще). К счастью, вы сохраняете свое игровое состояние, чтобы его можно было загрузить заново и продолжить, а не начинать все с самого начала. В некоторых играх прогресс автоматически сохраняется в критических точках. Теперь представьте, что вы живете в игровой вселенной. Время должно течь непрерывно, хотя в реальной жизни вы несколько (или много) раз возвращались к более ранним состояниям. Операция сохранения игры очень похожа на создание контрольной точки.



Новая концепция: состояние

Каждый, кто играет в видеоигры, знает, насколько важны сохраненные данные. Я не представляю, как бы я прошел любую игру (или завершил любую работу) без этой возможности. Более формально *контрольную точку* можно определить как блок данных, обычно находящийся в долгосрочном хранилище, который может использоваться экземпляром для восстановления предыдущего состояния. Итак, разберем концепцию *состояние*.

Вернемся во вселенную с зомби и посмотрим, какие данные могут понадобиться, чтобы восстановить и продолжить приключение. Данные сильно зависят от конкретной игры, но можно предположить, что в любом случае необходимо сохранить:

- текущий счет и уровни навыков героя;
- имеющееся снаряжение;
- выполненные задачи.

У этих данных есть одно важнейшее свойство: они изменяются в ходе игры. Данные, которые остаются неизменными, пока вы усердно трудитесь над спасением мира (например, карта и внешний вид зомби), включать в сохраненные не нужно.

А теперь вернемся к определению *состояния* в стриминговых системах: это внутренние данные каждого экземпляра, изменяющиеся при обработке событий. Например, в задании контроля за использованием системы каждый экземпляр анализатора использования отслеживает количество обработанных транзакций. Этот показатель изменяется при обработке новой транзакции и становится одной из составляющих состояния. При перезапуске экземпляра счетчик необходимо восстановить.

Хотя концепции контрольных точек и состояния не особенно сложны, следует понимать, что в распределенных системах управление контрольными точками — непростая задача. В системе над обработкой событий могут одновременно работать сотни и тысячи экземпляров. Ядро должно управлять контрольными точками всех экземпляров и следить за тем, чтобы все они были правильно синхронизированы. Сейчас мы временно оставим эту тему и вернемся к ней позже, в главе 10.

Контрольные точки в задании контроля за использованием системы для семантики «не менее одного»

Прежде чем вводить контрольные точки для семантики «не менее одного», необходимо добавить полезный компонент между шлюзом API и заданием контроля за использованием системы — *журнал событий*. На практике этот термин применяется не так часто; здесь он нужен нам для целей книги, но понять его

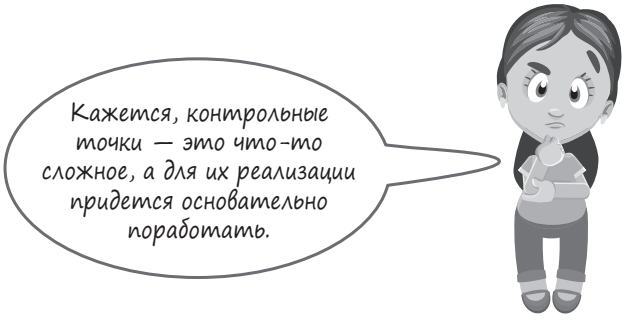
суть несложно. Журнал событий представляет собой очередь событий, в которой каждое событие отслеживается со смещением (или временной меткой). *Потребитель* (или *читатель*) может перейти к конкретному смещению и начать загрузку данных с этого момента. В реальной жизни события могут быть разбиты на несколько *разделов* с независимым управлением смещениями в каждом разделе, но для простоты будем считать, что в системе только одно смещение и один источник транзакций.

Если перед компонентом — источником транзакций находится журнал событий, каждую минуту (или с другим интервалом) экземпляр-источник создает контрольную точку с текущим состоянием — текущим смещением, с которым он работает. При перезапуске задания ядро определяет правильное смещение для экземпляра, к которому необходимо вернуться (*откат*) и начать обработку событий с этой точки. Следует заметить, что события, обработанные экземпляром от времени контрольной точки до момента перезапуска, будут обработаны снова, но в семантике «не менее одного» это нормально.

Контрольные точки и функции управления состоянием

Контрольные точки обладают чрезвычайно широкими возможностями. При выполнении задания с включенными контрольными точками совершается много действий. Вот основные из них.

- Периодически все экземпляры источников должны создать контрольную точку с их текущими состояниями.
- Контрольные точки сохраняются в хранилище (желательно отказоустойчивом).
- Стриминговое задание должно автоматически перезапуститься при обнаружении сбоя.
- Задание должно идентифицировать последние контрольные точки, и каждый перезапущенный экземпляр источника должен загрузить свой файл контрольной точки и восстановить свое предыдущее состояние.
- Объем хранилища ограничен, поэтому старые контрольные точки необходимо удалять для экономии ресурсов.



Если этот список вас пугает, без паники! Действительно, механизм контрольных точек в целом не так прост, а для его запуска необходимо потрудиться. К счастью, основную работу берут на себя стриминговые фреймворки, а владельцу стримингового задания придется побеспокоиться только об одном: о состоянии. А конкретнее — о двух функциях управления состоянием.

- Получение текущего состояния экземпляра. Эта функция будет вызываться периодически.
- Инициализация экземпляра объектом состояния, загруженным по контрольной точке. Функция будет вызвана в процессе запуска стримингового задания.

Если вы предоставите две функции, описанные выше, стриминговый фреймворк незаметно выполнит всю черную работу — например, упаковку состояний в контрольной точке, сохранение их на диске и использование контрольной точки для инициализации экземпляров.

Код управления состоянием в компоненте источника транзакций

Ниже приведен пример кода компонента `TransactionSource` в фреймворке Streamwork:

- Базовый класс `Source` заменяется на `StatefulSource`.
- С новым базовым классом появляется новая функция `getState()`, которая получает состояние экземпляра и возвращает его ядру.
- Еще одно изменение — функция `setupInstance()` получает дополнительный объект `State` для настройки экземпляра после его конструирования (у операторов без состояния его не было).

Классы **Source** *и* **StatefulSource.**

```
public abstract class Source extends Component {
  public abstract void setupInstance(int instance);
  public abstract void getEvents(EventCollector eventCollector);
}

public abstract class StatefulSource extends Component {
  public abstract void setupInstance(int instance, State state);
  public abstract void getEvents(EventCollector eventCollector);
  public abstract State getState();
}
```

Новый объект состояния используется для настройки экземпляра.

Эта новая функция используется для получения состояния экземпляра.

```
class TransactionSource extends StatefulSource {
  MessageQueue queue;
  int offset = 0;
  ......
  public void setupInstance(int instance, State state) {
    SourceState mstate = (SourceState)state;
    if (mstate != null) {
      offset = mstate.offset;
      log.seek(offset);
    }
  }
  public void getEvents(Event event, EventCollector eventCollector) {
    Transaction transaction = log.pull();
    eventCollector.add(new TransactionEvent(transaction));
    offset++;
  }

public State getState() {
    SourceState state = new SourceState();
    State.offset = offset;
    return new state;
  }
}
```

Данные из объекта состояния используются для настройки экземпляра.

Смещение (offset) изменяется при извлечении нового события из журнала событий и его отправке последующим компонентам.

Объект состояния экземпляра содержит текущее значение смещения (offset) из журнала событий.

Ровно один или фактически один?

Для задания контроля за использованием системы не идеальна ни семантика «не более одного», ни семантика «не менее одного», потому что точные результаты не гарантированы, однако они нужны нам для принятия правильных решений. Для достижения этой цели можно выбрать последнюю семантику: «ровно один». Эта семантика гарантирует, что каждое событие будет успешно обработано один и только один раз. Следовательно, результаты с такой семантикой будут точными.

Сначала обсудим, что же имеется в виду под семантикой «ровно один». Чрезвычайно важно понимать, что каждое событие на самом деле не обрабатывается или успешно обрабатывается ровно один раз, как предполагает название. Настоящий смысл в том, что если рассматривать задание как «черный ящик» (иначе говоря, если рассматривать только ввод и вывод, не касаясь внутреннего механизма работы задания), все *выглядит* так, словно каждое событие успешно обрабатывается один и только один раз. Тем не менее если заглянуть внутрь системы, выясняется, что каждое событие может обрабатываться более одного раза. Взгляните на тему этой главы: в ней не случайно речь идет о *семантике доставки*, а не о *семантике обработки*.

Когда ранее в этой главе мы кратко поясняли, что такое семантика, мы упоминали о том, что в некоторых фреймворках используется термин «фактически один». Формально термин «фактически один» может быть более точным, но термин «ровно один» получил широкое распространение; мы решили выбрать его в качестве основного для книги, чтобы вы не путались в будущем.

Если все эти оговорки «выглядит, словно» (или «фактически») все еще вас смущают, это вполне нормально. Чтобы вы лучше поняли, о чем идет речь, сделаем шаг в сторону и немного поговорим о следующей интересной концепции: *идемпотентности*. Хочется надеяться, что она поможет вам лучше понять, что имеется в виду под «фактически один».

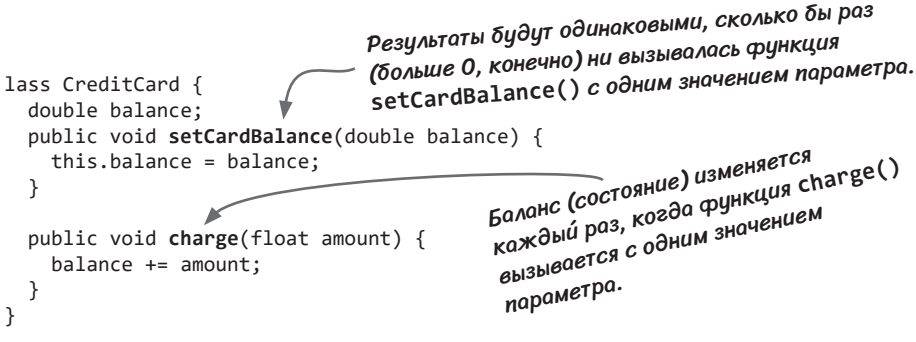
Реальную семантику «ровно один» в высшей степени трудно реализовать в распределенных системах — и это реальность.

Вспомогательная концепция: идемпотентные операции

Термин *идемпотентная операция* (idempotent operation) звучит сложно, верно? Этот термин из области математики и вычислительной теории означает, что сколько бы раз функция ни получала входные данные, ее результат всегда будет неизменным. Иначе: несколько идентичных вызовов операции приводят к *тому же результату*, что и один вызов. Непонятно? Не огорчайтесь. Рассмотрим пример в контексте класса, представляющего кредитную карту.

Этот класс содержит два метода: `setCardBalance()` и `charge()`.

- Функция `setCardBalance()` присваивает балансу карты новое значение, заданное параметром.
- Функция `charge()` увеличивает баланс на заданную величину.




```
lass CreditCard {
  double balance;
  public void setCardBalance(double balance) {
    this.balance = balance;
  }

  public void charge(float amount) {
    balance += amount;
  }
}
```

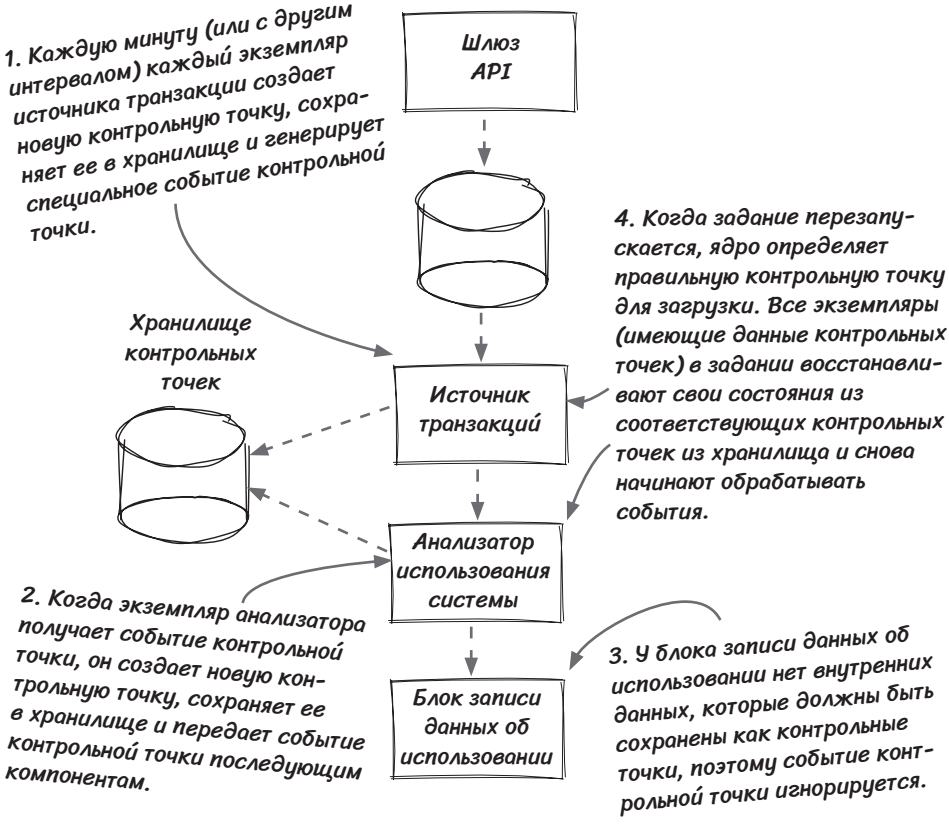
Одна интересная особенность функции `setCardBalance()` заключается в том, что после ее однократного вызова состоянию объекта кредитной карты (балансу) присваивается новое значение. Если функция затем вызывается во второй раз, то балансу снова будет присвоено новое значение — но состояние (баланс) при этом не изменится. Если рассматривать баланс карты, все выглядит так, словно функция вызывается только один раз, потому что вы не можете определить, вызывается она однократно или многократно. Иначе говоря, функция может быть вызвана один или несколько раз, но *фактически* она вызывается однократно, потому что эффект будет одинаковым. Из-за этого функция `setCardBalance()` является идемпотентной операцией.

С другой стороны, функция `charge()` идемпотентной операцией не является. Когда она вызывается только один раз, баланс увеличивается на заданную величину. Если случайно повторить вызов, баланс увеличится снова и объект карты перейдет в состояние ошибки.

Семантика «*ровно один*» в стриминговых системах работает так же, как приведенная выше функция `setCardBalance()`. По состояниям всех экземпляров задания все выглядит так, словно каждое событие обрабатывается ровно один раз, но во внутренней реализации событие может обрабатываться каждым компонентом более одного раза.

Наконец, «ровно один»

Теперь вы знаете реальный смысл семантики и концепцию идемпотентной операции, а также понимаете важность возвращения точных результатов. Вас больше интересует, как работает семантика «ровно один»? Название может показаться фантастическим, но на самом деле все не так сложно. Как правило, семантика «ровно один» поддерживается посредством контрольных точек, которые очень похожи на поддержку «не менее одного». Отличие в том, что контрольные точки создаются как для источников, так и для операторов, чтобы они могли вместе «вернуться во времени» при откате. Обратите внимание: контрольные точки необходимы только для операторов с внутренним состоянием.



Пока все просто? Не торопитесь. В нашем случае состояние экземпляра источника — всего лишь смещение. Но состояние экземпляра оператора может быть намного более сложным, потому что оно зависит от логики. Для операторов состояние может быть простым числом, списком, картой или более сложной структурой данных. Хотя за управление контрольными точками обычно отвечает стриминговое ядро, важно понимать, к каким затратам ресурсов это может привести.

Код управления состоянием в компоненте анализатора использования системы

Чтобы со фреймворком Streamwork компонент `SystemUsageAnalyzer` создавал и использовал состояние экземпляра, необходимо внести изменения, похожие на изменения в `TransactionSource`, представленные выше.

- Базовый класс `Operator` заменяется на `StatefulOperator`.
- Функция `setupInstance()` получает один дополнительный параметр с состоянием.
- Добавляется новая функция `getState()`.

```
public abstract class Operator extends Component {
  public abstract void setupInstance(int instance);
  public abstract void getEvents(EventCollector eventCollector);
  public abstract GroupingStrategy getGroupingStrategy();
}

public abstract class StatefulOperator extends Component {
  public abstract void setupInstance(int instance, State state);
  public abstract void apply(Event event, EventCollector eventCollector);
  public abstract GroupingStrategy getGroupingStrategy();
  public abstract State getState();
}

class SystemUsageAnalyzer extends StatefulOperator {
  int transactionCount;

  public void setupInstance(int instance, State state) {
    AnalyzerState mstate = (AnalyzerState)state;
    transactionCount = state.count;
    ……
  }

  public void apply(Event event, EventCollector eventCollector) {
    transactionCount++;

    eventCollector.add(transactionCount);
  }

  public State getState() {
    AnalyzerState state = new AnalyzerState();
    State.count = transactionCount;
    return state;
  }
}
```

Новый объект состояния используется для настройки экземпляра.

Эта новая функция используется для получения состояния экземпляра.

При конструировании экземпляра объект состояния используется для инициализации экземпляра.

Переменная count изменяется при обработке событий.

Новый объект состояния периодически создается для хранения данных экземпляров.

Фреймворк Streamwork поддерживает низкоуровневый API, показывающий, как работает внутренняя реализация. В наше время большинство фреймворков поддерживает высокоуровневые API — например, функциональные и декларативные. С новыми типами API разрабатываются компоненты, предназначенные для повторного использования, и пользователям не надо беспокоиться о технических подробностях.

Повторное сравнение семантик доставки

У всех семантик доставки существуют свои сценарии использования. Теперь, когда мы рассмотрели все семантики доставки, снова сравним их (в несколько упрощенном виде). Из следующей таблицы видно, что у каждой семантики доставки есть свои плюсы и минусы. Иногда ни один вариант не идеален. В таких случаях необходимо понимать достоинства и недостатки и принимать соответствующие решения. Возможно, вам также придется менять семантику при изменении требований.

Семантика доставки	**Не более одного**	**Не менее одного**	**Ровно один**
Точность	Точность не гарантирована из-за пропуска событий	Точность не гарантирована из-за дублирования событий	(На первый взгляд) гарантирован точный результат
Задержка (при возникновении ошибки)	Устойчивость к сбоям; отсутствие задержки при возникновении ошибок	Чувствительность к сбоям; возможны задержки при возникновении ошибок	Чувствительность к сбоям; возможны задержки при возникновении ошибок
Сложность/использование ресурсов	Очень простая, минимальные затраты	Средняя (в зависимости от реализации)	Высокая, значительные затраты ресурсов
Сложность сопровождения	Низкая	Средняя	Высокая
Пропускная способность	Высокая	Средняя	Низкая
Код	Изменения в коде не требуются	Некоторые изменения в коде	Значительные изменения в коде
Зависимости	Без внешних зависимостей	Без внешних зависимостей (с подтверждениями)	Требуется внешнее хранилище для сохранения контрольных точек

Резонное беспокойство тех, кто размышляет о выборе семантики «не более одного» или «не менее одного» с учетом задержки и эффективности, состоит в том, что точность не гарантирована. Существует популярный подход, чтобы избежать этой проблемы: лямбда-архитектура. В лямбда-архитектуре сопутствующий процесс пакетной обработки выполняется на тех же данных, чтобы получить точные результаты, но с более высокой задержкой. Поскольку в этой главе у нас и так много материала для усвоения, мы поговорим об этом подробнее в главе 10.

Итоги

В этой главе мы рассмотрели новую важную концепцию стриминговых систем: семантику доставки (или гарантию доставки). Для стриминговых заданий вы можете выбирать три типа семантик:

- «*Не более одного*» — каждое событие гарантированно будет обработано не более одного раза. Это означает, что события могут быть пропущены при сбоях в стриминговых заданиях.
- «*Не менее одного*» — событие гарантированно будет обработано стриминговыми заданиями, но может оказаться, что при сбоях некоторые события будут обработаны несколько раз.
- «*Ровно один*» — при такой семантике результат выглядит, как будто каждое событие обрабатывается только один раз. Также используется термин «фактически один».

В этой главе мы обсудили достоинства и недостатки каждой из этих семантик, кратко обсудили важный метод поддержки семантик «не менее одного» и «ровно один» — контрольные точки. Выбирайте наиболее подходящую семантику доставки для своих кейсов.

Упражнения

1. Какую семантику доставки вы выберете при построении следующих заданий? Почему?
   - Поиск наиболее популярных хештегов в Твиттере.
   - Импортирование записей из потока данных в базу данных.
2. В этой главе мы рассмотрели компонент анализатора использования в задании контроля за использованием системы и преобразовали его в идемпотентную операцию. А что вы скажете о компоненте блока записи данных об использовании? Является ли он идемпотентной операцией?

Что дальше?

В главах со 2-й по 5-ю были представлены наиболее распространенные и базовые концепции, которые понадобятся вам, когда вы начнете строить стриминговые системы. В следующей главе мы прервемся и обобщим то, что вы узнали к настоящему моменту. После этого перейдем к более сложным темам, таким как оконные вычисления и операции соединения.

Краткий обзор стриминговых систем и взгляд в будущее

6

В этой главе

- ✓ Обзор изученных концепций.
- ✓ Знакомство с более сложными концепциями, которые будут рассматриваться в части 2.

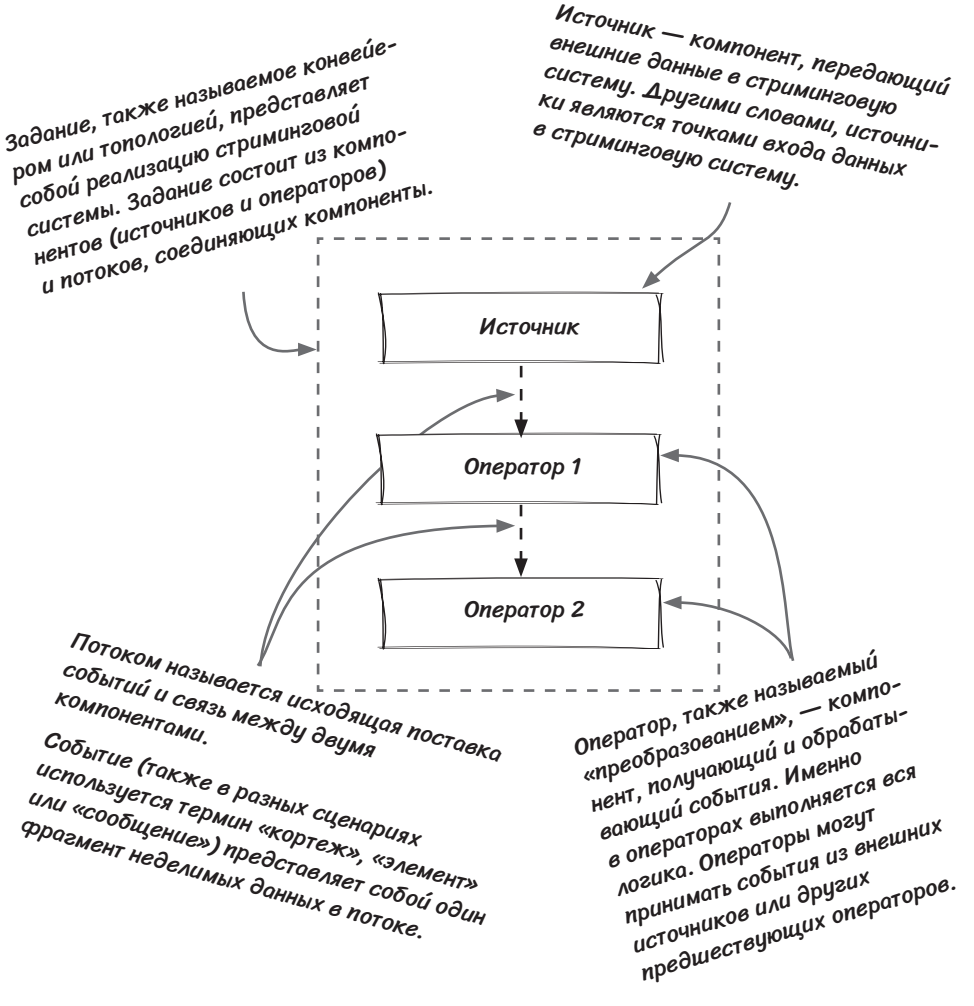
«Технология позволяет людям взять под контроль все что угодно, кроме самой технологии».

Джон Тьюдор

После изучения основных концепций стриминговых систем в предыдущих главах пришло время сделать небольшой перерыв и повторить их. Здесь также будет вкратце представлен материал последующих глав, и вы подготовитесь к новым приключениям.

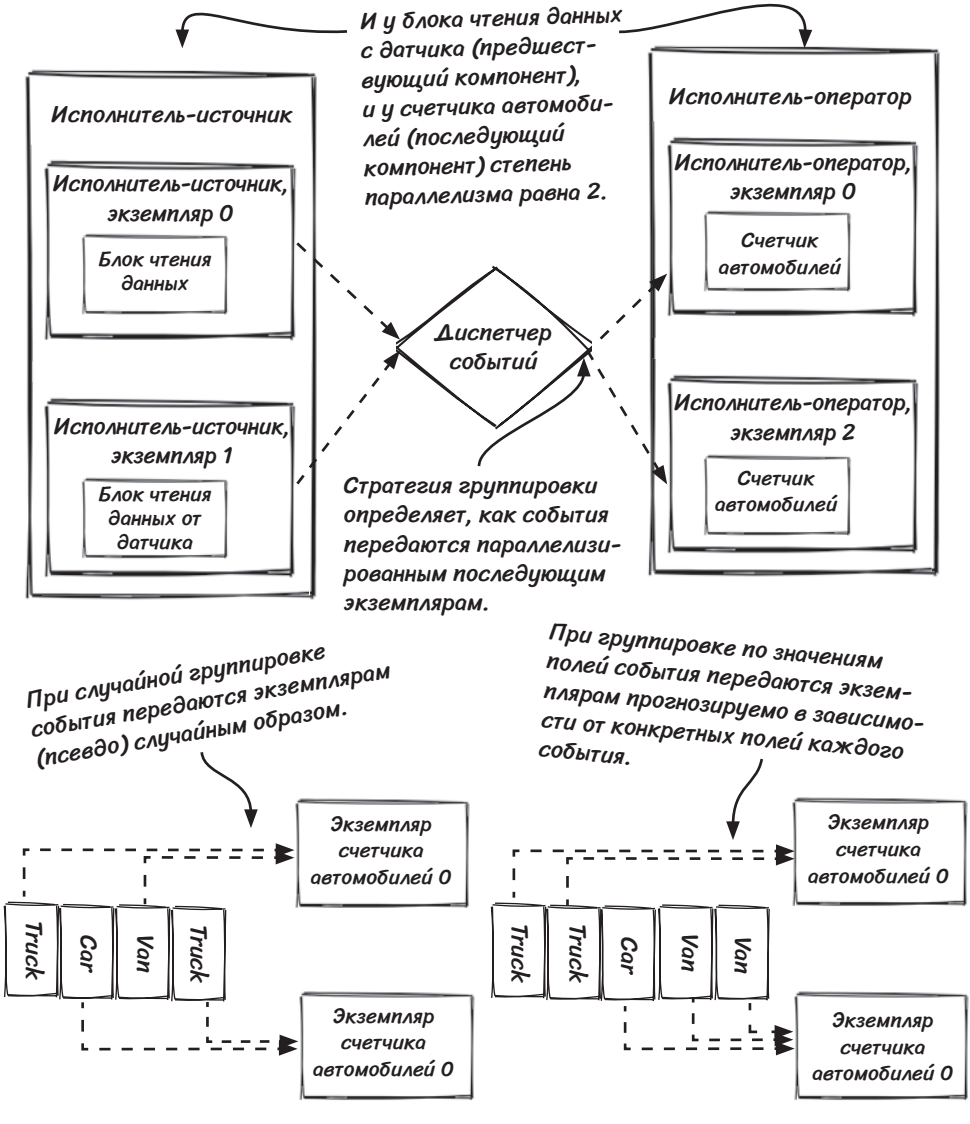
Компоненты стриминговых систем

Задание представляет собой приложение, которое загружает входные данные и обрабатывает их. Все стриминговые задания состоят из четырех компонентов: *событие*, *поток*, *источник* и *оператор*. Следует помнить, что эти понятия в разных фреймворках могут обозначаться разными терминами.



Параллелизация и группировка событий

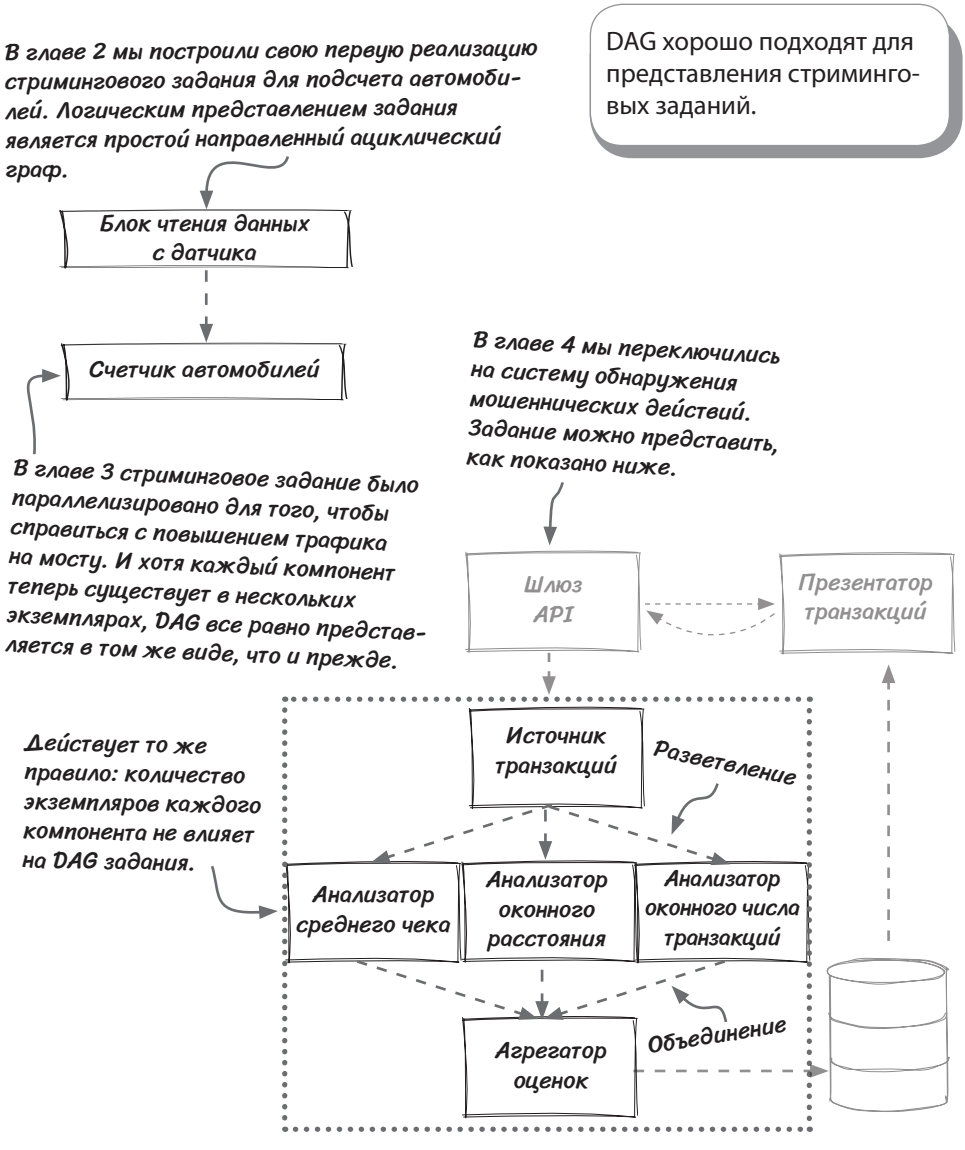
В реальном мире вариант с обработкой событий по очереди обычно неприемлем. *Параллелизация* важна для решения масштабных проблем (то есть для обработки большей нагрузки). При использовании параллелизации необходимо понимать, как организовать передачу событий со *стратегией группировки*.



DAG и стриминговые задания

DAG (направленный ациклический граф) используется для представления логической структуры стримингового задания и направлений передачи данных в нем. В более сложных стриминговых заданиях, таких как система обнаружения мошеннических действий, один компонент может иметь несколько предшествующих компонентов (*объединение*) и/или последующих компонентов (*разветвление*).

DAG хорошо подходят для представления стриминговых заданий.



Семантика (гарантия) доставки

Разобравшись с основными компонентами стриминговых заданий, мы вернулись к практической задаче. Какие требования действуют? Что важно для решения задачи? Пропускная способность, задержка и/или точность?

После выяснения требований необходимо соответствующим образом настроить семантику доставки.

Существуют три варианта семантики доставки:

- «*Не более одного*» — стриминговые задания обрабатывают события без гарантий успешной обработки.
- «*Не менее одного*» — стриминговые задания гарантируют, что каждое событие будет успешно обработано по крайней мере один раз, без гарантии количества его обработок.
- «*Ровно один*» — стриминговые задания гарантируют, что каждое событие будет успешно обработано один и только один раз (по крайней мере внешне). В некоторых фреймворках используется термин «*фактически один*».

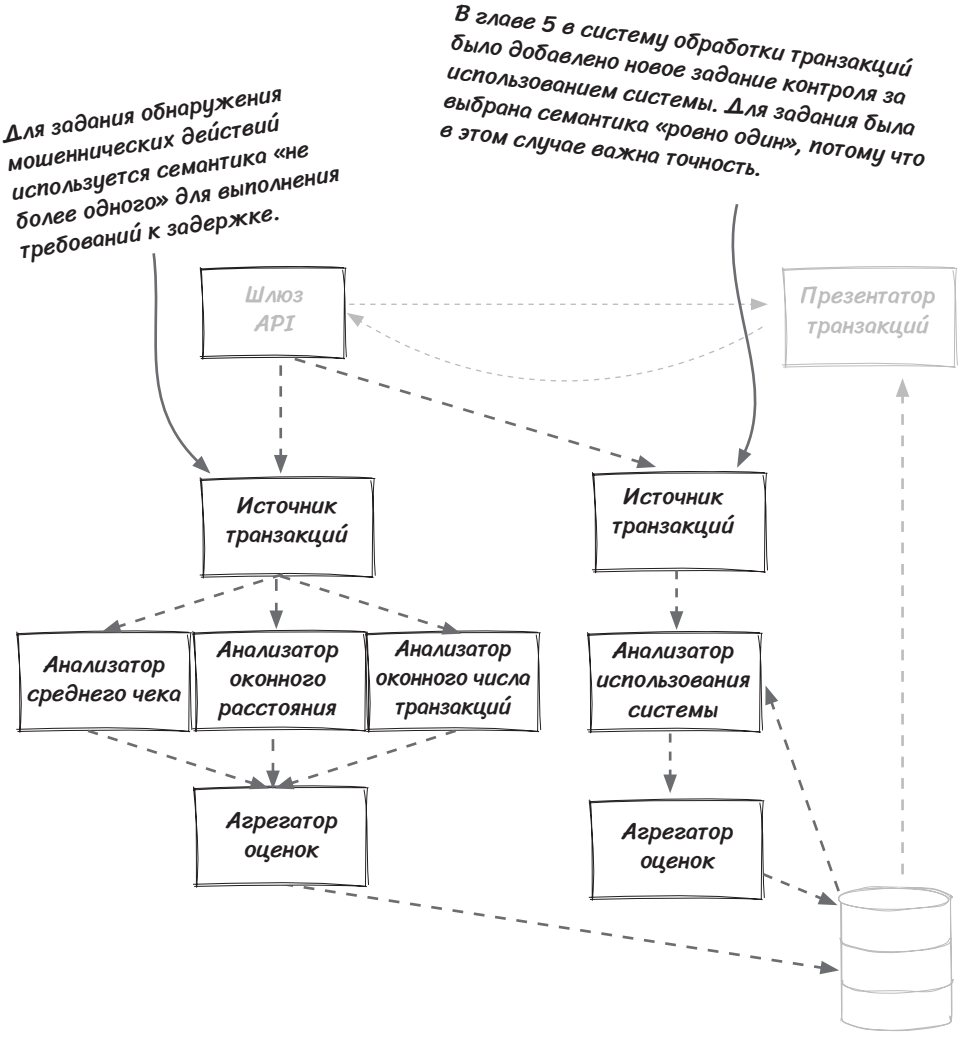
Семантика «ровно один» гарантирует точные результаты, но существуют определенные затраты, которые нельзя игнорировать, такие как задержка и сложность. Важно понимать, какие требования критичны в каждом случае, чтобы выбрать правильный вариант.

Семантика доставки в системе обнаружения мошеннических действий с кредитными картами

В главе 5 в систему обнаружения мошеннических действий с кредитными картами было добавлено новое задание для контроля за использованием системы. Оно предоставляет данные об использовании всей системы в реальном времени. Задание обнаружения мошенничества и новое задание имеют разные требования:

- Для исходного задания обнаружения мошенничества важнее низкая задержка.
- Для нового задания контроля за использованием системы важнее точность.

В результате для них были выбраны разные семантики доставки.



Что дальше?

До сих пор мы рассматривали базовые концепции стриминговых систем. Эти концепции должны помочь вам начать строить стриминговые задания для различных целей в выбранном фреймворке.

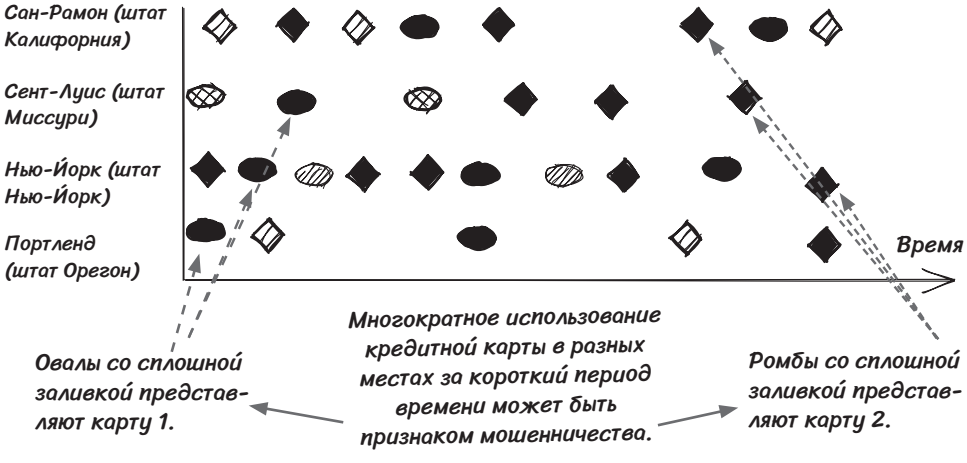
Но вы узнали далеко не все, что нужно знать о стриминговых системах! Когда вы начнете решать более масштабные и сложные задачи, то с большой вероятностью столкнетесь со сценариями, требующими более глубокого знания стриминговых систем. В главах части II будут рассматриваться более сложные темы:

- Оконные вычисления.
- Соединение данных в реальном времени.
- Обратное давление.
- Вычисления с состоянием и без состояния.

Для базовых концепций, которые изучались в предыдущих главах, важен порядок изложения материала, так что каждая глава строилась на материале предыдущих глав. Однако во второй части книги главы более самостоятельны, так что их можно читать либо по очереди, либо в любом порядке на ваше усмотрение. Чтобы вам было проще выбрать, с чего начать, приведем краткий обзор материала следующих глав.

Оконные вычисления

До сих пор события в примерах обрабатывались последовательно. Однако в задании обнаружения мошеннических действий анализаторы зависят не только от *текущего* события, но и от информации о том, когда, где и как *недавно* использовалась карта, — эта информация помогает выявить несанкционированное использование карт. Например, анализатор оконного расстояния выявляет попытки мошенничества, обнаруживая операции с кредитной картой, удаленные на большие расстояния, но совершенные за короткий период времени. Как строить стриминговые системы для решения подобных задач?



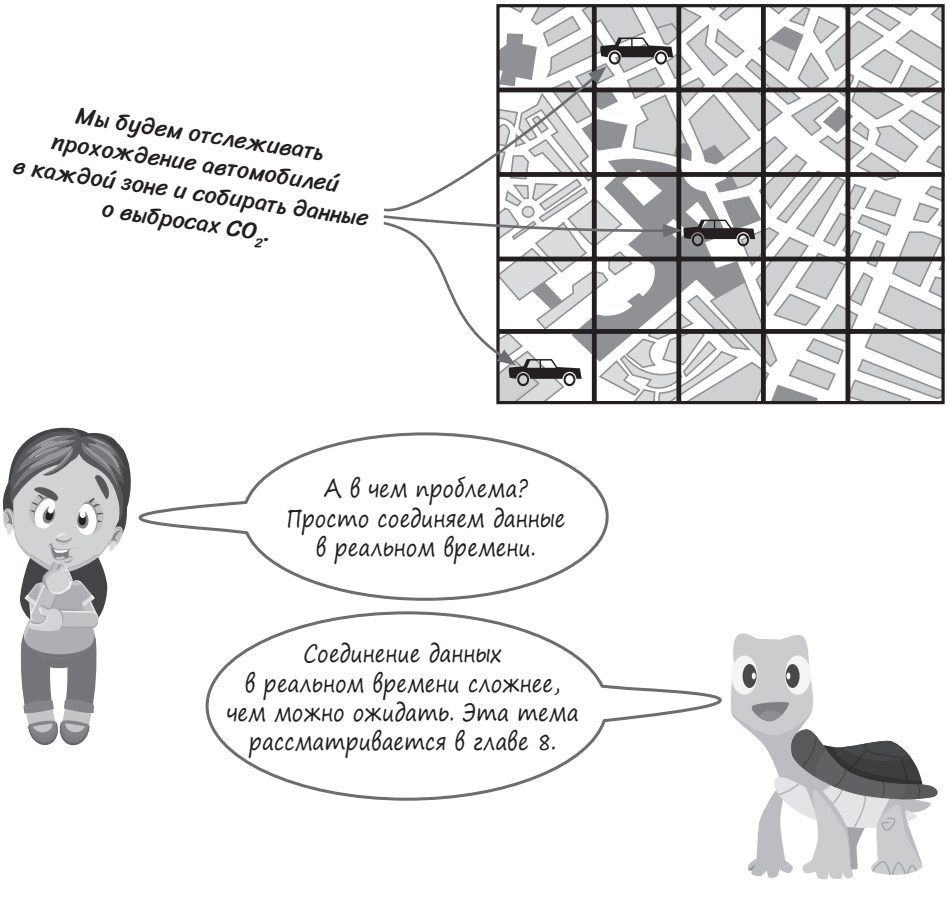
Чтобы сегментировать события на отдельные наборы для обработки, в стриминговых системах могут потребоваться *оконные вычисления*. В главе 7 будут рассмотрены различные оконные стратегии с анализатором оконного расстояния в заданиях обнаружения мошеннических действий.

Кроме того, у оконных вычислений часто существуют ограничения, которые играют важную роль для этого анализатора и многих других прикладных проблем. В главе 7 также рассматривается популярный прием — использование хранилищ «ключ — значение» (систем баз данных, похожих на словари) для реализации оконных операторов.

> В стриминговых системах оконные операторы обрабатывают наборы событий (вместо отдельных событий).

Соединение данных в реальном времени

В главе 8 мы построим новую систему для отслеживания в реальном времени выбросов CO_2 всех автомобилей Кремниевой долины. Автомобили ежеминутно передают информацию о своей модели и местонахождении. События соединяются с другими данными для генерирования карты выбросов CO_2 в реальном времени.

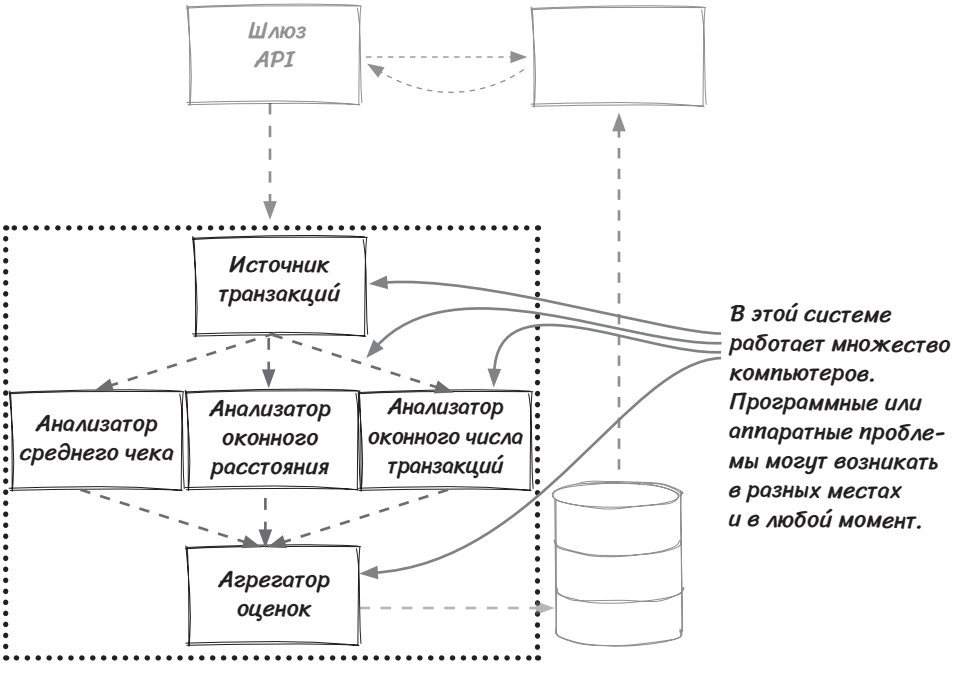


Для людей, уже работавших с базами данных, концепция *соединения* (join) должна быть знакома. Она используется при обращении к данным из нескольких таблиц. В стриминговых системах существует похожий оператор соединения со своими особенностями, он также рассматривается в главе 8. Обратите внимание: соединение — тип объединения потоков, который упоминался (но подробно не рассматривался) в главе 4.

Обратное давление

После запуска стримингового задания для обработки данных вы столкнетесь (хочется надеяться, что не сразу) с проблемой: компьютеры ненадежны! Точнее говоря, компьютеры по большей части надежны, но обычно стриминговые системы работают годами и в них возникают самые разные проблемы.

Ваша команда получила запрос от банка — необходимо проанализировать систему обнаружения мошеннических действий и составить отчет о надежности системы. А конкретно, перестанет ли работать задание при возникновении проблем с компьютерами или сетью и что с результатами — они будут неточными или их вообще не будет? Это разумное требование, так как в разработку системы вложены значительные средства. Собственно, этот вопрос важен даже без запроса от банка, не так ли?

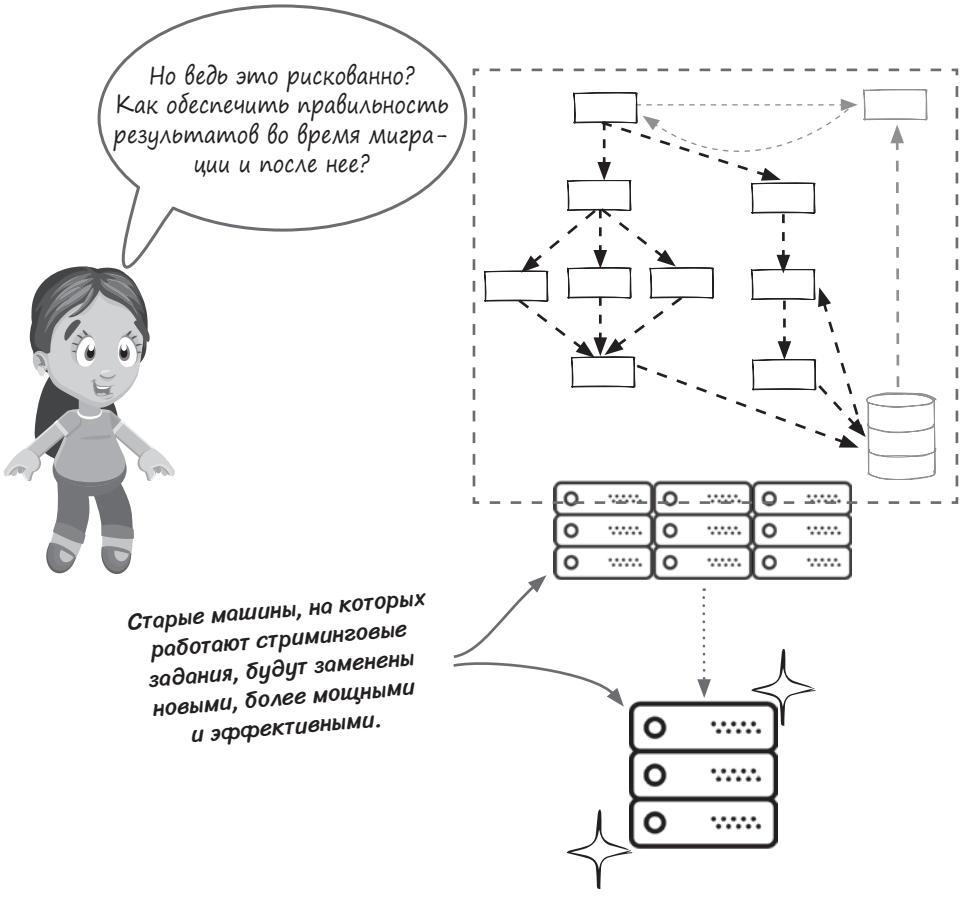


Обратное давление (backpressure) — стандартный защитный механизм, поддерживаемый многими стриминговыми фреймворками. Процессы временно

замедляются и пытаются дать системе возможность восстановиться от возникших проблем, таких как временные сетевые проблемы или внезапные выбросы трафика, создающие чрезмерную нагрузку на компьютеры. Иногда потеря событий лучше, чем замедление. Обратное давление — полезный инструмент для построения надежных систем. В главе 9 вы увидите, как стриминговые ядра обнаруживают и решают проблемы при помощи обратного давления.

Вычисления с состоянием и без состояния

Обслуживание играет важную роль во всех компьютерных системах. Чтобы сократить затраты и повысить надежность кода, Сид решил перенести стриминговые задания на новое, более эффективное оборудование. Переход создает серьезные трудности с обслуживанием, и вы должны действовать осторожно, чтобы все работало без сбоев.



В главе 5, посвященной *семантике доставки*, осталась одна нераскрытая тема: компоненты с состоянием (stateful components). Мы кратко обсудили, что такое компоненты с состоянием и как они используются в семантике доставки «не менее одного» и «ровно один». Тем не менее важно понимать плюсы и минусы разных вариантов, чтобы принимать более эффективные технические решения при построении и обслуживании стриминговых систем.

В главе 10 внутренняя работа компонентов с состоянием рассматривается более подробно. Также будут представлены альтернативные решения, которые позволят избежать некоторых затрат и ограничений.

Часть II
Движемся дальше

Во второй части книги мы углубимся в теорию, рассматривая реализацию более сложных задач в стриминговых системах без привязки к конкретному фреймворку. Глава 7 показывает, как разбить непрерывный поток данных на содержательные блоки, а в главе 8 описан процесс соединения данных в реальном времени. В главе 9 вы узнаете, как в стриминговых системах происходит восстановление после сбоев, а в главе 10 погрузитесь в сложности управления состоянием в стриминговых системах в реальном времени. Наконец, глава 11 подводит краткий итог изложенного в книге, а также дает рекомендации о том, что делать после прочтения.

7 Оконные вычисления

В этой главе

- ✓ Стандартные оконные стратегии.
- ✓ Временные метки в событиях.
- ✓ Оконный предел и поздние события.

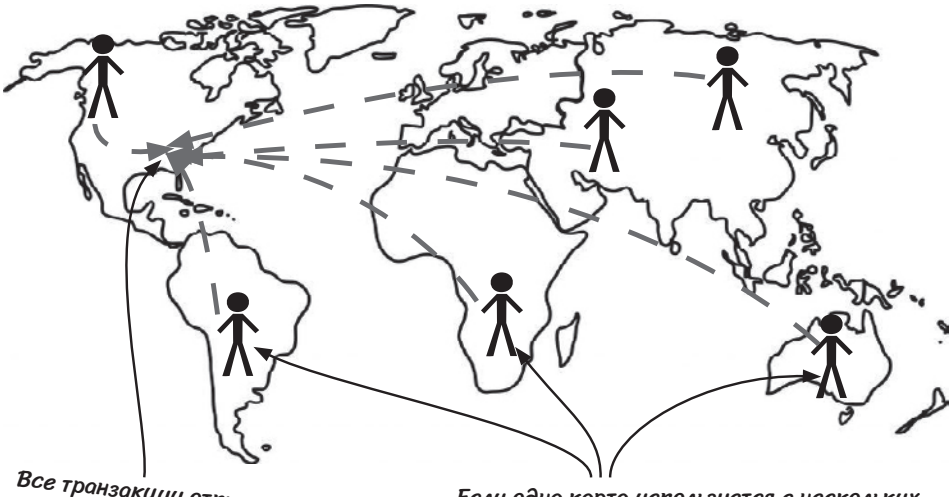
«Период устойчивого внимания компьютера не длиннее его шнура питания».

Автор неизвестен

В предыдущих главах мы построили стриминговое задание для обнаружения мошеннических транзакций по кредитным картам. Существует множество анализаторов, использующих разные модели, но основная идея заключается в сравнении транзакции с предыдущими операциями по той же карте. Оконные технологии предназначены именно для решения подобных задач, и в этой главе мы займемся оконной поддержкой в стриминговых системах.

Сегментация данных в реальном времени

С ростом популярности нового продукта он стал привлекать внимание злоумышленников. Группа хакеров запустила новую мошенническую схему на бензоколонках. Схема работает так: хакеры захватывают данные карты жертвы и дублируют ее на нескольких новых физических кредитных картах. Вновь созданные поддельные карты распространяются среди других участников группы, и одновременно совершаются покупки бензина по одной кредитной карте в нескольких точках мира. Хакеры надеются, что при одновременном снятии средств владелец карты этого не заметит, пока не будет слишком поздно. Результат — бесплатный бензин. Зачем затевать мошенничество в глобальном масштабе, чтобы бесплатно заправить машину? Не спрашивайте нас.



Все транзакции отправляются системе обнаружения мошеннических действий, которая работает здесь.

Если одна карта используется в нескольких точках мира, весьма вероятно, что эти транзакции являются мошенническими.

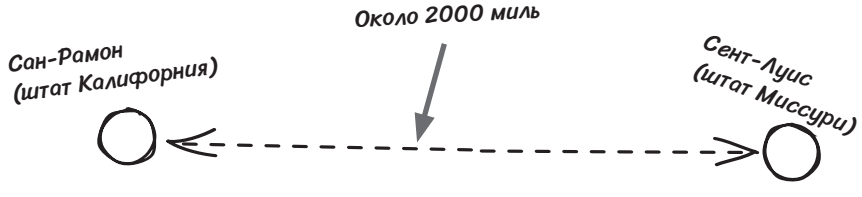
Как предотвратить мошенничество?

Мы будем использовать круглые числа для упрощения математических вычислений. Также будем считать, что максимальная скорость перемещения человека на самолете составляет 500 миль в час. К счастью, наша команда уже продумала возможность подобных мошеннических схем.

Анализ задачи

Мы пытаемся решить две задачи. Сначала система ищет перемещения на значительные расстояния для одной кредитной карты. Затем — значительные перемещения при использовании для нескольких кредитных карт. В первом сценарии конкретные транзакции по кредитным картам помечаются как мошеннические, а во втором продавцы (бензоколонки) помечаются как находящиеся под атакой со стороны опасных злоумышленников.

Из-за ограничения скорости перемещения (500 миль в час) можно уверенно сказать, что человек не сможет физически расплатиться картой в Сан-Рамоне, штат Калифорния, а через два часа — в Сент-Луисе, штат Миссури, потому что расстояние между этими населенными пунктами человек не может преодолеть за 2 часа.

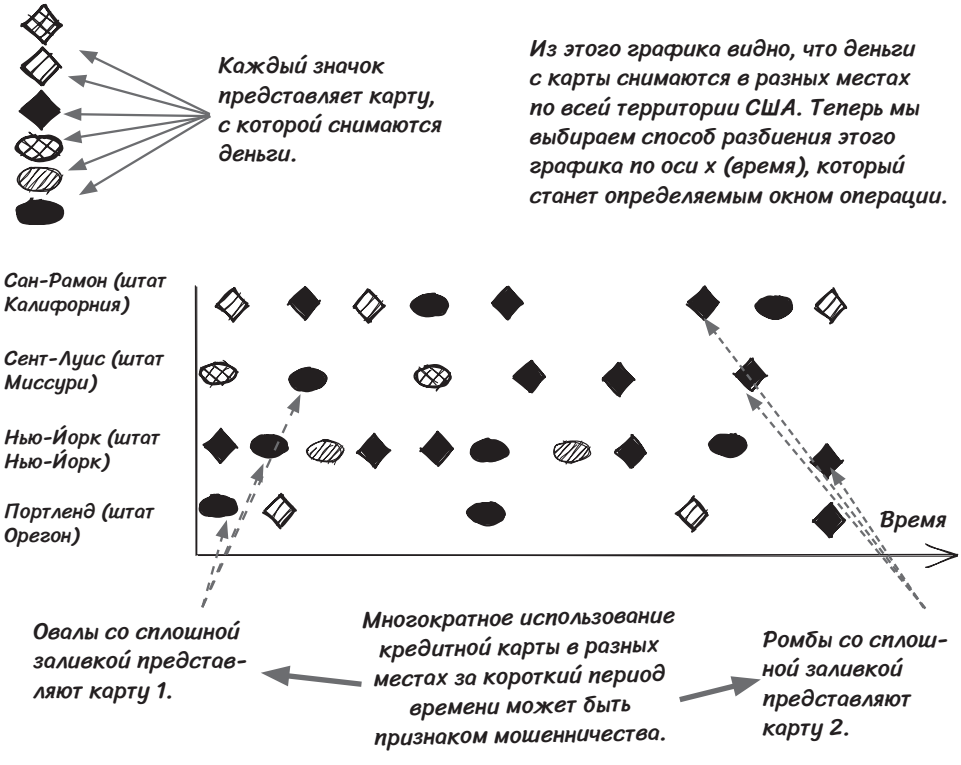


Наша формула:

```
final double maxMilesPerHour = 500;
final double distanceInMiles = 2000;
final double hourBetweenSwipes = 2;

if (distanceInMiles > hourBetweenSwipe * maxMilesPerHour) {
  // Транзакция помечается как потенциально мошенническая
}
```

Анализ задачи (продолжение)

Мошенники хотят проводить масштабные атаки по всему миру, чтобы одновременно заправляться за чужой счет. А значит, важно отслеживать не только поведение одной кредитной карты в системе, но и поведение системы в целом. При проведении таких масштабных атак необходимо иметь возможность блокировать операции по атакованным картам, чтобы повысить безопасность системы. На приведенной ниже диаграмме для примера используются города США, в которых могут проводиться операции по карте.



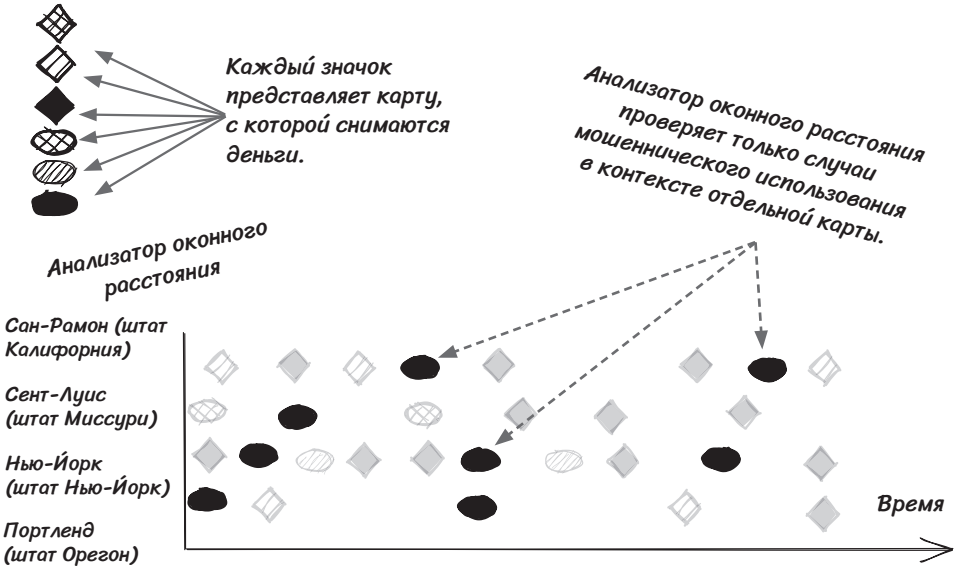
Подобные мошеннические схемы можно предотвращать двумя способами:

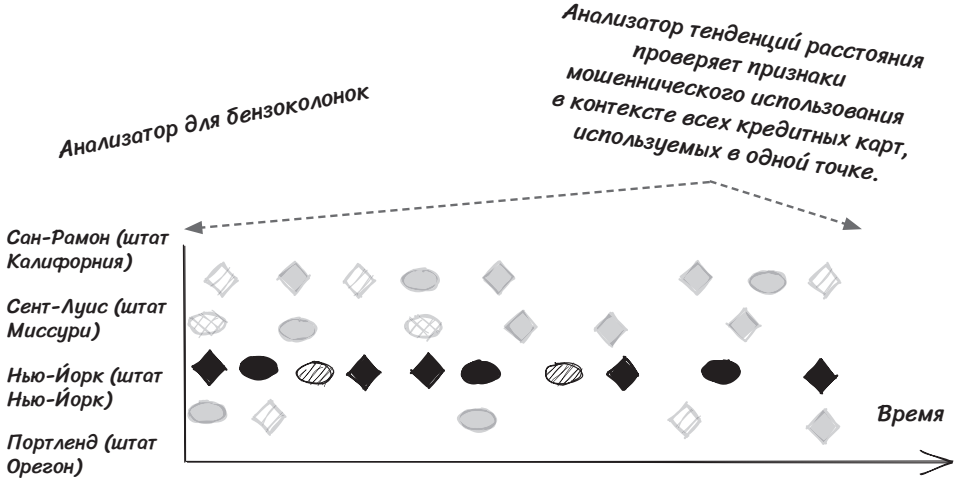
- Блокировать снятие средств с отдельных кредитных карт.
- Блокировать обработку кредитных карт на бензоколонках.

Но какими инструментами в стриминговой системе можно воспользоваться для обнаружения мошеннической активности?

Два разных контекста

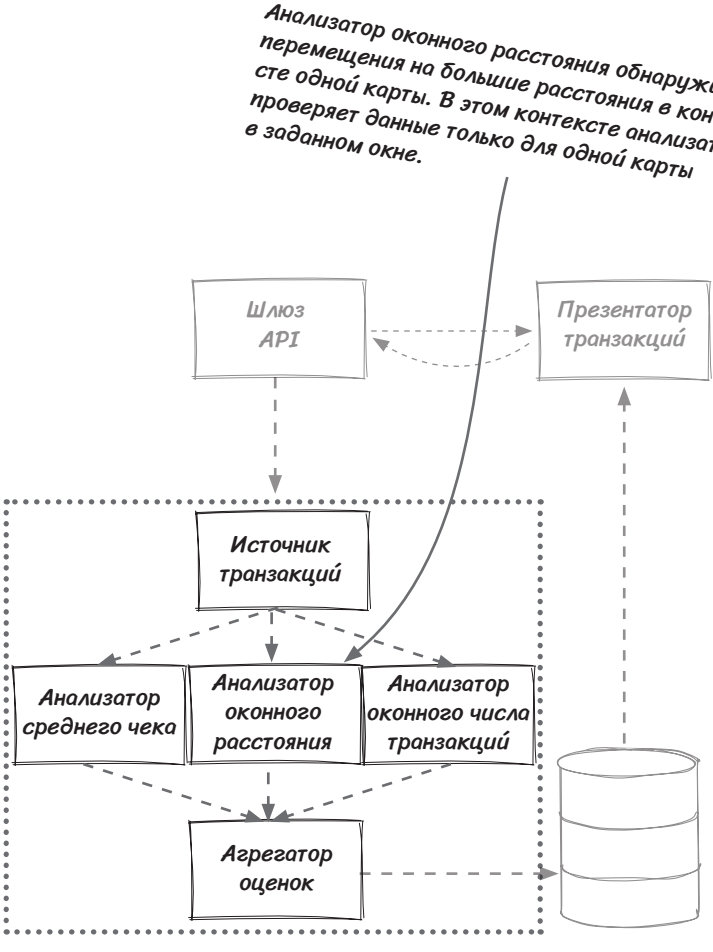
Чтобы реализовать два разных способа предотвращения мошенничества, рассмотрим график с предыдущей страницы — из него видно, как можно разделить контекст. Помните, что анализатор оконного расстояния ищет признаки мошенничества в контексте одиночных кредитных карт, а новый анализатор работает в контексте торговых точек.





Окна в задании обнаружения мошеннических действий

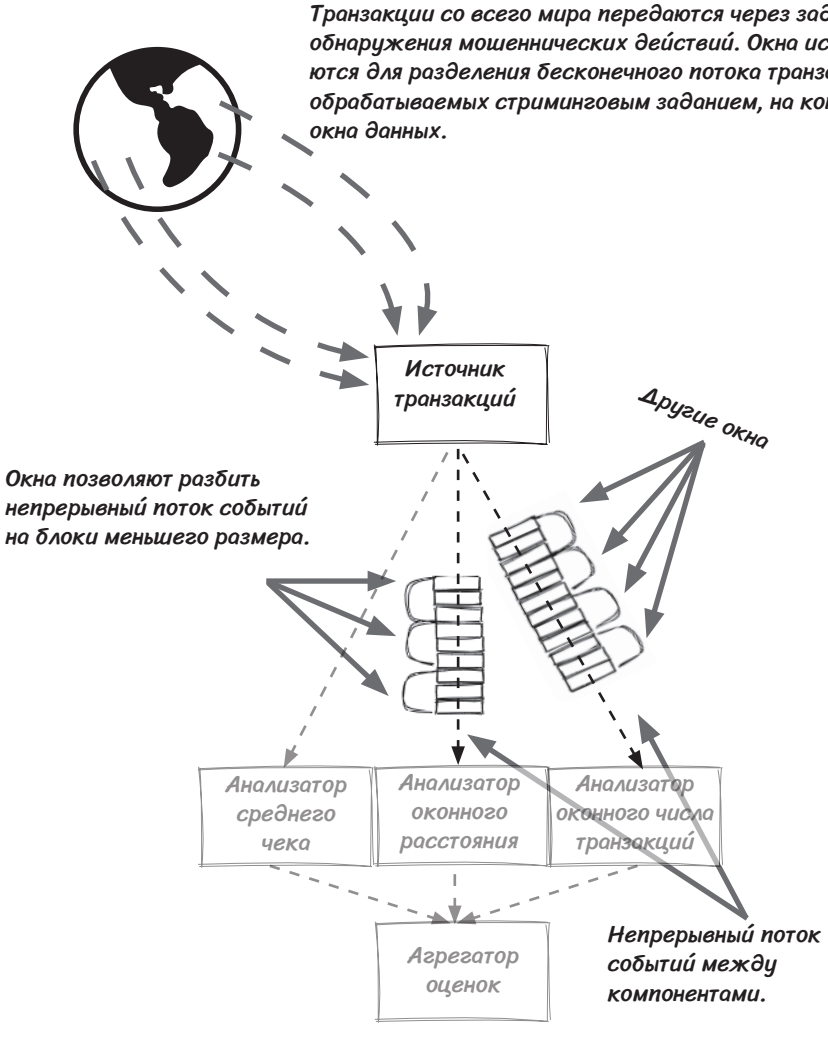
Многие компоненты-анализаторы в задании обнаружения мошеннических действий используют ту или иную разновидность *окна* (об этом подробнее чуть позже) для сравнения текущей транзакции с предыдущими. В этой главе мы остановимся на *анализаторе оконного расстояния* (windowed proximity analyzer), который обнаруживает быстрые перемещения кредитных карт в пространстве. Задачу для бензоколонок оставим нашим читателям для самостоятельной работы.



Что именно называется окном?

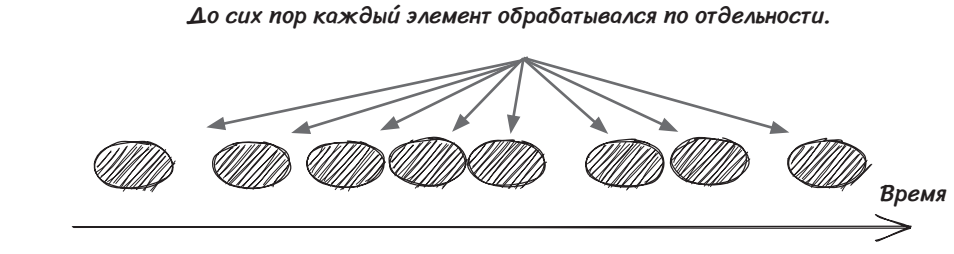
Так как транзакции по кредитным картам постоянно проходят через систему, создание точек отсечения или сегментов данных для обработки может оказаться непростой задачей. В конце концов, как выбрать окончание чего-то потенциально бесконечного — такого, как поток данных?

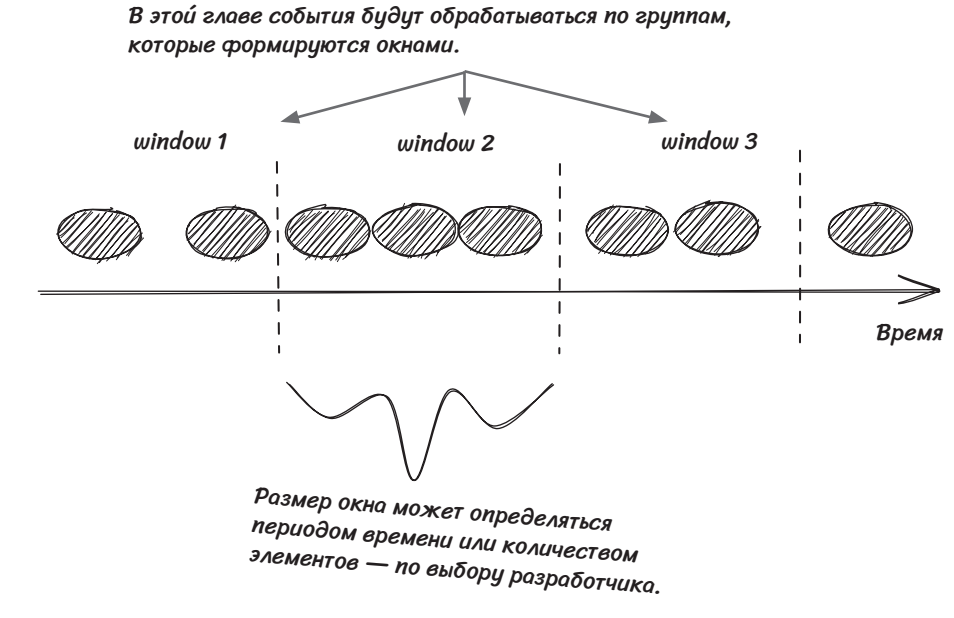
Использование окон в стриминговых системах позволяет разработчикам разбить бесконечный поток событий на части для обработки. Следует заметить, что сегментация может быть основана как на времени, так и на количестве событий. Позднее мы будем использовать окна, основанные на времени, так как они лучше соответствуют рассматриваемым сценариям.



Подробнее об окнах

До сих пор в нашей книге все, что происходило со стриминговыми системами, совершалось на уровне отдельных событий. Во многих случаях такой подход хорошо работает, но у него также могут быть свои ограничения, проявляющиеся в более сложных задачах. В других ситуациях события удобно группировать для обработки по некоторому критерию. Следующие диаграммы помогут вам чуть больше узнать о базовых концепциях использования окон.



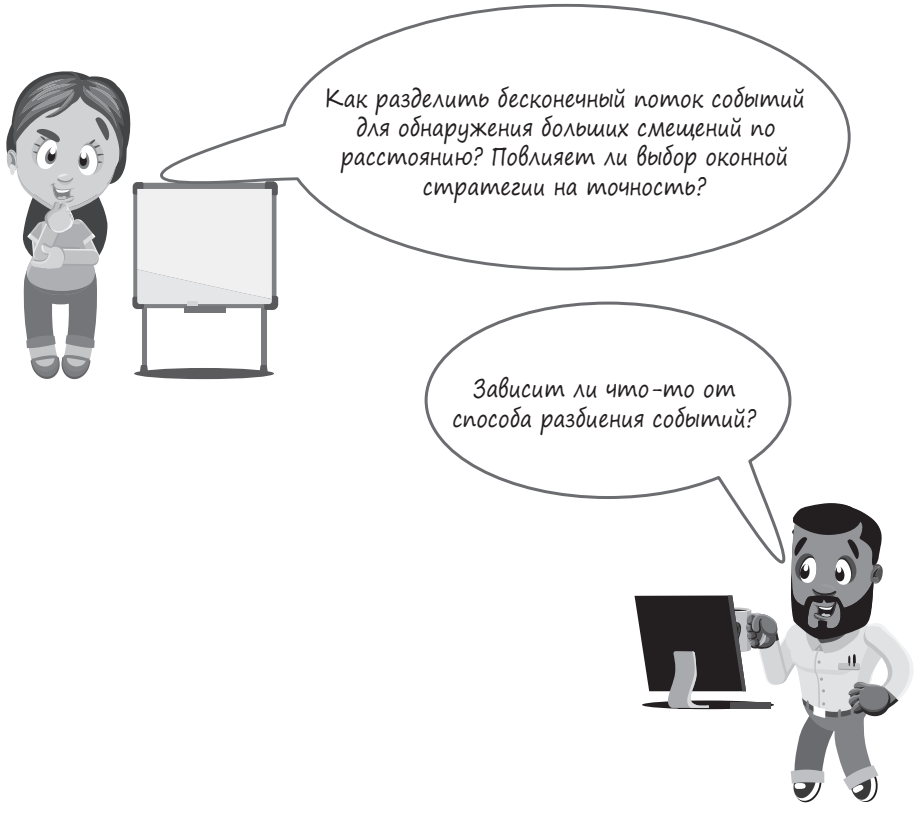


Новая концепция: оконная стратегия

Разобравшись с тем, как используются окна в стриминговых системах, рассмотрим способы группировки событий с применением *оконной стратегии*. Мы разберем три разных типа оконных стратегий и обсудим различия между ними в анализаторе оконного расстояния. Существуют три типа оконных стратегий:

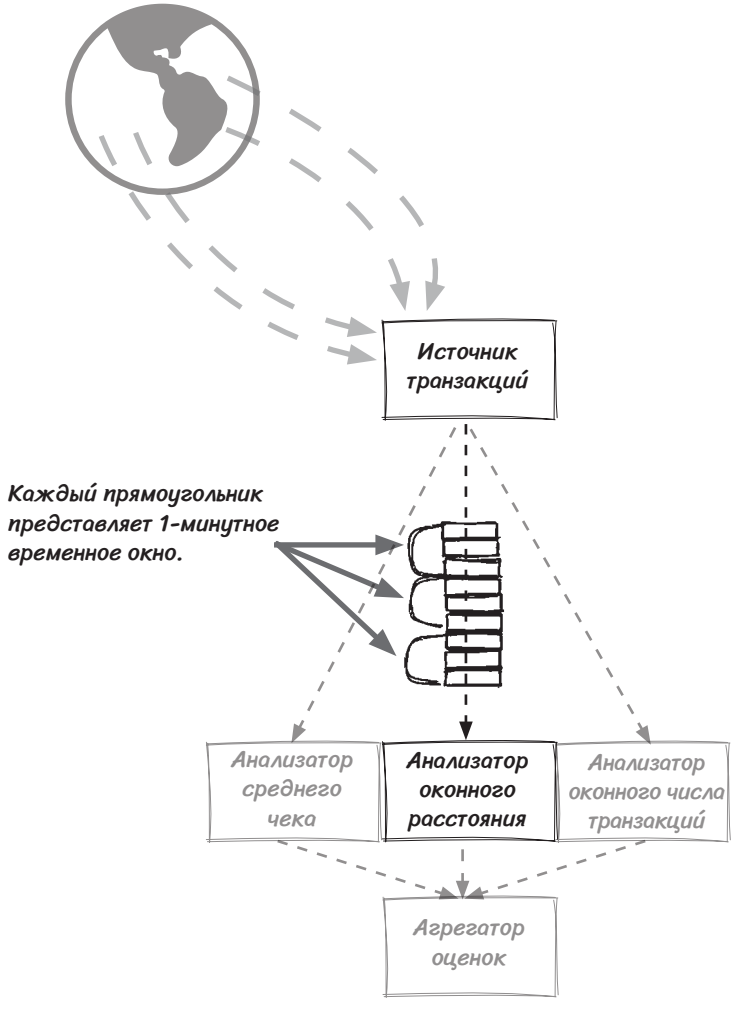
- Фиксированное окно.
- Скользящее окно.
- Сеансовое окно.

Обычно при постановке задачи нет жестких требований к выбору оконной стратегии (способа группировки событий). Вам потребуется пообщаться с другими техническими специалистами и владельцами продуктов в команде, чтобы принять оптимальное решение для конкретной задачи.



Фиксированные окна

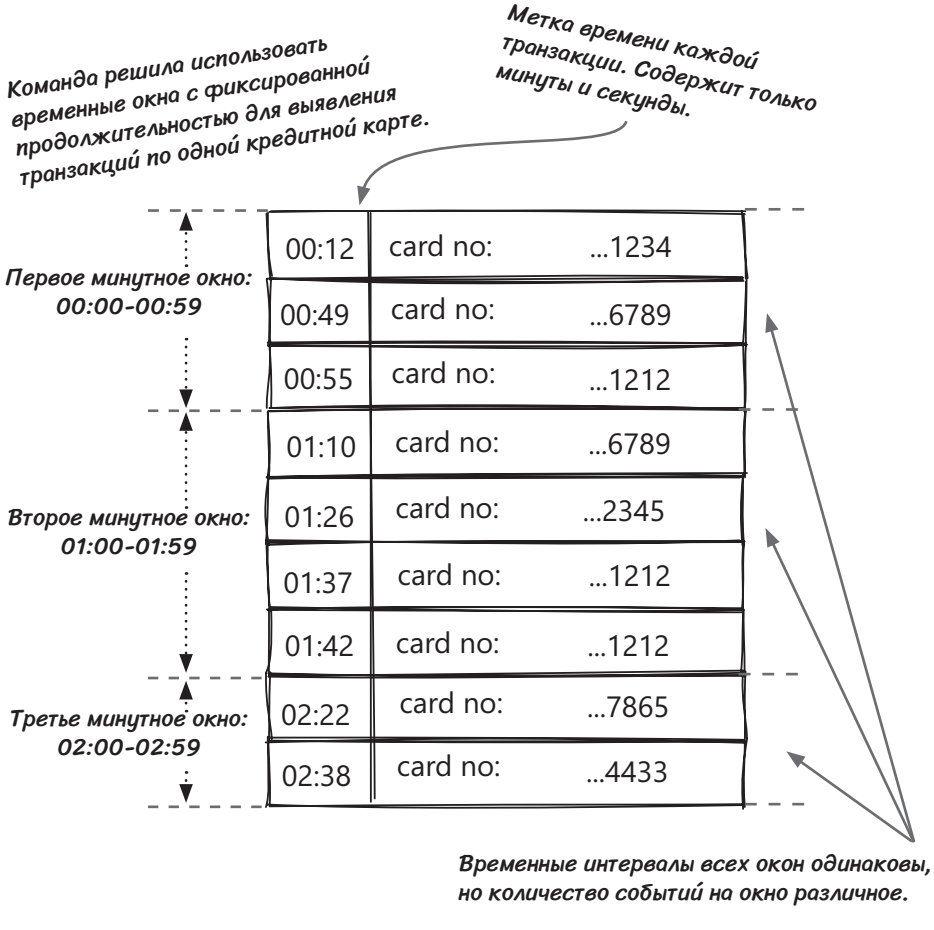
Первая и самая простая разновидность окон — *фиксированное окно*. Иногда их еще называют *переворачивающимися окнами* (tumbling windows). События, получаемые от начала до конца каждого окна, группируются в пакет, который обрабатывается как единое целое. Например, если настроить фиксированное временное окно с продолжительностью в 1 минуту (*минутное* окно), все события этого окна будут группироваться для обработки. Фиксированные окна просты и прямолинейны, они очень полезны во многих ситуациях. Вопрос в том, подойдут ли они для анализатора оконного расстояния?



Фиксированные окна в анализаторе оконного расстояния

В следующем примере рассматривается применение фиксированного окна для поиска повторяющихся списаний с одной карты. Для простоты мы используем минутные окна, чтобы понять, как будет выглядеть каждая группа событий. Цель — выявление повторяющихся транзакций по каждой кредитной карте внутри минутного окна. Другие аспекты (например, логику максимального перемещения в 500 миль в час) рассмотрим позже.

Важно заметить, что использование фиксированного временного окна означает лишь то, что временной интервал фиксирован. В каждом окне может быть больше или меньше событий в зависимости от того, сколько событий проходит через задание.

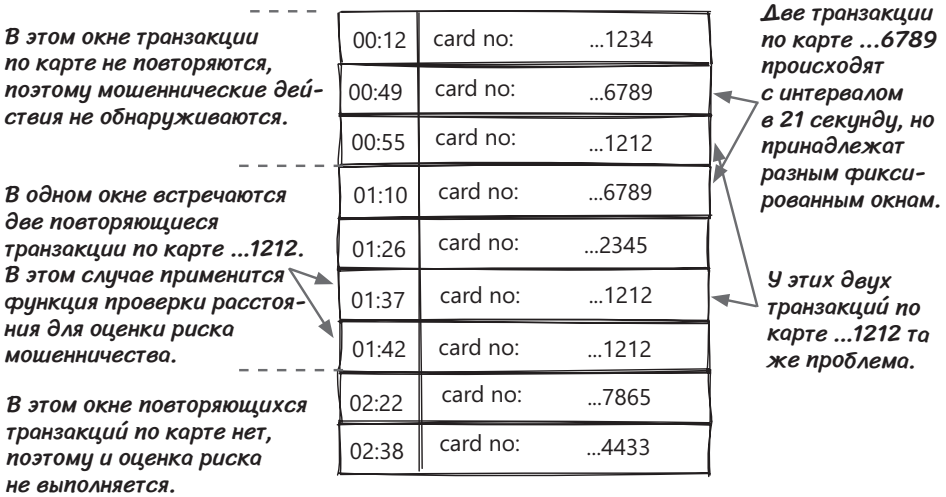


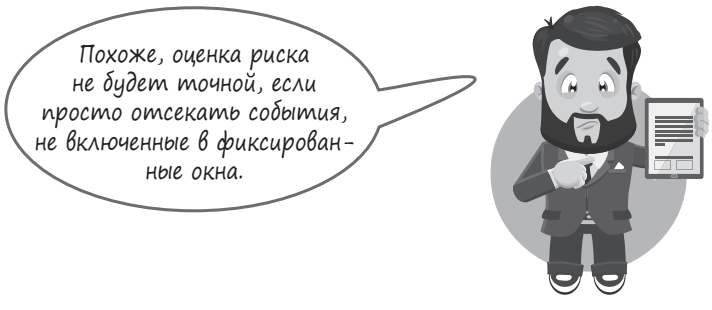
Обнаружение попыток мошенничества в фиксированном временном окне

Посмотрим, как будет вести себя анализатор расстояния при использовании фиксированных временных окон. Мы ограничили количество транзакций на окно, чтобы не усложнять описание основ работы с окнами.

Если внимательно присмотреться к диаграмме, становится ясно, как фиксированные временные окна влияют на потенциальные оценки рисков. С фиксированным временным окном вы просто отсекаете другие транзакции, проходящие через систему, даже если они находятся всего за секунду от границы окна. Как вы думаете, позволяет ли такой тип окон выявлять попытки мошенничества с максимальной точностью?

Нет, фиксированное временное окно в этом случае не идеально. Если две транзакции по одной карте происходят с интервалом всего в несколько секунд, но попадают в два разных временных окна (как две транзакции по карте ...6789), к ним нельзя будет применять функцию проверки расстояния.



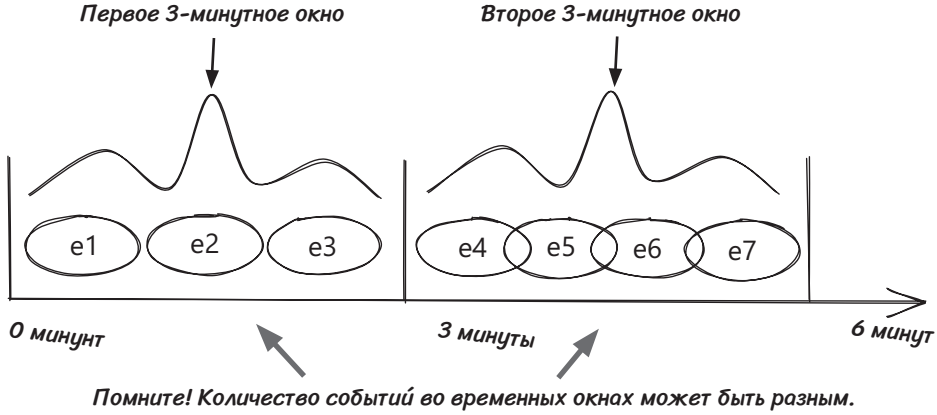


Фиксированные окна: время и количество

Прежде чем переходить к следующей оконной стратегии, выделим две разновидности фиксированных окон:

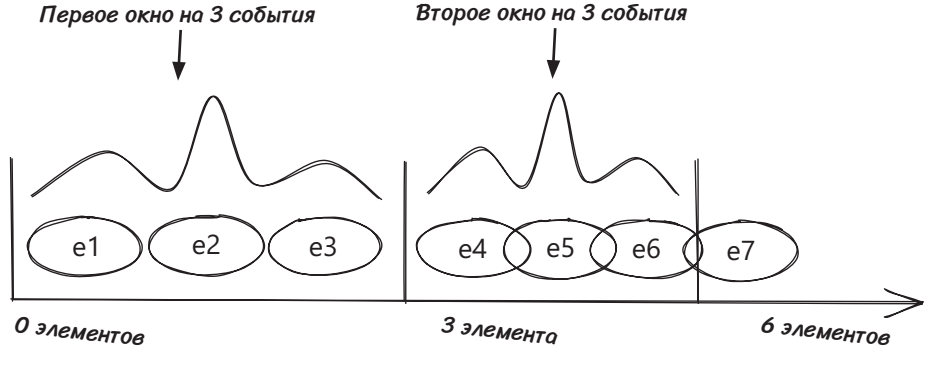
- *Временные окна* определяются неизменяемым интервалом времени.
- *Количественные окна* определяются неизменяемым количеством обработанных событий.

Временные окна. В данном случае интервал составляет 3 минуты. Количество событий в каждом окне может различаться.



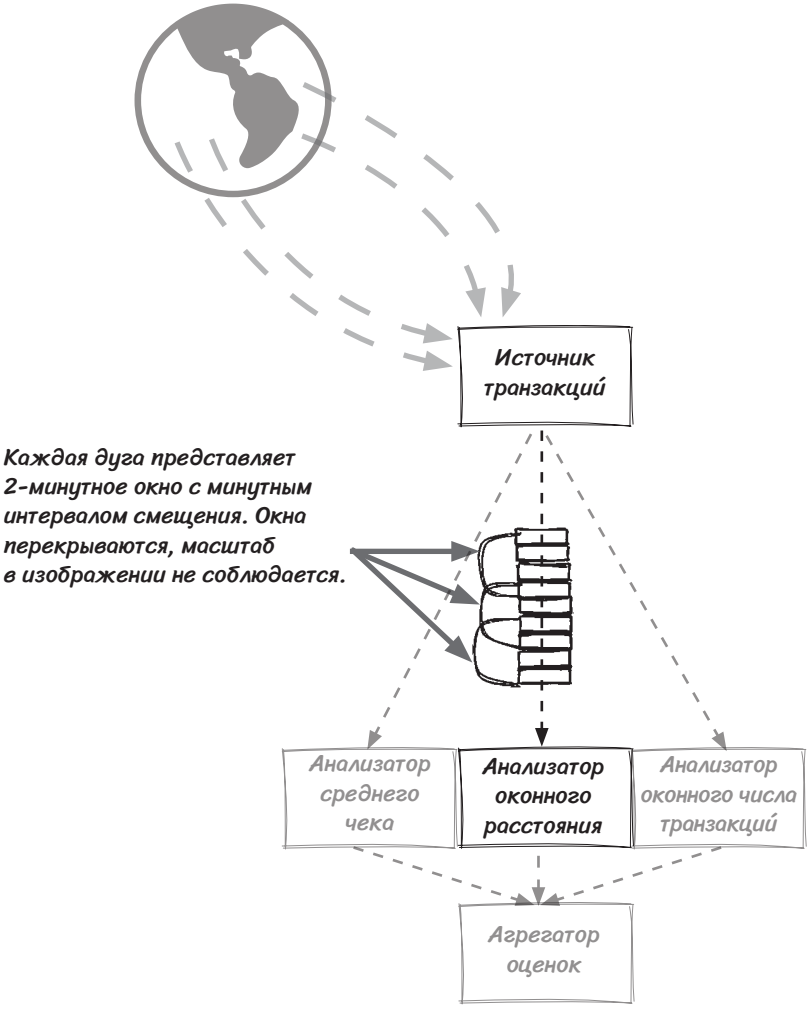
Помните! Количество событий во временных окнах может быть разным. Вполне может оказаться, что в одном окне 3 события, а в другом 4.

С количественными окнами количество событий в каждом окне будет одинаковым. При этом временные интервалы окон могут различаться.



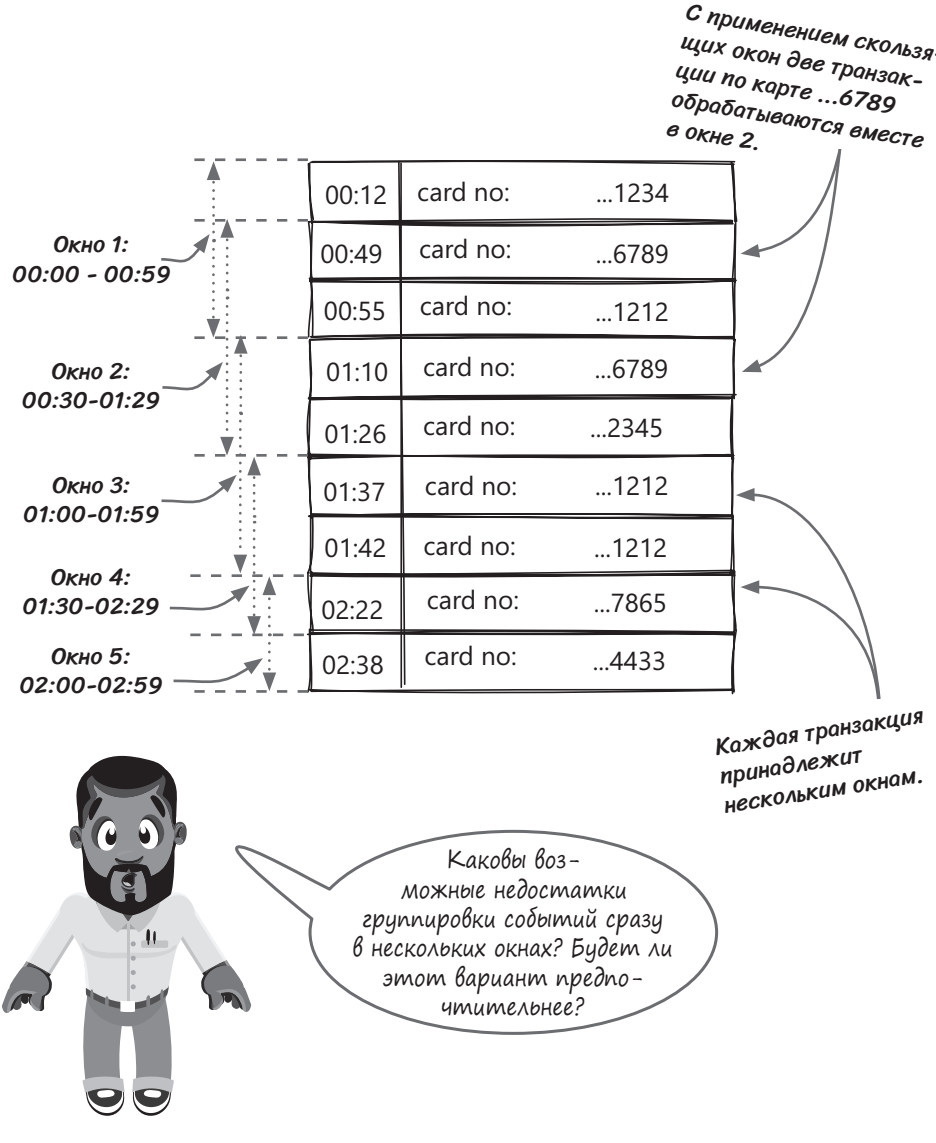
Скользящие окна

Другая популярная оконная стратегия — *скользящее окно* (sliding window). Скользящие окна похожи на фиксированные временные окна, но отличаются от них наличием определенного *интервала смещения* (slide interval). Новое окно создается не при завершении предыдущего окна, а через каждый интервал смещения. Интервал окна и интервал смещения позволяют окнам перекрываться, из-за этого каждое событие может быть включено в несколько окон. Формально можно сказать, что фиксированное окно является особой разновидностью скользящего окна, в которой оконный интервал равен интервалу смещения.



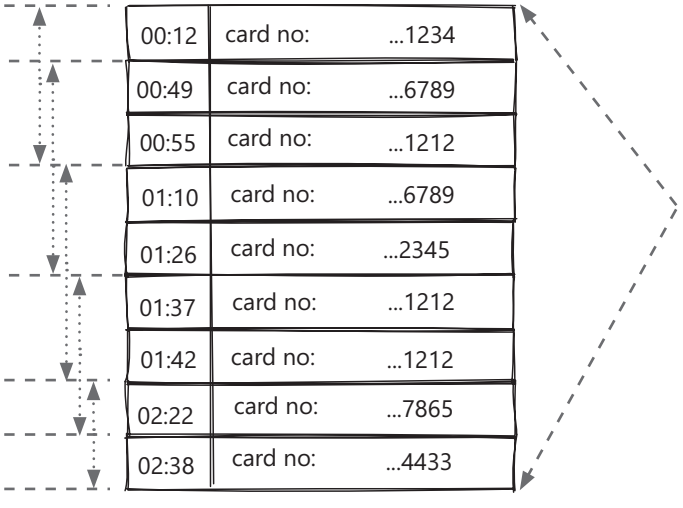
Скользящие окна: анализатор оконного расстояния

Скользящее окно может использоваться для поиска повторяющихся списаний с одной карты в перекрывающихся временных окнах. На следующей диаграмме изображены минутные скользящие окна с 30-секундными интервалами смещения. При использовании скользящих окон важно понимать, что одно событие может войти в несколько окон.



Обнаружение мошеннических действий благодаря использованию скользящих окон

Скользящие окна отличаются от фиксированных тем, что они перекрываются с заданным интервалом. Смещение обеспечивает удобный механизм более равномерно распределенного объединения событий, который позволяет определить, должна ли транзакция быть помечена как мошенническая. Скользящие окна помогают решить проблему отсечения событий, которую мы описывали для фиксированных окон.

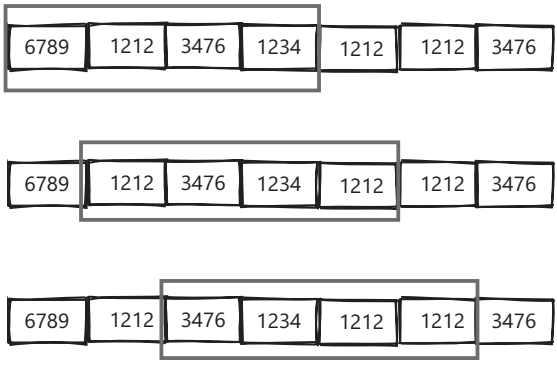


На самом деле скользящие окна просто помогают поддерживать смещаемый контекст данных, к которому можно обращаться для принятия решения о том, помечать ли событие как мошенническое. Другая точка зрения на скользящие окна представлена ниже.

Со смещением окна набор элементов данных, с которыми оно может выполнять операции, изменяется. Постепенное смещение данных, к которым может обращаться система, обеспечивает более последовательное и планомерное представление данных.

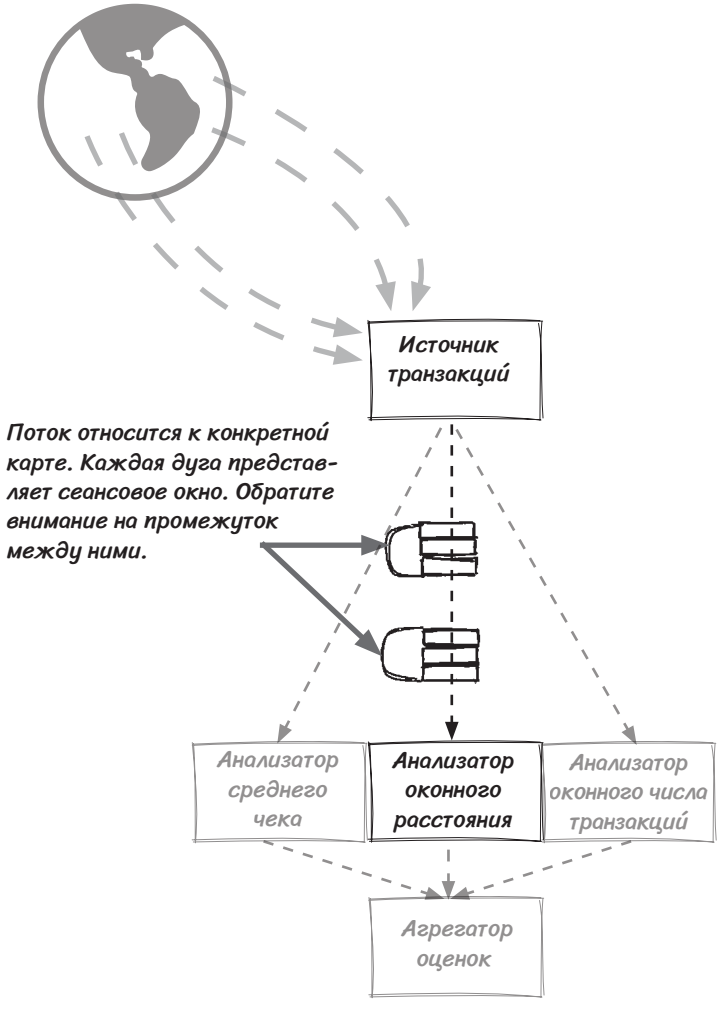
Вопрос на засыпку!

Как вы думаете, к чему приведет перекрытие скользящих окон при вычислении среднего значения — результат улучшится или ухудшится? Почему?



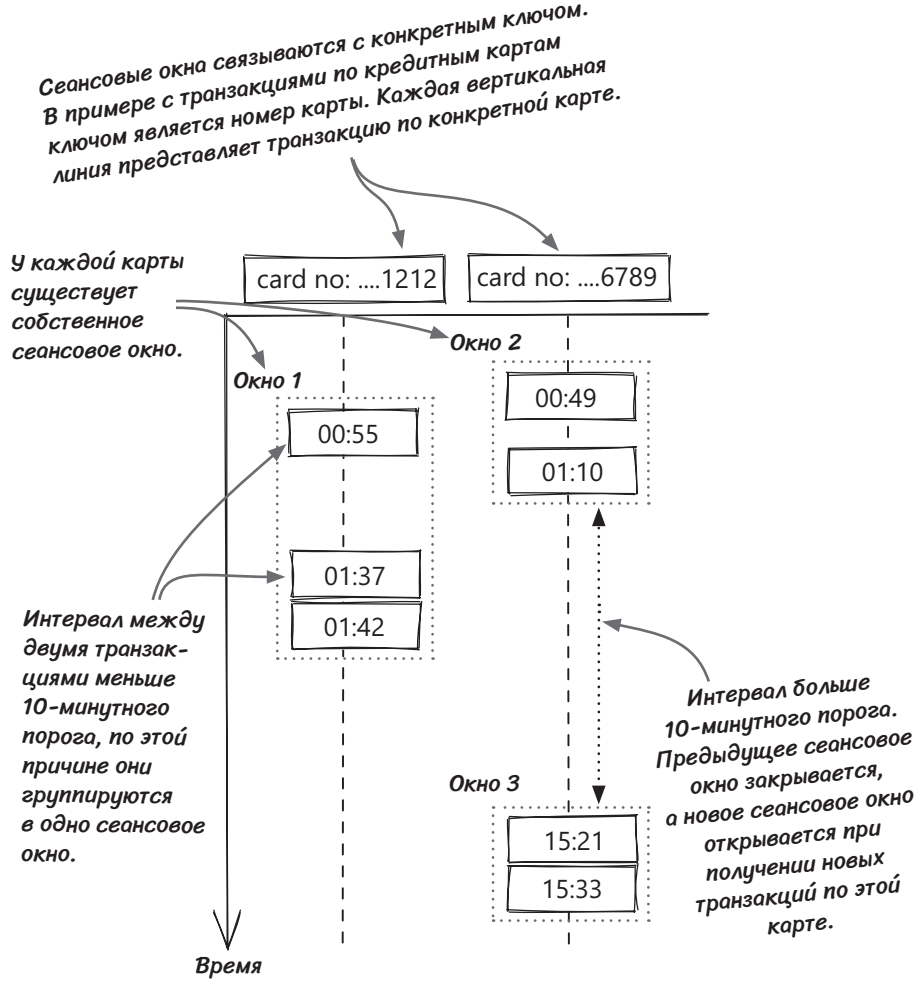
Сеансовые окна

Последняя оконная стратегия, которую нам хотелось бы рассмотреть, прежде чем переходить к реализации, — *сеансовые окна*. *Сеансом* (session) называется период активности, ограниченный определенными промежутками отсутствия активности; сеанс может использоваться для группировки событий. Как правило, сеансовые окна связываются с конкретным ключом и не являются глобальными для всех событий (в отличие от фиксированных и скользящих окон).



Сеансовые окна (продолжение)

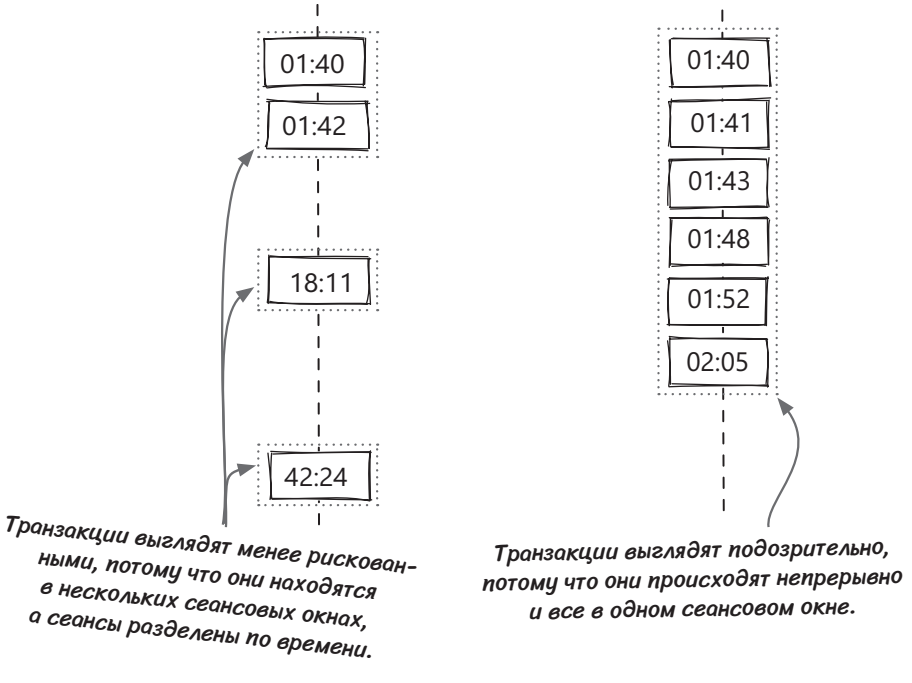
Сеансовые окна обычно определяются *тайм-аутом* — максимальным промежутком времени, в течение которого сеанс может оставаться открытым. Можно представить, что для каждого ключа устанавливается таймер. Если после получения ключа до истечения срока таймера не происходит ни одного события, сеансовое окно закрывается. В следующий раз при получении события с ключом будет запущен новый сеанс. На следующей диаграмме рассмотрим транзакции по двум картам (сеансовые окна обычно связаны с ключом, которым в нашем случае является номер карты). Порог отсутствия активности составляет 10 минут.



Обнаружение мошеннических действий благодаря использованию сеансовых окон

Сеансовые окна относительно сложны по сравнению с фиксированными и скользящими окнами. Посмотрим, как можно использовать сеансовые окна в задании обнаружения мошеннических действий. В текущей структуре анализатор для этой модели отсутствует; полезно рассмотреть такую возможность, к тому же это хороший пример применения сеансовых окон.

Когда посетитель совершает покупки в магазине, обычно он сначала ищет и сравнивает товар. По прошествии какого-то времени он наконец совершает покупку по кредитной карте. Затем покупатель может пойти в другой магазин и поступить так же или же завершить покупки (как известно, шопинг бывает утомителен). В любом случае вероятно, что в течение некоторого времени карта использоваться не будет.

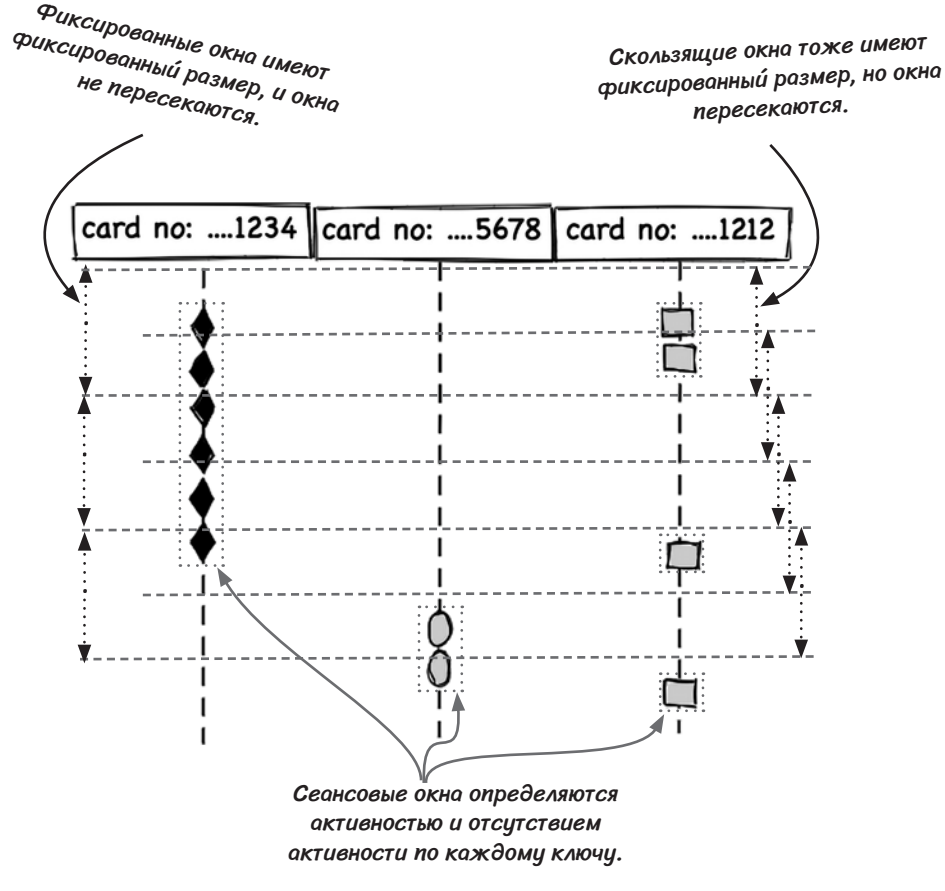


Из двух временных линий с транзакциями, приведенных выше, левая линия выглядит менее подозрительно, чем правая, потому что только одна или две транзакции происходят за короткий промежуток времени (сеансовое окно), а покупки разделены по времени. На правой линии списания средств с карты происходят практически непрерывно, без правдоподобных промежутков.

Обзор оконных стратегий

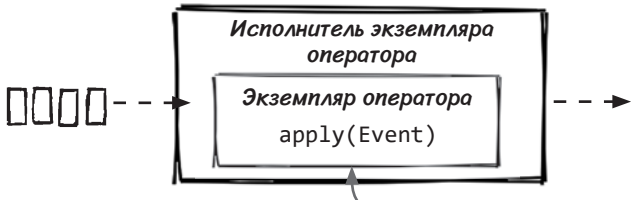
Мы рассмотрели принципы трех разных оконных стратегий. Давайте сравним их. Обратите внимание: при сравнении используются временные окна, но с таким же успехом можно использовать фиксированные и скользящие окна.

- *Фиксированные окна* имеют фиксированный размер, и новое окно открывается при закрытии предыдущего. Окна не пересекаются.
- *Скользящие окна* имеют одинаковый фиксированный размер, но новое окно открывается до закрытия предыдущего. Таким образом, окна пересекаются.
- *Сеансовые окна* обычно отслеживаются для каждого ключа по отдельности. Каждое окно открывается активностью и закрывается по промежутку отсутствия активности.



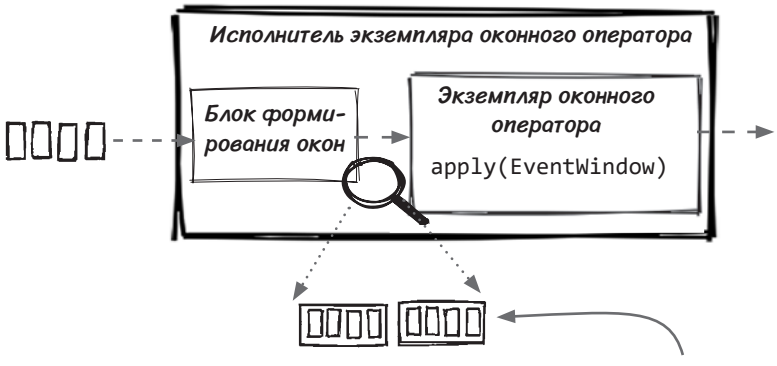
Разбиение потока событий на наборы данных

После рассмотрения всех концепций перейдем к конкретной реализации. С оконными стратегиями события обрабатываются небольшими группами (вместо изолированных событий). Из-за этого различия интерфейс `WindowedOperator` слегка отличается от обычного интерфейса `Operator`.



В обычных операторах события обрабатываются по очереди.

```
public interface Operator {
  public void apply(Event event, EventCollector eventCollector);
}
```



В оконных операторах события разбиваются на наборы, упаковываются в объекты EventWindow ядром и отправляются для обработки операторам, определяемым пользователем. Каждый объект EventWindow также включает информацию о времени — например, начало и конец окна.

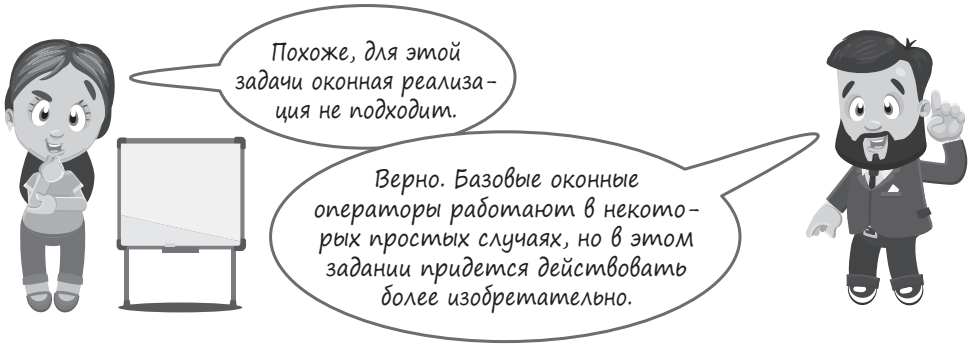
```
public interface WindowedOperator {
  public void apply(EventWindow window, EventCollector eventCollector);
}
```

Оконные системы: концепция или реализация

По сути, оконный оператор представляет собой механизм, преобразующий события в наборы событий, а стриминговые ядра обычно отвечают за управление наборами событий. По сравнению с заданиями, которые рассматривались до этой главы, стриминговым ядрам нужно больше ресурсов для оконных операторов. Чем больше событий в каждом окне, тем больше ресурсов понадобится стриминговому ядру. Иначе говоря, стриминговые задания работают более эффективно при небольшом размере окна. Тем не менее практические условия часто не идеальны. *C'est la vie*.

Возможно, некоторые читатели уже сталкивались с проблемами при использовании оконных операторов для реализации анализатора оконного расстояния в задании обнаружения мошеннических действий:

- В этом анализаторе мы хотели отслеживать точки, находящиеся на удалении друг от друга, и сравнивать расстояние и время транзакций в этих точках. А конкретно, если расстояние превышало 500 миль в час, умноженные на количество часов, прошедших между двумя транзакциями, оператор пометит транзакцию как *потенциально мошенническую*. Насколько эффективно в такой ситуации многочасовое скользящее окно? В этом окне могут содержаться сотни миллиардов транзакций, отслеживание и обработка которых потребуют значительных затрат.
- Ситуация только усложняется, если принять во внимание требование 20-миллисекундной задержки. Со скользящим окном необходимо определить *интервал смещения*, который не должен быть слишком большим. Если этот интервал слишком велик (например, одна секунда), то большинство транзакций (происходящих за первые 980 миллисекунд) нарушит 20-миллисекундное ограничение.

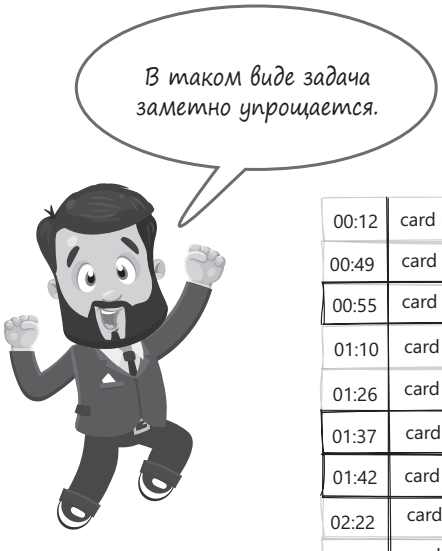


Итак, концепции помогают выбрать правильную стратегию для задачи, но чтобы реализовать анализатор в задании обнаружения мошенничества, недостаточно положиться на фреймворк — необходимо проявить смекалку. На практике такая ситуация достаточно обычна. Стриминговые фреймворки хороши для быстрых и низкозатратных задач, но реальные ситуации редко бывают именно такими.

Другой взгляд

Теперь посмотрим, как наша команда решает проблему и предотвращает хищение бензина. Прежде всего необходимо понять, как именно обрабатываются транзакции в анализаторе оконного расстояния.

Этот оператор должен отслеживать время и место каждой транзакции по карте и проверять, что время и расстояние между двумя транзакциями не нарушают правило. Тем не менее формулировка «любые две транзакции в окне» не является обязательной. Задачу можно упростить, если взглянуть на нее под слегка другим углом: в любой момент при поступлении новой транзакции можно сравнить ее время и место с параметрами *предыдущей транзакции* по той же карте и применить нашу формулу. Все предшествующие транзакции по карте, а также все транзакции по другим картам не влияют на результат, и их можно исключить из обработки.



00:12	card no:	...1234
00:49	card no:	...6789
00:55	card no:	...1212
01:10	card no:	...6789
01:26	card no:	...2345
01:37	card no:	...1212
01:42	card no:	...1212
02:22	card no:	...7865
02:38	card no:	...4433

Каждую транзакцию необходимо сравнить с предыдущей транзакцией по той же карте.

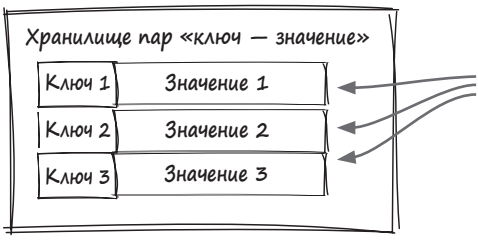
Так как формула уже готова, задача становится довольно простой; но как найти предыдущую транзакцию по той же карте?

Как насчет использования скользящего окна? Хороший вопрос; давайте рассмотрим и этот вариант. Длина окружности Земли составляет около 25 000 миль, так что расстояние между любыми двумя точками не превышает 12 500 миль. Согласно нашему правилу максимальной скорости 500 миль в час, на перемещение в любую другую точку Земли должно потребоваться не более 25 часов. Следовательно, транзакции со сроком более 25 часов обрабатывать не нужно. Обновленная формулировка задачи выглядит так: *как определить предыдущую транзакцию по той же карте за последние 25 часов?*

Хранилища пар «ключ — значение»

Немного поразмыслив над вычислениями для анализатора оператора оконного расстояния, наша команда решила реализовать их на базе системы хранилищ «ключ — значение». Этот метод построения оконных операторов без применения стандартной поддержки оконных операторов в стриминговых фреймворках чрезвычайно полезен, поэтому о нем стоит поговорить.

Хранилище пар «ключ — значение» (называемое также хранилищем K-V, от key-value) представляет собой систему хранения данных, спроектированную для сохранения и чтения объектов данных с ключами. За последнее десятилетие эта парадигма стала чрезвычайно популярной. Если термин вам еще не знаком, объясним: система работает как словарь, в котором каждая запись однозначно идентифицируется конкретным ключом. В отличие от более традиционных (и более известных) реляционных баз данных, записи в хранилищах «ключ — значение» полностью независимы друг от друга.



В хранилищах пар «ключ — значение» каждая запись ассоциируется с уникальным ключом. Обычно для работы с хранилищем достаточно двух базовых функций: get (ключ) и put (ключ, значение).

Зачем нужны системы хранения данных с ограниченной функциональностью? Их главные преимущества — производительность и масштабируемость. Так как хранилищам пар «ключ — значение» не нужно отслеживать отношения между разными записями, строками и столбцами, внутренние вычисления значительно упрощаются по сравнению с традиционными базами данных. В результате такие операции, как чтение и запись, выполняются намного быстрее. А поскольку записи не зависят друг от друга, также становится намного проще распределять данные по нескольким серверам и организовывать их совместную работу для предоставления сервиса хранения пар «ключ — значение», способного обрабатывать огромные объемы данных. Эти два преимущества важны для системы обнаружения мошеннических действий, а также многих других систем обработки данных.

Другая интересная возможность, поддерживаемая хранилищами пар «ключ — значение», — *ограничение срока действия*. Для пары «ключ — значение», добавляемой в хранилище, можно указать срок действия. По истечении этого срока пара будет автоматически удалена из системы и занимаемые ею ресурсы будут освобождены. Эта возможность чрезвычайно удобна для оконных операторов в стриминговых системах (а конкретно в части «за последние 25 часов» нашей задачи).

Реализация анализатора оконного расстояния

Благодаря паре «ключ — значение» стриминговому ядру не придется отслеживать и хранить в памяти все события окна. Это обязанность разработчиков системы. Но есть и плохая новость: хранилище пар «ключ — значение» может использоваться по-разному в зависимости от ситуации. Не существует простой и универсальной схемы реализации оконных стратегий с хранилищами пар «ключ — значение». В качестве примера возьмем анализатор оконного расстояния.

В анализаторе необходимо сравнить время и место каждой транзакции с временем и местом предыдущей транзакции по той же карте. Текущая транзакция хранится во входном событии, а предыдущая транзакция для каждой карты должна находиться в хранилище пар «ключ — значение». Ключом является идентификатор карты, а значением — время и место (для простоты в приведенном ниже исходном коде событие целиком сохраняется как значение).

```
public class WindowedProximityAnalyzer implements Operator {
  final static double maxMilesPerHour = 500;
  final static double distanceInMiles = 2000;
  final static double hourBetweenSwipes = 2;
  final KVStore store;

  public setupInstance(int instance) {
    store = setupKVStore();
  }

  public void apply(Event event, EventCollector eventCollector) {
    TransactionEvent transaction = (TransactionEvent) event;
    TransactionEvent prevTransaction = kvStore.get(transaction.getCardId());

    boolean result = false;
    if (prevTransaction != null) {
      double hourBetweenSwipe =
          transaction.getEventTime() - prevTransaction.getEventTime();
      double distanceInMiles = calculateDistance(transaction.getLocation(),
          prevTransaction.getLocation());
      if(distanceInMiles > hourBetweenSwipe * maxMilesPerHour) {
        // Транзакция помечается как потенциально мошенническая.
        result = true;
      }
    }

    eventCollector.emit(new AnazlyResult(event.getTransactionId(), result));
    kvStore.put(transaction.getCardId(), transaction);
```

Вместо Operator используется WindowedOperator.

Настройка хранилища пар «ключ — значение».

Предыдущая транзакция загружается из хранилища пар «ключ — значение».

Обнаруживается мошенническая транзакция.

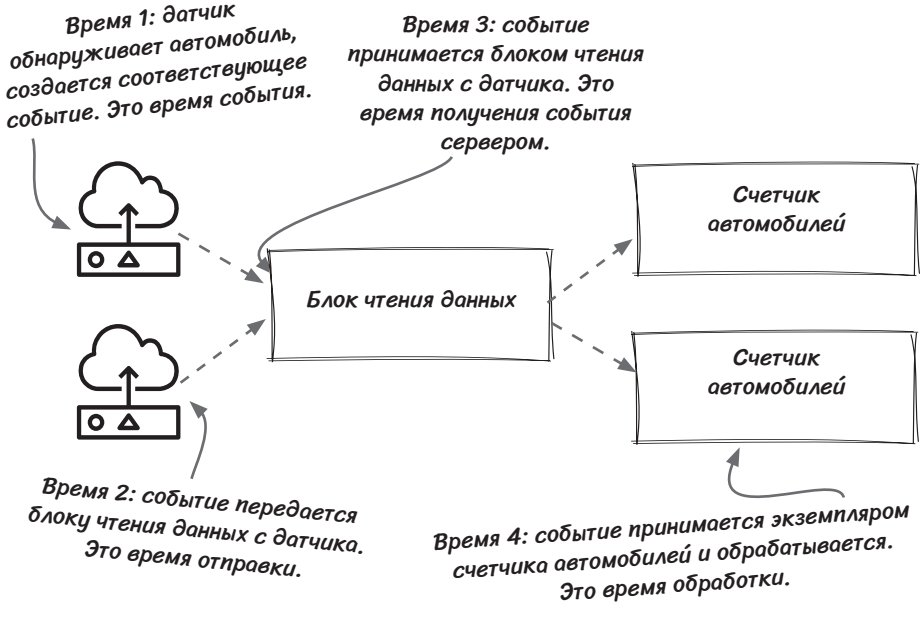
Текущая транзакция помещается в хранилище пар «ключ — значение», при этом идентификатор карты используется в качестве ключа. Предыдущее значение при этом заменяется.

Время события и другие вехи

Прежде чем завершить эту главу, осталось рассмотреть еще одну концепцию. В коде анализатора оконного расстояния присутствует одна важная строка, на которую стоит обратить особое внимание:

```
transaction.getEventTime();
```

Что же считать *временем события* (event time)? А есть ли другие ключевые моменты времени? Речь идет о времени фактического возникновения события. Многие операции обработки событий не совершаются немедленно. Вместо этого возникшее событие обычно сохраняется в некоторой служебной подсистеме, а реальная обработка происходит позднее. Все эти операции выполняются в разное время, так что да, есть и другие вехи времени. Используем в качестве примера нашу простую систему контроля дорожного движения и рассмотрим важные вехи времени, относящиеся к событию.

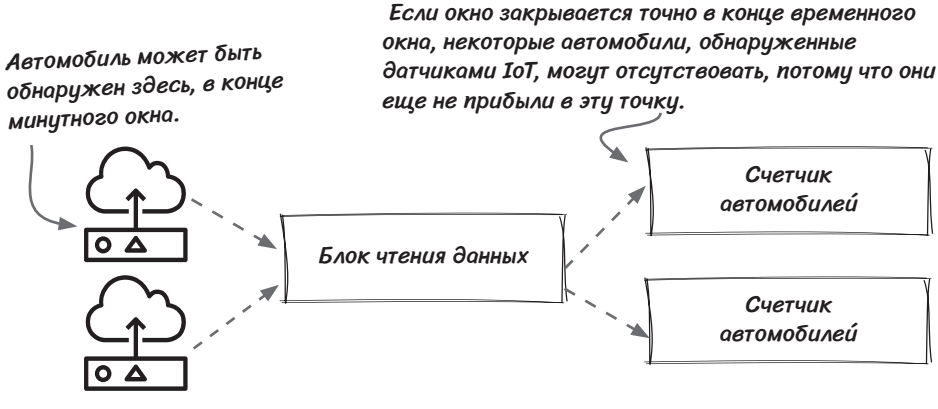


Среди всех вех времени самыми важными для каждого события являются *время события* и *время обработки*. Время события для каждого события можно сравнить с днем рождения для человека. А временем обработки считается время, в которое событие было обработано. В системе обнаружения мошеннических действий нас в действительности интересует момент физического считывания карты, то есть время события транзакции. Время события обычно включается в объекты событий, так что все вычисления с событием используют один момент времени для получения последовательных результатов.

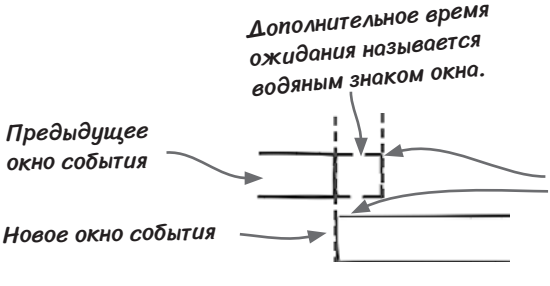
Водяной знак окна

Время события используется во многих оконных вычислениях, поэтому очень важно понимать промежуток между временем события и временем обработки. Из-за этого промежутка оконные стратегии, описанные в этой главе, не настолько просты, как может показаться.

Если взять в качестве примера систему контроля движения и настроить оператор счетчика автомобилей с простыми фиксированными окнами для подсчета автомобилей, обнаруженных за каждую минуту, каким должно быть время открытия и закрытия для каждого окна? Обратите внимание на то, что время появления каждого события в экземплярах операторов счетчиков автомобилей (время обработки) немного отстает от времени его создания в датчике IoT (время события). Если окно закрывается точно в конечное время, события, происходящие ближе к концу окна на датчиках IoT, будут отсутствовать, потому что они еще не были получены экземплярами счетчиков. Заметим, что их нельзя поместить в следующее окно, потому что на основании времени события они принадлежат уже закрытому окну.



Чтобы решить проблему с пропуском событий, можно держать окно открытым чуть дольше и ожидать получения событий. Дополнительное время ожидания для окна обычно называется *водяным знаком* (windowing watermark).



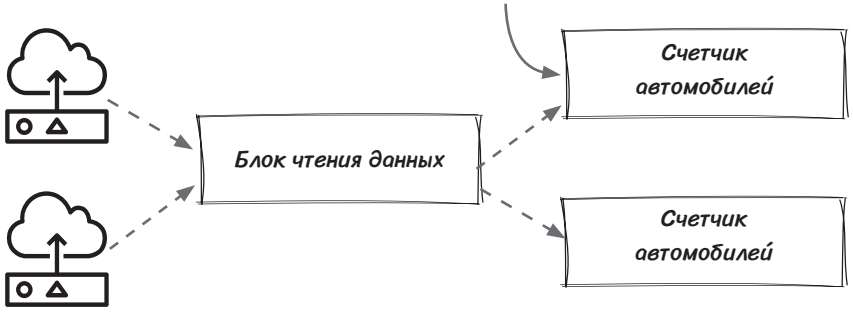
Если вернуться к реализации анализатора оконного расстояния, водяной знак становится еще одной причиной, по которой стандартный оконный оператор не идеален в рассматриваемой ситуации. Резервирование дополнительного времени перед обработкой наборов событий создает дополнительную задержку и только усложняет соблюдение требования о 20-миллисекундной задержке.

Поздние события

Водяной знак очень важен для предотвращения пропуска событий и генерирования полных наборов событий для обработки. Понять концепцию несложно, но выбрать время ожидания несколько сложнее.

Например, в системе контроля дорожного движения датчики IoT работают очень хорошо. В результате обычно все события автомобилей собираются успешно в пределах одной секунды. В этом случае может подойти 1-секундный водяной знак (то есть задержка на 1 секунду).

Обычно все события должны прибывать в пределах 1 секунды после их создания. 1-секундный водяной знак выглядит разумно.



Тем не менее слово «*обычно*» звучит как тревожный сигнал. Ранее мы не раз упоминали, что одной из серьезных проблем при построении любой распределенной системы становится преодоление сбоев. Выработайте в себе полезную привычку спрашивать: что, если система не будет работать так, как ожидалось? Даже в такой простой системе события могут задержаться более чем на секунду, если что-то пошло не так — например, датчик или блок чтения данных могут временно замедлиться либо пропускная способность сети может измениться из-за нестабильности соединения. События, полученные после закрытия соответствующего окна, называются *поздними событиями*. Что с ними делать?

Иногда потеря поздних событий допустима, но в других случаях важно обеспечить правильную обработку таких событий. Многие реальные стриминговые фреймворки предоставляют механизмы обработки поздних событий, но мы не будем описывать подробности, так как обработка привязана к конкретному

фреймворку. А пока ключевой вывод: учитывайте возможность появления поздних событий и не забывайте о них.

Итоги

Оконные вычисления становятся важнейшим аспектом стриминговых систем, потому что они предоставляют возможность объединения изолированных событий в наборы событий для обработки. В этой главе были рассмотрены три стандартные оконные стратегии, поддерживаемые в большинстве стриминговых фреймворков:

- Фиксированные окна.
- Скользящие окна.
- Сеансовые окна.

Базовая поддержка окон в стриминговых фреймворках имеет свои ограничения и может плохо работать во многих случаях. Поэтому кроме описания концепций и подхода к обработке оконных операторов в стриминговых фреймворках вы научились использовать хранилища пар «ключ — значение» для моделирования оконных операторов и преодоления ограничений.

В конце главы также были рассмотрены три взаимосвязанные концепции, которые играют важную роль при решении прикладных задач:

- Разные вехи времени, относящиеся к каждому событию, включая различия между временем события и временем обработки.
- Водяные знаки окна.
- Поздние события.

Упражнение

В начале главы мы сказали, что существуют два способа предотвращения мошеннических транзакций по кредитным картам:

- Блокировать снятие средств с отдельных кредитных карт.
- Блокировать обработку всех кредитных карт на бензоколонках.

Впоследствии мы сосредоточились на выявлении проблем с отдельными кредитными картами, но не рассматривали второй вариант. Как бы вы стали выявлять подозрительные бензоколонки, чтобы заблокировать обработку кредитных карт на них?

8 Операции соединения

В этой главе

- ✓ Сопоставление различных типов событий в реальном времени.
- ✓ Внутренние и внешние соединения.
- ✓ Оконные соединения.

«Запрос SQL заходит в бар, подходит к двум столикам и спрашивает: “Можно к вам присоединиться?”»[1]

Автор неизвестен

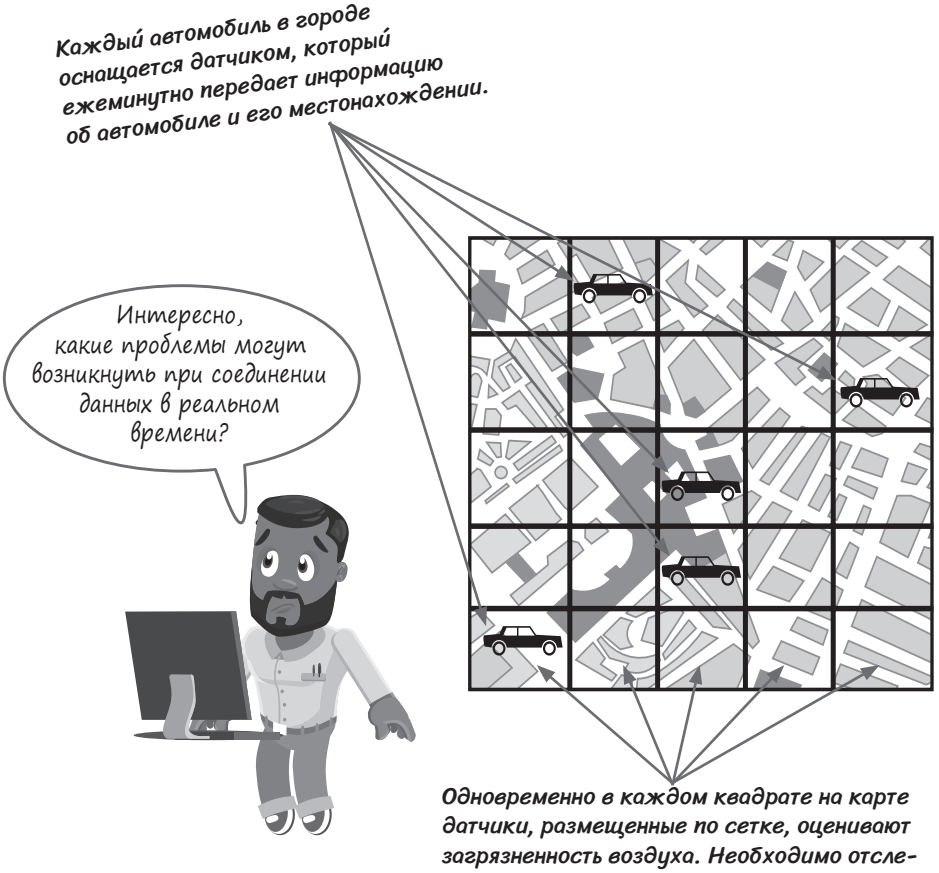
Если вы работали с любой базой данных SQL, то, скорее всего, вы использовали оператор `join` — или по крайней мере читали о нем. В стриминговых системах операция *соединения* (join) не настолько важна, как в базах данных, но все равно иногда очень полезна. В этой главе вы узнаете, как соединения работают в контексте потоковой обработки. Сначала мы рассмотрим пример использования соединения в базах данных, чтобы вы поняли его суть, а потом перейдем к соединениям в стриминговых системах. Если операция соединения вам уже знакома, можете пропустить вводную часть.

[1] An SQL query goes into a bar, walks up to two tables, and asks, can I join you? Шутка основана на многозначности слов table — «столик» и «таблица» и join you — «присоединиться к вам» и «соединить вас». Можно перевести как «запрос подходит к двум таблицам и спрашивает: можно вас соединить?». — *Примеч. пер.*

Соединение данных выбросов в реальном времени

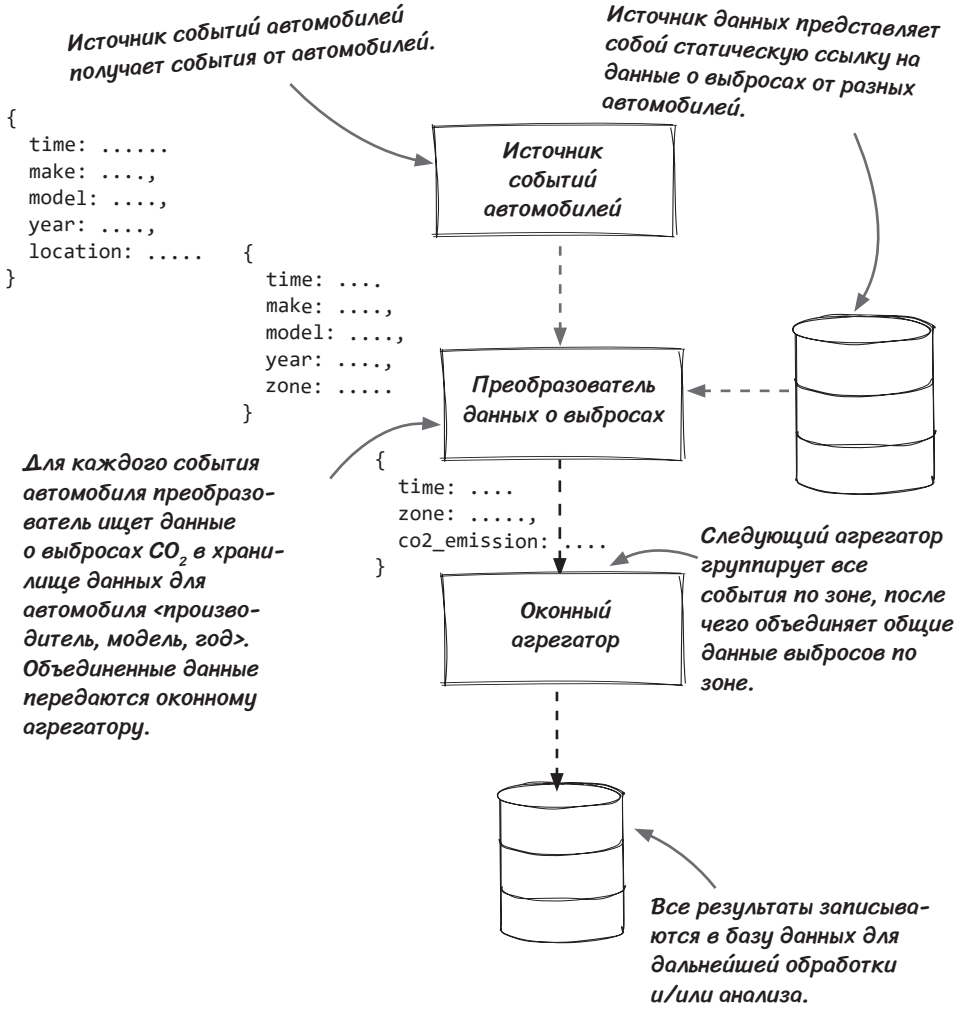
Нашей команде повезло: ее привлекли к работе над отслеживанием выбросов автомобилей в Кремниевой долине, штат Калифорния. Круто же?

Но каждая большая возможность несет в себе и сложности. Команде придется оперативно соединять события, генерируемые автомобилями в определенных районах города, с расчетными показателями выбросов. Как это сделать? Давайте разберемся.



Задание контроля выбросов, версия 1

Первая версия задания выбросов уже готова. Интересная часть задания — хранилище данных справа от преобразователя данных выбросов. Это статическая таблица, используемая преобразователем для поиска данных о выбросах, производимых каждым автомобилем. Предполагается, что в этой системе выбросы от автомобилей одного производителя, модели и года выпуска одинаковы.



Преобразователь данных о выбросах

Ключевым компонентом этого задания является преобразователь данных о выбросах. Он получает событие автомобиля, ищет данные выбросов для автомобиля в хранилище данных и выдает событие с информацией о зоне и данными о выбросах. Следует заметить, что исходящее событие содержит данные из двух источников: входящего события автомобиля и таблицы.


```
{
  make: ....,
  model: ....,
  year: ....,
  zone: .....
}
```

Производитель	Модель	Год	Выброс
XXX	AA	2020	3.3
YYY	CC	2021	4.2
...	...	...	...

```
class EmissionResolver extends Operator {
  private final Table emissionTable = ......;
  public void apply(Event event, EventCollector eventCollector) {
    VehicleEvent vehicleEvent = (VehicleEvent) event.getData();
    double emission = emissionTable.getEmission(
      vehicleEvent.make, vehicleEvent.model, vehicleEvent.year
    );
    eventCollector.add(
      new EmissionEvent(vehicleEvent.zone, emission)
    );
  }
}
```

```
{
  zone: .....,
  co2_emission: ....
}
```

Данные о выбросах включаются в исходящее событие.

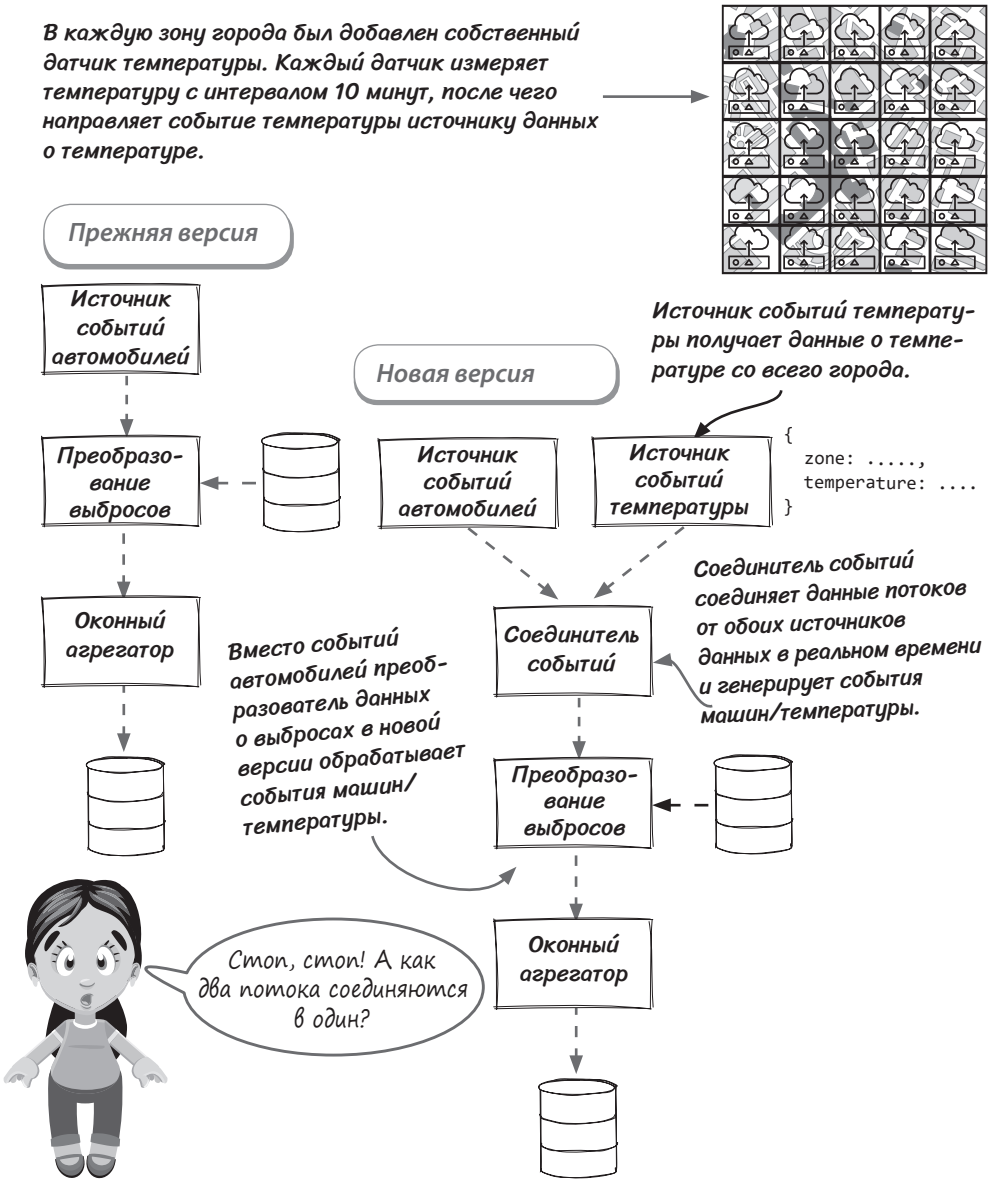
Этот оператор можно рассматривать как очень простой оператор *соединения*, который объединяет данные из разных источников на основании связывающих их характеристик (производитель, модель, год). Тем не менее данные о выбросах берутся из таблицы, а не из потока. Операторы соединения в стриминговых заданиях идут дальше, предоставляя данные в реальном времени.

Точность становится проблемой

Задание в целом работает нормально и успешно генерирует данные о выбросах в реальном времени. Тем не менее в вычислениях отсутствует один важный фактор: температура (как вы знаете, выбросы CO_2 зависят от температуры, и в Калифорнии погода тоже меняется). В результате данные о выбросах по зонам, возвращаемые системой, не обладают достаточной точностью. Добавлять датчик температуры в устройства, установленные на каждом автомобиле, уже поздно, поэтому проблему придется решать иначе.

Обновленное задание контроля выбросов

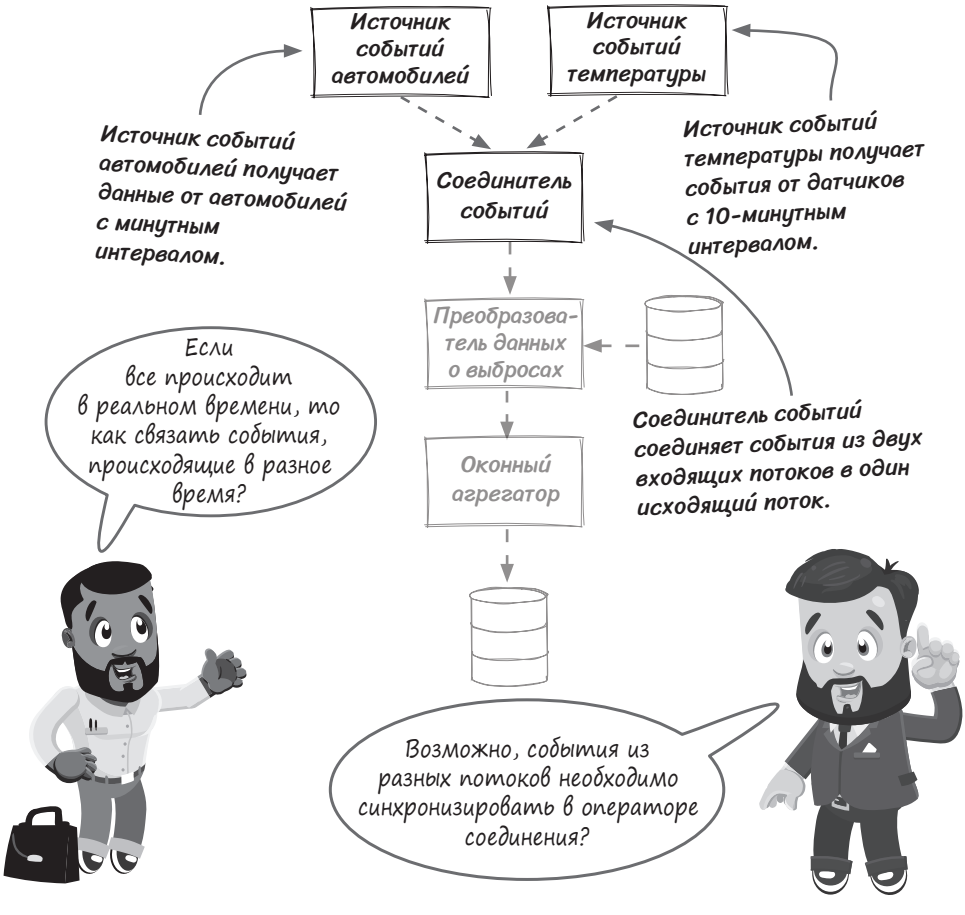
Чтобы повысить точность результатов, наша команда добавила еще один источник данных для введения в задание события текущей температуры. События температуры соединяются с событиями автомобилей по идентификатору зоны. Исходящие события выбросов передаются преобразователю данных о выбросах.



Все внимание на соединение

Основные изменения в новой версии:

- Дополнительный источник данных, получающий события температуры в задании.
- Соединитель событий, соединяющий два потока в один.



Источник событий температуры работает как нормальный источник, который отвечает за получение данных от стриминговых заданий. Ключевое изменение — появление оператора — соединителя событий, который имеет два входных потока событий и один выходной поток событий. События поступают в реальном времени, и события разных потоков редко бывают идеально синхронизированы друг с другом. Как заставить разные типы событий работать друг с другом в операторе-соединителе? Давайте разберем этот вопрос подробнее.

Еще раз: что такое соединение?

Вероятно, при любом упоминании оператора соединения join естественно представлять себе SQL. В конце концов, сам термин «соединение» пришел из мира реляционных баз данных.

Соединение — конструкция SQL, которая позволяет взять некоторое количество полей из одной таблицы и объединить их с набором полей из другой таблицы или таблиц для получения консолидированных данных. На следующей диаграмме представлен оператор соединения в контексте реляционных баз данных; стриминговое соединение рассматривается ниже.

Таблица событий автомобилей

make	model	year	zone
XXX	AA	2020	3
YYY	CC	2013	1
ZZZ	DD	2017	2
XXX	AA	2008	1
XXX	BB	2014	1
ZZZ	EE	2021	3
ZZZ	EE	2018	5

Таблица температуры

zone	temperature
1	95.4
2	94.3
3	95.1
4	95.2
5	95.3

Две таблицы содержат общее поле zone, которое образует связь между таблицами.

```
SELECT v.time, v.make, v.model, v.year, t.zone, t.temperature
FROM vehicle_events v
INNER JOIN temperature t on v.zone = t.zone;
```

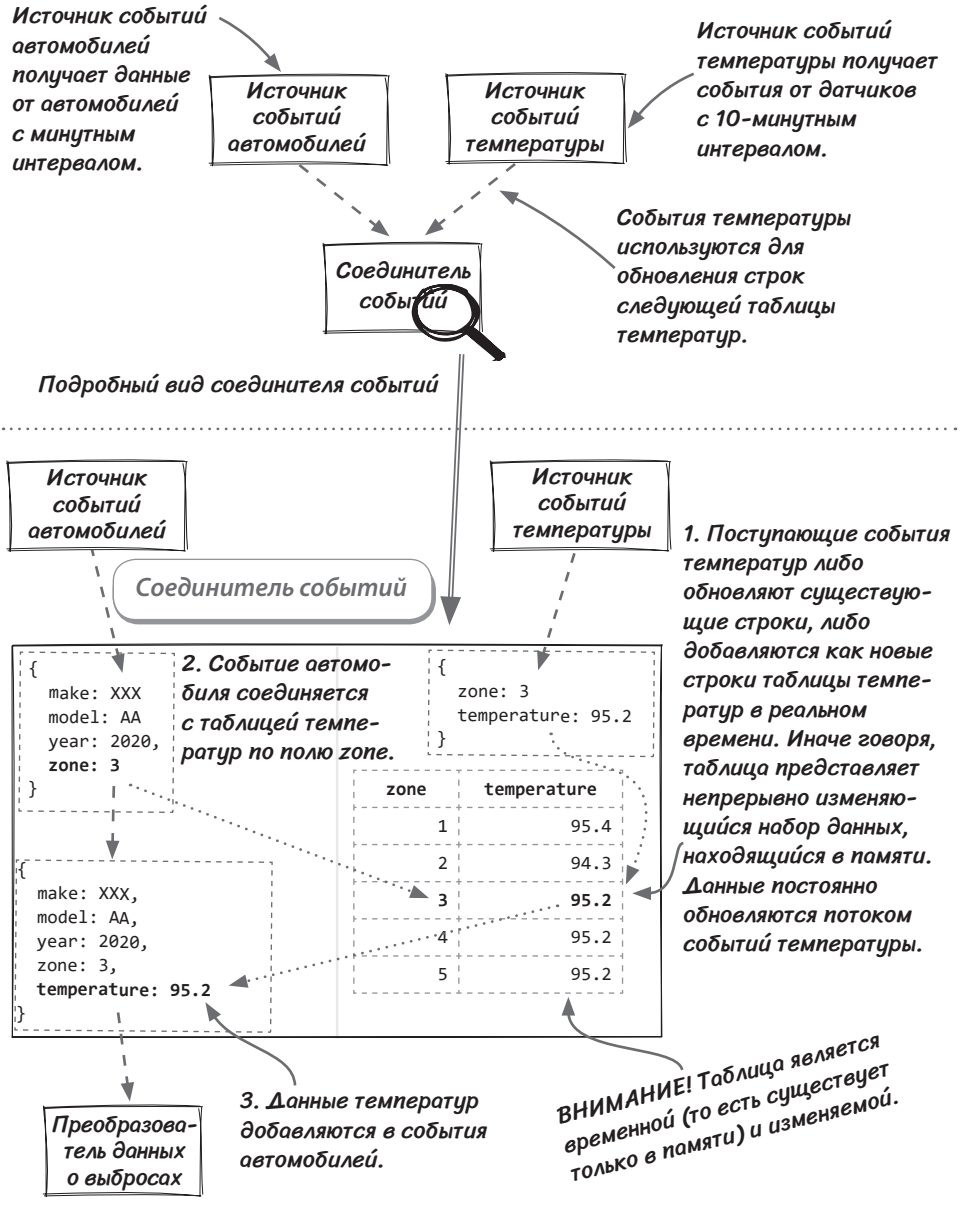
Соединенная таблица

Результат соединения этих таблиц может выглядеть примерно так:

make	model	year	zone	temperature
XXX	AA	2020	3	95.1
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.1
ZZZ	EE	2018	5	95.3

Как работает стриминговое соединение

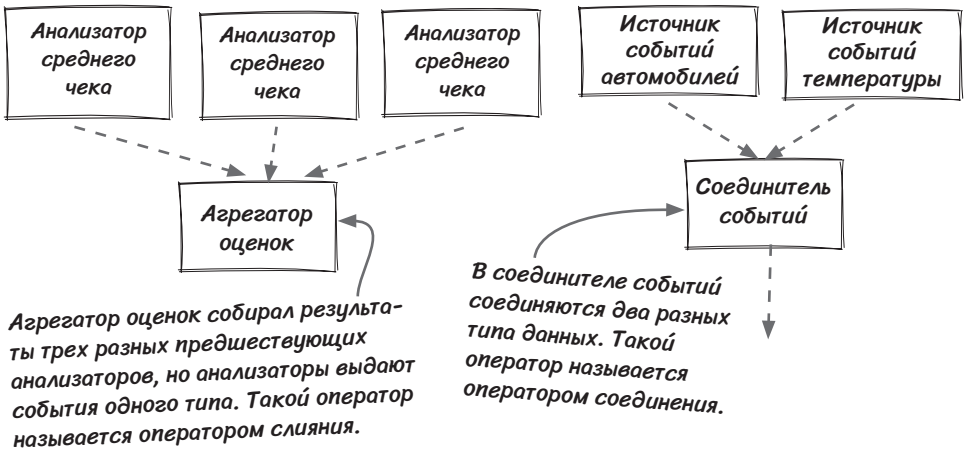
Как создать соединение для данных, которые непрерывно перемещаются и обновляются? Для этого следует преобразовать события температуры в таблицу.

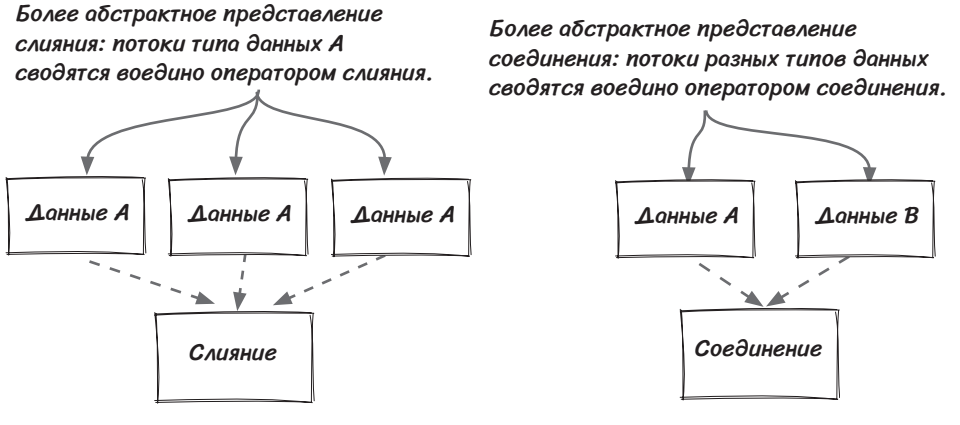


Соединение (join) потоков — разновидность объединения (fan-in)

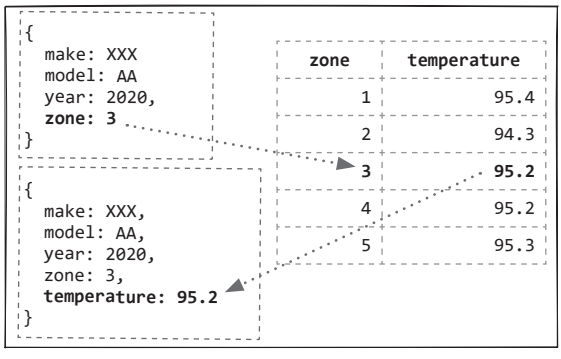
В главе 4 рассматривался сценарий обнаружения мошеннических действий, в котором оценки риска от предшествующих анализаторов использовались для определения того, является ли транзакция мошеннической. Относится ли агрегатор оценок к той же категории операторов?

Правильный ответ — нет, не является. В агрегаторе оценок все входящие потоки имеют один тип событий. Оператор не знает, из какого потока поступило каждое событие, и просто применяет одну и ту же логику. В соединителе событий события двух входящих потоков отличаются друг от друга и по-разному обрабатываются в операторе. Агрегатор оценок является оператором слияния, а соединитель событий является оператором соединения. Оба они являются операторами *объединения* (fan-in).





События автомобилей и события температуры



Следует заметить, что в операторе соединения события температуры преобразуются во временную таблицу температур, но события автомобилей обрабатываются как поток. Почему преобразуются события температуры, а не события автомобилей? Почему не преобразовать оба потока в таблицы? Важно продумать ответы на эти вопросы при построении систем.

Во-первых, предполагается, что для каждого входящего события автомобиля будет выдаваться одно исходящее событие. Следовательно, будет логично поддерживать течение событий автомобилей через оператор как поток. Во-вторых, управлять событиями автомобилей в форме таблицы поиска будет сложнее. Автомобилей намного больше, чем зон в системе, поэтому хранение событий автомобилей во временной таблице в памяти требует значительно больших затрат ресурсов. Более того, в каждой зоне для нас важна только последняя температура, а при управлении счетчиком автомобилей (добавлении и удалении) необходимо действовать осторожнее, потому что важно каждое событие.

Попробуем поместить события автомобилей в таблицу, а затем соединить их с потоком событий температуры. Для каждой зоны в таблице будет храниться несколько записей, а результатами станут пакеты событий вместо отдельных событий.

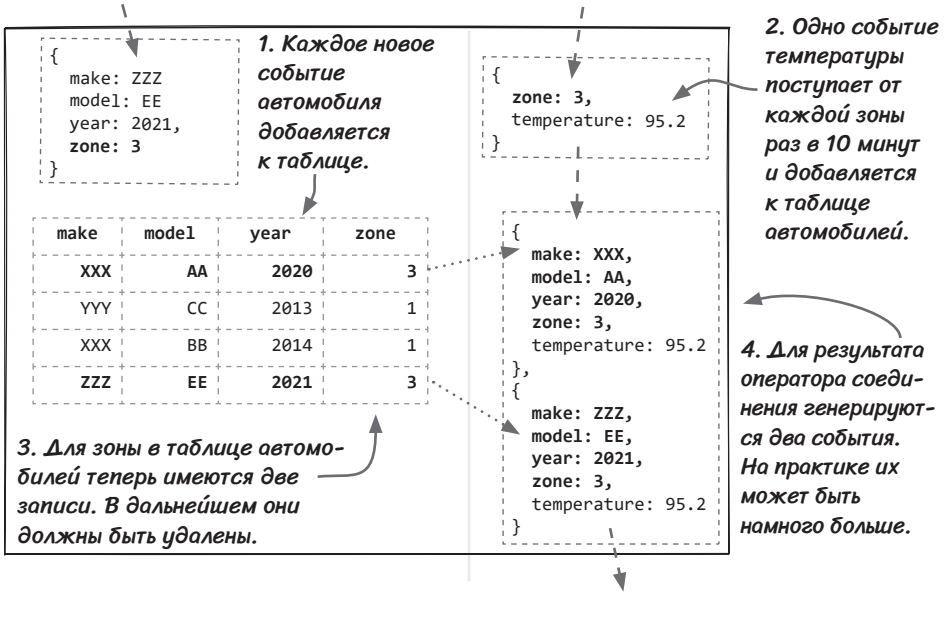
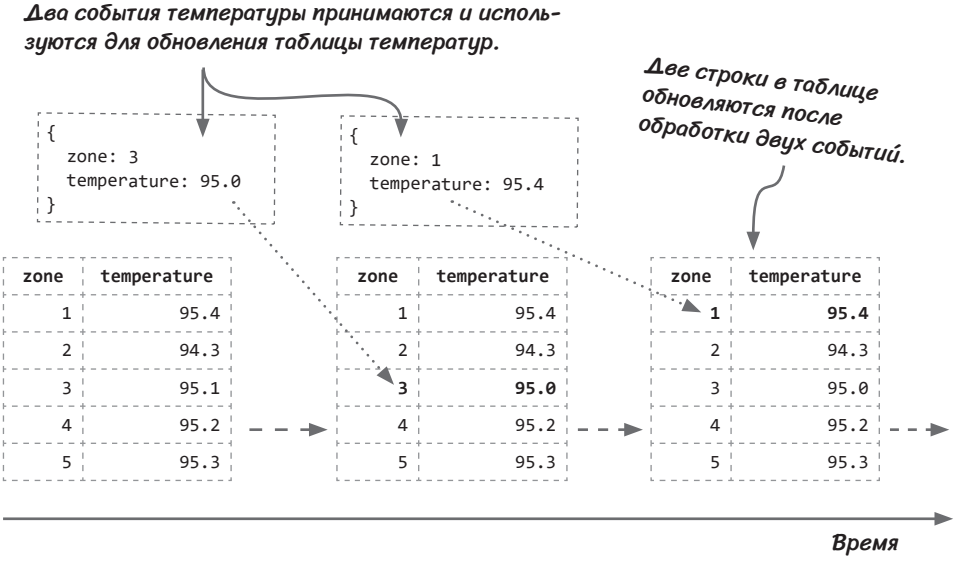


Таблица как материализованное представление стриминга

Порассуждаем на чуть более абстрактном уровне: как связаны между собой события температуры и таблица температур? Понимание отношений между ними поможет понять особенности событий температуры и принимать более эффективные решения при построении новых стриминговых систем.

Одна важная особенность данных температуры заключается в том, что в любой момент необходимо хранить только последнюю температуру для каждой зоны. Дело в том, что нас интересует именно *последняя* температура, а не история ее изменений. На следующей диаграмме показаны изменения в таблице температур до и после получения и обработки двух событий температуры.

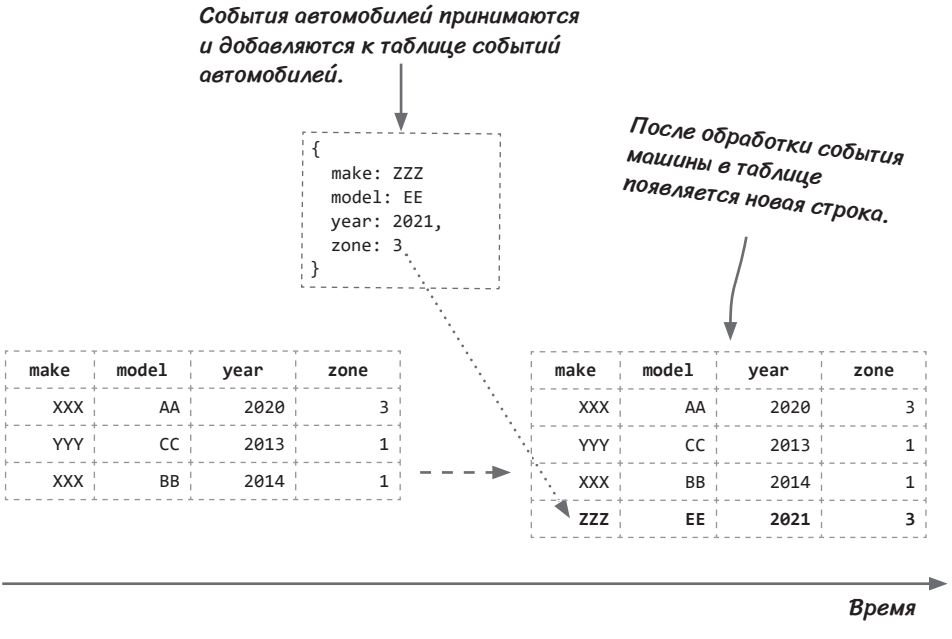


Каждое событие температуры используется для обновления таблицы последними данными. Таким образом, каждое событие может рассматриваться как *изменение* данных в таблице, а поток событий — как *журнал изменений*.

С другой стороны, при выполнении соединения ведется поиск по таблице температур. В любой момент таблица температур является результатом применения всех событий до конкретного момента времени. Таким образом, таблица может рассматриваться как *материализованное представление* событий температур. Интересный эффект материализации заключается в том, что интервал между событиями становится несущественным. В нашем примере интервал между событиями температур для каждой зоны составляет 10 минут, но система будет работать точно так же с интервалом в 1 секунду или 1 час.

Материализация событий автомобилей менее эффективна

С другой стороны, по сравнению с событиями температуры материализация событий автомобилей менее эффективна. Автомобили постоянно перемещаются по городу, и в соединение должно включаться каждое отдельное событие для одного автомобиля, а не только последнее. В результате таблица событий автомобилей по сути содержит *список событий, ожидающих обработки*. Кроме того, количество автомобилей обычно больше количества зон. В итоге по сравнению с событиями температуры события автомобилей сложнее, а их материализация менее эффективна.



На этой диаграмме показано, что события машин *добавляются* к таблице (вместо обновления данных в строках). Хотя существуют некоторые приемы для повышения эффективности, например добавление столбца счетчика и объединение строк с одним производителем, моделью, годом и зоной вместо простого добавления данных к таблице, достаточно ясно, что материализовать события температуры намного удобнее, чем события автомобилей. В реальных задачах это свойство может стать важным фактором при выборе способа обработки потоков, если используется оператор соединения.

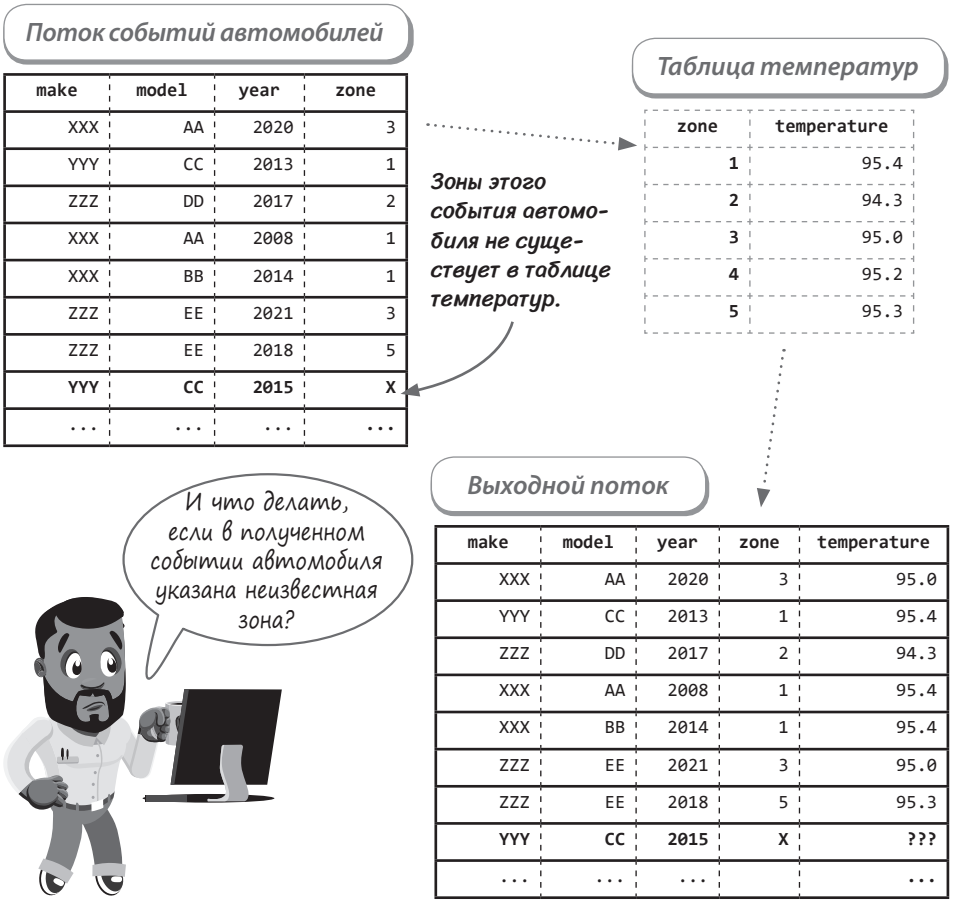
Проблемы с целостностью данных

Задание прекрасно работает для отслеживания выбросов в той зоне, которая изначально планировалась командой. К сожалению, люди постоянно норовят использовать приложение не так, как было задумано.

Почему возникает проблема и как ее решить? Необходимо рассмотреть различные типы операторов соединения.

Проблемы с оператором соединения

Ключевая функция оператора соединения — получение температуры для заданной зоны. Рассмотрим приведенное ниже табличное представление оператора. На диаграмме каждое событие автомобиля представляется строкой в таблице, но следует помнить, что события автомобилей обрабатываются одно за другим, как поток. Также следует учитывать, что таблица температур изменяется динамически, а значения температур могут изменяться при поступлении новых событий температуры.



make	model	year	zone
XXX	AA	2020	3
YYY	CC	2013	1
ZZZ	DD	2017	2
XXX	AA	2008	1
XXX	BB	2014	1
ZZZ	EE	2021	3
ZZZ	EE	2018	5
YYY	**CC**	**2015**	**X**
...	...	...	...

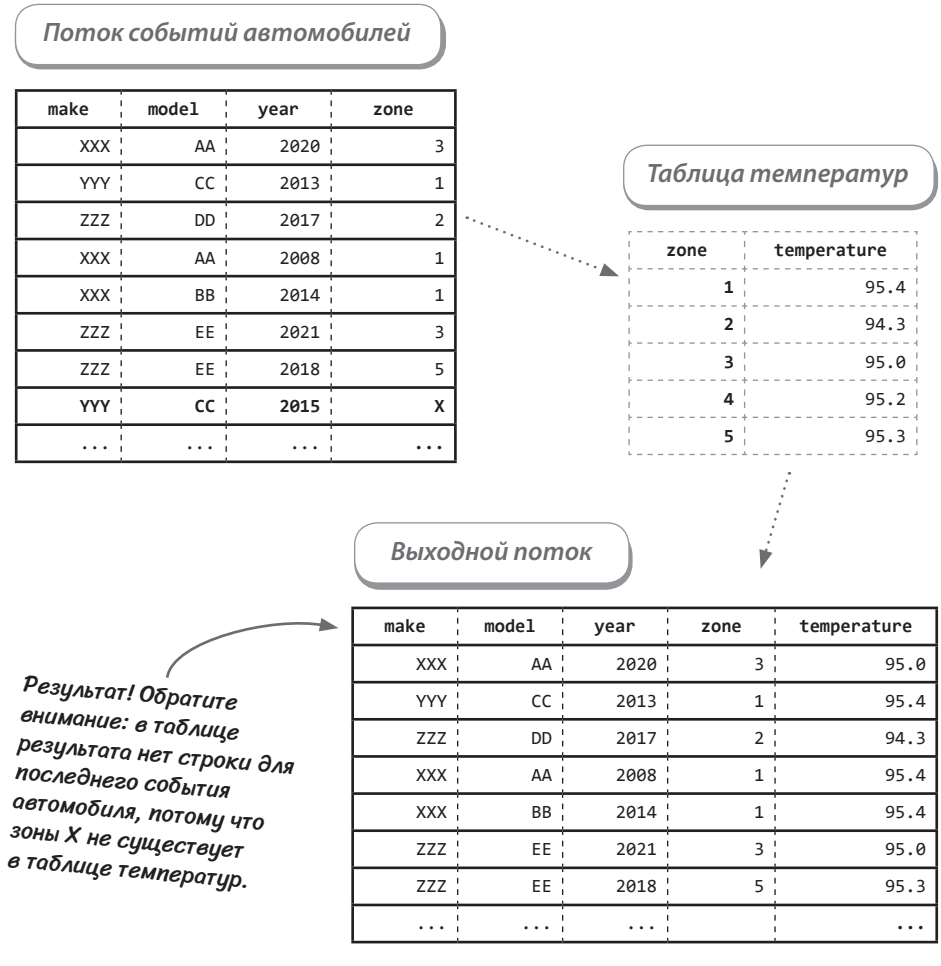
zone	temperature
1	95.4
2	94.3
3	95.0
4	95.2
5	95.3

make	model	year	zone	temperature
XXX	AA	2020	3	95.0
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.0
ZZZ	EE	2018	5	95.3
YYY	**CC**	**2015**	**X**	**???**
...	...	...		...

Проблема целостности данных возникла из-за того, что зона 7 из последнего события автомобиля отсутствует в таблице температур. Что делать? Чтобы ответить на этот вопрос, необходимо обсудить две новые концепции: *внутреннее соединение* (inner join) и *внешнее соединение* (outer join).

Внутреннее соединение

Внутреннее соединение обрабатывает только события автомобилей, для которых имеется соответствующая зона в таблице температур.



make	model	year	zone
XXX	AA	2020	3
YYY	CC	2013	1
ZZZ	DD	2017	2
XXX	AA	2008	1
XXX	BB	2014	1
ZZZ	EE	2021	3
ZZZ	EE	2018	5
YYY	**CC**	**2015**	**X**
...	...	...	...

zone	temperature
1	95.4
2	94.3
3	95.0
4	95.2
5	95.3

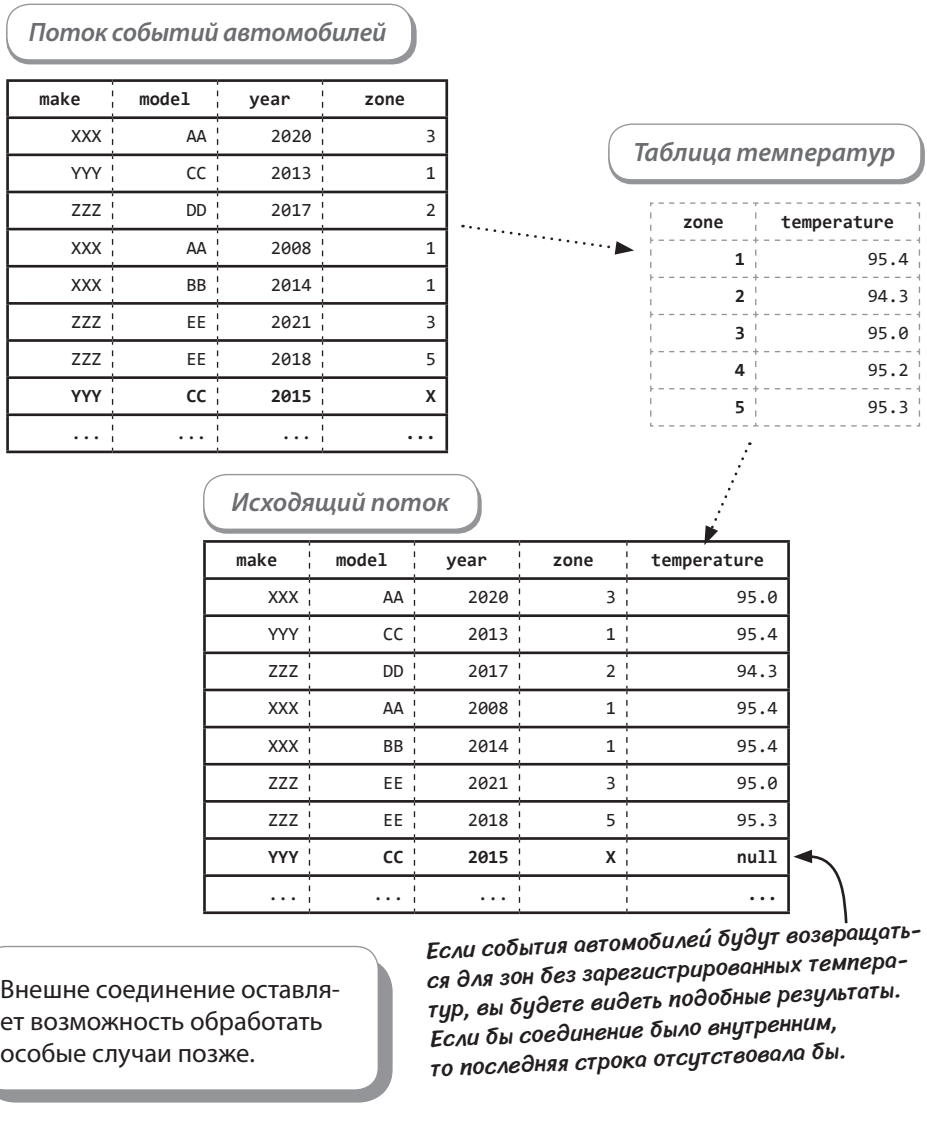
make	model	year	zone	temperature
XXX	AA	2020	3	95.0
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.0
ZZZ	EE	2018	5	95.3
...	...	...		...

Внимательно присмотревшись к результату работы оператора соединения, вы увидите, что в нем нет строки для зоны 7. Дело в том, что внутреннее соединение возвращает только строки данных, для которых существуют соответствующие значения, а в таблице температур нет зоны 7.

При использовании внутреннего соединения выбросы в таких зонах не будут учтены, так как события автомобилей игнорируются. Насколько это допустимо?

Внешнее соединение

Внешние соединения отличаются от внутренних тем, что в них *включаются* как имеющие соответствие, так и не имеющие соответствия строки для заданного столбца или данных. Таким образом, события не будут потеряны, хотя в результате могут появиться неполные события.



make	model	year	zone
XXX	AA	2020	3
YYY	CC	2013	1
ZZZ	DD	2017	2
XXX	AA	2008	1
XXX	BB	2014	1
ZZZ	EE	2021	3
ZZZ	EE	2018	5
YYY	**CC**	**2015**	**X**
...	...	...	...

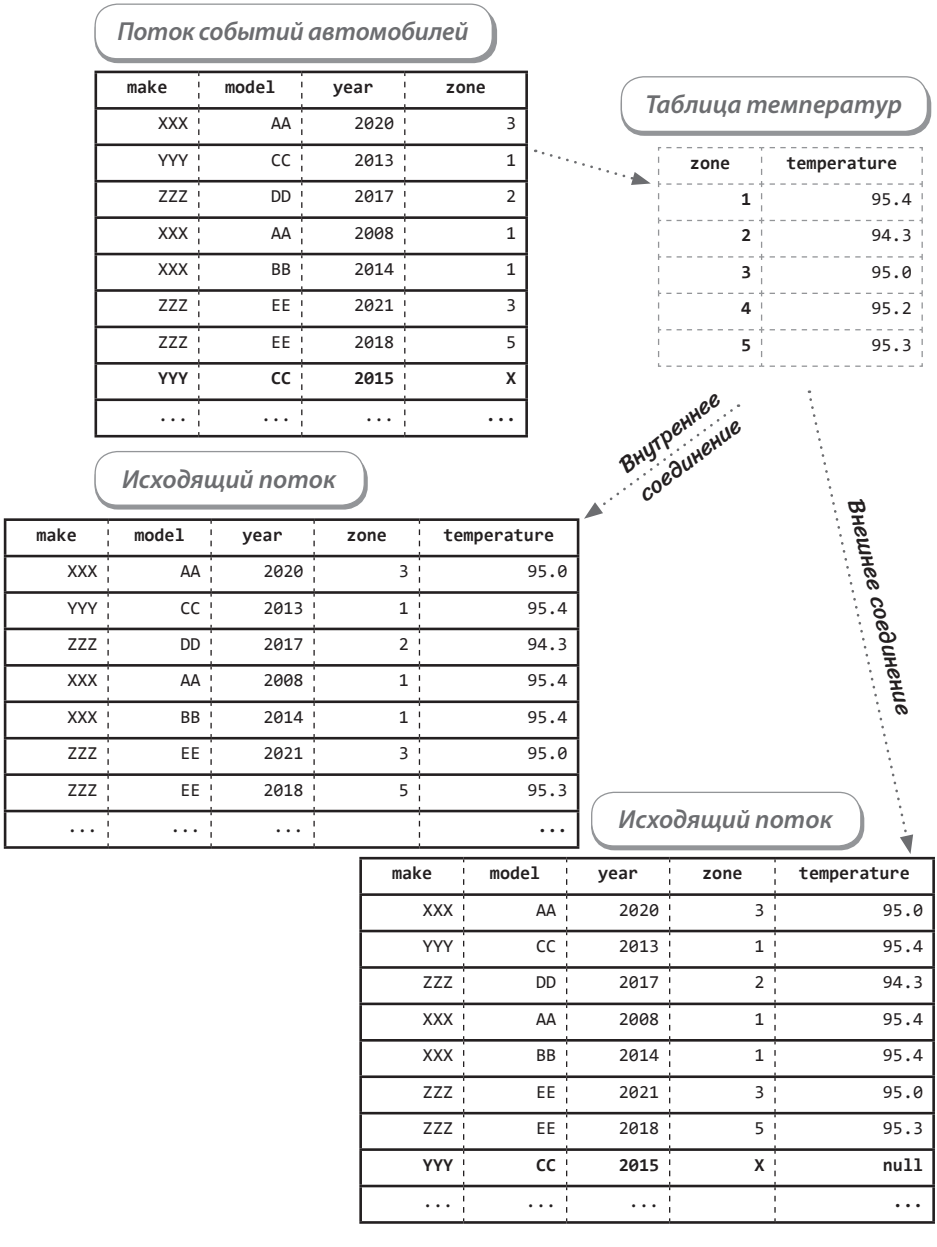
zone	temperature
1	95.4
2	94.3
3	95.0
4	95.2
5	95.3

make	model	year	zone	temperature
XXX	AA	2020	3	95.0
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.0
ZZZ	EE	2018	5	95.3
YYY	**CC**	**2015**	**X**	null
...	...	...		...

Наша команда решает применить внешнее соединение, чтобы сохранить неполные строки и обработать их позже.

Внутренние и внешние соединения

События автомобилей, не имеющие соответствующих данных в таблице температур, по-разному обрабатываются внутренними и внешними соединениями. Внутренние соединения возвращают результаты только в том случае, если для них есть соответствующие значения на обеих сторонах, а внешние соединения возвращают результаты независимо от того, есть ли соответствие данных.



make	model	year	zone
XXX	AA	2020	3
YYY	CC	2013	1
ZZZ	DD	2017	2
XXX	AA	2008	1
XXX	BB	2014	1
ZZZ	EE	2021	3
ZZZ	EE	2018	5
YYY	**CC**	**2015**	**X**
...	...	...	...

zone	temperature
1	95.4
2	94.3
3	95.0
4	95.2
5	95.3

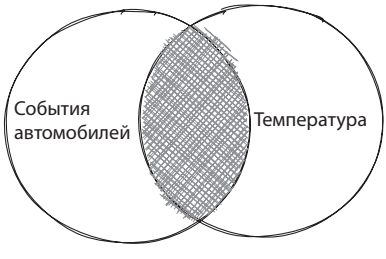
make	model	year	zone	temperature
XXX	AA	2020	3	95.0
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.0
ZZZ	EE	2018	5	95.3
...	...	...		...

make	model	year	zone	temperature
XXX	AA	2020	3	95.0
YYY	CC	2013	1	95.4
ZZZ	DD	2017	2	94.3
XXX	AA	2008	1	95.4
XXX	BB	2014	1	95.4
ZZZ	EE	2021	3	95.0
ZZZ	EE	2018	5	95.3
YYY	**CC**	**2015**	**X**	**null**
...	...	...		...

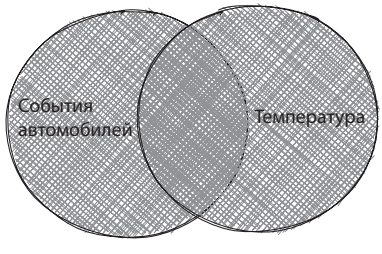
Разные типы соединений

Если вам известна конструкция `join` в базах данных, то вы наверняка помните, что существуют разные типы внешних соединений: *полные внешние соединения* (или просто *полные соединения*), *левые внешние соединения* (*левые соединения*) и *правые внешние соединения* (*правые соединения*). Все операторы соединения представлены на следующих диаграммах, чтобы продемонстрировать различия между ними в контексте баз данных SQL.

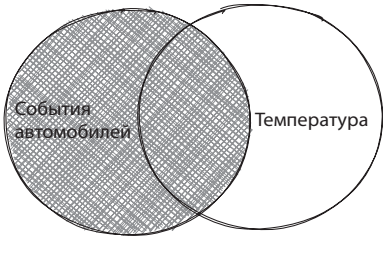
Внутренние соединения возвращают только те результаты, для которых существуют соответствующие значения в обеих таблицах.



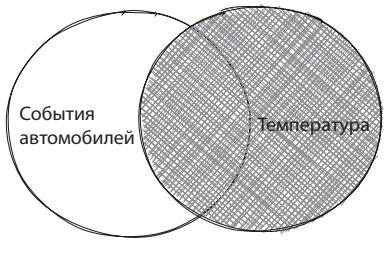
Полные внешние соединения возвращают все результаты из обеих таблиц.



Левые внешние соединения возвращают все результаты из таблицы событий автомобилей и только соответствующие строки из таблицы температур.



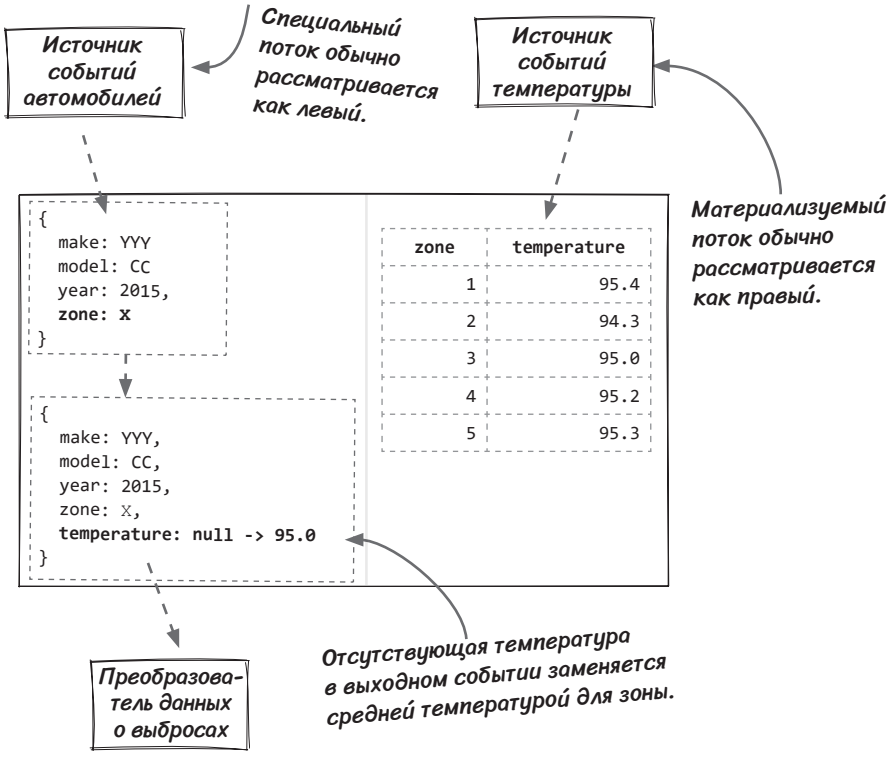
Правые внешние соединения возвращают все результаты из таблицы температур и только соответствующие строки из таблицы событий автомобилей.



Внешние соединения в стриминговых системах

Теперь вы знаете, как работают внутренние и внешние соединения в базах данных SQL. В сфере потоковой обработки все организовано примерно так же. Важное отличие заключается в том, что во многих случаях (например, в задании контроля выбросов CO_2) события одного из входящих потоков обрабатываются *одно за другим*, тогда как другие потоки материализуются в таблицах для соединения. Обычно специальный поток рассматривается как *левый*, а материализуемые потоки как *правые*. Таким образом, соединение, использованное в соединителе событий, является левым внешним соединением.

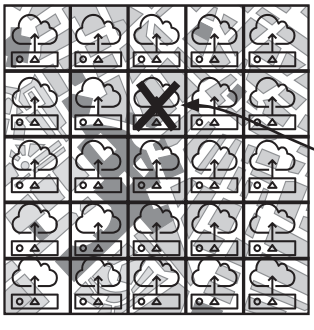
С левым внешним соединением наша команда может идентифицировать автомобили, передвигающиеся за пределами запланированной области, и решить проблемы целостности данных, подставляя среднюю температуру в генерируемые события «автомобиль — температура» (вместо того, чтобы удалять их). Такая подстановка повышает точность результатов.



Следует учитывать, что в более сложных (а значит, интересных) случаях может быть несколько правых потоков и к ним могут применяться разные типы соединений.

Новая проблема: ненадежное соединение

После решения проблемы целостности данных команда через несколько недель заметила другую проблему: некоторые значения в таблице температур выглядят странно. После анализа ситуации выяснилась первопричина: в одном датчике возникли проблемы с подключением, и иногда он сообщает температуру с периодичностью в несколько часов вместо 10 минут. Проблему можно решить ремонтом устройства, но нельзя ли повысить устойчивость системы к проблемам с подключением?



Подключение к этому датчику нестабильно, и это значение температуры в таблице устарело, потому что оно не обновлялось несколько часов.

zone	temperature
1	95.4
2	94.3
3	**91.2**
4	95.2
5	95.3

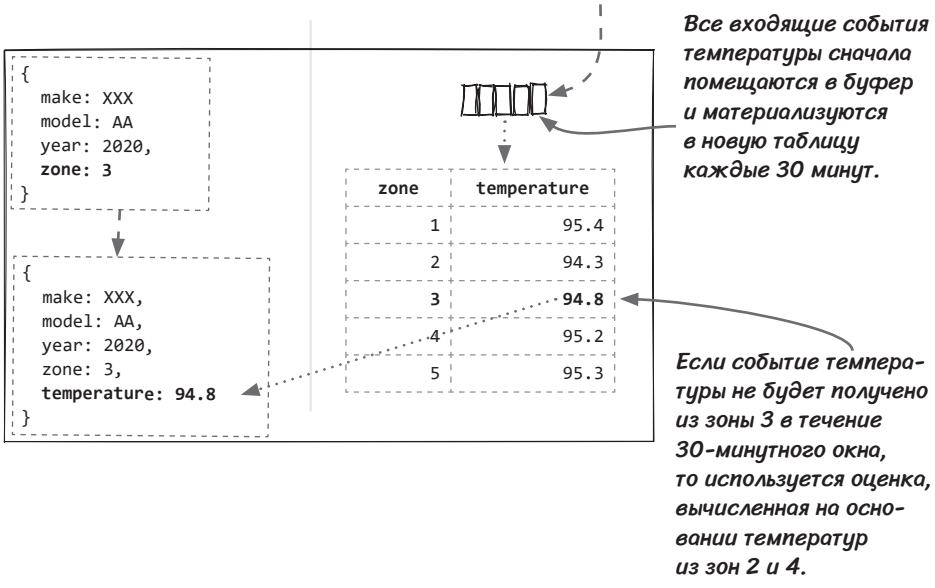
В общем случае стриминговые системы должны учитывать, что их источники событий могут быть ненадежными.

Оконные соединения

Новая концепция *оконных соединений* (windowed joins) поможет стриминговому заданию решить проблемы с ненадежностью канала связи. Смысл понятен из названия: оконное соединение представляет собой оператор, сочетающий окна с соединениями. В предыдущей главе оконные вычисления рассматривались подробно. Но подробности здесь не важны, так что не беспокойтесь, если вы решили начать чтение с этой главы.

С оконными соединениями задание работает почти так же, как исходная версия: события автомобилей обрабатываются одно за другим, а события температуры материализуются в таблицу поиска. Тем не менее материализация событий температуры основана на фиксированном временном окне вместо непрерывных событий. А если говорить конкретнее, события температуры сначала собираются в буфер и материализуются в пустой таблице в пакетном режиме каждые 30 минут. Если все датчики успешно передают данные в границах окна,

вычисления работают нормально. Но в случае, если в окне от датчика не было получено событие температуры, соответствующая строка таблицы поиска будет пустой, а соединитель события сможет оценить текущее значение по соседним зонам. На следующей диаграмме температура в зоне 3 используется для оценки температур в зонах 2 и 4. Использование оконного соединения позволяет гарантировать актуальность всех температурных данных в таблице.

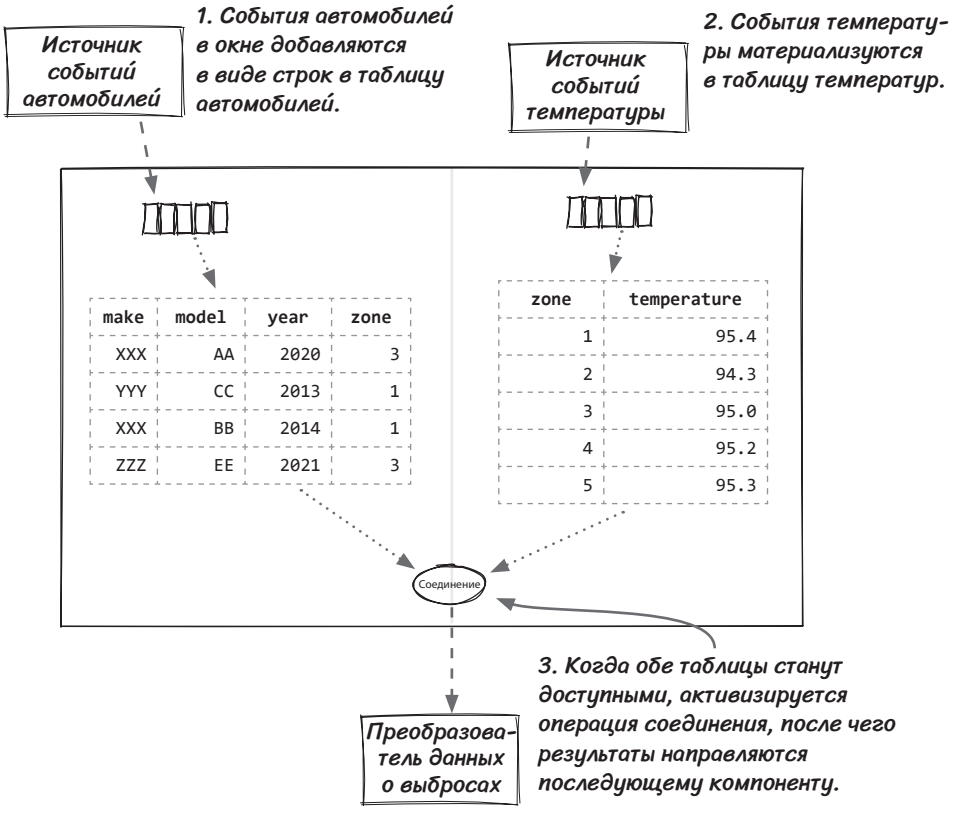


Переключаясь с непрерывной материализации на оконную, мы немного теряем в задержке температуры (значения температур обновляются каждые 30 минут вместо 10), но взамен получаем более защищенную систему, которая способна обнаруживать и автоматически обрабатывать некоторые непредвиденные проблемы.

Соединение двух таблиц вместо соединения потока с таблицей

Прежде чем завершить эту главу, рассмотрим для примера вариант, в котором оба потока сначала преобразуются в таблицы, а затем соединяются системой контроля выбросов CO_2. При таком решении процесс в компоненте состоит из двух шагов: материализации и соединения. Сначала два входящих события материализуются в две таблицы. Затем логика соединения применяется к таблицам, а результаты передаются последующим компонентам. Обычно на шаге материализации используются окна, а операция соединения очень похожа на

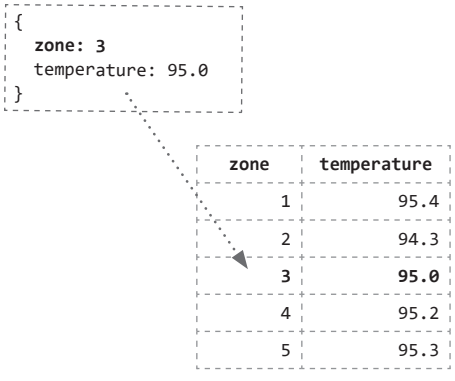
конструкцию `join` в базах данных SQL. Учтите, что к разным входящим потокам могут применяться разные оконные стратегии.



Так как весь процесс получается относительно стандартным, разработчики могут сосредоточиться на соединении, не беспокоясь о разной обработке потоков. Это может быть полезно при построении более сложных операторов соединения, и об этой возможности стоит знать. С другой стороны, задержка может быть не идеальной, потому что события обрабатываются не непрерывно, а небольшими пакетами. Разработчик сам выбирает лучший вариант в соответствии с требованиями.

Снова о материализации представлений

Ранее говорилось о том, что события температуры материализуются более эффективно, чем события автомобилей. Также упоминалось, что обычно события в одном специальном потоке обрабатываются одно за другим, а другие потоки материализуются во временные таблицы, но можно материализовать все потоки и соединить таблицы. Любознательный читатель наверняка спросит: можно ли соединять необработанные события температуры вместо материализованного представления?



Попробуем сохранить все события температуры в списке и обойтись без временной таблицы. Чтобы избежать нехватки памяти, мы будем удалять события температуры с возрастом более 30 минут. Для каждого события автомобиля необходимо найти последнюю температуру зоны в списке температур, сравнивая идентификатор зоны в событии автомобиля с идентификатором зоны для каждой температуры в списке. Конечный результат будет тем же, но с таблицей поиска (которая может быть представлена хеш-картой, деревом бинарного поиска или простым массивом с индексом — идентификатором зоны) поиск будет выполняться намного эффективнее. Из этого сравнения можно сделать вывод, что материализованное представление может рассматриваться как *оптимизация*. Собственно, материализованное представление стало популярным паттерном оптимизации во многих приложениях обработки данных.

> Материализация представления — популярный паттерн оптимизации в приложениях обработки данных.

Так как это оптимизация, мы можем более творчески подойти к управлению событиями, если существуют возможности повышения эффективности оператора. Например, на практике подобные датчики могут собирать намного больше информации — скажем, об уровне шума и качестве воздуха. Так как в этом задании нас интересует только температура в реальном времени в каждой зоне, всю остальную информацию можно отбросить, извлечь из событий только температуру и сохранить ее во временной таблице. В своих системах вы также можете попытаться создать несколько материализованных представлений одного потока или одно материализованное представление нескольких потоков, если это повысит эффективность задания.

Итоги

В этой главе была рассмотрена другая разновидность оператора объединения: соединение. Операторы соединения, как и операторы слияния, имеют несколько входящих потоков. Но вместо применения одной логики ко всем событиям разных потоков, в операторах соединения события разных потоков обрабатываются по-разному.

Как и в случае с конструкцией `join` в базах данных SQL, существуют разные типы соединений. Понимание соединений важно для решения проблемы целостности данных:

- *Внутренние соединения* возвращают только те результаты, которые имеют соответствующие значения в обеих таблицах.
- *Внешние соединения* возвращают результаты независимо от того, присутствуют соответствующие данные в обеих таблицах или нет.

Существуют три разновидности внешних соединений: полные внешние соединения (или полные соединения), левые внешние соединения (или левые соединения) и правые внешние соединения (или правые соединения).

В системе контроля выбросов CO_2 события автомобилей обрабатываются как поток, а события температуры используются как таблица поиска. Таблица является материализованным представлением потока. В конце главы вы также узнали, что соединения могут использоваться в сочетании с окнами, и познакомились с другими вариантами построения операторов соединения: материализацией всех входных потоков в таблицы и их соединением.

Обратное давление | 9

В этой главе

- ✓ Общие сведения об обратном давлении.
- ✓ Когда активизируется обратное давление.
- ✓ Как обратное давление работает в локальных и распределенных системах.

«Никогда не доверяйте компьютеру, который не сможете выкинуть из окна».

Стив Возняк

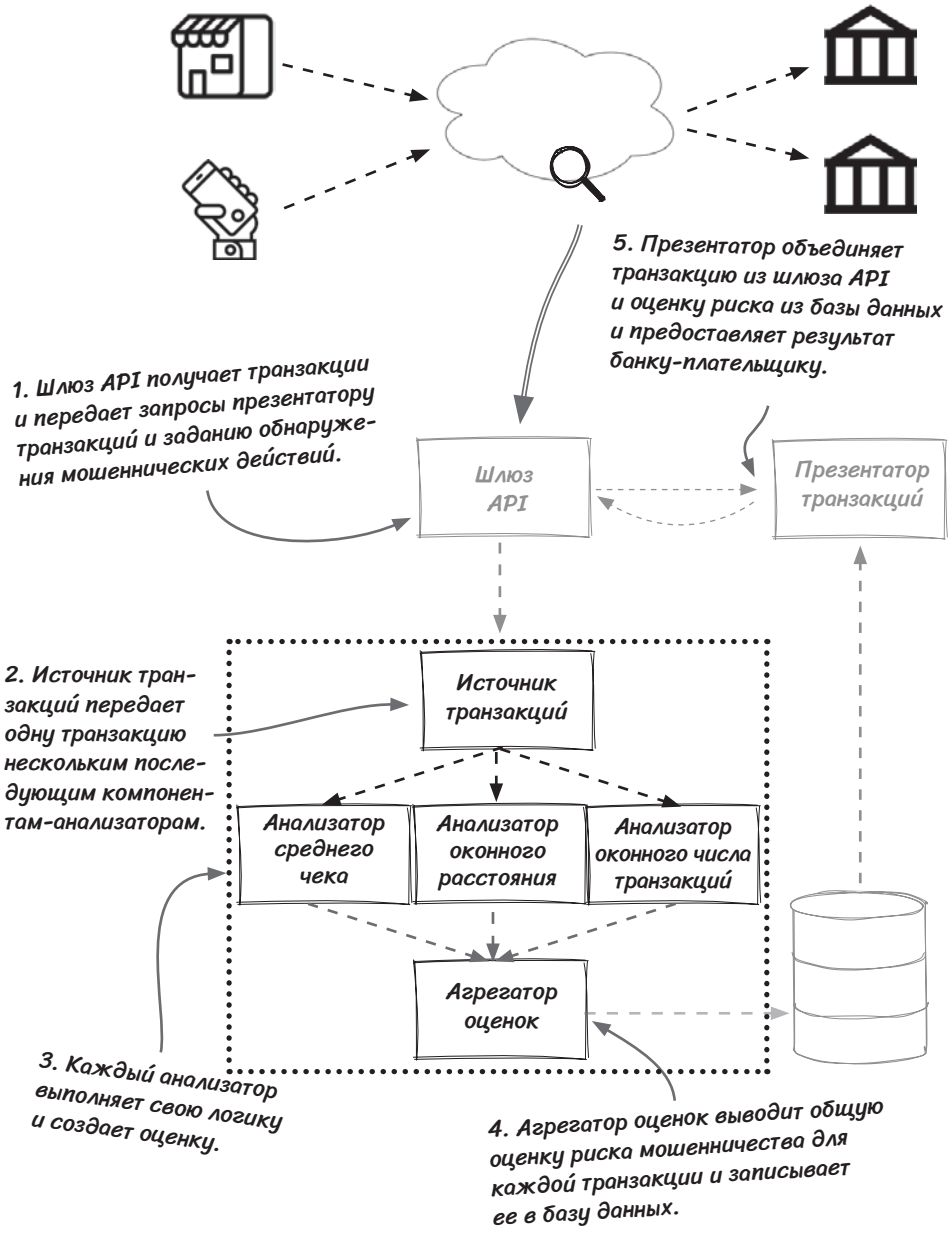
«Приготовьтесь к неожиданностям» — важнейшее правило при построении любых распределенных систем, и стриминговые системы не исключение. В этой главе рассматривается популярный механизм обработки ошибок в стриминговых системах: *обратное давление* (backpressure). Оно помогает защитить систему от сбоев в некоторых нестандартных ситуациях.

Надежность критична

В главе 4 наша команда построила стриминговую систему обработки транзакций и обнаружения попыток мошеннических действий с кредитными картами. Система хорошо работает, и заказчики пока довольны. Однако шеф беспокоится — и не зря.

Обзор системы

Прежде чем двигаться дальше, освежим в памяти структуру нашей системы.



Усовершенствование стриминговых заданий

Причина, по которой стриминговые системы находят все более широкое применение, — возможность получать данные по мере необходимости, но такую необходимость бывает сложно предсказать. Компоненты стриминговой системы или зависимой внешней системы (например, база данных оценок на диаграмме) могут не справляться с трафиком, и время от времени у них также могут возникать собственные проблемы. Рассмотрим несколько проблем, которые могут возникнуть в системе обнаружения мошеннических действий.

Обработка ошибок важна для всех распределенных систем, и наша система обнаружения мошеннических действий не исключение. В системе что-то может пойти не так, и очень важно принять защитные меры, предотвращающие возникновение проблем.

Повод для размышлений

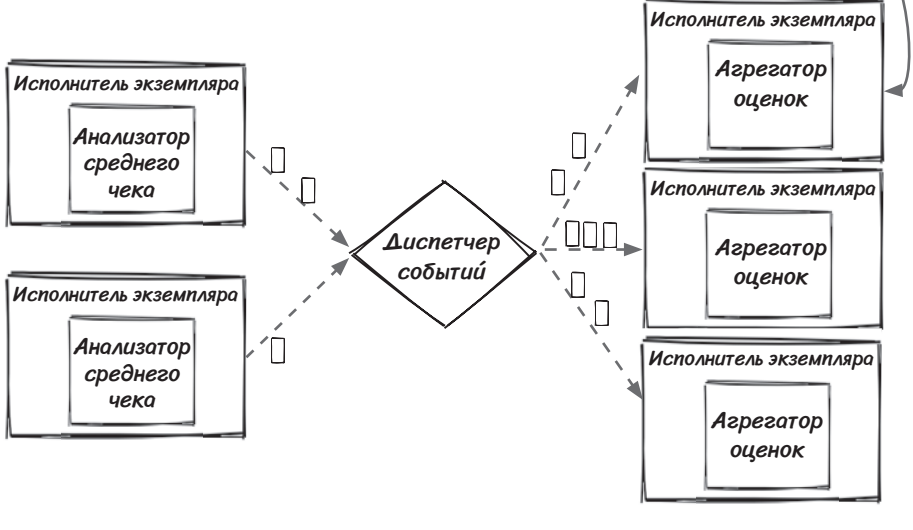
Что случится, если экземпляры начнут запаздывать или на них произойдет сбой?

Новые концепции: мощность, использование и резерв мощности

Следующие взаимосвязанные концепции пригодятся при обсуждении обратного давления:

- *Мощность* (capacity) — максимальное количество событий, которое может быть обработано экземпляром. На практике мощность измерить не так просто, поэтому для ее оценки часто применяются показатели использования процессора и памяти. Следует учитывать, что в потоковой системе количество событий, которые могут обрабатываться разными экземплярами, может сильно различаться.
- *Использование мощности* (capacity utilization) — отношение (выраженное в процентах) фактического количества обрабатываемых событий к мощности. В общем случае более высокое использование мощности означает более высокую эффективность.
- *Резерв мощности* (capacity headroom) — показатель, обратный использованию. Он представляет собой дополнительные события, которые могут обрабатываться экземпляром сверх текущего трафика. В большинстве случаев экземпляр с большим резервом мощности может быть более устойчивым к неожиданным данным или проблемам, однако его эффективность ниже, потому что больше ресурсов выделяется, но не используется в полной мере.

Допустим, максимальное количество событий, которые могут быть обработаны этим экземпляром, составляет 10 000 в секунду. Таким образом, мощность этого экземпляра составляет 10 000 событий в секунду (EPS, Events Per Second). Если предположить, что в настоящее время экземпляр обрабатывает 7500 событий в секунду, его использование составляет 75 %, а резерв мощности — 25 %.

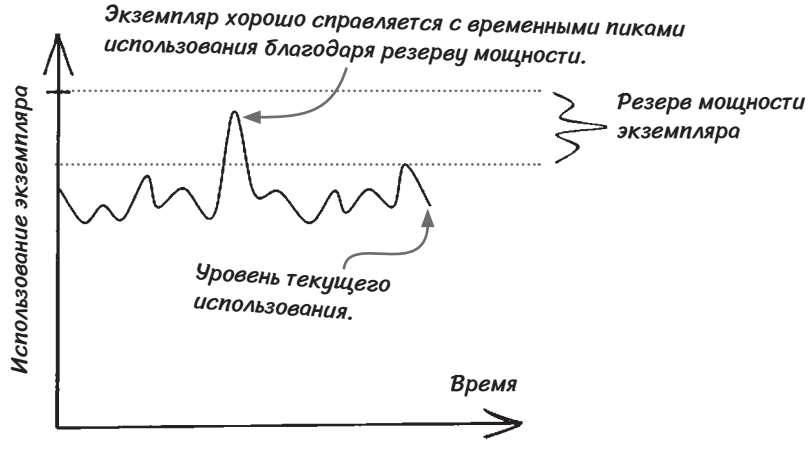


Подробнее об использовании и резерве мощности

В реальных системах время от времени происходит нечто неожиданное, что приводит к пиковым значениям использования. Примеры:

- Количество входящих событий может резко возрастать в отдельные моменты времени.
- Могут происходить сбои оборудования — например, перезапуск компьютера из-за сбоя питания или снижение пропускной способности, если канал связи занят другими процессами.

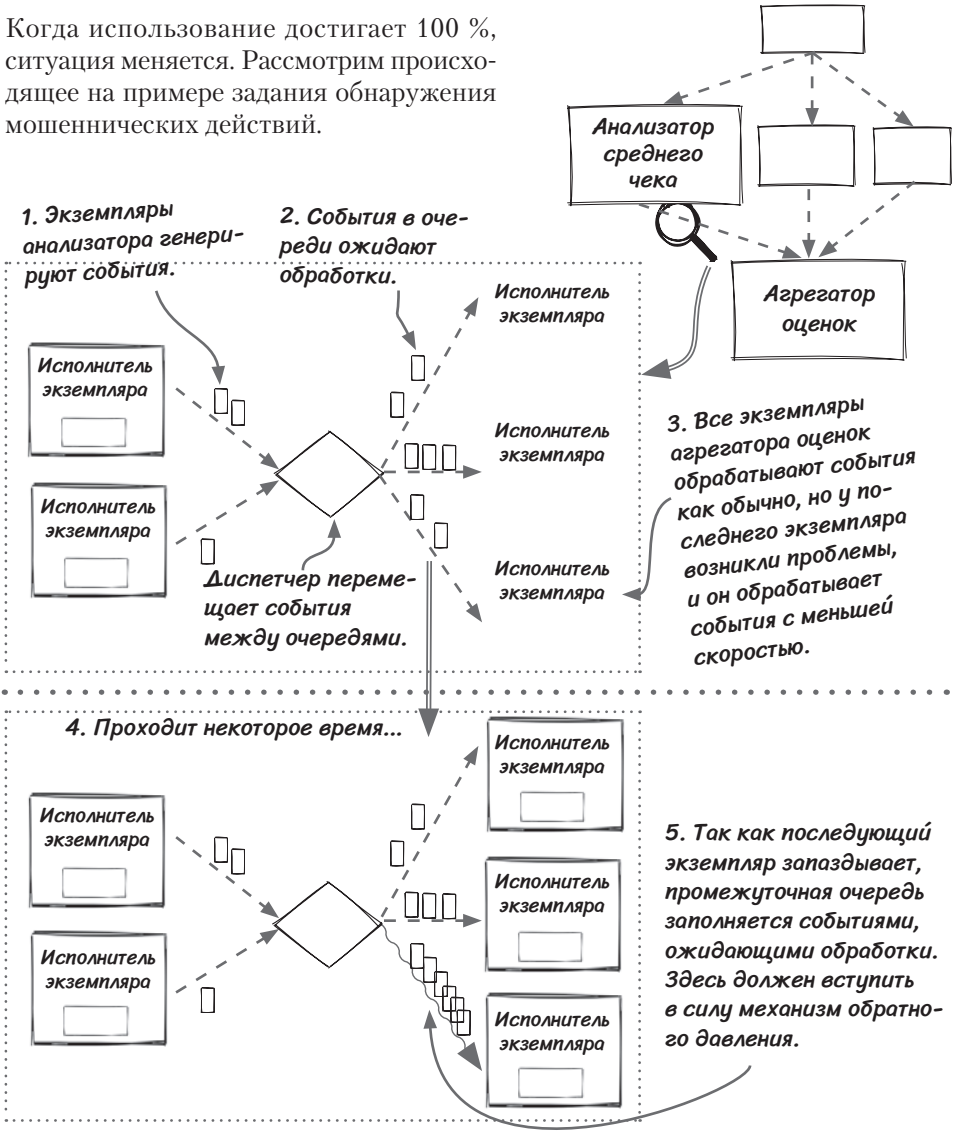
Важно учитывать эти потенциальные проблемы при построении распределенных систем. Грамотно написанное задание должно преодолевать эти временные трудности самостоятельно. В стриминговых системах при достаточном резерве мощности задание должно нормально выполняться без вмешательства со стороны пользователя. Тем не менее резерв мощности не может быть бесконечным (вдобавок он не бесплатен). Когда использование достигает 100 %, экземпляр становится полностью *загруженным* и следующей защитной мерой становится обратное давление.



- В стриминговом задании разные экземпляры могут иметь разные резервы мощности. Вообще говоря, резерв мощности компонента определяется минимальным резервом мощности всех его экземпляров; а резерв задания равен минимальному резерву всех экземпляров в задании. В идеале резервы мощности всех экземпляров в задании должны быть примерно одинаковы.
- В критических системах (таких как системы обнаружения мошеннических действий) желательно иметь достаточный резерв мощности для каждого экземпляра, чтобы задание было более устойчивым к возникновению неожиданных проблем.

Новая концепция: обратное давление

Когда использование достигает 100 %, ситуация меняется. Рассмотрим происходящее на примере задания обнаружения мошеннических действий.



Когда экземпляр становится полностью загруженным и не справляется с входящим трафиком, его входящая очередь растет и в конечном итоге заполняет память. Проблема распространяется на другие компоненты, и вся система перестает работать. Обратное давление — механизм, защищающий систему от сбоев.

Обратное давление по определению представляет собой давление в направлении, обратном направлению течения данных — от последующих экземпляров к предшествующим. Оно возникает тогда, когда экземпляр не может обрабатывать

события со скоростью входящего трафика, или, иначе говоря, когда использование достигает 100 %. Целью *обратного давления* является замедление входящего трафика, если трафик превышает возможности системы по его обработке.

Измерение использования мощности

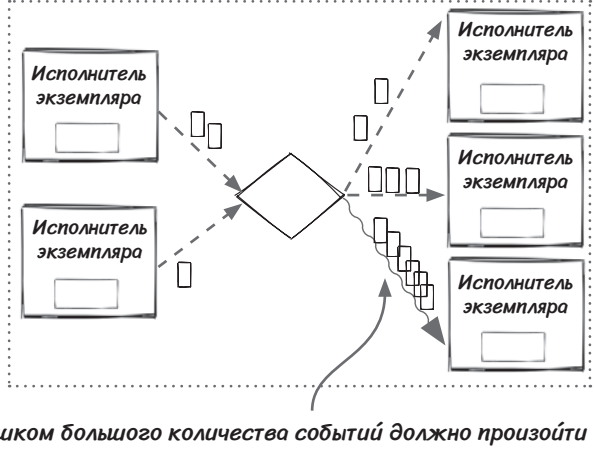
Обратное давление должно срабатывать, когда использование достигает 100 %. Но мощность и ее использование не так просто измерить или оценить. Существует множество факторов, ограничивающих количество событий, которые могут быть обработаны экземпляром: ресурсы, оборудование, данные и т. д. Уровни использования процессора и памяти полезны, но это ненадежная характеристика мощности. Требуется более эффективное решение; к счастью, оно существует.

Вы узнали, что работающая стриминговая система состоит из процессов и очередей событий, которые их связывают. Очереди событий отвечают за передачу событий между экземплярами (по аналогии с конвейерными лентами, связывающими рабочих на сборочной линии). Когда использование экземпляра достигает 100 %, скорость обработки начинает отставать от входящего трафика. В результате события во входящей очереди экземпляра начинают накапливаться. Таким образом, по длине входящей очереди экземпляра можно определить, достигло ли использование экземпляра своего максимума.

Обычно длина очереди должна возрастать и убывать в пределах относительно устойчивого диапазона. Если она возрастает непрерывно, то, скорее всего, экземпляр слишком занят и не справляется с обработкой трафика.

Далее мы сначала рассмотрим обратное давление на примере локального ядра Streamwork, чтобы вы поняли некоторые основные принципы, а затем перейдем к более общим распределенным фреймворкам.

Обратное давление особенно полезно для решения внешних проблем: перезапуска экземпляров, сопровождения зависимых систем, внезапных выбросов событий от источников. Стриминговая система устраняет их, временно замедляя работу и затем возобновляя ее без вмешательства пользователя. Следовательно, очень важно понимать, что может и что не может сделать обратное давление, чтобы при возникновении проблем не впадать в панику и держать ситуацию под контролем.

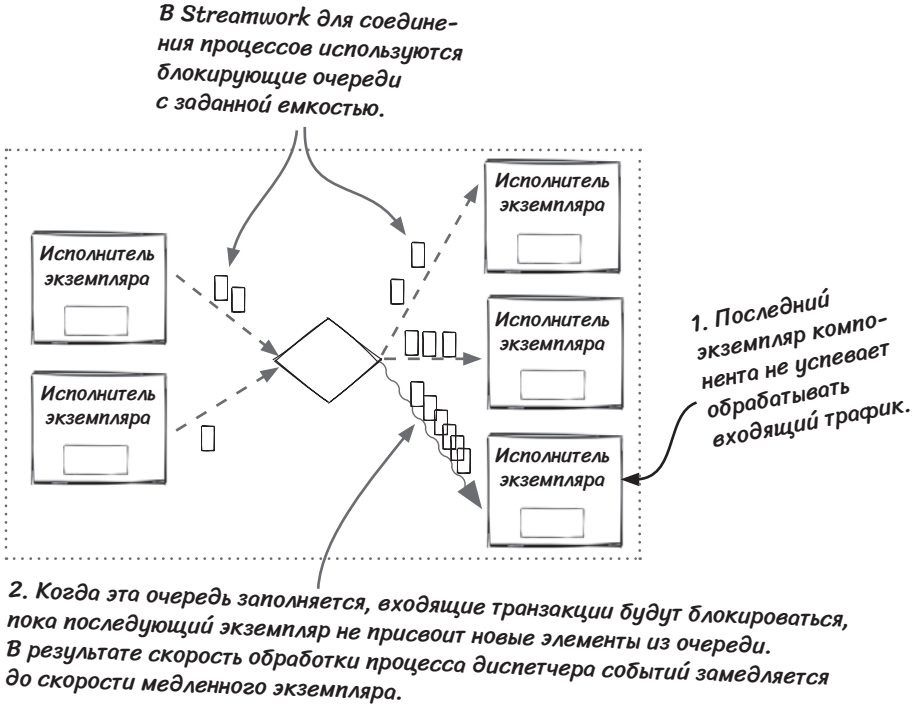


После накопления в очереди слишком большого количества событий должно произойти событие обратного давления, замедляющее поступление событий от предшествующих компонентов.

Обратное давление в ядре Streamwork

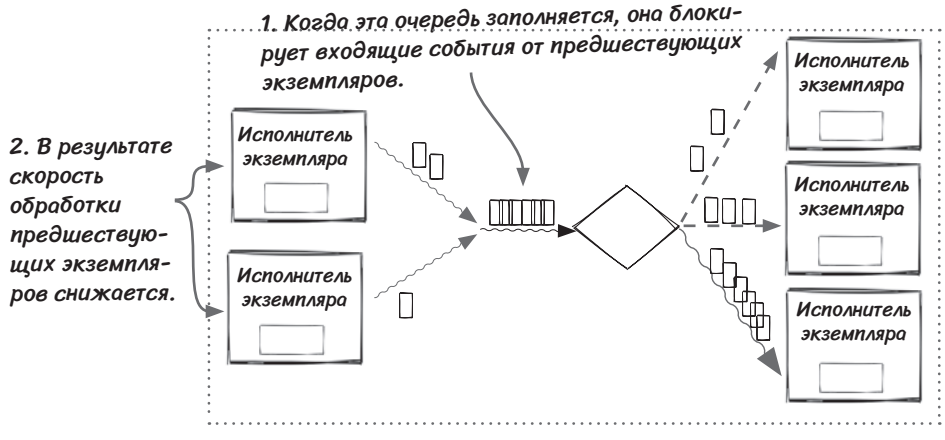
Начнем с нашего ядра Streamwork, так как оно устроено достаточно просто. Ядро Streamwork как локальная система не имеет сложной логики обратного давления. Тем не менее приведенная информация поможет понять, как работает обратное давление в реальных фреймворках.

В ядре Streamwork *блокирующие очереди* (очереди, которые могут приостанавливать потоки, пытающиеся присоединять новые события при заполненной очереди, или получать элементы при пустой очереди) используются для соединения процессов. Длины очередей ограничены — у каждой очереди существует максимальная *емкость*, которая играет ключевую роль в механизме обратного давления. Когда экземпляр не успевает достаточно быстро обрабатывать события, скорость уменьшения очереди перед ним будет ниже скорости вставки. Очередь начинает расти и со временем *заполняется*. В дальнейшем операция вставки блокируется до того, как событие перейдет в последующий экземпляр. В результате скорость вставки снижается до скорости обработки событий последующим экземпляром.

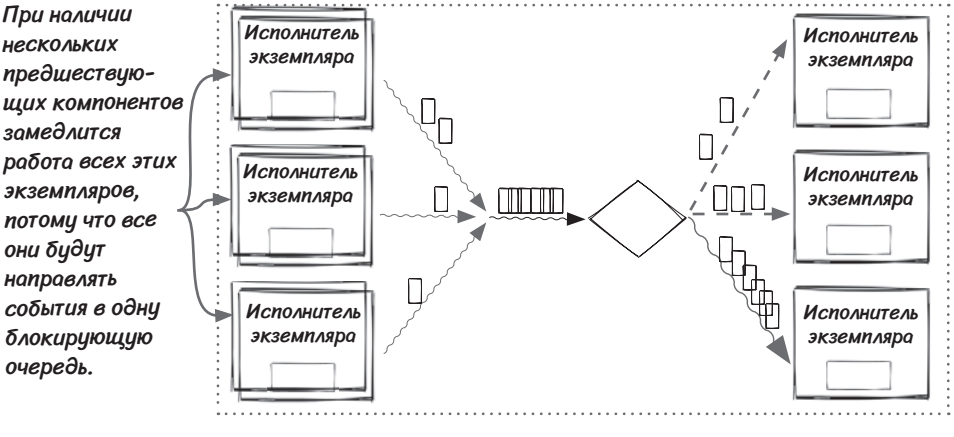


Обратное давление в ядре Streamwork: распространение

Замедление диспетчера событий не решает всех проблем. После того как диспетчер событий замедлится, то же самое произойдет с очередью, расположенной между ним и предшествующими экземплярами. Когда эта очередь будет заполнена, это повлияет на все экземпляры предшествующего компонента. Чтобы увидеть перед диспетчером событий блокирующую очередь, общую для всех предшествующих компонентов, рассмотрим работу обратного давления в ядре Streamwork более подробно на следующей диаграмме.

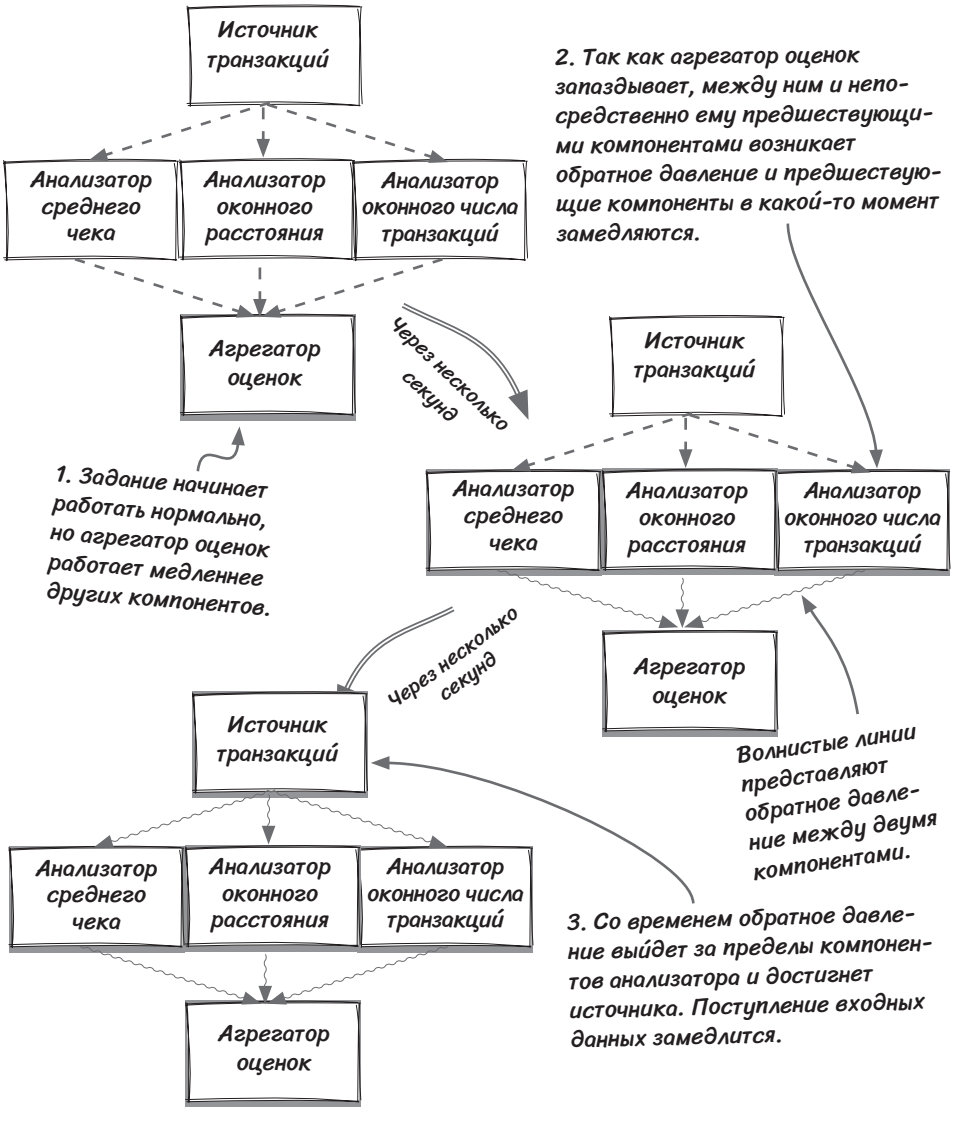


Если перед этим компонентом происходит объединение (то есть несколько непосредственно предшествующих компонентов для одного последующего компонента), это повлияет на все такие компоненты, потому что события блокируются в одной блокирующей очереди.



Стриминговое задание с обратным давлением

Посмотрим, как обратное давление с ядром Streamwork влияет на работу задания обнаружения мошеннических действий, когда экземпляр агрегатора оценок не справляется с входящим трафиком. Сначала только агрегатор оценок работает на пониженной скорости. Затем предшествующие анализаторы замедляются из-за обратного давления. Со временем обратное давление замедляет весь процесс выполнения задания, и его производительность понизится, пока проблема не будет решена.



Обратное давление в распределенных системах

В целом обнаружение и реализация обратного давления в локальных системах на базе блокирующих очередей достаточно просты. Однако в распределенных системах ситуация усложняется. Рассмотрим возможные сложности поэтапно:

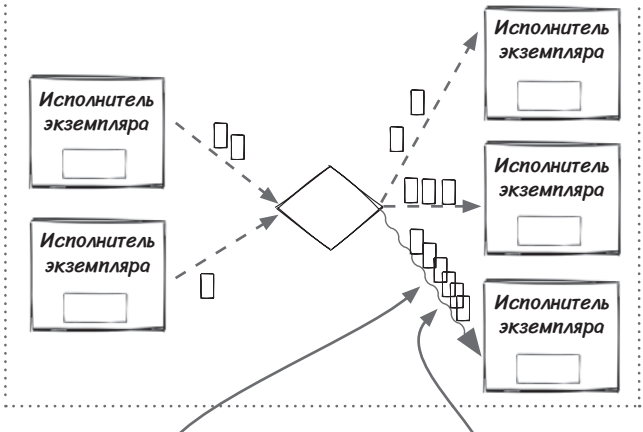
1. Обнаружение занятых экземпляров.
2. Состояние обратного давления.

Обнаружение занятых экземпляров

На первом шаге важно обнаружить занятые экземпляры, чтобы система могла реагировать с опережением. В главе 2 уже упоминалось, что такая структура данных, как очередь событий, широко применяется в стриминговых системах для соединения процессов. Хотя обычно используются неограниченные очереди, отслеживание размера очередей позволяет определить, успевает ли экземпляр обрабатывать входящий трафик. Если говорить конкретнее, есть минимум две разные метрики, которые могут использоваться для определения порога:

- Количество событий в очереди.
- Размер памяти, занимаемой событиями в очереди.

Когда количество событий или размер памяти достигают порогового значения, это означает, что у присоединенного экземпляра возникли проблемы. Ядро объявляет состояние обратного давления.



Состояние обратного давления

После объявления состояния обратного давления, как и в случае с ядром Streamwork, входящие события должны замедлиться. Тем не менее в распределенных системах эта задача часто оказывается намного более сложной, чем в локальных, потому что экземпляры могут работать на разных компьютерах и даже в разных регионах. По этой причине стриминговые фреймворки обычно останавливают обработку входящих событий вместо ее замедления, чтобы занятый экземпляр хотя бы временно получил ресурс мощности:

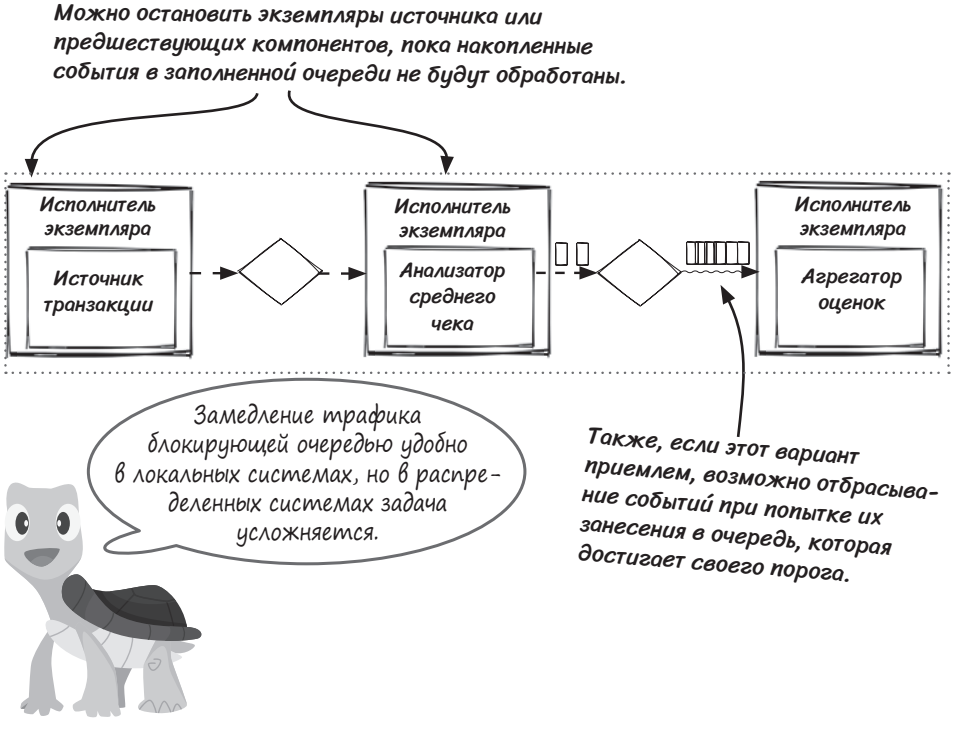
- останавливаются экземпляры предшествующих компонентов

или

- останавливаются экземпляры источников.

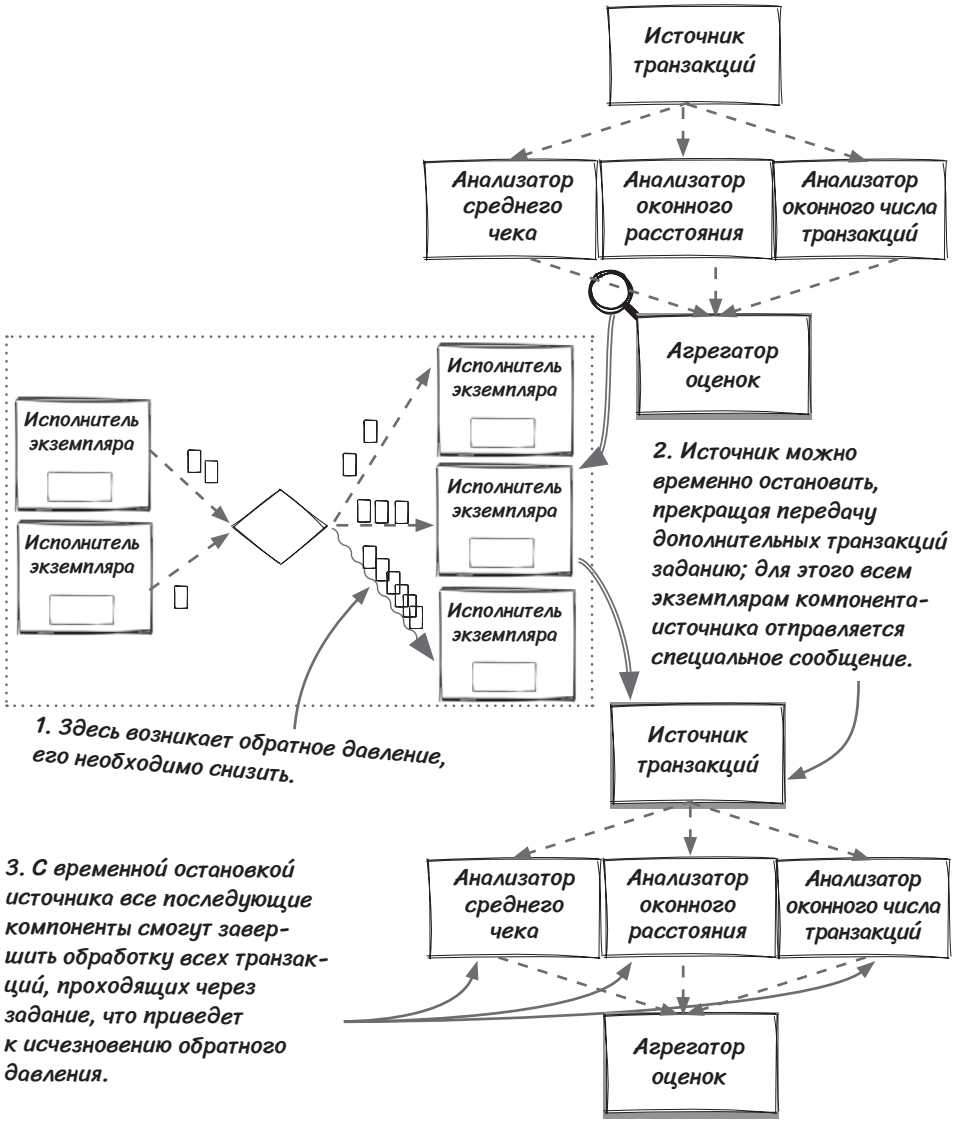
Далее в этой главе будет рассмотрен еще один вариант, хотя и намного менее популярный: отбрасывание событий. Этот вариант может показаться нежелательным, но и он может быть полезен, если задержка между конечными точками более критична, а потеря событий приемлема. По сути, выбор между этими двумя вариантами означает компромисс между точностью и задержкой.

Два варианта поясняются на следующих диаграммах. Мы добавили экземпляр источника, чтобы упростить объяснения, и для краткости опустили подробности промежуточных очередей и диспетчеров очередей.



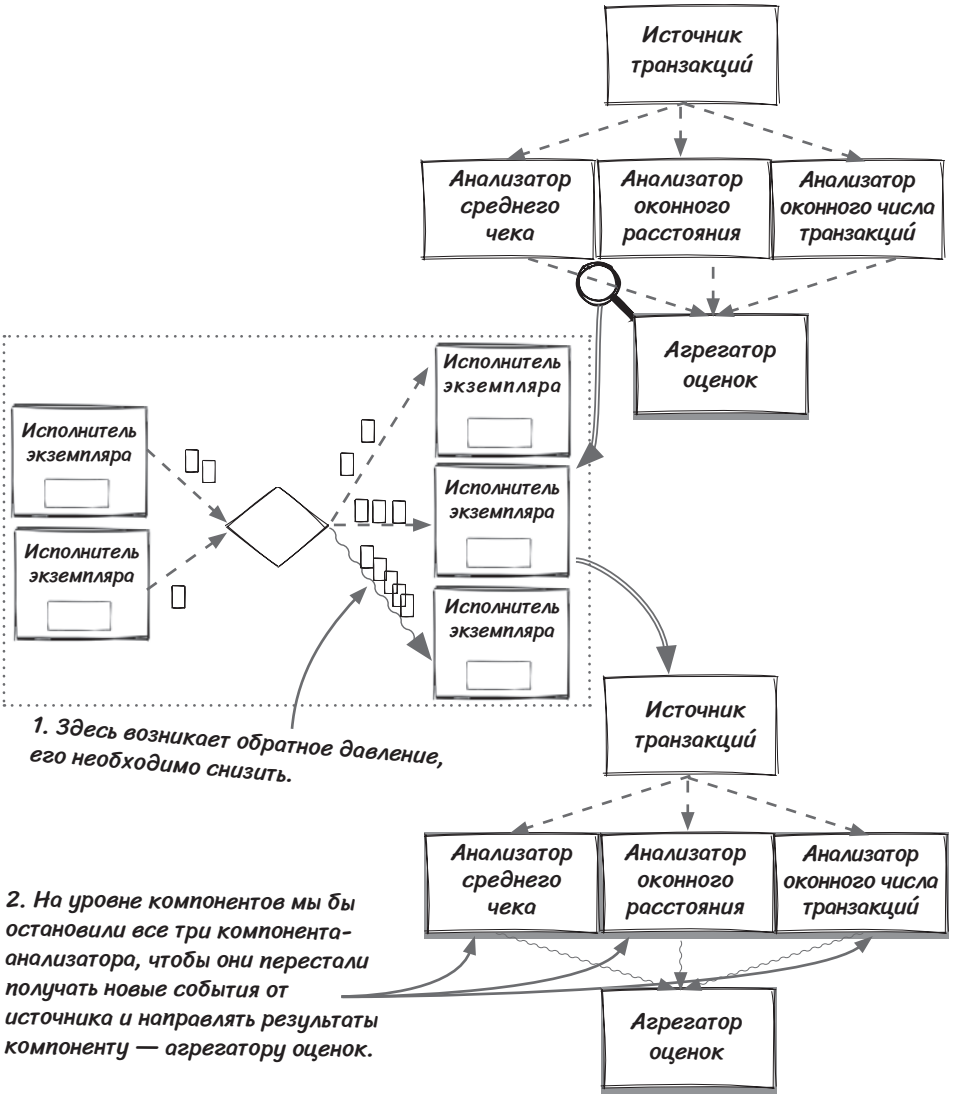
Обратное давление: остановка источников

Вероятно, остановка компонента-источника — самый простой способ снизить обратное давление в распределенных системах. Она позволяет приостановить поток входящих событий медленного экземпляра, а также всех остальных экземпляров стримингового задания; это может быть уместно в том случае, если существует вероятность появления нескольких занятых экземпляров.



Обратное давление: остановка предшествующих компонентов

Остановить поступление событий также можно на уровне компонентов. Этот способ будет более детализированным (до определенной степени), чем предыдущая реализация. Предполагается, что можно остановить только конкретные компоненты или экземпляры (вместо всех сразу) и снизить обратное давление до того, как оно распространится на все экземпляры. Если проблема не решается, со временем компонент-источник так или иначе будет остановлен. Учтите, что этот вариант может быть сложен для реализации в распределенных системах и требует больших затрат ресурсов.

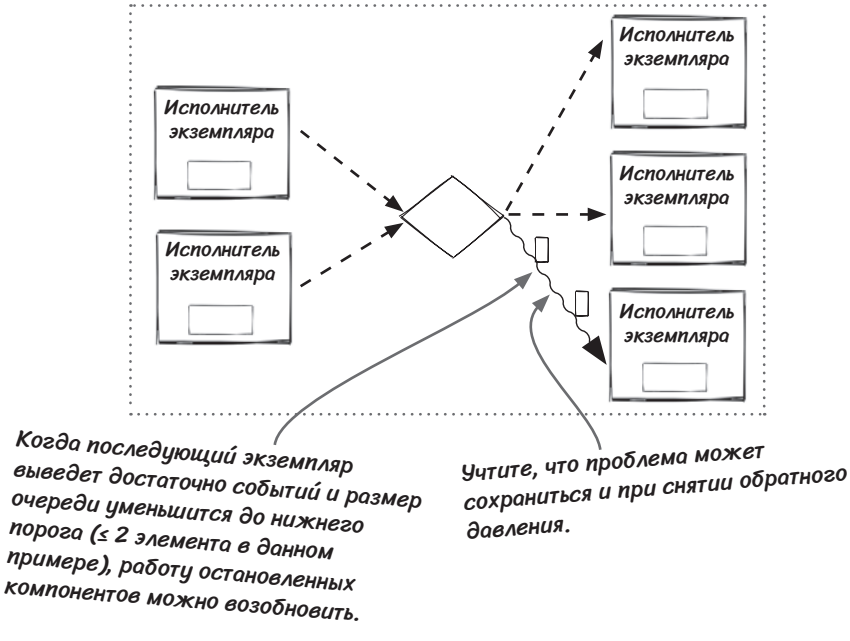


Снятие обратного давления

После того как задание пробудет в состоянии обратного давления в течение некоторого времени и занятый экземпляр восстановится (хочется надеяться), возникает следующий вопрос: что считать завершением состояния обратного давления, чтобы можно было восстановить нормальный трафик?

Решение вас вряд ли удивит, так как оно похоже на первый этап — обнаружение: нужно отслеживать размер очередей. Но, в отличие от этапа обнаружения, на котором мы проверяли, не переполнилась ли очередь, на этот раз мы проверяем, *достаточно ли она пуста*, то есть опустилось ли количество событий ниже минимального порога и появилось ли в очереди достаточно места для новых событий.

Обратите внимание: снятие обратного давления не означает, что медленный экземпляр восстановил работоспособность. Оно означает лишь то, что в очереди появилось место для новых событий.



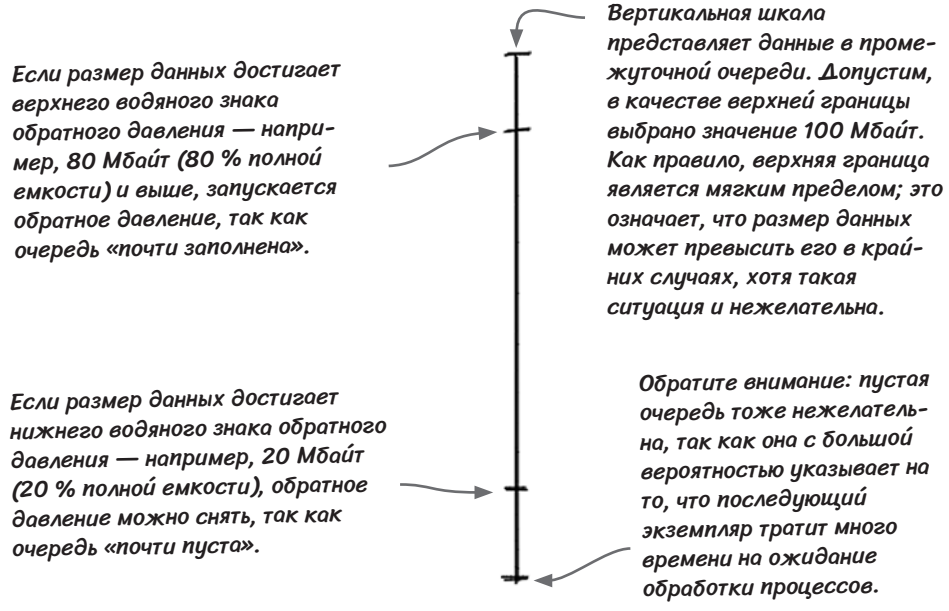
Важно помнить, что обратное давление представляет собой *пассивный* механизм, разработанный для защиты медленного экземпляра и всей системы от более серьезных проблем (например, аварийного останова). Этот механизм не решает проблему в медленном экземпляре и не заставляет его работать быстрее. В результате обратное давление может возникнуть снова, если медленный экземпляр все еще не может работать нормально после возобновления входящих событий. Сначала мы поближе рассмотрим пороги обнаружения и снятия обратного давления, а затем обсудим суть проблемы.

Новая концепция: водяные знаки обратного давления

Размеры промежуточных очередей проверяются и сравниваются с пороговыми значениями для включения и снятия состояния обратного давления. Рассмотрим эти два порога в рамках нового понятия: *водяных знаков обратного давления*. Обычно они являются параметрами конфигурации, предоставляемыми стриминговыми фреймворками:

- Водяные знаки обратного давления представляют собой верхний и нижний уровень использования промежуточных очередей между процессами.
- Когда размер данных в очереди выше верхнего водяного знака обратного давления, система должна объявить состояние обратного давления (если оно не было объявлено ранее).
- Если обратное давление активно, а размер данных в очереди, вызвавшей обратное давление, меньше нижнего водяного знака, обратное давление можно снять. Учтите, что нижний водяной знак обратного давления нежелательно задавать равным нулю, поскольку это означает, что предыдущий занятый экземпляр будет простаивать в промежуток времени между снятием обратного давления и поступлением новых событий в очередь.

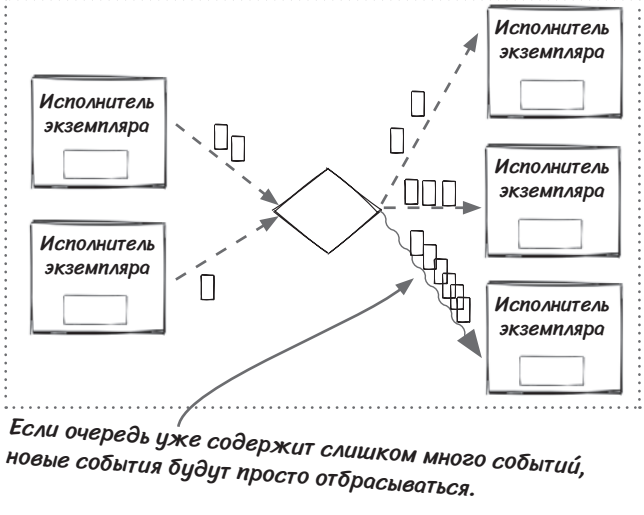
Размеры данных в очередях увеличиваются и уменьшаются при обработке событий заданием. В идеале они всегда находятся в промежутке между нижним и верхним водяными знаками обратного давления, чтобы события обрабатывались на полной скорости.



Другой подход к управлению отстающими экземплярами: отбрасывание событий

Обратное давление полезно для защиты систем и сохранения их работоспособности. Оно хорошо работает в большинстве случаев, но в некоторых особых ситуациях возможно другое решение: *отбрасывание событий*.

При таком подходе при обнаружении отстающего экземпляра вместо остановки/возобновления обработки входящих событий система просто игнорирует новые события, передаваемые во входящую очередь экземпляра.



Этот вариант выглядит пугающе, потому что результаты будут неточными. Да, это, безусловно, так. Если вы помните семантику доставки, о которой говорилось в главе 5, то заметите, что этот вариант должен применяться только в ситуациях *«не более одного»*.

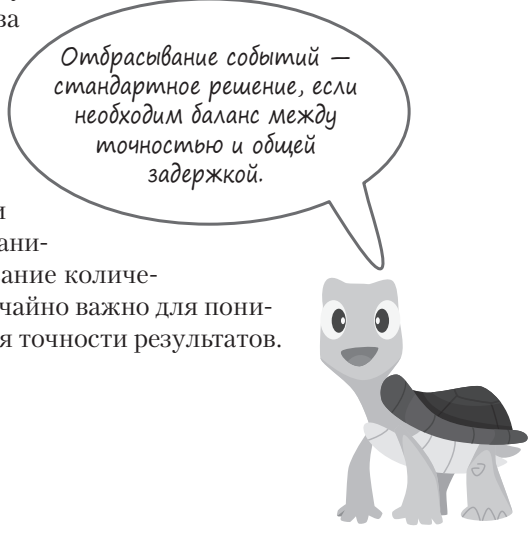
Тем не менее все не так страшно, как кажется. Результаты оказываются неточными, только если экземпляр не справляется с трафиком, что при правильной конфигурации системы происходит редко. Иначе говоря, результаты должны быть точными *почти* всегда. Мы уже несколько раз отмечали, что обратное давление представляет собой механизм самозащиты в экстремальных ситуациях, предотвращающий аварийный останов системы. Для стриминговых заданий состояние обратного давления не оптимально. Если оно возникает слишком часто, проверьте систему и попытайтесь найти и устранить первопричину.

Когда отбрасывание событий допустимо?

Разве кто-нибудь согласится, чтобы система теряла события? Не только вы задаете такой вопрос. В процессе разработки заданий спросите себя: готовы ли вы поступиться точностью ради снижения задержки в случае, если какой-то экземпляр не будет справляться с рабочей нагрузкой?

Возьмем для примера социальные сети и задачу отслеживания количества взаимодействий с пользователями (например, *лайков*) в реальном времени. Во втором варианте счетчик всегда будет содержать самые новые данные, хотя и не на 100 % точные. Если проблема затрагивает 1 экземпляр из 100, можно ожидать, что ошибка составит менее 1 %. Если применять обратное давление для остановки событий, счетчик будет точным, но вы не получите самые последние данные, пока система находится в состоянии обратного давления, потому что работа системы замедлится. После снятия этого состояния системе также понадобится время, чтобы наверстать отставание и вернуться к новейшим событиям. Вы не получите последнее значение, пока проблема не будет решена, что в случае длительного сбоя может быть хуже погрешности < 1 %. По сути, при отбрасывании событий вы получаете систему, *приближенную к реальному времени*, при *предположительно достаточной точности* результатов.

Вернемся к заданию обнаружения мошеннических действий — соблюдение сроков в нем критично. Приостановка обработки данных на несколько минут и нарушение требований к задержке до того, как проблема будет решена обратным давлением, в данном случае неприемлемы. Если сравнивать два варианта, то предпочтительнее обеспечить работу без задержек, даже при небольшой потере точности. Безусловно, о происходящем следует сообщить техническим специалистам, чтобы они проанализировали проблему и устранили ее как можно скорее. Отслеживание количества отброшенных событий чрезвычайно важно для понимания текущего состояния и уровня точности результатов.



Обратное давление как возможный симптом

Обратное давление — важный механизм решения временных проблем в стриминговых системах (например, сбоев экземпляров и внезапных пиков входного трафика), чтобы предотвратить более серьезные сбои. Стриминговые системы могут возвращаться в нормальное состояние автоматически после устранения причины без вмешательства со стороны пользователя. Иначе говоря, с обратным давлением стриминговые системы более устойчивы к непредвиденным проблемам, что обычно желательно для распределенных систем. Конечно, было бы идеально, если бы в системе никогда не возникало обратное давление, но, к сожалению, реальные условия от идеальных далеки. Обратное давление — необходимая мера предосторожности.

Хотя мы надеемся, что проблема временная и обратное давление справится с ней за нас, все зависит от ситуации. Вполне возможно, что экземпляр не восстановится сам по себе и для устранения причины потребуется внешнее вмешательство. В таких случаях постоянное обратное давление становится симптомом. Обычно существуют два варианта развития событий, и каждый требует особых действий.

- Экземпляр просто перестает работать, и обратное давление никогда не снимается.
- Экземпляр продолжает работать, но не успевает за входящим трафиком. Обратное давление снова активизируется вскоре после снятия.

Экземпляр перестает работать, обратное давление не снимается

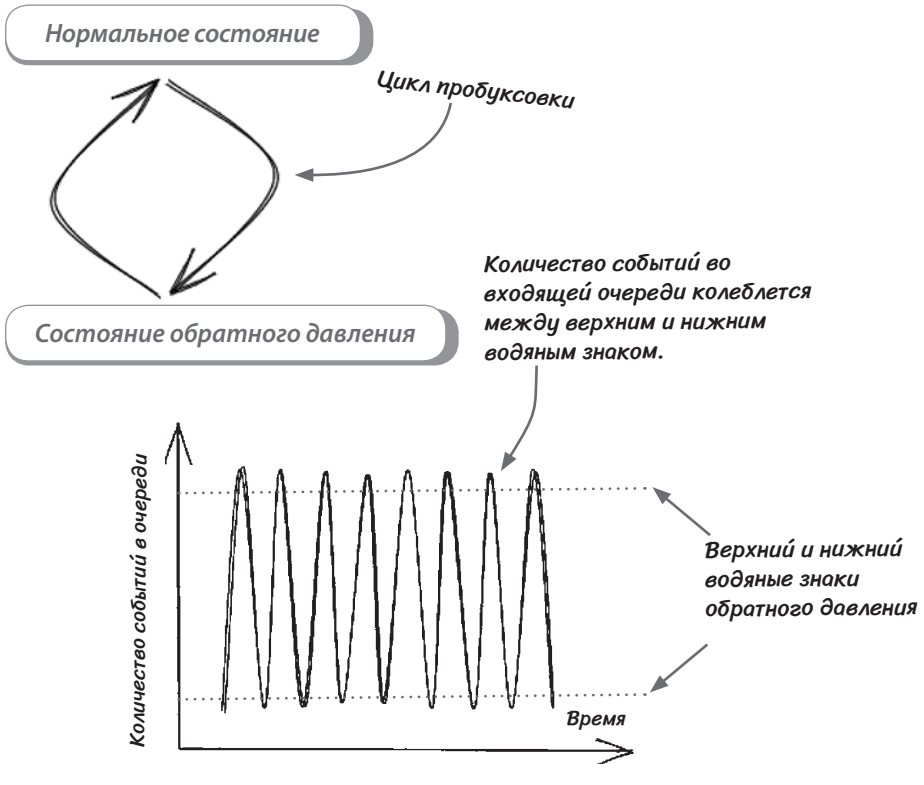
В этом случае события не уходят из очереди, а состояние обратного давления вообще не снимается. Проблема решается довольно просто: восстановлением работоспособности экземпляра. Восстановить ее можно путем перезапуска, но также важно найти первопричину проблемы и устранить ее. Часто проблема приводит к обнаружению ошибок, которые необходимо исправить.

Экземпляр не справляется, обратное давление возникает снова

Ситуация, в которой экземпляр не справляется с трафиком, более интересна. В этом случае обработка данных может временно восстановиться после уменьшения данных в очереди, но обратное давление вскоре возникает снова. Разберем этот случай подробнее.

Останов и возобновление работы могут привести к пробуксовке

А теперь рассмотрим эффект, называемый *пробуксовкой* (thrashing). Если проблема имеет долгосрочный характер, то при объявлении заданием состояния обратного давления события в очередях будут приниматься всеми экземплярами; но как только состояние обратного давления будет снято, новые события данных снова переполняют экземпляр и состояние вскоре объявляется заново. Пробуксовка — циклическое объявление и снятие обратного давления.



Пробуксовка ожидаемо возникает в том случае, если ситуация не изменяется. Если один экземпляр по-прежнему не справляется с обработкой трафика, размер данных в очереди увеличится снова, пока не достигнет верхнего предела, что приведет к повторной активизации обратного давления. А после следующего снятия обратного давления это с большой вероятностью произойдет снова. Количество событий во входящей очереди экземпляра изменяется так, как показано на диаграмме выше. Чтобы выйти из пробуксовки, необходимо найти первопричину и устранить ее.

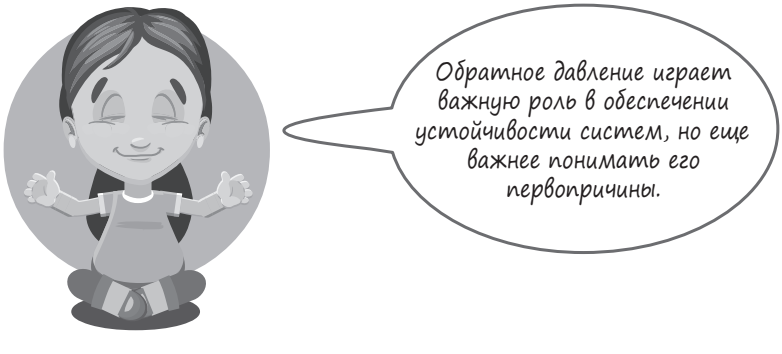
Решение проблемы пробуксовки

Если в системе наблюдается пробуксовка, выясните, почему экземпляр не обрабатывает события с нужной скоростью. Может быть, в нем возникла внутренняя проблема, замедляющая работу, или пришло время масштабировать систему? Как правило, у подобных проблем два источника — трафик и компоненты.

- Возможно, поток событий от источника повысился до постоянного уровня, с которым задание не справляется. В этом случае задание необходимо масштабировать, чтобы обрабатывать новый трафик. Если говорить конкретнее, возможно, стоит начать с повышения параллелизма (количества экземпляров компонента — подробности см. в главе 3) медленных компонентов задания.
- Скорость обработки некоторых компонентов может по какой-то причине снизиться. Проверьте компоненты, попробуйте их оптимизировать или перенастроить. Учитывайте зависимости, используемые компонентами. Нередко зависимости могут работать медленнее при изменении модели трафика.

Важно понимать данные и систему

Обратное давление возникает, когда экземпляр не может обрабатывать события со скоростью входящего трафика. Это мощный механизм, защищающий систему от сбоев, но для вас как владельца системы очень важно понимать данные и систему и разбираться в причинах возникновения обратного давления. В реальных системах могут возникать многочисленные проблемы, и рассмотреть их все в книге не удастся. Тем не менее мы надеемся, что понимание базовых концепций поможет вам сориентироваться.



Итоги

В этой главе рассматривался часто возникающий механизм *обратного давления*. Вы узнали:

- Когда и почему возникает обратное давление.
- Как стриминговые фреймворки обнаруживают проблемы и устраняют их, используя обратное давление.
- Остановка входящего трафика или отбрасывание событий — принципы работы, плюсы и минусы.
- Что можно сделать, если первопричина продолжает существовать.

Обратное давление — важный механизм. Надеемся, что знание принципов его работы пригодится вам при обслуживании и усовершенствовании стриминговых систем.

10 Вычисления с состоянием

В этой главе

- ✓ Что такое компоненты с состоянием и без состояния.
- ✓ Как работают компоненты с состоянием.
- ✓ Сопутствующие методы.

«А вы пробовали выключить, а потом снова включить?»

Сериал «Компьютерщики»

Концепция *состояния* рассматривалась в главе 5. Она играет важную роль во многих компьютерных программах. Например, прогресс в компьютерной игре, текущее содержимое текстового редактора, строки электронной таблицы, открытые страницы в браузере — все это *состояния* соответствующих программ. При повторном запуске программы после ее закрытия мы хотим восстановить нужное состояние. В стриминговых системах очень важно правильно организовать работу с состоянием. В этой главе мы более подробно обсудим, как используется состояние и как управлять им в стриминговых системах.

Миграция стриминговых заданий

Обслуживание — часть повседневной работы с распределенными системами. Несколько примеров: выпуск новой сборки с исправлением ошибок и новым функционалом, обновление программ или оборудования для повышения безопасности или эффективности систем, обработка отказов оборудования для обеспечения беспрерывной работы системы.

ЭйДжей и Сид решили перенести стриминговые задания на новое, более эффективное оборудование, чтобы сократить затраты и повысить надежность. Это довольно масштабная задача обслуживания, и очень важно действовать осторожно.

Компоненты с состоянием в задании контроля за использованием системы

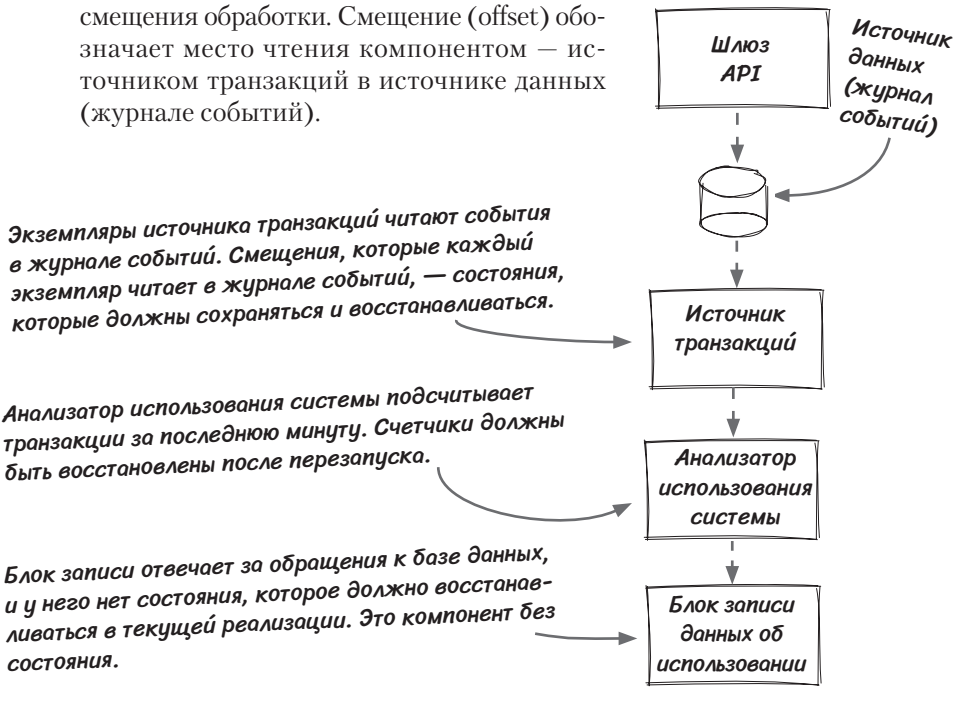
Компоненты с состоянием (stateful) очень полезны для построения компонентов с внутренними данными. Мы кратко говорили о них в главе 5 в контексте задания контроля за использованием системы. Пришло время поближе рассмотреть их и понять, как они работают.

> Компоненты с состоянием кратко рассматривались в предыдущих главах. В нашем стриминговом задании есть несколько точек, в которых они будут уместны.

Чтобы возобновить обработку после перезапуска стримингового задания, каждый экземпляр компонента должен заранее сохранить свои основные внутренние данные — состояние — в контрольной точке во внутреннем хранилище. После перезагрузки экземпляра данные загружаются обратно в память и используются для настройки экземпляра до перезапуска процесса.

Состав сохраняемых данных зависит от компонента. В нашем задании — контроля за использованием системы.

- Источник транзакций должен отслеживать смещения обработки. Смещение (offset) обозначает место чтения компонентом — источником транзакций в источнике данных (журнале событий).

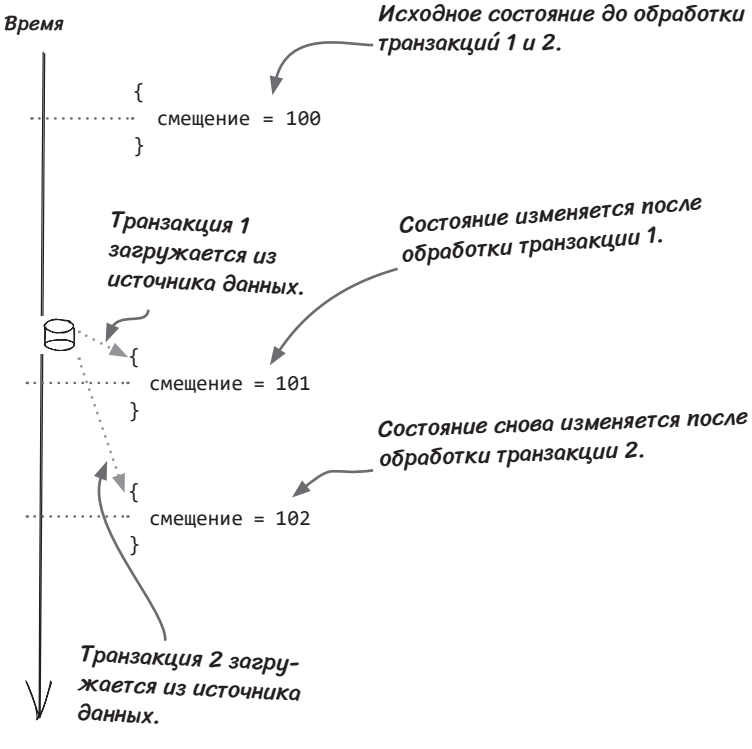


- Счетчики транзакций критичны для работы анализатора использования системы, их тоже необходимо сохранить.
- У блока записи данных об использовании нет данных, которые нужно сохранять или восстанавливать.

Следовательно, первые два компонента должны быть реализованы как *компоненты с состоянием*, а последний — как *компонент без состояния* (stateless).

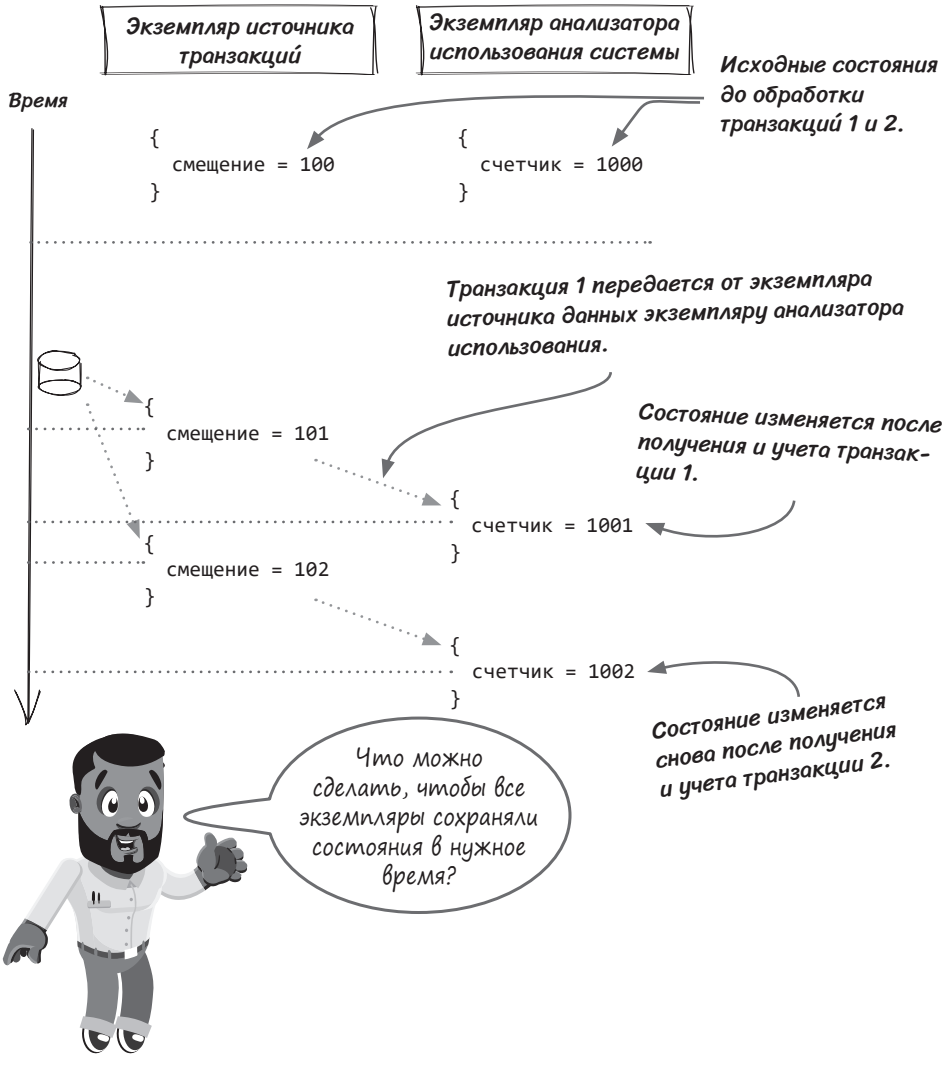
Снова о состоянии

Прежде чем двигаться дальше, вернемся к одной из фундаментальных концепций: что такое состояние? Как объяснялось в главе 5, состояние представляет собой внутренние данные каждого экземпляра, изменяемые по мере обработки событий. Например, состояние компонента — источника транзакций определяет место во внешнем источнике, откуда экземпляр загружает данные (смещение). Оно перемещается вперед после загрузки новых событий. Посмотрим, как состояние экземпляра источника транзакций изменяется до и после обработки двух экземпляров.



Состояния в разных компонентах

Все становится интереснее, когда мы рассматриваем совокупность состояний разных компонентов. В главе 7, посвященной оконным вычислениям, говорилось, что время обработки события должно быть разным для разных экземпляров, потому что событие передается от одного экземпляра к другому. Аналогичным образом для одного события в разных экземплярах изменения состояния происходят в разное время. Рассмотрим изменения состояния экземпляра источника транзакций и экземпляра анализатора использования системы до и после обработки двух транзакций.

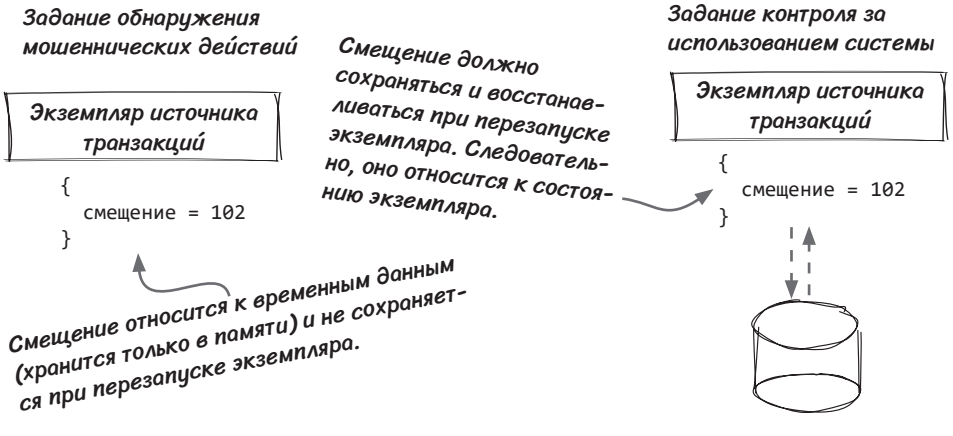


Данные состояния и временные данные

До сих пор *состояние* определялось просто: внутренние данные экземпляра, которые изменяются при обработке событий. Да, определение правильное, но некоторые данные состояния могут быть временными, и их не нужно восстанавливать при восстановлении экземпляра. Обычно временные данные *не* включаются в состояние экземпляра.

Например, *кэширование* — популярный метод улучшения быстродействия и/или эффективности. В процессе кэширования компонент, находящийся перед затратными или медленными вычислениями (например, сложной функцией или запросом к удаленной системе), сохраняет результаты запроса, чтобы вычисления не приходилось выполнять повторно. Обычно содержимое кэша не считается данными состояния экземпляра, хотя оно и может изменяться при обработке событий. В конце концов, экземпляр должен правильно работать с совершенно новым кэшем после перезапуска. Подключение к базе данных в каждом экземпляре блока записи данных об использовании также относится к временным данным, так как подключение придется создавать заново при перезапуске экземпляра.

Другой пример — компонент — источник транзакции в задании обнаружения мошеннических действий. Внутри каждый экземпляр хранит смещение события последней транзакции, загруженного из источника данных. Но, как упоминалось в главе 5, так как задержка критична для этого задания, при перезапуске экземпляра предпочтительно сразу перейти к последней транзакции, вместо того чтобы восстанавливать предыдущее смещение. В этом задании смещение относится к временным данным, и его не следует относить к данным состояния. А значит, этот компонент является *компонентом без состояния*.



Итак, состояние экземпляра включает только ключевые данные, чтобы экземпляр можно было развернуть в предыдущем состоянии и он мог продолжить нормально работать. Временные данные в стриминговых системах обычно не считаются данными состояния.

Компоненты с состоянием и без состояния: код

Компонент — источник транзакции есть как в задании контроля за использованием системы, так и в задании обнаружения мошеннических действий и работает в обоих случаях примерно одинаково. Единственное различие заключается в том, что в задании контроля он обладает состоянием, а в задании обнаружения мошенничества — нет. Ниже мы приводим коды обоих заданий, чтобы вы могли проследить за изменениями в компоненте с состоянием.

- Функция `setupInstance()` получает дополнительный параметр `state`.
- Вводится новая функция `getState()`.

Версия с состоянием в задании контроля.

```
class TransactionSource extends StatefulSource {
  EventLog transactions = new EventLog();
  int offset = 0;
  ......
  public void setupInstance(int instance, State state) {
    SourceState mstate = (SourceState)state;
    if (mstate != null) {
      offset = mstate.offset;
      transactions.seek(offset);
    }
  }

  public void getEvents(Event event, EventCollector eventCollector) {
    Transaction transaction = transactions.pull();
    eventCollector.add(new TransactionEvent(transaction));
    offset++;
    system.out.println("Reading from offset %d", offset);
  }

public State getState() {
    SourceState state = new SourceState();
    State.offset = offset;
    return new state;
  }
}
```

Данные в объекте состояния используются для настройки экземпляра.

Объект состояния экземпляра содержит текущее смещение в журнале событий.

Версия без состояния в задании обнаружения мошеннических действий.

```
class TransactionSource extends Source {
  EventLog transactions = new EventLog();
  int offset = 0;
  ......
  public void setupInstance(int instance) {
    offset = transactions.seek(LATEST);
  }

  public void getEvents(Event event, EventCollector eventCollector) {
    Transaction transaction = transactions.pull();
    eventCollector.add(new TransactionEvent(transaction));
    offset++;
    system.out.println("Reading from offset %d", offset);
  }
}
```

Источник с состоянием и оператор в задании контроля за использованием системы

В главе 5 был представлен код классов `TransactionSource` и `SystemUsageAnalyzer`. Теперь объединим их и сравним. В целом процессы с состоянием в источниках с состоянием и операторах протекают примерно одинаково.

```
class TransactionSource extends StatefulSource {
  MessageQueue queue;
  int offset = 0;
  ......
  public void setupInstance(int instance, State state) {
    SourceState mstate = (SourceState)state;
    if (mstate != null) {
      offset = mstate.offset;
      log.seek(offset);
    }
  }

  public void getEvents(Event event, EventCollector eventCollector) {
    Transaction transaction = log.pull();
    eventCollector.add(new TransactionEvent(transaction));
    offset++;
  }

public State getState() {
    SourceState state = new SourceState();
    State.offset = offset;
    return new state;
  }
}
```

Данные в объекте состояния используются для настройки экземпляра.

Значение offset изменяется при извлечении нового события из журнала событий и передаче его последующим компонентам.

Объект состояния экземпляра содержит текущее смещение в журнале событий.

```
class SystemUsageAnalyzer extends StatefulOperator {
  int transactionCount;

  public void setupInstance(int instance, State state) {
    AnalyzerState mstate = (AnalyzerState)state;
    transactionCount = state.count;
  }

  public void apply(Event event, EventCollector eventCollector) {
    transactionCount++;

    eventCollector.add(transactionCount);
  }

  public State getState() {
    AnalyzerState state = new AnalyzerState();
    State.count = transactionCount;
    return state;
  }
}
```

При формировании экземпляра объект состояния используется для его инициализации.

Переменная-счетчик изменяется при обработке событий.

Новый объект состояния периодически создается для хранения данных экземпляра.

Состояния и контрольные точки

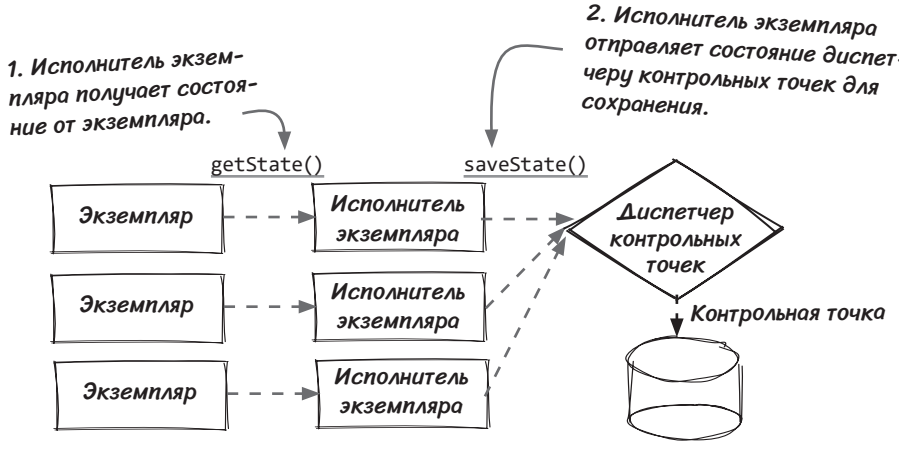
По сравнению с компонентами без состояния, рассмотренными выше, в компоненты с состоянием добавляются две функции, которые необходимо реализовать разработчикам:

- Функция `getState()` преобразует данные экземпляра в объект состояния.
- Функция `setupInstance()` использует объект состояния для реконструкции экземпляра.

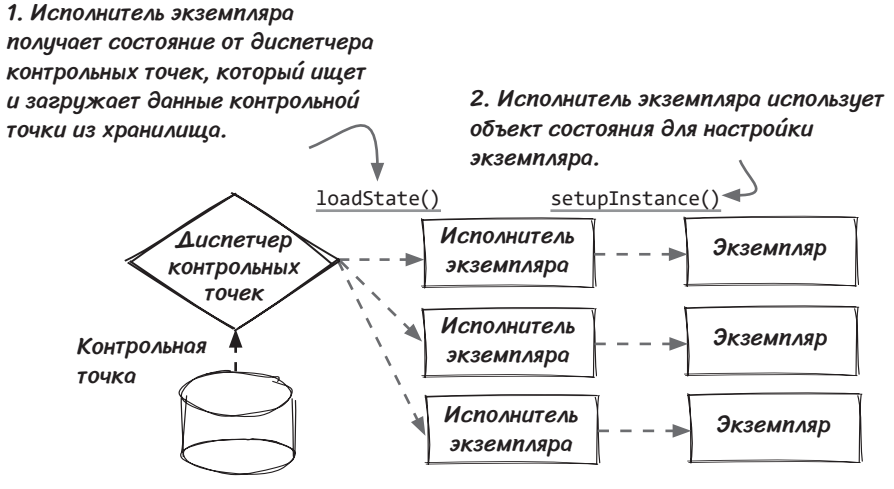
А теперь посмотрим, что происходит на самом деле. Понимание этого механизма поможет вам создавать эффективные и надежные задания и находить причины возникновения проблем.

В главе 5 *контрольная точка* была определена как «блок данных, который может использоваться экземпляром для восстановления предыдущего состояния». Стриминговое ядро — а конкретнее *исполнитель экземпляра* и *диспетчер контрольных точек* (помните принцип единой ответственности?) — отвечает за вызов двух функций в следующих двух случаях соответственно:

- Функция `getState()` периодически вызывается исполнителем экземпляра для получения последнего состояния каждого экземпляра, после чего объект состояния передается диспетчеру контрольных точек для создания контрольной точки.



- Функция `setupInstance()` вызывается исполнителем экземпляра после создания экземпляра, и диспетчер контрольных точек загружает последнюю контрольную точку.



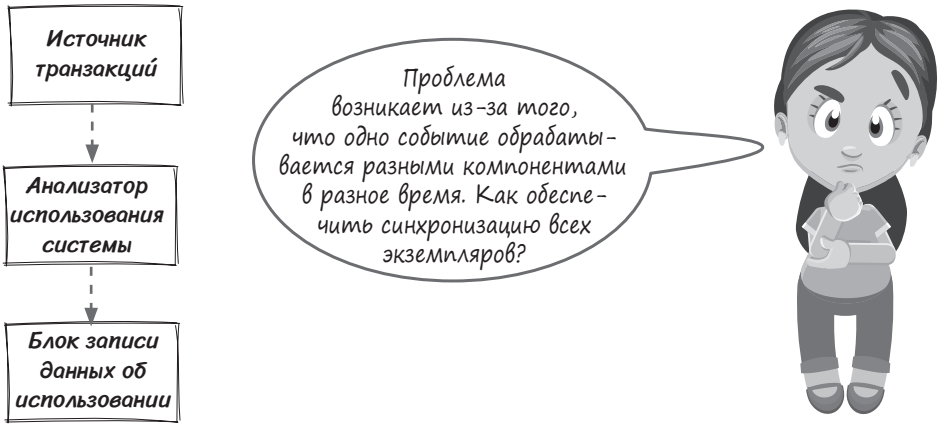
Создание контрольных точек: сложность выбора момента времени

В обязанности исполнителей экземпляров входит вызов функции `getState()` экземпляров для получения текущих состояний и их передачи диспетчеру контрольных точек для сохранения в контрольной точке. Остается понять, как исполнители экземпляров определяют правильный момент для запуска этого процесса.

Первая естественная мысль — создавать контрольные точки по часам. Все исполнители экземпляров запускают функцию одновременно. Создается полный «*моментальный снимок*» всей системы, как при переводе компьютера в спящий режим, когда все содержимое памяти сохраняется на диске и данные снова загружаются в память при пробуждении компьютера.

Тем не менее в стриминговых системах такой подход не работает. При запуске создания контрольной точки некоторые события могут быть обработаны частью компонентов, но при этом остаются необработанными последующими компонентами. Если контрольная точка будет создана подобным способом

и использована для реконструкции экземпляров, состояния разных экземпляров будут *рассинхронизированы*, и впоследствии вы получите неправильные результаты.



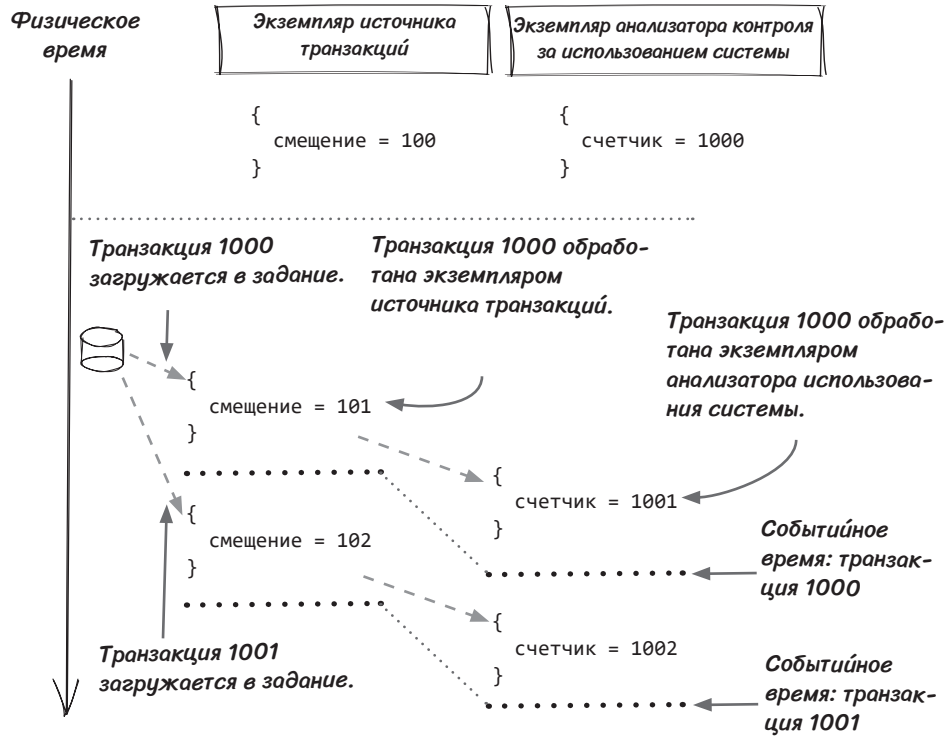
Например, в работающем стриминговом задании каждое событие обрабатывается экземпляром компонента-источника (источником транзакций в задании контроля за использованием системы), после чего отправляется правильному экземпляру последующего компонента (анализатору использования системы в задании контроля). Процесс повторяется, пока не останется ни одного последующего компонента. Таким образом, каждое событие обрабатывается в разное время в разных компонентах и в то же время разные компоненты работают над разными событиями.

Чтобы избежать проблемы рассинхронизации и сохранить правильность результатов, вместо сохранения состояний в один и тот же момент физического времени все экземпляры должны сохранять свои состояния в один и тот же момент *событийного* времени: сразу же после обработки одной и той же транзакции.

Событийный отсчет времени

При создании контрольных точек в стриминговых системах время измеряется не по часам, а по идентификатору события. Например, в задании контроля за использованием системы источник транзакции будет находиться во времени *транзакции 1001*, когда транзакция 1001 была обработана и выдана им. Анализатор использования системы в тот же момент физического времени будет находиться во времени *после транзакции 1000* и достигнет времени транзакции 1001 после получения, обработки и выдачи транзакции 1001. На следующей диаграмме физическое время и событийное время изображены параллельно. Для простоты будем предполагать, что каждый компонент существует только

в одном экземпляре. Случай множественных экземпляров будет рассмотрен позже, когда мы будем говорить о реализации.



При использовании событийного времени все экземпляры могут сохранить свое состояние одновременно для создания действительной контрольной точки.

Создание контрольных точек с использованием событий контрольных точек

Как же событийное время реализуется в стриминговых фреймворках? Как и события, отсчет времени строится в стриминговом контексте, о котором постоянно говорится в этой книге. Интересно?

Событийный отсчет времени в целом выглядит не особо сложно, но есть проблема: как правило, каждый компонент создается в нескольких экземплярах и каждое событие обрабатывается одним из экземпляров. Как эти экземпляры синхронизируются друг с другом? В этом помогает новая разновидность событий — *управляющие события*, использующие другую стратегию маршрутизации по сравнению с событиями данных.

До сих пор все наши стриминговые задания обрабатывали *события данных*, такие как события автомобилей и транзакции по кредитным картам. Управляющие события не содержат данных для обработки. Вместо этого в них передаются данные, при помощи которых все модули в стриминговом задании взаимодействуют друг с другом. В случае контрольных точек используется *событие контрольной точки*, обязанностью которого становится оповещение всех экземпляров стримингового задания о том, что пришло время создать контрольную точку. Возможны и другие типы управляющих событий, но в книге рассматриваются только события контрольных точек.

Периодически диспетчер контрольных точек в задании создает событие контрольной точки с уникальным идентификатором и направляет его компоненту-источнику или, говоря точнее, исполнителям экземпляров компонента-источника. Исполнители экземпляров включают события контрольных точек в поток обычных событий данных. С этого начинается путешествие событий контрольных точек.

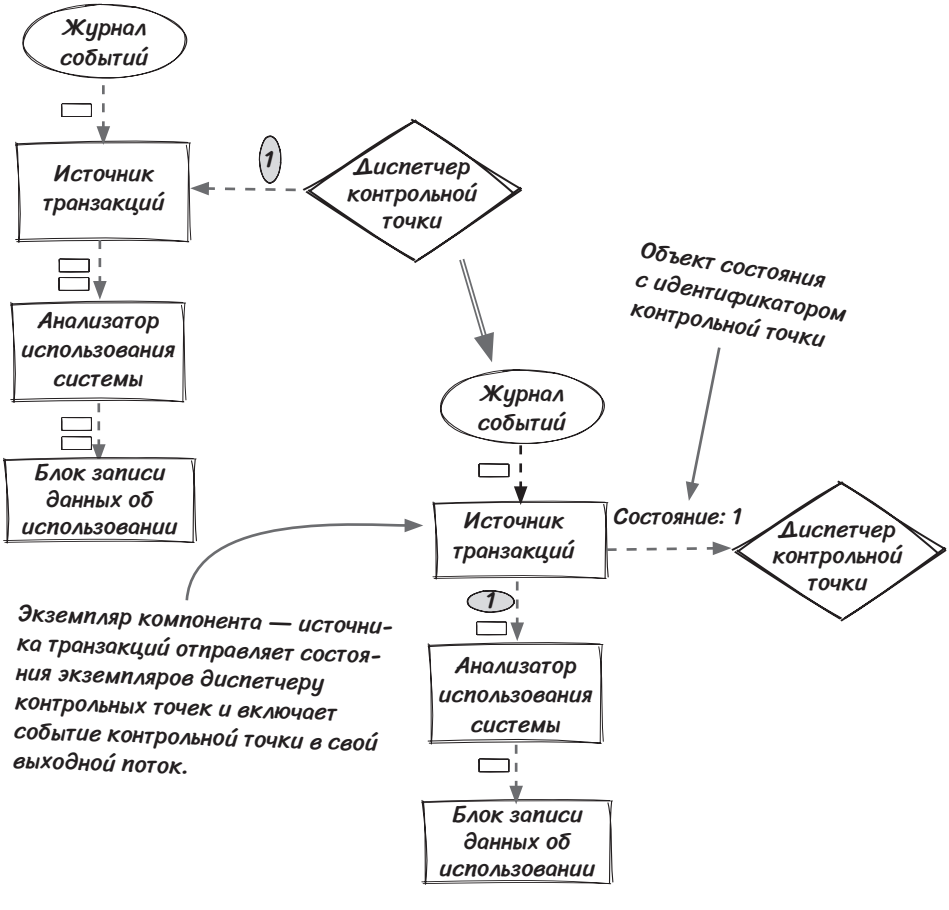
Учтите, что экземпляры компонента-источника, содержащие пользовательскую логику, не знают о существовании события контрольной точки. Им известно лишь то, что функция `getState()` вызывается исполнителем экземпляра для извлечения текущих состояний.

Событие контрольной точки обрабатывается исполнителями экземпляров

Все исполнители экземпляров повторяют одни и те же действия:

- Вызов функции `getState()` и отправка состояния диспетчеру контрольных точек.
- Включение события контрольной точки в ее выходной поток.

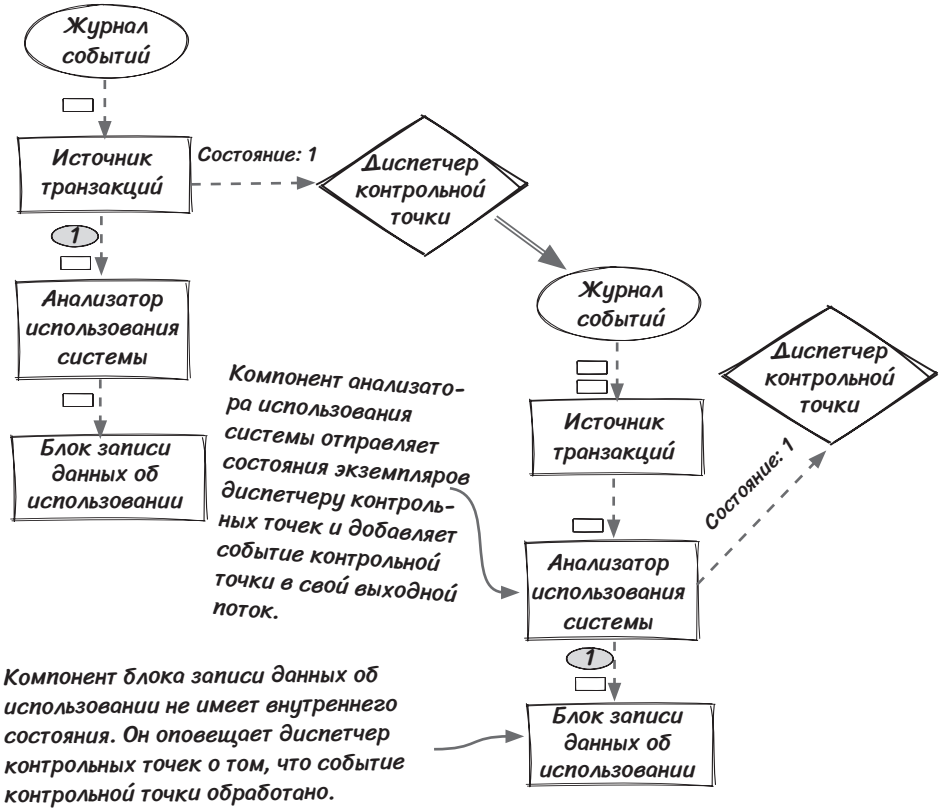
Внимательно посмотрев на следующую диаграмму, вы увидите, что каждое событие контрольной точки также содержит *идентификатор контрольной точки*. Идентификатор можно рассматривать как событийное время. Когда исполнитель экземпляров отправляет объект состояния диспетчеру контрольных точек, в него включается идентификатор состояния, так что диспетчер контрольных точек знает, что экземпляр на тот момент находится в данном состоянии. Идентификатор также включается в объект контрольной точки для той же цели.



Путь события контрольной точки через задание

После того как событие контрольной точки будет добавлено в поток событий исполнителями экземпляров источника, оно пройдет через задание и всех исполнителей экземпляров операторов в нем. На двух диаграммах, приведенных ниже, показано, что событие контрольной точки с идентификатором 1 последовательно обрабатывается компонентами источника транзакций и анализатора использования системы.

Последний компонент, блок записи данных об использовании, не имеет состояния, поэтому он оповещает диспетчер контрольных точек о том, что событие обработано без объекта состояния. Диспетчер контрольных точек узнает, что событие контрольной точки посетило все компоненты в задании, контрольная точка *завершена* и ее можно зафиксировать в хранилище.



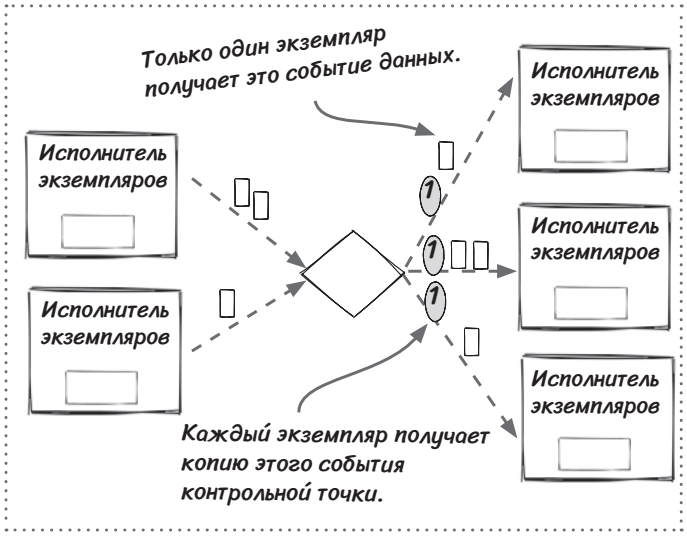
В целом событие контрольной точки проходит по заданию примерно так же, как обычное событие, но отличия все же имеются. Рассмотрим происходящее более подробно.

Создание контрольных точек с использованием событий контрольных точек на уровне экземпляров

Событие контрольной точки переходит от компонента к компоненту. Объекты состояния отправляются диспетчеру контрольных точек по очереди исполнителями экземпляров при получении события контрольной точки. В результате все состояния создаются между двумя конкретными событиями (200 и 201) для каждого компонента в приведенном примере.

Не стоит забывать, что каждый компонент может существовать в нескольких экземплярах. В главе 4 вы узнали, что каждое событие передается конкретному экземпляру в зависимости от стратегии группировки. Маршрутизация событий контрольных точек осуществляется иначе; рассмотрим ее более подробно. (Некоторым читателям материал на этой и следующей странице может показаться слишком подробным. Вы можете пропустить его и перейти к теме загрузки контрольных точек.)

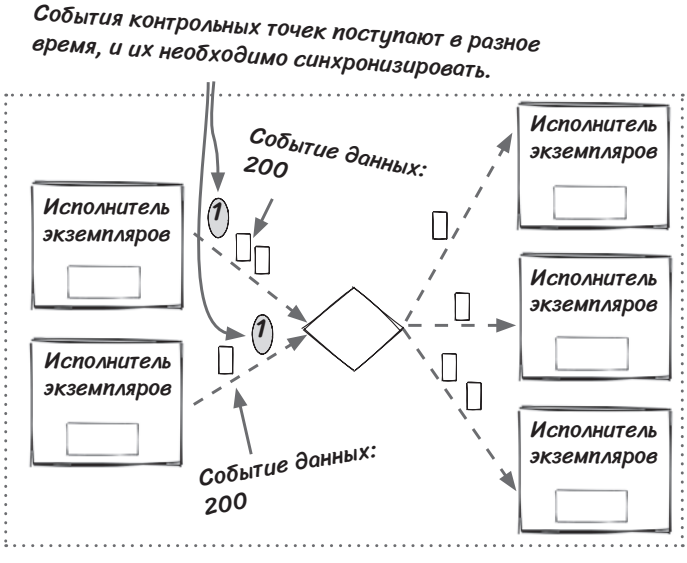
Простой ответ выглядит так: *все экземпляры* должны получить событие контрольной точки, чтобы вызовы `getState()` были правильно инициированы. В нашем фреймворке Streamwork диспетчер событий отвечает за *синхронизацию* и *диспетчеризацию* события контрольной точки. Начнем с диспетчеризации (так как эта тема проще), а синхронизацию рассмотрим ниже.



Когда диспетчер событий получает событие контрольной точки от предшествующего компонента, он направляет по одной копии события *каждому* экземпляру последующего компонента. С другой стороны, событие данных обычно получает только один экземпляр последующего компонента.

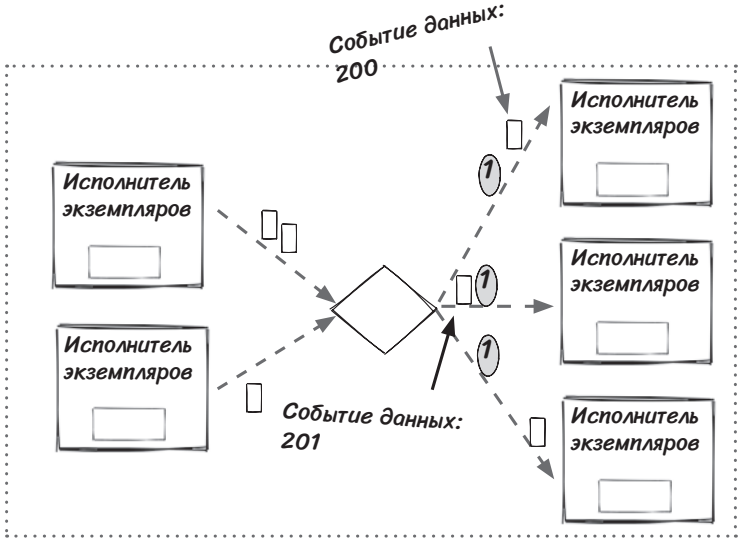
Синхронизация событий контрольных точек

Механизм диспетчеризации событий контрольных точек достаточно прост, но с синхронизацией дело обстоит сложнее. Синхронизация событий контрольных точек — процесс получения диспетчером событий входящих событий контрольных точек. Каждый диспетчер событий получает события от нескольких экземпляров (собственно, он также может получать события от экземпляров нескольких компонентов), то есть ожидается одно событие контрольной точки от каждого предшествующего исполнителя экземпляра. Эти события контрольных точек редко прибывают одновременно, как в примере на приведенной диаграмме. Что же делать в подобных случаях?



Если взглянуть на приведенную выше диаграмму и принять во внимание событийный отсчет времени, *время*, представляемое событием контрольной точки 1, *лежит между событиями данных 200 и 201*. Событие контрольной точки будет получено всеми исполнителями экземпляров, поэтому возможно, что событие контрольной точки будет обработано одним экземпляром раньше других, как на диаграмме. В этом случае после получения первого события контрольной точки диспетчер событий *блокирует* поток событий, из которого поступило событие контрольной точки, до того момента, когда событие контрольной точки будет получено из всех остальных входящих соединений. Иначе говоря, событие контрольной точки работает как *барьер* (или *блокировщик*). В приведенном примере событие контрольной точки сначала поступает из нижнего соединения. Диспетчер событий блокирует обработку события

данных 201 и продолжает обрабатывать события (событие данных 200 и то, которое ему предшествует) от входящего соединения, пока не будет получено событие контрольной точки.



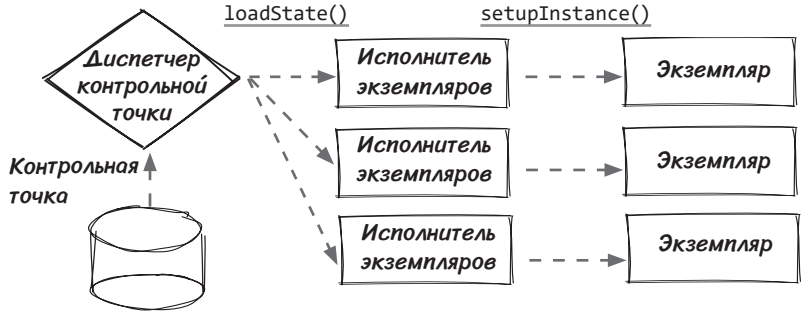
После того как событие контрольной точки 1 будет получено из обоих соединений, поскольку другие входные соединения, которые следовало бы ожидать, отсутствуют, диспетчер событий передает событие контрольной точки всем последующим исполнителям экземпляров и начинает принимать события данных. В результате событие данных 200 передается диспетчером событий до события контрольной точки 1 и события данных 201.

Загрузка контрольных точек и обратная совместимость

Итак, мы рассмотрели процесс создания контрольных точек. Теперь разберемся, как контрольные точки загружаются и используются. В отличие от повторяющегося процесса создания, загрузка контрольных точек осуществляется только один раз в каждом жизненном цикле стримингового задания — при запуске.

Когда стриминговое задание запускается (например, в случае сбоя — когда в экземпляре произошел аварийный останов и задание перезапускается на тех же машинах; экземпляры задания переместились на другие машины, как при миграции, над которой работают ЭйДжей и Сид), каждый исполнитель экземпляра запрашивает данные состояния соответствующего экземпляра от диспетчера

контрольных точек. В свою очередь, диспетчер контрольных точек обращается к хранилищу контрольных точек, ищет последнюю контрольную точку и возвращает данные исполнителям экземпляров. Каждый исполнитель экземпляра затем использует полученные данные состояния для настройки экземпляра. После того как все экземпляры будут успешно созданы, стриминговое задание начинает обрабатывать события.

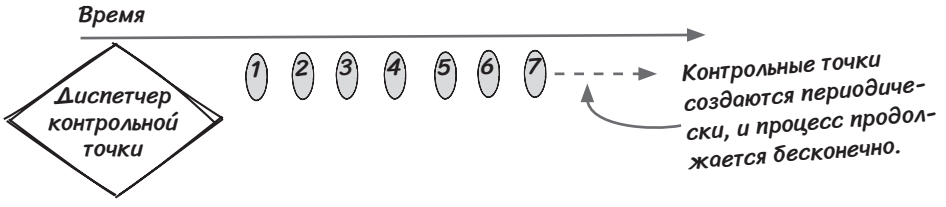


Весь процесс достаточно прост, но в нем есть одна сложность — *обратная совместимость*. Контрольная точка была создана при *предыдущем* запуске задания, и данные состояния в контрольной точке используются для создания *новых* экземпляров. Если задание просто перезапускается (вручную или автоматически), проблем быть не должно, так как логика экземпляров остается прежней. Но если логика существующих компонентов с состоянием изменится, разработчики должны позаботиться о том, чтобы новая реализация работала с прежними контрольными точками для правильного восстановления состояний экземпляров. Если это требование не выполняется, задание может запуститься в некорректном состоянии или вообще перестать работать.

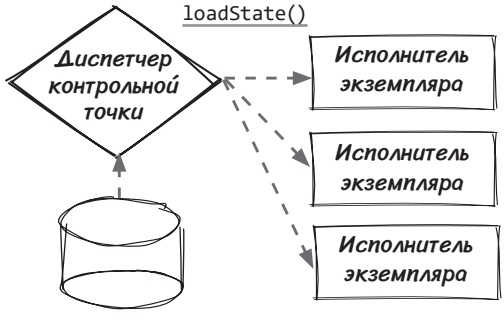
Некоторые стриминговые фреймворки создают особую разновидность контрольных точек: *точки сохранения*. Они похожи на обычные контрольные точки, но активируются вручную, и разработчик имеет больше возможностей для управления ими. Это фактор, который разработчики должны учитывать при выборе стримингового фреймворка для систем.

Хранилище контрольных точек

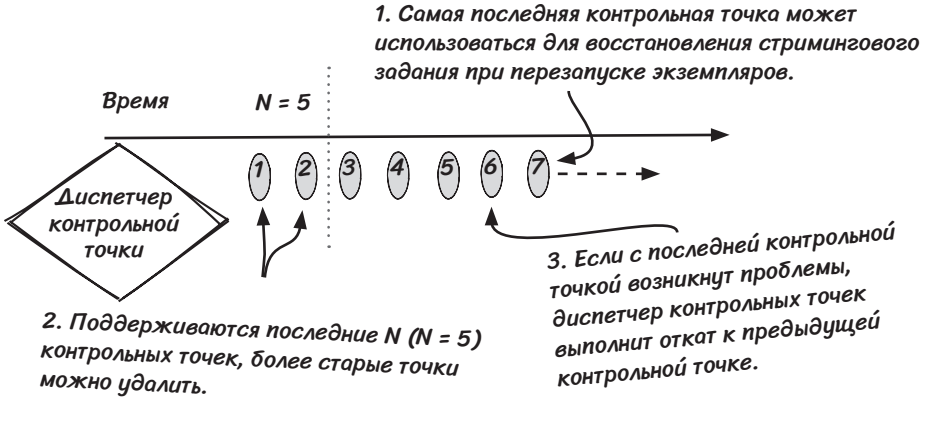
Последняя тема, относящаяся к контрольным точкам, — хранилища. Контрольные точки обычно создаются периодически с *монотонно возрастающими* идентификаторами, и этот процесс, управляемый ядром, продолжается до остановки стримингового задания.



При перезапуске экземпляров только *самая последняя* контрольная точка используется для их инициализации. Теоретически можно хранить только одну контрольную точку для стримингового задания и обновлять ее «на месте» при создании новой точки.



Тем не менее на практике все сложнее. Например, создание контрольной точки может завершиться неудачей, если какие-либо экземпляры будут потеряны и контрольная точка не будет завершена, или данные контрольной точки могут быть повреждены из-за отказа диска и невозможности тем самым загрузить их. Для повышения надежности стриминговых систем в хранилище обычно оставляют N последних контрольных точек, а более старые точки удаляют (как правило, значение N можно настраивать). Если самая последняя контрольная точка не настраивается, диспетчер контрольных точек выполняет откат к предыдущей контрольной точке и пытается восстановить стриминговое задание. При необходимости откат может выполняться повторно, пока контрольная точка не будет успешно загружена.



Компоненты с состоянием и компоненты без состояния

Вы достаточно узнали о компонентах с состоянием и без состояния. Пора прерваться, взглянуть на общую картину и оценить достоинства и недостатки компонентов с состоянием. В конце концов, у компонентов с состоянием есть и обратная сторона. На самом деле вопрос звучит так: стоит ли использовать компоненты с состоянием?

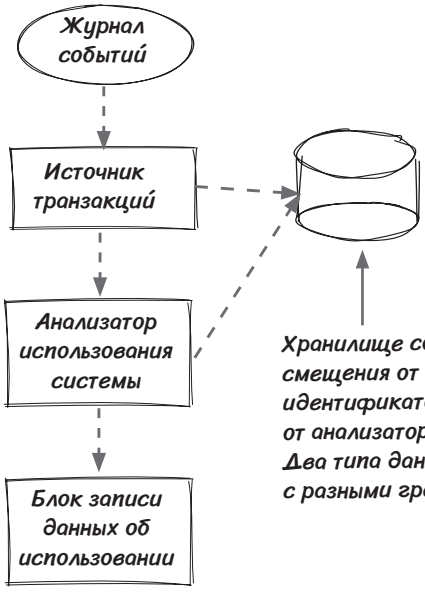
Окончательный ответ на него можете дать только вы — разработчик. В разных системах действуют разные требования. Хотя функциональность систем может быть примерно одинаковой, они могут работать абсолютно по-разному, поскольку входящий трафик событий обладает разными характеристиками: пропускной способностью, размером данных, количеством экземпляров и т. д. Надеемся, приведенное ниже краткое сравнение поможет вам принимать лучшие решения и строить лучшие системы. В оставшейся части этой главы мы поговорим о двух практических методах поддержки полезной функциональности компонентов с состоянием в компонентах без состояния.

	Компонент с состоянием	**Компонент без состояния**
Точность	Вычисления с состоянием важны для семантики «ровно один», которая гарантирует точность (фактически)	Гарантии точности отсутствуют, потому что фреймворк не управляет состоянием экземпляров
Задержка (при возникновении ошибок)	При возникновении ошибок экземпляры выполняют откат к предыдущему состоянию	При возникновении ошибок экземпляры продолжают работать над новыми событиями
Потребление ресурсов	Управление состоянием экземпляров требует больших затрат ресурсов	Управление состоянием экземпляров не требует ресурсов
Трудоемкость сопровождения	Приходится поддерживать больше процессов (например, диспетчер контрольных точек или хранилище контрольных точек), обратная совместимость критична	Дополнительные затраты на сопровождение отсутствуют
Пропускная способность	Если управление контрольными точками не оптимизировано, пропускная способность может снизиться	Высокая пропускная способность не требует дополнительных затрат
Код	Необходимо реализовать управление состоянием экземпляров	Дополнительная логика отсутствует
Зависимости	Необходимо хранилище контрольных точек	Внешние зависимости отсутствуют

Компоненты с состоянием используются только там, где это необходимо. Обычно это делается для того, чтобы по возможности упростить задание и снизить затраты на обслуживание.

Ручное управление состоянием экземпляров

Из сравнения становится видно, что преимуществом компонентов с состоянием является точность. Когда в системе что-то происходит и некоторые экземпляры приходится перезапускать, стриминговые ядра помогают в управлении и выполнении отката состояний экземпляров. Кроме дополнительных затрат, управляемые ядром состояния имеют и другие ограничения. Одно очевидное ограничение заключается в том, что контрольные точки не должны создаваться слишком часто, потому что в этом случае затраты на обслуживание повышаются и система становится менее эффективной. Может оказаться, что для некоторых компонентов желательно использовать разные интервалы, а это невозможно с управлением на уровне ядра. Таким образом, иногда стоит рассмотреть возможность ручного управления состоянием. Рассмотрим его на примере задания контроля за использованием системы.



На следующей диаграмме показано задание контроля за использованием системы с подключенным хранилищем состояния. Разные экземпляры сохраняют свои состояния в хранилище независимо. Как обсуждалось ранее, абсолютное время не работает, потому что разные экземпляры работают с разными событиями. И поскольку мы управляем состояниями вручную, теперь у нас нет событий контрольных точек, обеспечивающих событийный отсчет времени. Что можно сделать для синхронизации разных экземпляров?

Ключ к решению проблемы — создание чего-то общего, что могут использовать все компоненты и экземпляры для взаимной синхронизации. Одно из возможных решений — идентификатор транзакции. Например, экземпляры источников транзакций хранят смещения, а экземпляры анализаторов использования системы каждую минуту сохраняют идентификаторы транзакции и текущие счетчики в хранилище. При перезапуске задания экземпляры источников транзакций загружают смещение из хранилища и затем *возвращаются* на несколько событий (или даже несколько минут) назад и перезапускаются с этой точки. Экземпляры анализатора использования системы загружают последние идентификаторы транзакций и счетчики из хранилища. Впоследствии экземпляры анализаторов могут пропускать входящие события, пока не обнаружат идентификаторы транзакций в состояниях, после чего можно возобновить обычный подсчет. При таком решении источники транзакций и анализаторы использования системы могут управлять состояниями своих экземпляров по-разному, потому что два компонента уже не связаны идентификаторами контрольных точек. В результате затраты ресурсов снижаются, а также обеспечивается большая гибкость, что может быть важно в некоторых реальных сценариях использования.

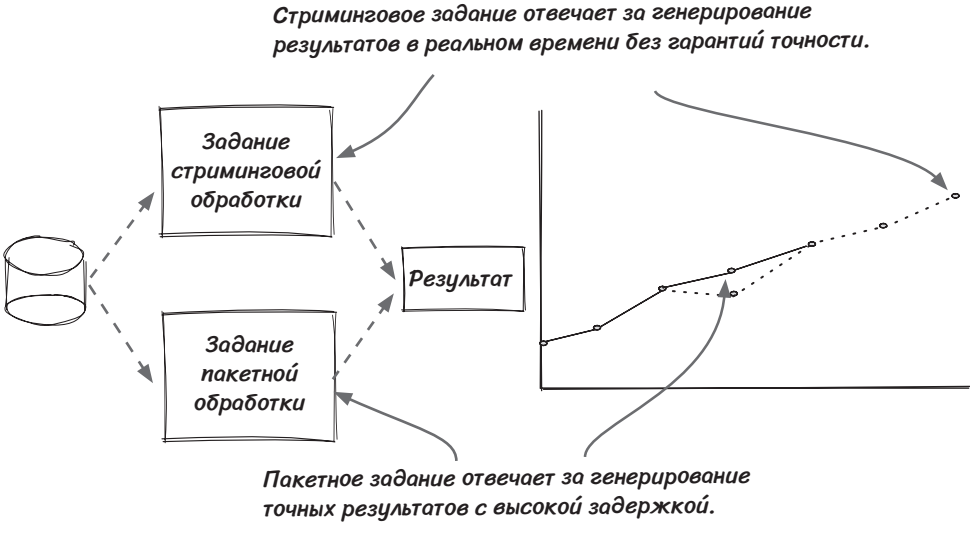
Лямбда-архитектура

Другое популярное и интересное решение называется *лямбда-архитектурой*. Название кажется сложным, но не беспокойтесь — это не так.

Чтобы понять суть этого метода, вспомните главу 1, где сравнивались системы пакетной и стриминговой обработки. При стриминговой обработке системы могут генерировать результаты в реальном времени, а системы пакетной обработки обычно обладают большей отказоустойчивостью, потому что при возникновении проблем можно легко отбросить все временные данные и заново обработать пакет событий с самого начала. Как следствие, окончательные результаты точны, потому что каждое событие обрабатывается ровно один раз для получения окончательного результата. Кроме того, поскольку системы пакетной обработки могут быть более эффективными при обработке очень большого количества событий, в некоторых случаях можно применить более сложные вычисления, которые трудно выполнять в реальном времени.

Идея лямбда-архитектуры весьма проста: стриминговое задание и пакетное задание выполняются параллельно с одними и теми же данными событий. В такой архитектуре стриминговое задание отвечает за генерирование результатов

в реальном времени, которые в основном точны, но при возникновении проблем точность не гарантирована; с другой стороны, пакетное задание отвечает за генерирование точных результатов с более высокой задержкой.



С лямбда-архитектурой приходится строить и обслуживать две системы, а представить два набора результатов довольно сложно. Тем не менее требование к точности стримингового задания может быть менее жестким, и задание может работать именно над тем, для чего оно разрабатывалось и с чем особенно хорошо справляется, — над обработкой событий *в реальном времени*.

Итоги

В этой главе мы вернулись к теме состояния экземпляров и рассмотрели ее более подробно. Затем так же подробно рассмотрели управление состояниями экземпляров и контрольными точками в стриминговых заданиях, включая:

- Создание контрольных точек с использованием событий контрольных точек.
- Загрузку контрольных точек и проблему обратной совместимости.
- Хранение контрольных точек.

После краткого сравнения компонентов с состоянием и без состояния мы узнали два популярных приема, которые можно использовать, чтобы сохранить некоторые преимущества компонентов с состоянием:

- Ручное управление состоянием экземпляров.
- Лямбда-архитектура.

Упражнения

1. Если преобразовать задание контроля за использованием системы в задание без состояния, каковы будут достоинства и недостатки такого решения? Можно ли улучшить его, вручную управляя состоянием экземпляров? И что произойдет в случае аппаратного сбоя, когда экземпляры будут перезапущены на других машинах?
2. Задание обнаружения мошеннических действий оптимизировано для обработки в реальном времени из-за требований к задержке. Каковы его достоинства и недостатки и как его можно усовершенствовать с применением лямбда-архитектуры?

11 Продвинутые концепции в стриминговых системах

В этой главе

- ✓ Более сложные аспекты стриминговых систем.
- ✓ Что дальше?

«Важно не то, сбили ли тебя с ног, — важно то, поднялся ли ты снова».

Винс Ломбарди

Вы молодец! Вы добрались до конца второй части книги, и мы достаточно подробно обсудили целый ряд тем. Давайте кратко повторим их, чтобы информация лучше закрепилась в памяти.

Это действительно все?

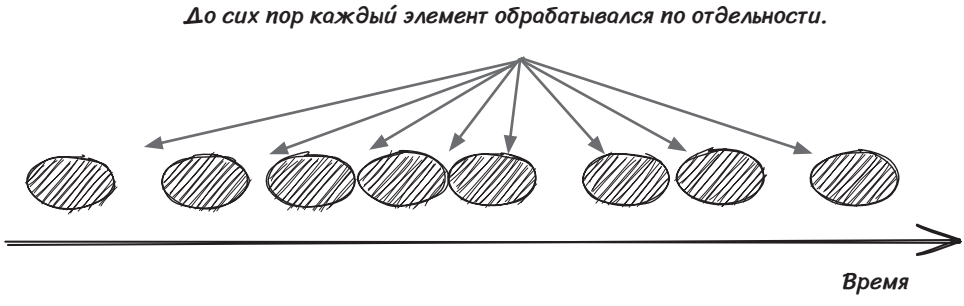
Мы, авторы, считаем, что подошла к концу книга, но не обучение и эксперименты, которые вам предстоят. И когда мы пишем эту главу, мы вспоминаем свой долгий путь учения. Сколько интересного мы узнали на этом пути! Хочется верить, что после прочтения этой книги вы почувствуете пользу новых знаний — как почувствовали мы.

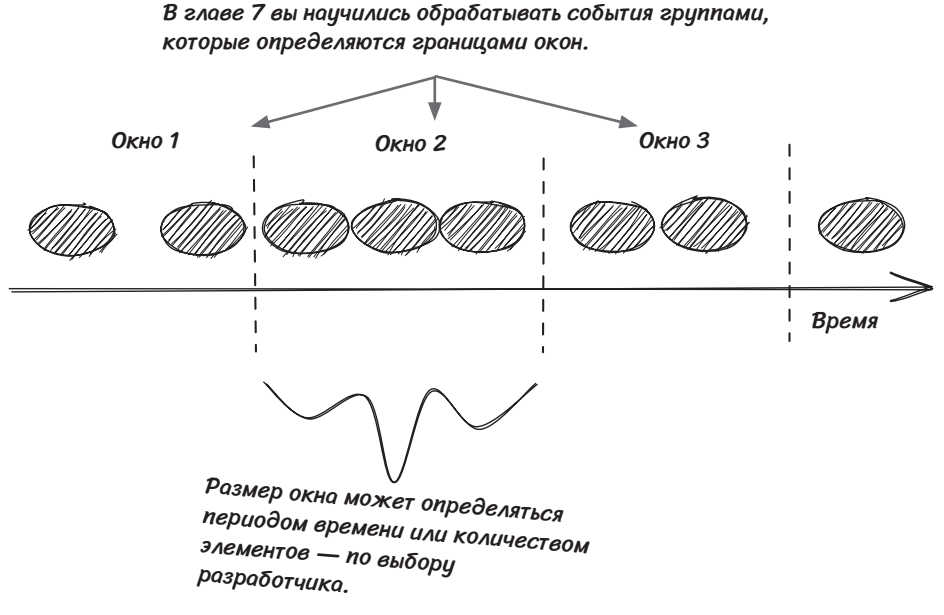
О чем эта глава

Во второй части книги рассматривались сложные темы. Мы хотим обобщить ключевые моменты. Не обязательно глубоко разбираться во всех этих темах в начале карьеры, но их знание поможет вам стать специалистом в области систем реального времени. В конце концов, само изучение этих тем — дело далеко не простое.

Оконные вычисления

Мы узнали, что не все стриминговые задания должны обрабатывать события по одному. Иногда может быть полезно группировать события — по времени или по количеству.

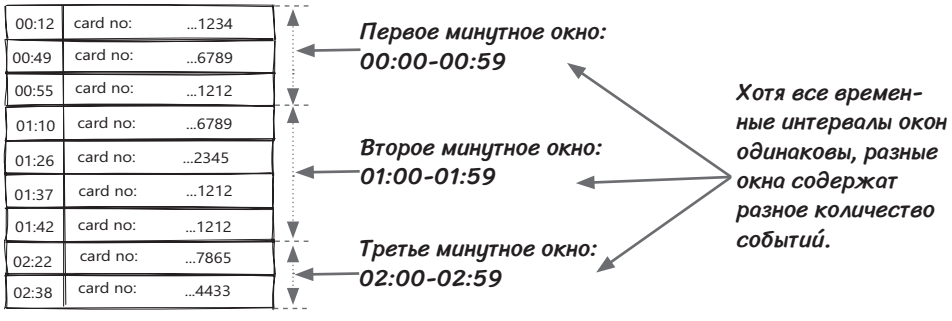




Основные виды окон

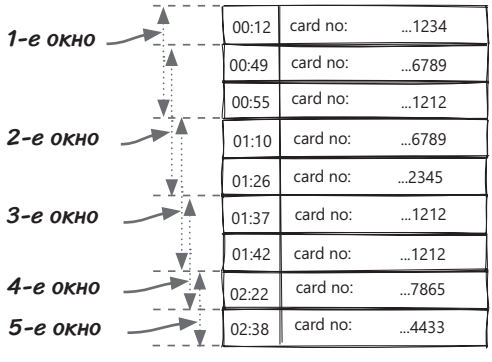
За создание или определение окна отвечает исключительно разработчик. Мы представили три основных вида окон на примере задания обнаружения мошеннических действий. На диаграммах ниже используются временные окна.

Фиксированные окна

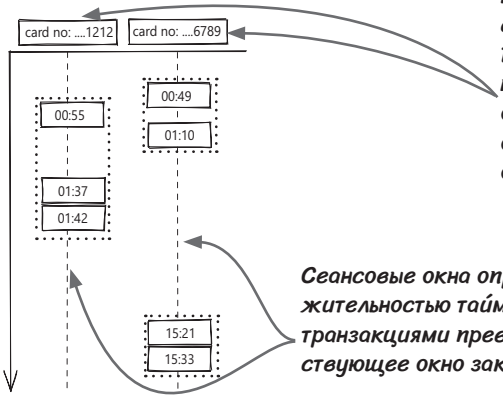


Скользящие окна

Со скользящим окном вы просто поддерживаете смещаемый контекст данных, к которому можно обращаться для принятия решения, помечать ли событие как мошенническое.



Сеансовые окна

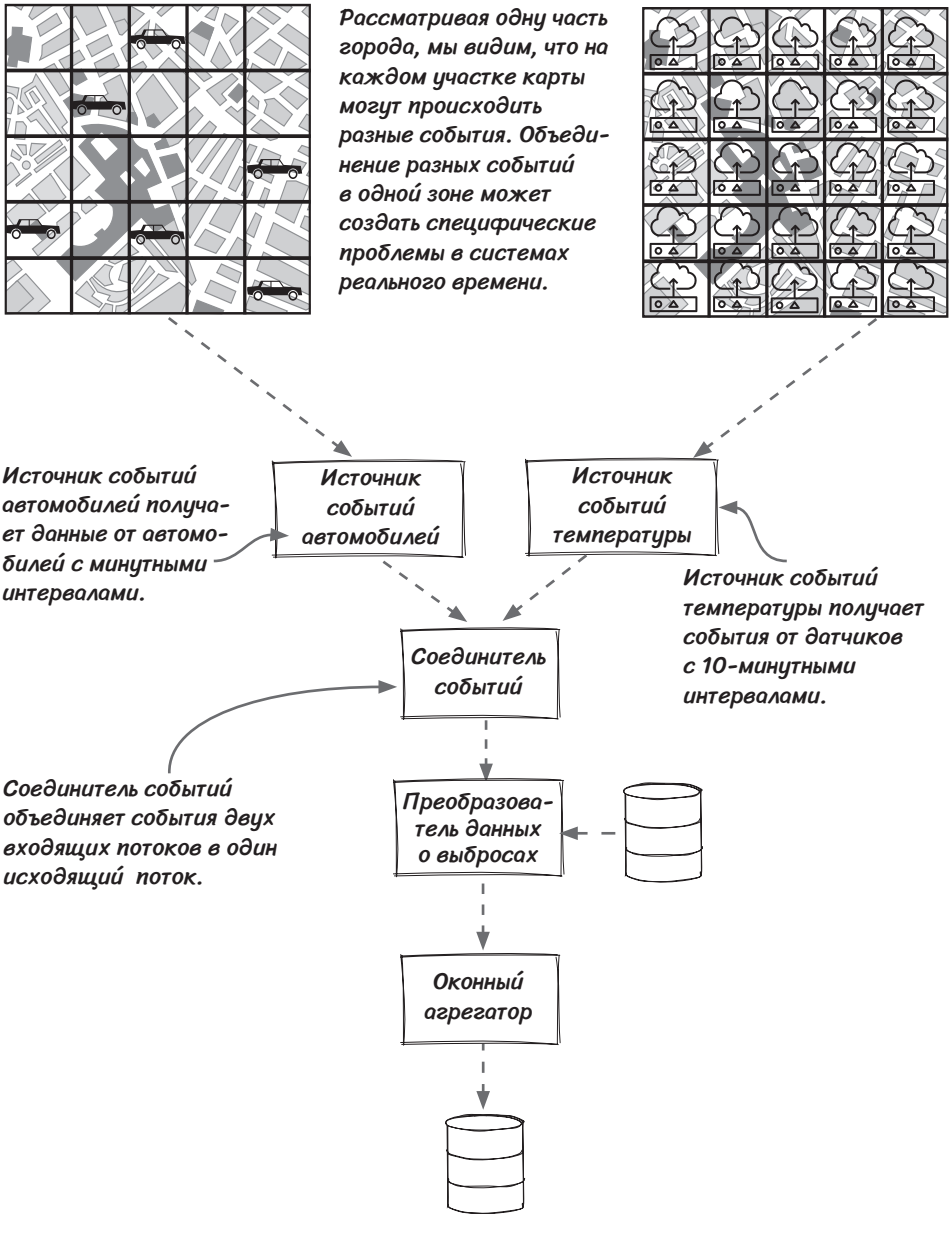


Как правило, сеансовые окна связываются с конкретным ключом. В данном примере ключом является номер кредитной карты. Таким образом, каждая кредитная карта обладает собственным уникальным окном.

Сеансовые окна определяются максимальной продолжительностью тайм-аута. Если интервал между транзакциями превышает длину тайм-аута, то существующее окно закрывается и открывается новое окно.

Соединение данных в реальном времени

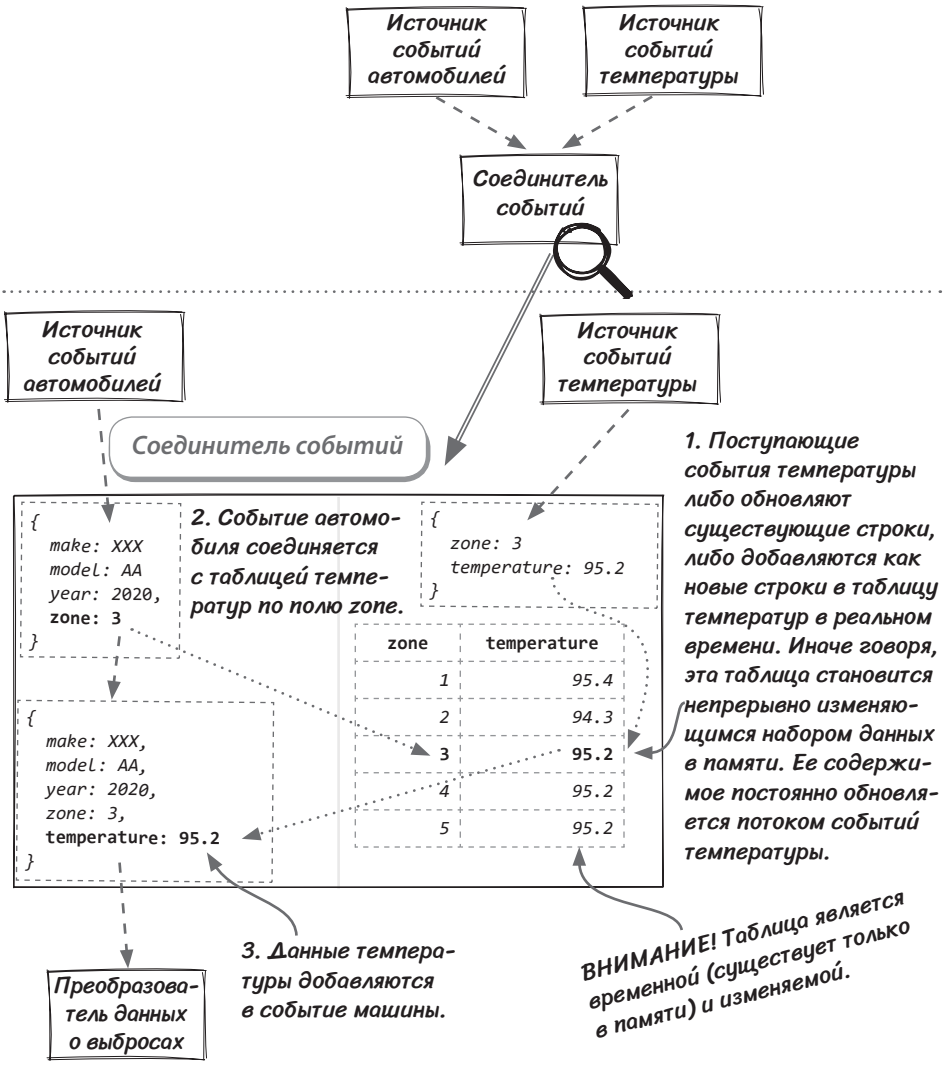
В главе 8 рассматривалось соединение данных в реальном времени. В предложенном сценарии в одной географической зоне генерировались события двух видов. Нужно было решить, как соединять события двух разных типов, поступающие с разными интервалами.



SQL и стриминговые соединения

Многим из нас известна (более или менее) конструкция `join` в SQL. В стриминговых системах соединение работает похожим образом, но есть и различия. В одном из типичных решений один входящий поток работает как поток, а другой поток (или потоки) преобразуется во временную таблицу в памяти. Эта таблица может рассматриваться как материализованное представление потока. Нужно помнить два ключевых факта:

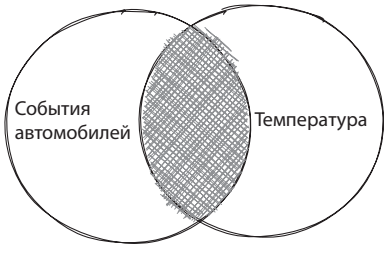
1. Потоковое соединение является разновидностью объединения.
2. Поток может материализоваться в таблицу непрерывно или с использованием окна.



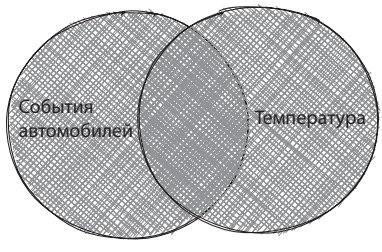
Внутренние и внешние соединения

Как и у конструкции `join` в SQL, в стриминговых системах существуют четыре типа соединений. Выбирайте наиболее подходящий для каждого случая.

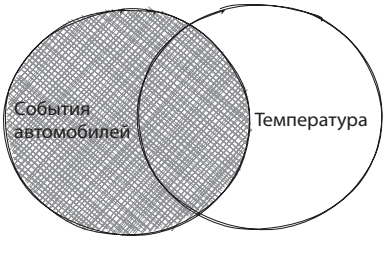
Внутренние соединения возвращают только результаты, у которых имеются соответствующие значения в обеих таблицах.



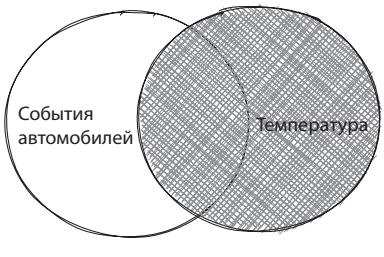
Полные внешние соединения возвращают все результаты из обеих таблиц.



Левые внешние соединения возвращают все результаты из таблицы событий автомобилей и только соответствующие строки из таблицы температур.

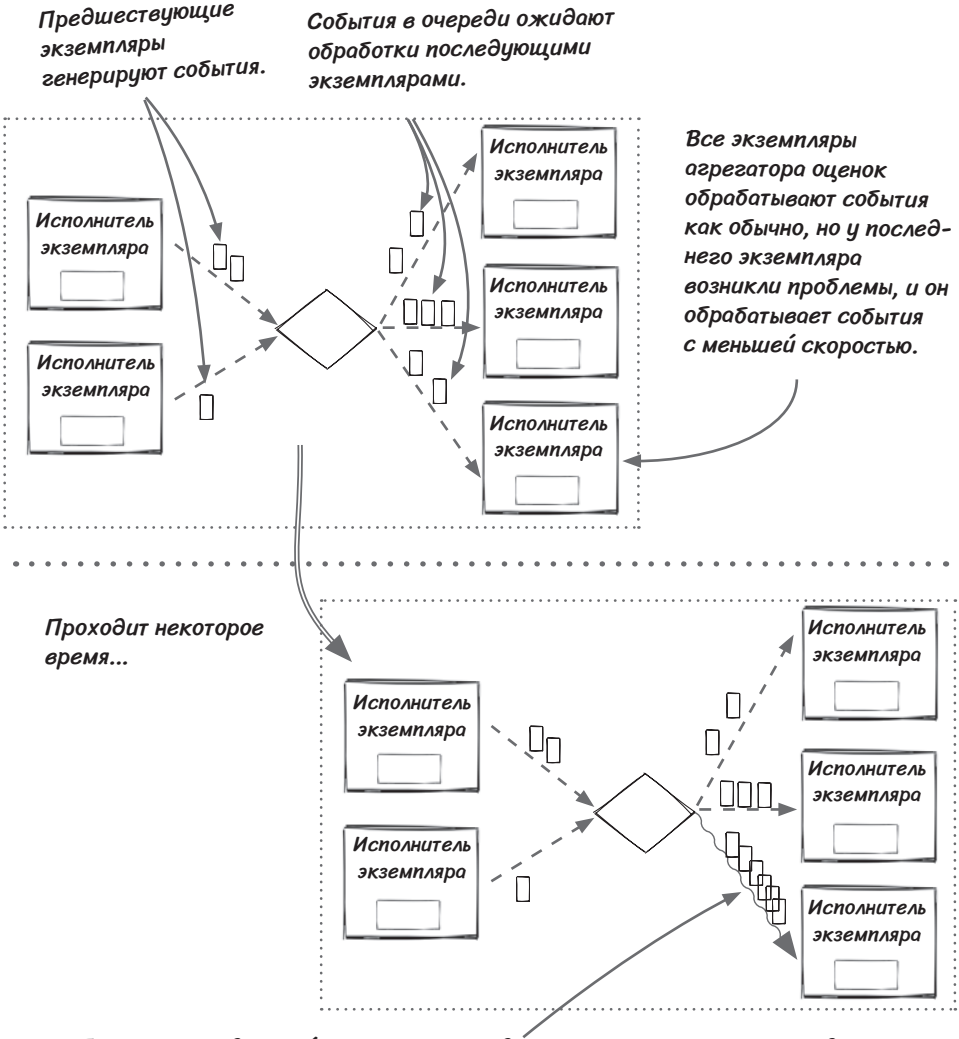


Правые внешние соединения возвращают все результаты из таблицы температур и только соответствующие строки из таблицы событий автомобилей.



Неожиданности в стриминговых системах

Создание надежных распределенных систем — непростое и интересное дело. В главе 9 были рассмотрены типичные проблемы, которые возникают в стриминговых системах и приводят к замедлению работы некоторых экземпляров, а также распространенный метод решения временных проблем — обратное давление.



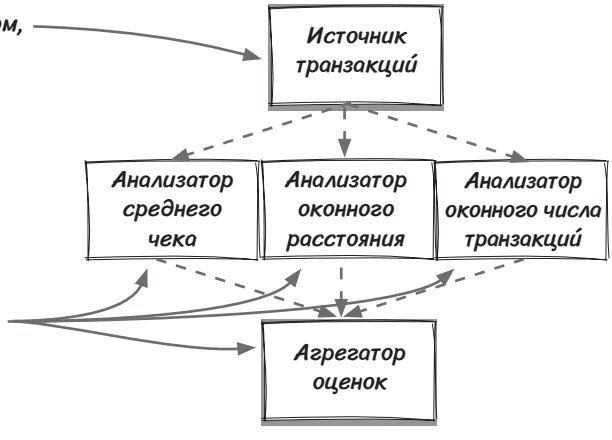
Обратное давление: замедление источников или предшествующих компонентов

Обратное давление действует в направлении, обратном направлению следования данных, и замедляет трафик событий. В книге были рассмотрены два способа создания обратного давления: остановка источников и остановка предшествующих компонентов.

Остановка источников

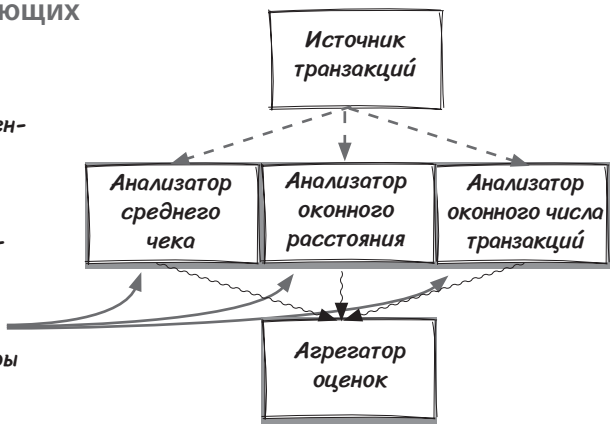
1. Чтобы временно остановить отправку дополнительных транзакций заданию источником, можно отправить специальное сообщение всем экземплярам компонента-источника.

2. С временной остановкой источника все последующие компоненты смогут завершить обработку всех транзакций, проходящих через задание, после чего источник можно будет запустить снова.



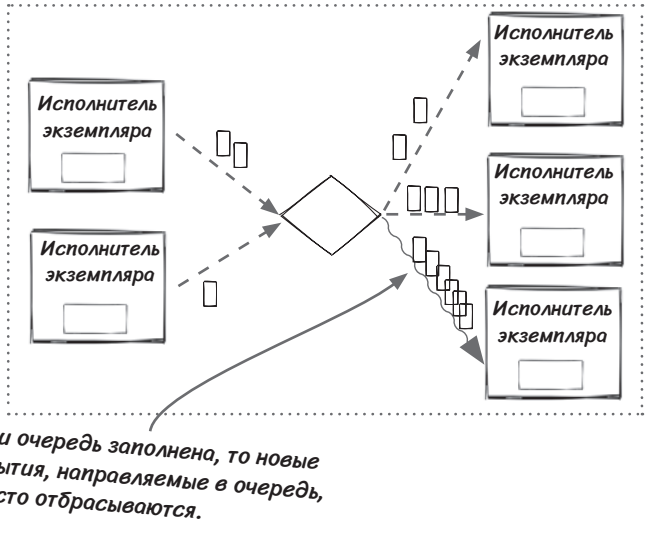
Остановка предшествующих компонентов

На уровне компонентов мы останавливаем все три компонента-анализатора, чтобы они временно перестали получать новые события от источника и направлять результаты компоненту — агрегатору оценок. Агрегатор сможет завершить обработку незавершенных событий, после чего анализаторы продолжат работу.



Другой подход к управлению отстающими экземплярами: отбрасывание событий

В этом варианте при отставании экземпляра вместо того, чтобы останавливать и возобновлять обработку источника предшествующих компонентов, система будет просто отбрасывать новые события, передаваемые экземпляру.



Несомненно, этот вариант нужно выбирать с осторожностью, так как события будут теряться. Тем не менее это не так плохо, как может показаться. Результаты неточны *только* при наличии обратного давления, что теоретически должно быть редко. Таким образом, почти все время они будут точными. С другой стороны, отбрасывание событий может быть предпочтительным в тех случаях, когда задержка между конечными точками важнее точности. Не забывайте, что отбрасывание событий также требует меньших затрат ресурсов, чем приостановка и возобновление обработки событий.

Обратное давление может быть симптомом постоянной проблемы

Мы несколько раз упоминали о том, что обратное давление представляет собой механизм самозащиты для предотвращения более серьезных проблем в экстремальных ситуациях. Хотя мы надеемся, что проблема, из-за которой некоторые экземпляры не справляются с обработкой, временна и обратное давление справится с ней автоматически, не исключено, что экземпляр не восстановится и потребуется вмешательство владельца для устранения первопричины. В таких случаях постоянное обратное давление оказывается симптомом и разработчик должен заняться проблемой.

Экземпляр перестает работать, и обратное давление не снимается

В этом случае события не уходят из очереди и состояние обратного давления никогда не будет снято. Проблема решается довольно просто: восстановлением работоспособности экземпляра. Восстановить ее можно путем перезапуска, но также важно найти первопричину проблемы и устранить ее. Часто проблема ведет к обнаружению ошибок, которые необходимо исправить.

Экземпляр не справляется, обратное давление возникает снова: пробуксовка

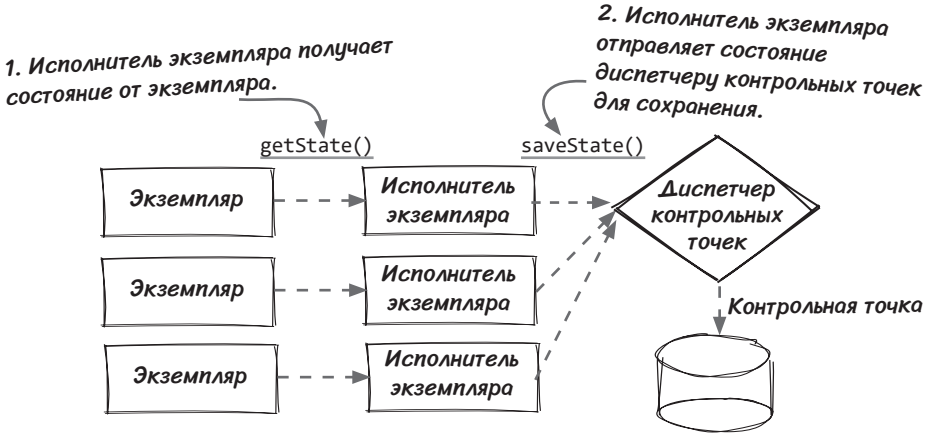
Если в системе наблюдается пробуксовка, выясните, почему экземпляр не обрабатывает события с нужной скоростью. Как правило, у подобных проблем два источника — трафик и компоненты. Если трафик увеличился или изменился его паттерн, возможно, система нуждается в оптимизации или настройке. Если экземпляр стал работать медленнее, необходимо найти первопричину. Также необходимо учитывать зависимости, используемые компонентами. В конечном итоге вы, владелец системы, должны понимать данные и логику системы и разобраться, из-за чего возникает обратное давление.

Компоненты с состоянием и контрольные точки

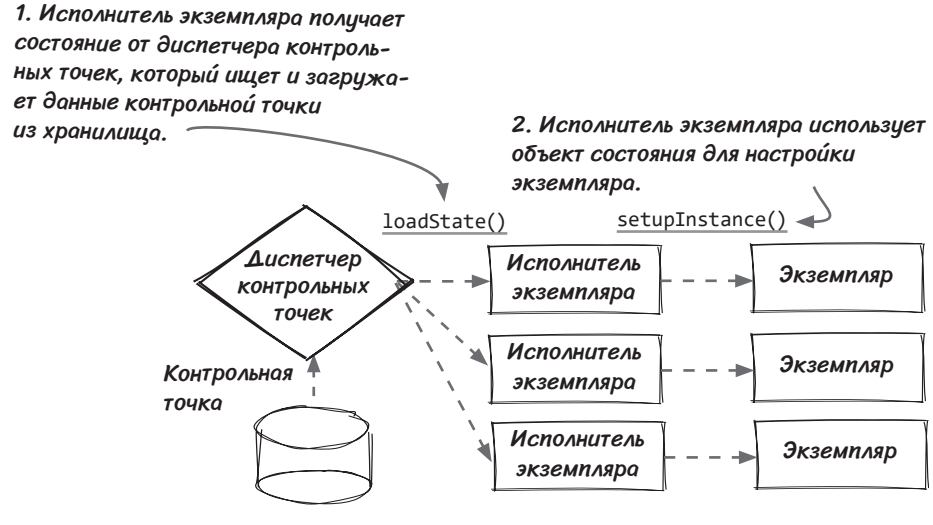
В главе 10 вы узнали, что стриминговые задания можно запускать и останавливать без потери данных. Компоненты с состоянием позволяют воссоздать контекст, так что компоненты возобновляют обработку с состояния, в котором она была остановлена ранее. В нашем конкретном случае ЭйДжей м Миранда искали возможность останавливать и перезапускать задание контроля за использованием системы на новых машинах в прозрачном режиме.

Контрольная точка — блок данных, который может использоваться экземпляром для восстановления предыдущего состояния, — играет ключевую роль в сохранении и восстановлении состояний экземпляров.

- Функция `getState()` периодически вызывается исполнителем экземпляра для получения новейшего состояния каждого экземпляра, после чего объект состояния передается диспетчеру контрольных точек для создания контрольной точки.



- Функция `setupInstance()` вызывается исполнителем экземпляра после создания экземпляра, и диспетчер контрольных точек загружает новейшую контрольную точку.



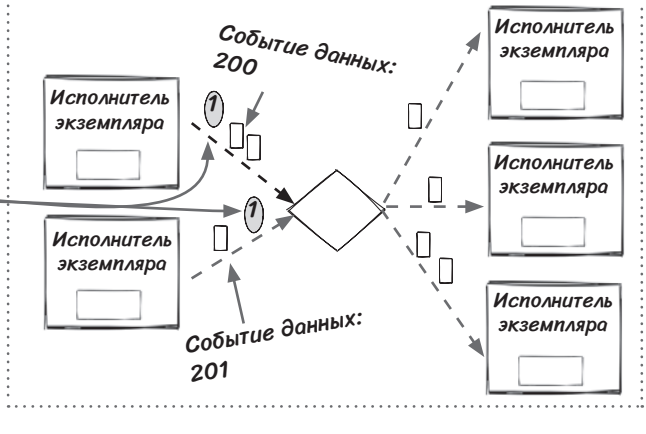
Событийный отсчет времени

Все экземпляры в стриминговом задании должны зафиксировать свое состояние одновременно, чтобы при необходимости задание можно было восстановить в состоянии на предыдущий момент времени. Однако в этом случае используется не абсолютное, а событийное время.

Периодически диспетчер контрольных точек в задании выдает событие контрольной точки с уникальным идентификатором всем экземплярам источника. Событие проходит через все задание и оповещает каждый экземпляр, что пришло время отправить свое внутреннее состояние диспетчеру контрольных точек. В отличие от обычных событий данных, которые передаются одному экземпляру последующего компонента, событие контрольной точки передается всем экземплярам последующего компонента.

На уровне экземпляров каждый диспетчер событий соединен с несколькими предшествующими экземплярами и несколькими последующими. Входные события контрольных точек диспетчера событий могут прибыть не одновременно, и их необходимо синхронизировать, прежде чем отправлять последующим экземплярам.

Два события контрольных точек должны быть синхронизированы перед их отправкой последующим экземплярам.



Компоненты с состоянием и компоненты без состояния

Вы, создатель или специалист по сопровождению стриминговых заданий, должны выбрать между использованием компонентов с состоянием и без состояния. В такой ситуации приходится опираться на интуицию или взаимодействовать с коллегами по команде для принятия решения. Не всегда понятно, какую разновидность использовать в каждой конкретной ситуации, и тогда выбор скорее становится искусством, нежели наукой. В следующей таблице сравниваются некоторые аспекты компонентов с состоянием и компонентов без состояния.

	Компонент с состоянием	**Компонент без состояния**
Точность	Вычисления с состоянием важны для семантики «ровно один», которая гарантирует точность (фактически)	Гарантии точности отсутствуют, потому что фреймворк не управляет состоянием экземпляров
Задержка (при возникновении ошибок)	При возникновении ошибок экземпляры выполняют откат к предыдущему состоянию	При возникновении ошибок экземпляры продолжают работать над новыми событиями
Потребление ресурсов	Управление состоянием экземпляров требует больших затрат ресурсов	Управление состоянием экземпляров не требует ресурсов
Трудоемкость сопровождения	Приходится сопровождать больше процессов (например, диспетчер контрольных точек или хранилище контрольных точек), обратная совместимость критична	Дополнительные затраты на сопровождение отсутствуют
Пропускная способность	Если управление контрольными точками не оптимизировано, пропускная способность может снизиться	Высокая пропускная способность не требует дополнительных затрат
Код	Необходимо реализовать управление состоянием экземпляров	Дополнительная логика не нужна
Зависимости	Необходимо хранилище контрольных точек	Внешние зависимости отсутствуют

Компоненты с состоянием превосходно работают на улучшение надежности стримингового задания, но помните, что начинать нужно с простых решений. Как только вы включаете состояние в свои стриминговые задания, сложность планирования, отладки, диагностики и прогнозирования может сделать их более чем громоздкими. Прежде чем принимать каждое решение, убедитесь в том, что вы понимаете связанные с ним затраты.

У вас все получилось!

Вы заслужили похвалу: материал был довольно объемным. Вы преодолели около 300 страниц описания работы стриминговых систем! Что же дальше? Например, можно начать углублять знания и опыт по этой теме. У вас нет диплома? Не беспокойтесь, он вам не понадобится. С упорством и самоотдачей вы, безусловно, сможете освоить стриминговые системы (и выйти на новый уровень карьеры). Вот некоторые идеи, чем заняться дальше (конечно, не обязательно делать это в указанном порядке).

Выберите проект с открытым исходным кодом

Попробуйте воспроизвести задачи, над которыми вы работали в этой книге, в реальном стриминговом фреймворке с открытым исходным кодом. Узнаете ли вы части нашего ядра Streamwork в реальных стриминговых фреймворках? Как называются экземпляры, исполнители экземпляров и диспетчеры событий в выбранном вами фреймворке?

Запустите блог и начните обучать тому, чему вы научились

Лучший способ чему-то научиться — преподавать это. Начните строить собственные решения и приготовьтесь к критическим отзывам. Всегда интересно видеть, как люди интерпретируют одни и те же концепции с разных точек зрения.

Посещайте встречи и конференции

В стриминговых системах и других системах обработки событий важны подробности и реальные сценарии использования. Из историй, рассказанных на профессиональных тусовках и конференциях, можно узнать много полезного. А можно пойти еще дальше — выступать с докладами, проводить виртуальные презентации и обсуждения!

Участвуйте в проектах с открытым исходным кодом

На наш взгляд, это самый полезный пункт в этом списке. По нашему опыту, ничто так не улучшает навыки работы с технологиями и людьми. Участвуя в проектах с открытым кодом, вы сталкиваетесь с современными технологиями и учитесь планировать, проектировать и реализовывать функционал вместе с настоящими профессионалами со всего мира. Что еще важнее, мы ручаемся, что работа над проектами с открытым кодом принесет вам больше удовлетворения, чем все, за что вам когда-либо платили.

Никогда не опускайте руки… никогда

Чтобы достичь выдающихся результатов, вам неизбежно придется терпеть неудачи — одну за другой. Примите их. Они делают вас лучше.

Ключевые концепции, рассмотренные в книге